HONEYWELL

CP-6 APL REFERENCE MANUAL

SOFTWARE

CONTROL PROGRAM-SIX (CP-6) APL REFERENCE MANUAL

SUBJECT

Description of the APL Programming Language Elements, Statements, Functions, and System Commands

SOFTWARE SUPPORTED

APL D00 on CP-6 Operating System Release D00.

ORDER NUMBER

CE38-04

May 1986

Honeywell

Preface

This document contains reference information for the D00 release version of CP-6 APL.

The Los Angeles Development Center (L.A.D.C.) of Honeywell Information Systems, Inc. has developed Computer Aided Publications (CAP). CAP is an advanced text processing system providing automatic table of contents, automatic indexing, format control, automatic output of camera-ready masters, and other features. This manual is a product of CP-6 CAP, with special handling for APL characters.

Readers of this document may report errors or suggest changes through a STAR on the CP-6 STARLOG system. Prompt response is made to any STAR against a CP-6 manual, and changes will be incorporated into subsequent releases and/or revisions of the manuals.

The information in this publication is believed to be accurate in all respects. Honeywell Information Systems cannot assume responsibility for any consequences resulting from unauthorized use thereof. The information contained herein is subject to change. New editions of this publication may be issued to incorporate such changes.

The information and specification in this document are subject to change without notice. Consult your Honeywell Marketing Representative for product or service availability.

File No.:1W13

CONTENTS

Page

CONTENTS (cont)

Page

Page

CONTENTS (cont)

TABLES

CONTENTS (cont)

FIGURES

About This Manual

This manual is organized in the following manner:

Section 1 presents an overview of CP-6 APL, its features, capabilities, and compatibility with the CP-6 operating system.

Section 2 describes the use of APL.

Section 3 describes common elements of APL.

Section 4 describes APL expression evaluation.

Section 5 describes APL primitive functions.

Section 6 describes APL statements.

Section 7 describes APL defined functions.

Section 8 describes system commands and APL workspace concepts.

Section 9 describes APL report formatting.

Section 10 describes APL execution stops.

Section 11 describes system defined functions and variables.

Section 12 describes APL file I/O.

Section 13 describes APL I-D-S/II system functions.

Section 14 describes system functions for manipulating packages.

Section 15 describes functions and variables used in APL graphics.

Section 16 describes blind I/O.

Appendix A contains information on APL parameters.

Appendix B contains a comprehensive list of APL symbols.

Appendix C contains information on error messages.

Appendix D contains information on APL's compatibility with CP-V workspace management, information on APL's compatibility with CP-V file input/output, and summarizes CP-V APL intrinsic functions.

Appendix E contains a summary of CP-6 APL.

On-Line HELP Facility

CP-6 APL has an on-line HELP facility. APL users can list syntax formats, parameters, and function or command descriptions at the terminal. For a list of HELP topics from the system level (!), enter:

```
    HELP (APL) TOPICS
```

Section 1

Introduction

APL is an acronym for A Programming Language, the language created by Kenneth Iverson. It is a problem-solving language the symbology of which closely approximates mathematical notation, making the language extremely attractive for use by engineers, financial planners, scientists, and statisticians. APL is an interpretive language designed for use on time-sharing computers. The term "interpretive language" means that APL does not wait to receive a complete program prior to compiling each statement into object code and executing it; instead APL interprets each statement as it is entered and immediately executes it. An answer is received by the user each time a portion of the total problem is stated.

APL is a powerful language: concise in notation, easy to learn and easy to use. It has many features that make it attractive for use in business applications where user interaction and rapid feedback are key requirements. One of APL's major strengths is its ability to manipulate vectors and multi-dimensional arrays as easily as it does scalar (single) values. For example, a matrix addition that might require a number of statements and several loops in other languages can be accomplished as A+B in APL. It is this type of simplification which best exemplifies APL's concise power.

This enhanced version of the processor is designed for operation under Control Program-6 and is hereafter referred to as CP-6 APL. This manual is intended primarily for use as a reference document by experienced APL programmers. Beginning APL users may find it useful to consult an APL primer to augment the information contained in this manual. Three such publications are "APL-An Interactive Approach" by Leonard Gilman and Allen J. Rose (John Wiley Sons, Inc., New York), "A Course in APL with Applications" by Louis D. Grey (Addison-Wesley Publishing Company, Inc., Reading, Mass), and "APL User's Guide" by Harry Katzan, Jr. (Van Nostrand and Reinhold Company, New York).

CP-6 APL incorporates a broad range of improvements, including a number of improvements that are unavailable on other APL systems. Some highlights of CP-6 APL include:

o APL Standards Compatibility

  CP-6 APL is a superset of the ISO APL standard.

o On-line and Batch Operation

  Complete flexibility of operation is provided. Programs may be developed and executed in any mode. The batch mode is advantageous for either long execution times or voluminous output. On-line mode is more advantageous for interactive program development and moderate amounts of execution time and output.

o Operation from Terminals without APL Characters

  APL characters may be represented by combinations of alphanumeric and special characters in order to allow programs to be created or modified on any terminal supported by CP-6.

o Input/Output Assignment Control

  The CP-6 APL system command,)SET, allows the assignment of normal and 'blind' I/O to files and devices such as line printers or magnetic tapes. It is also used to establish format control over printed output.

o Formatted Output

  Three separate formatting functions are available (monadic ⍕, dyadic ⍕, and ⎕*FMT*) to facilitate the preparation of reports and tables.

- File Input/Output

  A program-controlled mechanism is provided for file input/output. Any variable in an APL workspace may be written to a file and later retrieved for subsequent processing, permitting an APL program to operate on more data than can be contained in a workspace. APL entities may also be written as data records without their APL attributes, and non-APL records can be read.

  The CP-6 APL file I/O system operates with all CP-6 file types. File access may be with numeric keys or character keys. Files may be accessed in shared update mode, using the CP-6 Enqueue-Dequeue feature to coordinate shared access control.

- Compound Statements

  More than one statement can be included on a line using diamonds for separation. Since an item of a compound statement can be a branch, this feature permits conditional execution control within a single statement of a function.

- Blind Input/Output

  Blind input/output is a form of device input/output that permits input and output of character data. It is designed to facilitate the use of screen access modes, graphics terminals or other special devices with CP-6 APL. Using the *)SET* command, blind I/O may be used to create or access sequential files or to access devices such as line printers or magnetic tapes.

- Easy Function Copying

  An entire function can be copied simply by changing the name of an already defined function.

- Replicate

  The / function has been extended to permit non-negative integers in the left argument. The selected items of the right argument are "replicated" the number of times indicated in the left argument.

- Powerful Function Editor

  CP-6 APL permits a range of lines to be specified for display or editing. Within the range specification, it is possible to request a display of all lines containing a string or identifier, or to replace all occurrences of a string with another string.

- Enhancements to System Commands

  - The *)SEAL* command provides protected workspaces. When *)SEAL* is executed, the current workspace is saved with all user functions locked. A sealed workspace cannot be accessed by other users unless they are running APL. The workspace owner retains full access.

  - The *)TERMINAL* command allows independent setting of input and output terminal translation tables.

  - The Quiet commands (*)QLOAD*, *)QCOPY* and *)QPCOPY*) suppress the SAVED message when loading or copying successfully.

  - Options have been added to the *)SI* command to control function suspension due to errors.

  - The *)COPY* and *)PCOPY* commands allow system variables to be copied if named explicitly.

  - The *)SIL* command lists the lines in execution within the state indicator.

- Availability of Other CP-6 Facilities

  A user of CP-6 APL may use other CP-6 processors such as EDIT, PCL, and FORTRAN from the same terminal during the same session. An APL workspace may pass commands to a command processor (e.g., IBEX) and may link to other run units.

- The Execute Function

  The execute function has been extended to allow the execution of system commands.

- Observation of Intermediate Results

  The)*OBSERVE* command permits the user to view intermediate results as APL executes a statement.

- Single Stepping

  The)*STEP* command is used as a debugging aid. This command causes execution of one line of a defined function, and then immediately suspends execution.

- Catching Assignments

  The)*CATCH* command is a debugging aid which permits the user to catch (or intercept momentarily) every assignment to a named variable immediately following each assignment. The assignment is "caught" by means of a function defined by the user according to their debugging requirements.

- Error and Break Control

  CP-6 APL has a facility to provide the user with selective and dynamic control over errors and breaks. Since this facility permits bypassing of standard APL handling of breaks and errors, it is called the "sidetracking" capability.

- Text Editing Functions

  Five system functions are available to facilitate the manipulation of character vectors in CP-6 APL.

- Shared Variable System Functions

  Nine system functions are provided to support the sharing of variables between the workspaces of consenting CP-6 users. Any CP-6 user may access this facility.

- Defined Function Extensions

  A dyadic defined function may be used monadically or dyadically. If used monadically, the dummy name that references the missing left argument will be undefined.

- Database Access

  System functions are provided to access I-D-S/II databases. All of the standard Codasyl DML functions are provided and they are augmented by unique information functions tailored to the APL environment.

- Packages

  Packages provide the ability to manipulate aggregates containing variables and functions.

- Extended Error Messages

  Additional information concerning an error that APL has detected may be displayed with the)*?* command.

- Nested Arrays

  Items of an array in CP-6 APL may themselves contain APL arrays. In addition to extending most existing functions to accept nested arrays, new functions (enclose, disclose, equivalence, type, first, and depth) and a new operator (each) have been added. Defined functions, system functions, and derived functions are permitted as arguments to operators.

- Vector Notation

  CP-6 APL syntax has been extended to provide a simple notation for the entry of nested arrays.

- Vector Assignment

  This mechanism is used to assign each item of a vector to a different name in a single operation.

- Selective Assignment

  This capability allows items of an array that are selected by an APL expression to be assigned new values.

- Sorting

  The grade-up and grade-down functions have been extended to sort character arrays and arrays of any rank.

- Least Common Multiple Function

  The OR primitive function (v) has been extended to provide the Least Common Multiple function.

- Greatest Common Divisor Function

  The And (^) primitive function has been extended to provide the Greatest Common Divisor function.

Section 2

Using APL

Logging On

The user must first prepare the terminal for use, establish a connection with the CP-6 system, and then invoke the APL processor. This is done as follows:

1. Connecting to the CP-6 system:

   a. Press the number 8 several times until CP-6 responds with:

   ```
   PLEASE TYPE A LEFT PARENTHESIS
   ```

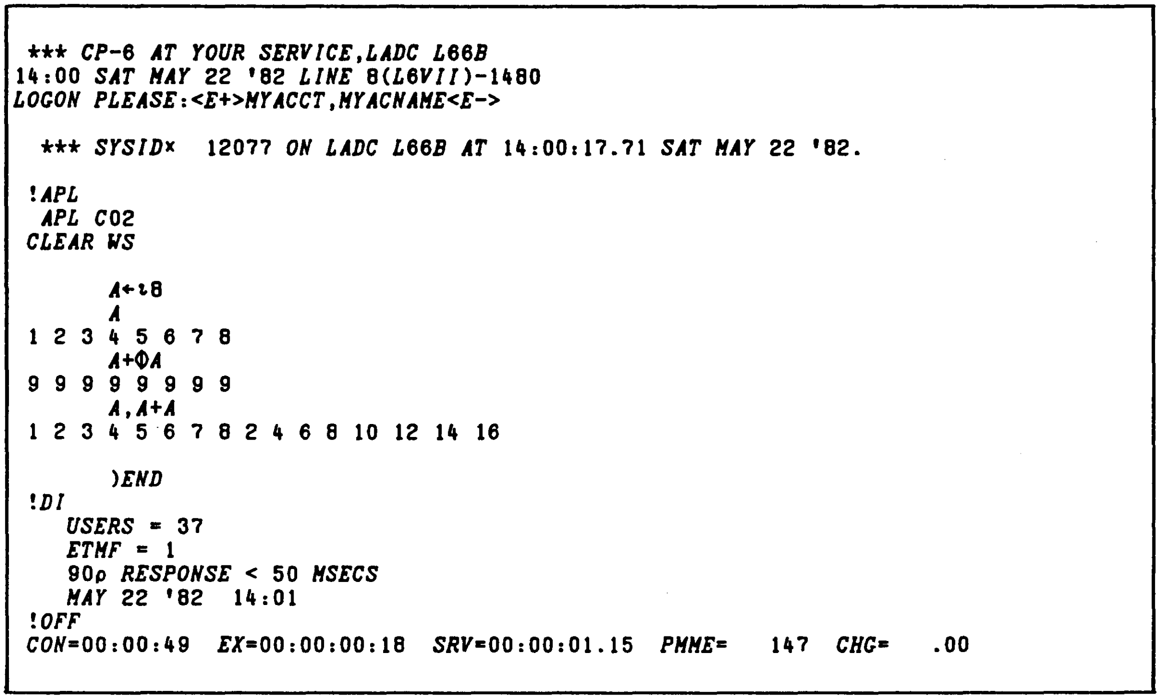
   b. The system requests that the user enter a left parenthesis. Once a left parenthesis is entered, a salutation is printed after which the system requests a logon. At this time a valid logon should be entered. A logon consists of an account, name, and optional password, separated by commas. This information is not echoed (printed) on the terminal to provide privacy.

   ```
   *** CP-6 AT YOUR SERVICE, LADC L66A
   14:30 THU OCT 17 '85 LINE 8(L6VI)-1480
   LOGON PLEASE:
   ```

   c. The CP-6 system will then allow the user to log on to the system with an attendant greeting, or inform the user of the reason for not logging on.

   d. When the CP-6 system prompts with !, the user is at the IBEX Command Processor level and may invoke APL by typing APL and pressing RETURN.

Figure 2-1 shows a sample APL session including logon and logoff, as performed from a Diablo 1620 or equivalent terminal with an APL typewheel.



```
 *** CP-6 AT YOUR SERVICE,LADC L66B
14:00 SAT MAY 22 '82 LINE 8(L6VII)-1480
LOGON PLEASE:<E+>MYACCT,MYACNAME<E->

  *** SYSIDx  12077 ON LADC L66B AT 14:00:17.71 SAT MAY 22 '82.

 !APL
  APL C02
 CLEAR WS

       A←⍳8
       A
1 2 3 4 5 6 7 8
       A+⌽A
9 9 9 9 9 9 9 9
       A,A+A
1 2 3 4 5 6 7 8 2 4 6 8 10 12 14 16

       )END
!DI
   USERS = 37
   ETMF = 1
   90ρ RESPONSE < 50 MSECS
   MAY 22 '82  14:01
!OFF
CON=00:00:49  EX=00:00:00:18  SRV=00:00:01.15  PMME=    147  CHG=    .00
```

Figure 2-1. Sample APL Session

General APL Input

The following paragraphs define the APL character set, APL names, and various input/output characteristics.

Character Set

One of CP-6 APL's unique characteristics is the richness of its character set. An APL keyboard normally has 94 printing graphics. All of these are legal characters. In addition, backspacing may be used to create the following overstrikes, all of which are legal characters:

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

⌽ ⍉ ⊖ ⍟ ⍒ ⍋ ! ⌶ ⍞ ⌹ ⍝ ⍱ ⍲ ⍫ ⌿ ≠̲
⍀ $ ⍙ 0⃞ 1⃞ 2⃞ 3⃞ 4⃞ 5⃞ 6⃞ 7⃞ 8⃞ 9⃞ ⍑ ⍦ ⍘ ≡̲

Other legal characters are blank (the space bar), tab (the TAB key, treated as one or more blanks), and carriage return (the RETURN key). Two other characters are also accepted for control purposes: the <CTL-D> sequence and the BREAK key discussed below under "Line Corrections during Input" and "Control Keys".

Names

Names are used to identify certain CP-6 APL constructs. All variables, functions, groups, workspaces, and statement labels have names; the following restrictions apply to these names:

1. All names except workspace names can contain from 1 to 79 characters. Workspace names can contain from 1 to 31 characters (see Section 8).

2. Names may be composed of letters, numbers, Δ, underlined letters, underlined Δ, and underscore.

3. Names cannot begin with a number or underscore.

4. There can be no blanks embedded within a name.

5. A particular kind of name, called a distinguished name, begins with ⎕.

Some examples of names are:

Δ PAYROLL BΔ1 S1234 TEMPERATURE ⎕PW

User Input versus Computer Output

The user can enter input whenever the carrier or cursor is indented six spaces from the left margin. As soon as the user has typed any input and pressed the RETURN key, APL takes control. Characters entered by the user while APL is processing will be "stored" until APL has completed processing the previous input, printed any results, and prompted for more input (usually by indenting six spaces from the left margin).

User input and computer output are easily distinguished. Computer output usually begins at the left margin while user input is usually indented six spaces. For example:

```
      )DIGITS 2
WAS 10
      3÷9
0.33
      2+2
4
      4÷2
2
```

Everything at the left margin in this example is printed by APL, while everything which is indented is typed by the user.

Line Corrections during Input

A line can be corrected during input as long as the RETURN key has not been struck. Simply strike the RUBOUT key, to delete characters up to the error and enter <ESC> R to retype the correct portion of the line. Then proceed with the entry of the line. For example, suppose the user mistakenly types 30-20 instead of 30+20. The user can correct this as follows:

```
      30-20\\\<R>          enter three RUBOUTS and <ESC> R
      30+20                the system displays 30; user enters +20
50                         system responds with 50.
```

Perhaps the simplest line correction method is to delete all of the input with the control X character. Another correction method can be employed if the user discovers that a character has been omitted. As long as the RETURN key has not been struck, the user can simply backspace to where the character is to be inserted (or enter <ESC> V followed by the character at which to position), enter <ESC> J, and type it. For example, suppose the user types the following line and notices that one left parenthesis is missing:

```
      (1○H)*2)+(2○H)*2
```

By simply backspacing and typing the required left parenthesis, the user can enter

```
      ((1○H)*2)+(2○H)*2
```

This illustrates that it is not always necessary to enter characters in order. The user can leave blanks in a line, then backspace and fill them in. As a rule, APL interprets what the user sees at the terminal; this is known as visual fidelity. For more information on standard CP-6 input line editing, see the CP-6 Programmer Reference Manual (CE40).

Execution and Definition Modes

From the user's viewpoint, CP-6 APL operates in two modes, execution mode and definition mode. In execution mode, the processor responds to each line of input by taking a specified action or by performing requested calculations and printing a result. In the following printout, for example, the first line is a system command that causes the processor to take some action and to respond with a message, and the third line (3÷9) performs a calculation, printing the results on the fourth line:

```
      )DIGITS 2
WAS 10
      3÷9
0.33
```

System commands can be entered during execution or definition mode. Calculations are performed only in execution mode.

In definition mode, statements (that is, calculations) are saved as part of a defined function instead of being executed immediately. System commands issued in this mode, however, are executed immediately. After functions are defined, they can be referenced in other defined functions or in statements entered in execution mode. The user must type the del symbol ∇ to begin definition mode, and another ∇ to return to execution mode. See section 7 under Defined Functions, for a detailed description of definition mode.

Prompts

CP-6 APL has four ways of prompting for (that is, requesting) input: direct line prompt, function line prompt, evaluated input prompt, and quote-quad prompt. These are described below.

Direct-Line Prompt

When APL is ready for user input in immediate execution mode, it automatically moves six spaces in from the left margin. This is a signal to the user to enter a statement or system command. Direct-line prompts are shown in the following example:

```
      2+2
4
      4÷2
2
```

In this example, APL indented six spaces to prompt for user input, and the user entered the statement 2+2. The processor then printed the result of the calculation at the left margin, moved to the next line, and again indented six spaces to prompt for more input.

Function-Line Prompt

Within definition mode (that is, when a function is being defined) CP-6 APL prompts for user input by printing a line number in brackets at the left margin. After printing the line number, it moves three spaces to the right and waits for user input. As an example, look at the following portion of a function definition:

```
      ∇SQUARE
[1]   A←(B×B)
[2]
```

In this example, the user entered a function header (*∇SQUARE*), and APL typed the [1] and moved three spaces to the right to prompt for user input. The user then entered the statement *A←(B×B)*, and APL typed the [2] to prompt for more user input. This continues until the user ends the function definition with another del symbol ∇.

Quad Prompt

The quad symbol ⎕ can be used in a statement to indicate evaluated input. When APL encounters the quad on execution of the statement, it halts and requests input by printing the symbols ⎕:, moving to the next line, and indenting six spaces. The user can enter any valid APL expression. This expression will be evaluated, and its value substituted for the quad contained in the statement. Execution of the statement then resumes. Examples of the quad prompt are shown below:

```
      A←⎕÷8
⎕:
      7×2×4
      A
7
      ANSWER←⎕
⎕:
      'YES'
      ANSWER
YES
```

Quote-Quad Prompt

The quote-quad symbol ⍞ (a quote symbol overstruck with a quad) is used to enter character data. It is executed similarly to the quad symbol except that nothing is printed to signal the user, and no six-space indentation takes place. The user enters character data without enclosing it in quotes. For example:

```
      A1←⍞
YES
      A1
YES
```

Comments

Comments can be written on separate lines or can follow (that is, be tacked onto) statements. They may be included on any line except a system command line or a function edit control line. To enter a comment, type the symbol ⍝ and follow it with the comment. This symbol is produced by typing a ∩ symbol (upper shift C) and overstriking it with a ∘ symbol (upper shift J). Any valid APL characters may appear to the right of the ⍝ symbol. The ⍝ and any characters to the right are ignored in APL expression evaluation, but will be printed if the line is displayed. Examples of comments are shown below:

```
      ⍝ THIS IS A COMMENT.

      A̲←B×B  ⍝SET A̲ = B-SQUARED.

[3]   X←Y+5  ⍝ COMMENT: X IS SET TO Y+5
```

Control Keys

The BREAK key is used to interrupt execution or stop a lengthy display on the terminal.

Statements and System Commands

Each completed line of input in CP-6 APL is classified as either a statement or a system command. Statements specify the operations to be performed by APL, such as calculations, branching, and assignments of values or expressions. Some examples of statements are:

```
      4+2
      B←A÷2
      →START
      ∇A PLUS B
[3]   'ENTER VALUES FOR A'
```

System commands are used to communicate directly with the APL system itself. They are concerned primarily with the mechanical aspects of the processor, such as logging on and off, saving, loading, and deleting workspaces. System commands always begin with a right parenthesis. A few examples of system commands follow:

```
      )SAVE NEWJOB
      )LOAD OLDJOB
      )END
      )DIGITS
```

Statements and system commands are described in detail in sections 6 and 8, respectively.

Variables and Functions

Data (numeric or character) can be assigned a name and stored in the active workspace. The name and the associated value are collectively known as a variable. The value may be a single data item (scalar) or a group of data items (array), and may be changed as needed during the course of a program. Examples of assignments of variables are shown below:

```
A←5
B2←1 2 3
ABC←5+4
B3←A+B2
```

Some character symbols indicate that basic APL operations, such as addition or multiplication, are to be performed. These symbols are called primitive functions. Functions can be monadic (have one argument) or dyadic (have two arguments). Some examples of functions are:

```
×
+
⌈
÷
```

The domain and range of function arguments and a list of all the functions are presented in Section 3 under Primitive Functions. Section 5 is devoted to a detailed discussion of each function.

Defined Functions

In addition to the primitive functions, APL permits users to define new functions, name them, and store them in a workspace. Defined functions can then be referenced by name in subsequent statements, either as programs by themselves or as mathematical operations used in a formula. To define a function, the user enters it statement by statement while APL is in definition mode. This mode begins when the user types a del symbol ∇ and ends when another ∇ is typed.

Section 3

Common Elements in APL

Constants

Constants are either numeric or character.

Numeric Constants

Numeric constants can take the form of integer or real numbers. An integer is a whole number, requiring neither decimal point nor exponential form. A real number is a number, usually with a decimal point, expressed in either exponential form or decimal form. The user need not generally be concerned with whether a number is integer or real, or exponential or decimal, since APL automatically takes care of any necessary conversions. The representation of numeric data is accomplished with the following characters:

0 1 2 3 4 5 6 7 8 9 . ¯ *E*

The numbers are the ordinary keyboard digits, and the decimal point is the keyboard period. The ¯ character, called the negative sign, is found over the digit 2 on an APL keyboard and is used to indicate negative numbers. It should be distinguished from the - character, which is found over the + symbol and is used for subtraction. The negative sign is only valid for numeric constants; it is not valid in any other context. The *E* is the letter E on the keyboard and is used to indicate an exponent. Embedded blanks, commas, and other punctuation are not allowed in APL numbers.

APL ignores leading and trailing zeros, so that the user need enter only the parts of numbers required for calculations. Thus, there is no need for the user to enter data as all integer or all fractional. For example, the number one may be entered as 1.00, 001.0, 1, etc. Examples of numeric constants entered in decimal form are shown below:

```
      5 + 5.55
10.55
      6.8 ÷ 20
0.34
```

The negative symbol (¯) can be used only with a numeric constant to indicate a negative number; it can never be used with a name. The symbol immediately precedes the applicable number; that is, no blanks are allowed between the symbol and the number. The use of the negative symbol is shown below:

```
      ¯2
¯2
      ¯4 + ¯5
¯9
      4 - ¯3
7
```

It is often easier to enter very large numbers in exponential form rather than decimal form. Exponential representation is written as a number, followed by *E*, followed by an integer indicating a power of 10. (*E* can be interpreted as "times 10 to the following power".) The exponent (the number following the *E*) can be a positive or negative number. Following are some examples of numeric data in exponential form:

APL Exponential Notation	Mathematical Notation
¯8.37E14	-8.37×10^{14}
4.2E¯6	4.2×10^{-6}
.99E5	$.99 \times 10^{5}$
3.8E¯60	3.8×10^{-60}

The maximum and minimum magnitude representable numbers in CP-6 APL are approximately:

```
    8.379879956E152

    4.661462957E¯156
```

Note that non-integer values are handled internally as "double precision floating point" numbers. Fractions that are representable exactly in decimal notation, such as .1, are not exactly representable in this internal form. In some instances, this will cause results of operations to deviate from expected results, particularly if the anticipated result is displayed to 20 decimal places or is a value near zero.

Character Constants

Character constants are enclosed in quote symbols and can contain any keyboard character including legal overstrikes and the space character. The quote symbols are used to distinguish a character constant from a number, the name of something, or a constant in the language. They are not printed in the display of the literal. For example:

```
      A←'?'
      A
?
```

In this example, the name A has been assigned the value of a character constant.

Vector Notation

When two or more values appear together separated by one or more blanks, a vector is formed. The vector that is formed has the properties of length (the number of items), type (numeric, character or nested), and rank (vector). Some examples of numeric vectors are:

```
      1 1 3
      1 2.5 ¯726E12
```

Character vectors may be formed either as a series of character scalars with each item enclosed in quotes, or by enclosing the entire string in quotes. For example:

```
      'H'  'I'  ' '  'T'  'H'  'E'  'R'  'E'
HI THERE
      'HI THERE'
HI THERE
```

Both character vectors are equivalent. If a quote is to be used within text, it must be represented by two quotes. The use of the quote character is shown below:

```
    A←'THE ''⍝'' CHARACTER IS USED FOR COMMENTS.'
    A
THE '⍝' CHARACTER IS USED FOR COMMENTS.
```

Character arrays may be generated, compared for equality, indexed and catenated just like any other arrays.

A character constant may contain one or more carriage returns. If a carriage return is entered before the closing quote is given, APL will automatically type the closing quote at the beginning of the next line to indicate that a closing quote is required to end this string. If the constant is to be extended, a RUBOUT may be entered to delete the closing quote.

Parentheses may also be used to separate items in vector notation. For example:

```
      A←1(2)
      A←(1)  2
      A←(1)  (2)
```

The three examples above are all equivalent ways of forming the two item vector 1 2. Multiple blanks and extra parenthesis are also always permitted:

```
      A←19  20
      A←((19))  ( ( ( 20 ) ) )
```

The use of parenthesis in vector notation is used to produce a single item out of any array that they enclose. The parenthesis may also enclose any array. For example:

```
      A←('YEAR') 1983  ('SALES')  (2619 5250)
      A[3]
SALES
```

In this example, vector notation has produced a four item vector which contains the vector *'YEAR'* as the first item, the scalar 1983 as the second item, the vector *'SALES'* as the third item, and the numeric vector 2619 5250 as the final item. Parentheses are not required around character vectors because the enclosing quotes are already grouping them. For example:

```
      A←'YEAR' 1983 'SALES' (2619 5250)
```

This example produces the same four item vector as the previous example.

Names

All of the following constituents of the APL language have names (sometimes known as identifiers) so that they may be easily referenced: variables, functions, groups, statement labels, and workspaces.

Name Format

A name can include only letters, the letters underscored, digits, ∆, ⍙, and _ characters. A name cannot start with a digit or an underscore. Distinguished names follow the other rules for names, but always start with a single ⎕ character.

Lengths of names may vary, depending on their use. The names of variables, functions, groups, and statement labels can be of any length up to 79 characters. Workspace names (also known as fids in CP-6 APL) can be up to 31 characters in length.

Name Usage

The uses of APL names are described below:

1. A variable refers to the name given to scalar or array values by the assignment symbol (the character '←') described later in this section under Assignment.

2. Defined function names are treated briefly later in this section under Function References, and in detail in Section 7, Defined Functions.

3. A collection of names can be referenced using groups. Included in the group can be the names of variables, functions, and other groups (see the)*GRP* command in Section 8).

4. A label is given to a statement within a user-defined function so that it may be referenced by other statements of that function. Statement labels are used as branch reference points.

5. A workspace name is used to identify an active workspace so that it can be saved and later recalled. Workspace names are referenced in system commands which are described in Section 8 (also see item 8).

6. A password is assigned to a workspace or file to prevent other users from accessing it. The password must always be used in order to access the workspace or file. Passwords are described in Section 8 (also see item 8, below). Passwords may contain any characters.

7. An account is the identifier of a recognized user's account. The account must be specified when logging on to the CP-6 system and when accessing a workspace or file in another user's account. The use of accounts is described in Section 8 (also see item 8, next). Accounts may be, but are not restricted to, letters or digits.

8. In CP-6 APL, a saved workspace is a CP-6 file. A file identifier (fid) refers to the information needed in a system command to save a workspace or to reference it after it has been saved. A file identifier takes the following form:

   ```
   workspace[.[acct][.[password]]]
   ```

   where

   workspace   is the name assigned to the workspace, or file. It can consist of up to 31 characters from the set *A-Z*, *A̲-Z̲*, -, :, _, $, and 0-9.

   acct   is the identifier of a recognized user's account. It can consists of up to eight characters from the set of accounts authorized by the installation manager.

   password   is assigned to a workspace, or file, in order to restrict user access. It can consist of up to eight characters.

   The bracketed items in the above form indicate optional items. File identifiers are used in the following system commands, all of which are described in Section 8:)*LIB*,)*COPY*,)*DROP*,)*LOAD*,)*PCOPY*,)*QCOPY*,)*QLOAD*,)*QPCOPY*,)*SAVE*,)*SET*, and)*WSID*.

   Accounts and Passwords may include any characters except the period, comma, semicolon, or embedded blanks.

   For further information on file identifiers, see the the documentation on the command processor IBEX in the CP-6 Programmer Reference Manual (CE40). Set names and serial numbers are also discussed there.

Variables

A variable must be assigned a value before it can be used. The value assigned can be numeric, character, or nested and can be a scalar or an array (a vector, a matrix, or a higher-order array). The user can display the value of a variable at any time simply by typing the variable name. Examples of the assignment and use of variables are shown below:

```
      A←2
      B←2 3 4 5
      A+B
4 5 6 7
      C←4 5⍴⍳20
      C
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
      D←B÷2
      D
1 1.5 2 2.5
```

A variable can be respecified at any time simply by assigning a new value to the variable name. The most recent value specification replaces any previous value. For example, notice the following:

```
      ABC←1
      ABC×0 1 2 3 4 5
0 1 2 3 4 5
      ABC←2
      ABC×0 1 2 3 4 5
0 2 4 6 8 10
```

In this example, *ABC* is first assigned a value of 1 and calculations are performed with that value. The variable *ABC* is then assigned a value of 2 and the calculations are performed using this new value.

Another way of respecifying a variable value is to decrease or increase its value by a certain amount. For example, suppose variable A has a value of *A* has a value of 2 and the user want to increase this value by 1. This can be accomplished as follows:

```
      A←A+1
      A
3
```

Notice that the calculation 2+1 is performed first, and then the result 3 is assigned to a variable *A*. This type of operation is particularly useful for setting up a counter to test the number of occurrences of an event, such as the number of passes through a program loop. Each time through the loop the counter can be increased or decreased by 1 and then tested against a desired value to determine further action.

Local and Global Variables

Local variables exist while user-defined functions (Section 7) are active, that is, while the function is pendent or suspended. Local variables, described below, are classified as follows:

- Dummies
- Result
- Locals
- Labels

Dummies, result, and locals are indicated by their presence in the header of a defined function. Labels are indicated on statements within a defined function.

At a given point in time if a variable is not local, it is global. It is possible (in fact useful) to allow global variables to be identified by the same name as local variables (or local variables for one function to use the same name as local variables for another function). This concept is useful in APL because it allows a defined function to be formed without regard to name conflicts. Its local variables are totally independent of any previously assigned variables. Furthermore, if the function calls itself, a new set of variables exist independent of the original local variables. As each such function call exits (that is, becomes inactive again), the current set of local variables disappear and the earlier values associated with their names once more become accessible.

When a function call occurs, its local variables are said to "shadow" previous definitions for the names used by the local variables. Shadowing can be repeated extensively as functions are called. As these functions exit, their shadowing effect is removed. Only globals will exist when no function is active. Global variables also exist if their names are not shadowed by any currently active functions (for example, the local variables use unique names). Shadowing is illustrated in Figure 3-1.

Local Variables

The following local variables are named in a function header: result, dummies, and locals. These are all optional; a function is not required to use any local variables. Notice the following example:

```
    ∇R←Y F X;A;B;C
```

In this example, the function *F* names the following local variables in its header line:

R (result) - note that *R* is followed by a ← symbol, which designates that *R* is the result name.

X (dummy) - one name to the right of *F* separated by blanks(s), designates the right dummy. When *F* is called, the right argument's value is automatically assigned to local variable *X*.

Y (dummy) - one name to the left of *F*, separated by blank(s), designates the left dummy. When *F* is called, the left argument's value is automatically assigned to local variable *Y*.

A, *B*, and *C* (locals) - note that each local name is preceded by a semicolon.

The remaining type of local variable is the label. Its name appears in a function line as in the example below.

```
    [3]  L:⍝ THIS LINE IS LABELED.
```

Notice that the label's name, *L*, follows the line number, [3], and is in turn followed by a colon. Although labels are classified as local variables, it is more appropriate to consider them local constants. They cannot be assigned values; that is, the following expression is a syntax error when *L* is a label:

```
    L←4
```

The value of a label is the line number of its function line (which cannot change during execution of the function).

The example in Table 3-1 illustrates the effect of shadowing as functions *F1* and *F2* become active and inactive.

Table 3–1. Effect of Shadowing

Example	Description

)CLEAR
CLEAR WS
V←'V=GLOBAL'
W←'W=GLOBAL'
X←'X=GLOBAL'
Y←'Y=GLOBAL'
	Set V, W, X, Y
	to be global variables.

∇ F1;X;Y [1] ' .....F1 CALLED......' [2] V [3] W [4] ,X←'X=LOCAL (F1)' [5] ,Y←'Y=LOCAL (F1)' [6] F2 ⍝ CALL F2 [7] ' .....F1 EXITS......'∇	Define F1, naming X and Y as its locals.
∇F2;W;X [1] ' .....F2 CALLED......' [2] V [3] ,W←'W=LOCAL (F2)' [4] ,X←'X=LOCAL (F2)' [5] Y [6] ' .....F2 EXITS......'∇	Define F2, naming W and X as its locals.
V W X Y V=GLOBAL W=GLOBAL X=GLOBAL Y=GLOBAL	Verify V, W, X, Y.
F2 .....F2 CALLED...... V=GLOBAL	Call F2.

W=LOCAL (F2)
X=LOCAL (F2)
Y=GLOBAL
.....F2 EXITS......
	V and Y are still global.
	W and X are local to F2.

V W X Y V=GLOBAL W=GLOBAL X=GLOBAL Y=GLOBAL	V, W, X, Y are global again.
F1 .....F1 CALLED...... V=GLOBAL	Call F1.

W=GLOBAL
X=LOCAL (F1)
Y=LOCAL (F1)
	V and w are still global.
	X and Y are local to F1.

.....F2 CALLED...... V=GLOBAL	F1 calls F2.

W=LOCAL (F2)
X=LOCAL (F2)
Y=LOCAL (F1)
.....F2 EXITS......
.....F1 EXITS......
	V is still global.
	W and X are local to F2.
	Y is still local to F1.

V W X Y V=GLOBAL W=GLOBAL X=GLOBAL Y=GLOBAL	V, W, X, Y are again global.

Arrays and Indexing

As mentioned earlier, a variable may represent a scalar or an array. A scalar is always a single item, an item being a character, number, or nested array. One example of a scalar is:

```
      SCLR←33
      SCLR
33
```

Although an array may be made up of more than one item, it can also consist of a single item or even no items. An array with no items is called an empty array.

In addition, arrays can be classified as vectors, matrices, or higher-order arrays. A vector is an array of one dimension, and is displayed as a collection of items arranged on one line. As a typical example, notice the vector named *VECT* which has four items:

```
      VECT←5 7 9 11
      VECT
5 7 9 11
```

A matrix is an array with two dimensions, (a dimension is sometimes called a coordinate) and is displayed as a collection of items arranged in a rectangular pattern. An example of a two-dimensional matrix, named *MAT*, is shown below:

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

Notice that this matrix has three rows and five columns. It is two-dimensional because it is made up of rows and columns.

A higher-order array is an array with three or more dimensions, displayed as a collection of items in a set of rectangular patterns. An example of a higher-order array is:

```
      CUBE
 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
```

This higher-order array is three-dimensional. It has two planes, and each plane has three rows and five columns.

The user can find out if a variable is a scalar, a vector, a matrix, or a higher-order array by using *ρρ* to test for the rank (that is, number of dimensions) of the variable. For example, testing the previous variables *SCLR*, *VECT*, *MAT*, and *CUBE* will give

```
      ρρSCLR
0
      ρρVECT
1
      ρρMAT
2
      ρρCUBE
3
```

A 0 indicates a scalar, a 1 indicates a vector, a 2 indicates a two-dimensional array, a 3 indicates a three-dimensional array, and so on, up to a maximum of 62 dimensions.

The user can also determine the size of each dimension in an array (that is, the "shape" of the array) by using ρ. For example, testing the same variables *SCLR*, *VECT*, *MAT*, and *CUBE* will give

```
      ρSCLR
      ρVECT
4
      ρMAT
3 5
      ρCUBE
2 3 5
```

Since a scalar has no dimensions, ρ of a scalar produces an empty (vector) result; nothing is displayed (other than the next input prompt). The above example confirms that *SCLR* is a scalar (no dimension); that *VECT* is a vector with four items; that *MAT* is a two-dimensional matrix with three rows and five columns (15 items); and that *CUBE* is a three-dimensional array with two planes, each with three rows and five columns. One other situation should be noted, ρ of an empty vector will return the value zero, and ρ of an empty array will return one or more zeros depending on which dimension or dimensions have length zero.

Indexing of Arrays

Items in an array can be referenced by their positions within the array. The position number is called an index. The index can also be used for several items, and to index other indexed arguments. The following topics are discussed in this subsection:

- Referencing a Single Item
- Referencing More Than One Item
- Assigning a Value to an Array
- Indexing an Indexed Argument

Referencing a Single Item

An item in an array is referenced by its position within the array, which is indicated by one or more numbers called indexes. One number is used as the index of an item in a vector array; two numbers, as the index of an item in a two-dimensional matrix; three numbers, as the index of an item in a three-dimensional array; and so on, with one number for each dimension.

The indexes of all arrays start with 0 or 1, depending on the index origin. When the user first enters APL, the index origin is 1 by default. It can be set to 0 by assigning the ⎕*IO* system variable to 0, and reset to 1 by reassigning the ⎕*IO* variable to 1.

```
      V←'ABCDE'
      ⎕IO←1
      V[2]
B
      ⎕IO←0
      V[2]
C
      V[1]
B
      ⎕IO←1
```

The indexes of a two-dimensional matrix also start with 0 or 1, depending on the index origin, but two numbers are used in each index. The first number selects the items from a row, and the second number selects the items from a column. The indexes are ordered with the rightmost position varying the fastest, then the next rightmost, and so on. For purposes of illustration, consider the matrix named *MAT*3:

```
      MAT3
 3  1 11 2 12
13 15  4 8 14
 6 10  7 9  5
```

The indexes for this matrix, with index origin 1, will be

```
[1;1]  [1;2]  [1;3]  [1;4]  [1;5]
[2;1]  [2;2]  [2;3]  [2;4]  [2;5]
[3;1]  [3;2]  [3;3]  [3;4]  [3;5]
```

Thus *MAT*3[1;1] is 3; *MAT*3[1;2] is 1; *MAT*3[1;3] is 11; *MAT*3[1;4] is 2; and so on. Notice that semicolons must be used to separate the numbers of each dimension.

An item in an array of more than two dimensions is selected in the same way as an item of a two-dimensional array, except that more numbers are included in the index. An index contains one number for each coördinate of the associated array. For example, consider the following three-dimensional array:

```
      MAT4
 1  4 14  7
15 13  2  8

11 12  6 16
 5  3  9 10
```

To reference the value 8 in this array, one uses the index *MAT*4[1;2;4], where 1 denotes the first plane, 2 denotes the second row, and 4 denotes the fourth column. Notice that each additional coordinate always adds a number to the beginning of an index. The rightmost number of an index always refers to a column; the next rightmost to a row; the next rightmost to a plane; the next to a panel of planes; and so on.

Referencing More Than One Item

To reference items within an array, simply include the index of each desired item in brackets after the array name. For example, notice the following vector:

```
      A←5 4 ¯1 3 9 ¯2 7 4
```

To select the items 5, ¯1, and 3 from this vector (assuming an index origin of 1), one uses the expression *A*[1 3 4] as shown here:

```
      A[1 3 4]
5 ¯1 3
```

Other examples of referencing several items in vector A are shown below. Notice in the second example that indexing can be used to create larger and differently shaped arrays:

```
      A[1 1 8 8 8]
5 5 4 4 4

      A[3 2⍴1 3 4 2 6 5]
 5 ¯1
 3  4
¯2  9
```

There are a variety of ways to reference several items in a matrix. Consider the following matrix:

```
      MAT5
 1 10  9  8 11
 2 15  4  5  6
15  3 12 13  7
```

Examples of referencing several items in this matrix are shown below. These examples assume an index origin of 1.

```
      MAT5[1;4 5 2]
8 11 10
      MAT5[1 2;]
1 10 9 8 11
2 15 4 5  6
      MAT5[1 2;1 2 3 4 5]
1 10 9 8 11
```

```
2 15 4 5  6
      MAT5[1 2 3;4]
8 5 13
      MAT5[1 2;4 5]
8 11
5  6
      MAT5[;2 4]
10  8
15  5
 3 13
      MAT5[1 2 3;2 4]
10  8
15  5
 3 13
```

In fact, the shape of the indexing result has a rank equal to the shape of each of the index expressions joined together. If an index expression is elided, the result shape has the length of the elided coordinate inserted.

Several items in a three-dimensional array are referenced similarly to a matrix, except that the third coordinate must also be added to the index. Consider the following three-dimensional array:

```
      MAT6
 1  2  3  4  5
 6  7  8  9 10

11 12 13 14 15
16 17 18 19 20
```

Examples of referencing several items in this array are shown below. These examples assume an index origin of 1.

```
      MAT6[1;2;5]
10
      MAT6[;2;]
 6  7  8  9 10
16 17 18 19 20
      MAT6[;2;1 3]
 6  8
16 18
      MAT6[1 1 2;1 2 1;1 2 4]
 1  2  4
 6  7  9
 1  2  4


 1  2  4
 6  7  9
 1  2  4


11 12 14
16 17 19
11 12 14
```

Assigning a Value into an Array

One or more items in an already existing array can be assigned values via the assignment symbol ←. The user simply places the variable name and the index designation to the left of the symbol, and the new value to the right. Examples follow, all of which assume an index origin of 1.

Example of vector:

```
      V←3+⍳10
      V
4 5 6 7 8 9 10 11 12 13
      V[1 3 5]←1 0 1
      V
1 5 0 7 1 9 10 11 12 13
      V[1 3 5 7 9]←0
      V
0 5 0 7 0 9 0 11 0 13
      WHOOPS←V[]←2
      V
2 2 2 2 2 2 2 2 2 2
      WHOOPS
2
```

Example of matrix:

```
      MAT7←2 5⍴⍳10
      MAT7
1 2 3 4  5
6 7 8 9 10
      MAT7[2;5]←0
      MAT7
1 2 3 4 5
6 7 8 9 0
      MAT7[1 2;3 5]←¯1
      MAT7
1 2 ¯1 4 ¯1
6 7 ¯1 9 ¯1
      MAT7[;]←2
      MAT7
2 2 2 2 2
2 2 2 2 2
```

Notice from examples above (*MAT6*[;2;], *V*[]←2, and *MAT7*[;]←2) that if an index position is not filled, all index values for that position are assumed to be applicable. Assigning a new value to an indexed variable does not change the rank or shape of the variable, it merely changes some items in the variable.

The value that is assigned to a variable or indexed variable is also the "result" of the assignment. This is illustrated by the example *WHOOPS*←*V*[]←2. Since *V* is a 10-item vector, all 10 index values received the value 2. But the result as far as the assignment operation is concerned is the scalar 2. Thus, *WHOOPS* becomes a scalar variable having the value 2. When analyzing APL expressions, it is helpful to imagine that assignments are "invisible". For example,

```
      3+M[;4]←5
```

can be analyzed as if the assignment were not present, i.e.,

```
      3+      5
```

making the result (8) apparent.

Indexing an Indexed Argument

In APL, an indexed argument may itself be indexed. For example:

```
      A[1;][2]
```

which is equivalent to the expression (*A*[1;])[2] and is interpreted as follows. Obtain the first row of matrix *A*. This row temporarily forms a vector, call it *T*, whose length is the number of columns originally given for *A*. Select the second item from vector *T*, and (in this case) display the value of that item.

Only arguments can be followed by multiple indexes. Specifications and coordinates cannot; thus the following is a syntax error:

```
      A[1;][2]←X
LINESCAN ERR  ^
```

The user instead is advised in this case to use

```
        A[1;2]←X
```

Functions and Arguments

APL expressions are derived from three fundamental entities: operators, functions, and values. Functions may be formed by the user (see section 7 under Defined Functions) or are included as an inherent part of the language. In the latter case, they are called primitive functions. Most primitive functions are represented by a single character. A general treatment of these functions is given in this section; for a detailed treatment, see Section 5, APL Functions.

Operators usually take APL functions as arguments and return a new (or derived) function. The derived function typically applies the function arguments to the value arguments in an operator defined order. Examples of APL operators include axis, inner product, and outer product.

Values are APL arrays and have certain attributes: type, rank, depth, and length or shape. The domain of an array may be character type, numeric type, or nested type. There are three numeric domains: logical, integer, or real; however, the user seldom needs to be concerned with this distinction. Logical data represents 1's or 0's and is stored in bit form. Integer data represents positive and negative numbers (using neither decimal point nor exponential form) whose range is limited to the size of one computer word. Real data is stored in doubleword form (that is, in floating-point form). Text or character data is stored in byte form. The nested domain type can have an array item which contains other APL arrays or both character and numeric data items. If a numeric argument contains numbers that could fit in more than one domain, it is made to uniformly contain numbers in the largest size domain necessary. Thus the following vector argument has integer domain since that is necessary to represent the 2:

```
        1 0 1 0 2
```

The rank of an array is the number of its dimensions (or coordinates). A scalar has a rank of zero, a vector has a rank of one, a matrix has a rank of two, and so forth. The maximum allowed rank in CP-6 APL is 62.

The length of a vector is its number of items or components (zero for an empty vector). The shape or dimension of an array (including a vector) is an ordered vector containing the lengths of its coordinates. Single-item vectors and single-item arrays of higher order (for instance, a 1 1 1 reshape of 5 is a single-item three-dimensional array) are not equivalent to scalars but may be used interchangeably with scalars in many operations. Vectors and arrays of higher ranks may also be 'empty'. This is the case when the length of a coordinate is zero.

The depth of an array indicates the maximum level of nesting of items within the array. A simple scalar character or number has depth 0. An array containing only simple scalar character or scalar numeric items has depth 1. An array containing items of depth 0 and 1 has depth 2. In general, an array containing items of depth less than or equal to N has a depth of N+1. Simple arrays have depth 0 or 1. Nested arrays have depth 2 or more.

Functions are classified as monadic or dyadic according to the number of their arguments. A monadic function has one argument to the right of the function. A dyadic function has two arguments, one to the right of the function and one to the left.

In many cases, the same function can be used both monadically and dyadically, but the resulting functions are different, although usually related in a natural way. Each function has its own domain, rank, and length or shape requirements, and the result of a function may have a new set of these characteristics.

Axis Operator

Certain functions are coordinate-dependent. For example, a matrix rotation can occur about the first coordinate (rotation of rows) or about the second coordinate (rotation of columns). For such functions, the user has the option of specifying this coordinate in the form of a bracketed expression to the right of the function. The value of this expression must be an integer of appropriate range. These coordinate specifications are called the Axis operator. The Axis operator takes the coordinate specified and the function to its left and creates a new "derived" function which operates on the requested coordinate. The following functions may use a coordinate specification:

Reduction
Reversal
Rotation
Scan
Compression
Expansion
Catenation
Enclose
Disclose

NOTE: Catenation may also use a fractional coordinate specification. This form of catenation is called lamination. Enclose and disclose permit the specification of a vector of axes.

APL Functions and Operators

Tables 3-2, 3-3, and 3-4 include summary information about Scalar Functions, Mixed Functions and Operators, respectively. Each table lists dyadic and monadic operations, if any, and gives simple examples. For a detailed description of these functions and operators, see Section 5.

Scalar Function Summary

Scalar functions are pervasive. That is, when they are applied to nested arrays, the function is applied to every numeric and every character scalar in the array.

Table 3-2. Scalar Functions

Function	Usage
+	Monadic – Conjugate: Leaves argument unchanged. Example: +10 10 Dyadic – Addition: Adds two arguments. Example: 10+20 30

Table 3-2. Scalar Functions (cont.)

Function	Usage

–	Monadic – Minus:
	Negates the argument that follows it. Example:
	`      -(10+5)`
	`¯15`
	Dyadic – Subtraction:
	Subtracts the right argument from the left argument. Example:
	`     10-5`
	`5`

x	Monadic – Signum:
	Returns ¯1, 0, or 1, depending on whether its argument is negative, zero or positive. Example:
	`      x¯15`
	`¯1`
	Dyadic – Multiplication:
	Multiplies the left argument by the right argument. Example:
	`      10×15 150`
	`150 1500`

÷	Monadic – Reciprocal:
	Divides 1 by the value of its argument. Example:
	`      ÷1 3 5`
	`1 0.3333333333 0.2`
	Note that this is equivalent to the dyadic use:
	1÷1 3 5.
	Dyadic – Division:
	Divides the left argument by the right argument. Example:
	`     10÷5 2 1 .5`
	`2 5 10 20`

Table 3–2. Scalar Functions (cont.)

Function	Usage
*	Monadic – Exponential: Raises e (i.e., the base of the natural logarithms, having the value of approximately 2.71828...) to the power of its argument. Examples: *1 2.718281828 *10 22026.46579 *2.2 9.025013499 Dyadic – Exponentiation: Raises the left argument to the power indicated by the right argument. Examples: 10 10 2*2 10 3 100 1E10 8
⍟	Monadic – Natural logarithm: Computes the natural logarithm of its argument (that is, log base e of the argument). Examples: ⍟1 0 ⍟2 0.6931471806 ⍟3 10 1.098612289 2.302585093 Dyadic – Logarithm: Computes the logarithm of the right argument to the base indicated by the left argument; that is, computes the power to which the left argument must be raised to equal the right argument. Examples: 10⍟100 2 10⍟1 10 100 1000 0 1 2 3 2⍟4 2 2⍟1 2 4 8 0 1 2 3
⌊	Monadic – Floor: Returns the greatest integer less or equal to its arguments. Examples: ⌊ 10.7 10 ⌊2 4.1 ¯8.9 ¯2 2 4 ¯9 ¯2

Table 3–2. Scalar Functions (cont.)

Function	Usage
	Dyadic – Minimum: Compares two arguments and returns the value of the smaller argument. Examples: 5⌊2 2 9⌊3 11 8 3 9 8 4 3 2⌊3 3 3 2
⌈	Monadic – Ceiling: Returns the least integer greater than or equal to its argument. Examples: ⌈ 10.7 11 ⌈ 2 4.1 ¯8.9 ¯2 2 5 ¯8 ¯2 Dyadic – Maximum: Compares two arguments and returns the value of the larger argument. Examples: 5⌈2 5 9⌈3 11 8 ¯2 10 9 11 9 9 10
|	Monadic – Absolute value: Returns the absolute value of its argument. Example: |¯10 10 Dyadic – Residue: Returns the remainder from dividing the right argument by the left argument. Examples: 2|4 0 5|15 16 17 18 0 1 2 3 2 3|7 1 1

Table 3-2. Scalar Functions (cont.)

Function	Usage
!	Monadic – Generalized factorial: For integer arguments, returns the factorial of its argument. The argument may not be a negative integer. (See Section 5 for explanation of ! with non-integer argument.) Examples: !3 6 !0 1 2 1 1 2 Dyadic – Generalized combination: For positive integer arguments, the right argument represents a population size and the left argument represents a sample size. The result is the number of different samples that can be drawn from the population (see Section 5 for explanation of ! with non-integer arguments.) Examples: 13!52 6.350135596*E*11 2!10 45 3!10 120
○	Monadic – Pi times: Multiplies the value of pi (approximately 3.14159265353589793) times its argument. Examples: ○1 3.141592654 ○2 .1 6.283185307 0.3141592654 Dyadic – Circular: Returns the result of any of a number of trigonometric functions. The left argument specifies the trigonometric function and must be one of the integers from –7 to 7, as follows: 0 (1-*X**2)*0.5 1 sine *X* –1 arcsine *X* 2 cosine *X* –2 arccos *X* 3 tangent *X* –3 arctan *X* 4 (1+*X**2)*0.5 –4 *B*×(1-*B**¯2)*0.5 5 sinh *X* –5 arcsinh *X* 6 cosh *X* –6 arccosh *X* 7 tanh *X* –7 arctanh *X* Examples: 2○(10×2.5) 0.9912028119 1○2 4 0.9092974268 ¯0.7568024953

Table 3-2. Scalar Functions (cont.)

Function	Usage
<	Dyadic – Less than: Tests if the left argument is less than the right argument. Returns 1 if the test is true, and 0 if the test is false. (See Section 5 for effect of comparison tolerance on relational functions.) Examples: 2<3 1 3<4 1 2 5 1 0 0 1
≤	Dyadic – Less than or equal to: Tests if the left argument is less than or equal to the right argument. Returns 1 if the test is true, and 0 if the test is false. (See Section 5 for effect of comparison tolerance on relational functions.) Examples: 2≤3 1 2≤1 2 3 4 0 1 1 1
>	Dyadic – Greater than: Tests if the left argument is greater than the right argument. Returns 1 if the test is true, and 0 if the test is false. (See Section 5 for effect of comparison tolerance on relational functions.) Examples: 2>3 0 2>¯2 0 2 3 1 1 0 0
≥	Dyadic – Greater than or equal to: Tests if the left argument is greater than or equal to the right argument. Returns 1 if the test is true, and 0 if the test is false. Examples: 2≥3 0 2≥¯2 0 2 3 1 1 1 0

Table 3–2. Scalar Functions (cont.)

Function	Usage
=	Dyadic – Equal to: Tests if the left argument is equal to the right argument. Returns 1 if the test is true, and 0 if the test is false. (See Section 5 for effect of comparison tolerance on relational functions.) Examples: 1=0 0 2=0 1 2 3 0 0 1 0 'A'='CANADA' 0 1 0 1 0 1
≠	Dyadic – Not equal: Tests if the left and right arguments are unequal. Returns 1 if the test is true, and 0 if the test is false. (See Section 5 for effect of comparison tolerance on relational functions.) Examples: 2≠1 1 3≠¯3 0 3 6 1 1 0 1 'A'≠'CANADA' 1 0 1 0 1 0
∧	Dyadic – And: (The arguments must be 0 or 1.) Returns 1 if both arguments are 1, and 0 for any other combination of arguments. Examples: 0∧0 0 (1=2)∧(3<4) 0 (1<2)∧3<1 0 (1=1)∧3<4 1 Least Common Multiple: Returns the least common multiple of the left and right arguments. The LCM of a set of numbers is defined as their product divided by the GCD of the numbers. Examples: 3∧2 6 4∧6 12 0.5∧0.3 1.5

Table 3–2. Scalar Functions (cont.)

Function	Usage
∨	Dyadic – Or: Returns 1 if either or both arguments are 1, and 0 if neither argument is 1. Examples: 0∨1 1 (1=2)∨(4<3) 0 (3<4)∨4<5 1 Greatest Common Divisor: Returns the greatest common divisor of the left and right arguments. The GCD of a pair of numbers is defined as the largest divisor of both which produces an integer or near–integer result. Examples: .5∨÷3 0.1666666667 2∨8 2 2∨3 1
⍲	Dyadic – Nand: Returns 0 if both arguments are 1, and 1 for all other combinations. Examples: 0⍲0 1 (2<1)⍲(5<1) 1 (1<2)⍲(1<5) 0
⍱	Dyadic – Nor: Returns 1 if both arguments are 0, and returns 0 for all other combinations. Examples: 0⍱0 1 0⍱1 0 (1=2)⍱(2<1) 1 (1=2)÷⍱2<3 0

Table 3–2. Scalar Functions (cont.)

Function	Usage
~	Dyadic – Not: Returns 0 if the argument is 1, and returns a 1 if the argument is 0. Examples: ~0 1 1 0 ~(6>4) 0 ~1 0 1 0 0 1 0 1

Mixed Function Summary

The mixed functions produce results with a structure that is different from that of its arguments. Mixed functions can be sub–divided into the structural mixed functions and the transformation mixed functions:

- o The structural mixed function subset re–orders the array right argument under the optional control of a left argument. The re–ordering is generally dependent on the right argument's rank and shape but independent of the actual elements within it.
- o The transformation mixed functions produce results which typically depend upon the value of the array arguments.

The following table is a summary of APL mixed functions.

Table 3–3. Mixed Functions

Function	Usage
⍳	Monadic – Index generator: Generates a vector whose length is the value of the argument. If the index origin (⎕*IO*) is 1, the vector will contain positive integers 1 through value of the argument. If the index origin is 0, the vector will contain the positive integers 0 through the value of the argument minus 1. Examples: ⍳5 1 2 3 4 5 ⎕*IO*←0 ⍳5 0 1 2 3 4 ⎕*IO*←1 Dyadic – Index of: Returns the position of the right argument in the left argument. If the right argument is not found in the left argument, it is given a value of the last index position of the left argument plus 1. Examples:

Table 3–3. Mixed Functions (cont.)

Function	Usage

	`      6 4 3ι6`
	`1`
	`      6 4 3ι3 5 4`
	`3 4 2`

,	Monadic – Ravel:
	Generates a vector from either a scalar or an array of higher dimension. Examples:
	`      ⎕←A←2 4ρι8`
	`1 2 3 4`
	`5 6 7 8`
	`      ,A`
	`1 2 3 4 5 6 7 8`
	Dyadic – Catenation:
	Joins together scalars or arrays of conforming dimension. Examples:
	`      A←1 2 3`
	`      B←4 5 6 7`
	`      A,B`
	`1 2 3 4 5 6 7`
	`      C←3`
	`      (C÷2),C×3-2`
	`1.5 3`

ρ	Monadic – Shape:
	Returns an empty vector if the argument is a scalar, the length (or number of items) if the argument is a vector, or a vector containing the length of each dimension if the argument is a higher-order array. Examples:
	`      ρA←2`
	`      ρB←1 5 6 7`
	`4`
	`      ρC←3 3ρι9`
	`3 3`
	`      ρρA`
	`0`
	Dyadic – Reshape (restructure):
	Generates an array whose dimensions are the left arguments and whose items are taken from the right argument. Examples:
	`      5ρ1`
	`1 1 1 1 1`
	`      2 4ρ8`
	`8 8 8 8`
	`8 8 8 8`
	`      2 4ρι8`
	`1 2 3 4`
	`5 6 7 8`

Table 3-3. Mixed Functions (cont.)

Function	Usage
⍋	Monadic – Grade-up: Ranks the components of its argument in ascending order, and returns the positions (i.e., indexes of the components). Example: A←1 4 1 2 3 1 ⍋A 1 3 6 4 5 2 Dyadic – Grade-up: Ranks the components of its right argument in ascending order defined by the collating sequence given by the left argument. Similar to the monadic grade-up function except that both arguments must be character and the ordering is defined by the left argument. Examples: A←3 4⍴'ABRACODEBACK' 'ABCDEFGHIJK'⍋A 1 3 2
⍒	Monadic – Grade-down: Similar to Grade-up, except that it returns the indexes in descending order. Examples: A←1 4 1 2 3 1 ⍒A 2 5 4 1 3 6 Dyadic – Grade-down: Ranks the components of its right argument in the descending order defined by the collating sequence given by the left argument. This is similar to the monadic grade-down function except that both arguments must be character and the ordering is defined by the left argument. Examples: A←3 4⍴'ABRACODEBACK' 'ABCDEFGHIJK'⍒A 2 3 1
?	Monadic – Roll: Returns an integer pseudorandomly selected from ⍳ B. Examples: ?5 3 ?3 3 3 3 2 1 ?5 8 11 13 2 5 8 2 Note that this function is modified by ⎕IO (index origin).

Table 3-3. Mixed Functions (cont.)

Function	Usage

	Dyadic – Deal:
	Returns the number of integers specified in the left argument, each pseudorandomly selected from the integers specified in the right argument, and with no repetition of numbers in the result. Examples:
	`      4?8`
	`8 3 4 2`
	`      4?4`
	`1 3 2 4`
	Note that this function is modified by ⎕*IO* (index origin).

⊥	Dyadic – Base value:
	Switches from one number system to another. The right argument contains the numbers to be converted and the left argument contains the increments needed to convert from one unit to another. The left argument, usually called the radix vector, can be thought of as the base of the number system. Examples:
	`      10 10 10⊥5 6 5`
	`565`
	`      0 60⊥10 20`
	`620`
	`      2 2 2 2⊥1 0 0 1`
	`9`
	`      2⊥1 0 0 1`
	`9`

⊤	Dyadic – Encode:
	Converts a number to some predetermined representation. It works in reverse of the base value operation above. The following shows how to reconvert to the initial arguments used above in the base value. Examples:
	`      10 10 10⊤565`
	`5 6 5`
	`      0 60⊤620`
	`10 20`
	`      2 2 2 2⊤9`
	`1 0 0 1`

⍕	Monadic – Format:
	Converts numeric arrays to character arrays. The result is the same as if the argument were printed. Examples:
	`      ⍕ 3 3.1`
	`3 3.1`

Table 3-3. Mixed Functions (cont.)

Function	Usage
	Dyadic – Format: Converts numeric arrays to character arrays while controlling the format with the left argument. The left argument specifies the width and precision to be used in the display of the right argument. Examples: 2 0⍕3 4.1 5 3 4 5 5 2⍕ 3 0.61 5.5 3.00 0.61 5.50
↑	Monadic – First Returns an array whose value is the first item of the right argument. If the right argument is empty, then the result is the prototype of the right argument. For a scalar right argument, this function is the inverse of the enclose function. Examples: ↑⍳10 1 ↑'ONE' 'TWO' 'THREE' ONE Dyadic – Take: Selects the number of components indicated by the left argument from the right argument. If the left argument is positive, the take function selects the components from the beginning of the right argument. If the left argument is negative, the take function selects the components from the end of the right argument. Examples: A←2 4 6 8 3↑A 2 4 6 ¯3↑A 4 6 8
↓	Dyadic – Drop: Similar to take except that the indicated items are dropped instead of selected. Examples: A←2 4 6 8 2↓A 6 8 ¯1↓A 2 4 6

Table 3-3. Mixed Functions (cont.)

Function	Usage
∊	Monadic – Type: Returns an array containing 0 where argument items are numeric or blank where argument items are text. Example: ∊1 'BRUCE' 2 (3 4) 0 0 0 0 ' '=∊1 'HI' 2 0 1 1 0 Dyadic – Membership: Returns 1 if a given item of the left argument is an item of the right argument, and 0 if it is not. The result has the same dimensions as the left argument. Examples: A←⍳6 B←2×⍳4 B∊A 1 1 1 0 C←'ABCDEFGHIJK' D←3 3⍴'HOWAREYOU' D∊C 1 0 0 1 0 1 0 0 0
⍎	Monadic – Execute: Treats its argument (a character scalar or vector) as an APL statement. Examples: ⍎'2+3' 5
⊂	Monadic – Enclose: Increases the depth of the argument by 1 and decreases the rank. If an axis is not specified, all axes are enclosed and the result is a scalar. When an axis is specified, the rank of the result is the rank of the argument minus the number of axes being enclosed. The enclose of a simple scalar yields the scalar unchanged. Examples: ⊂'B' B ⍴⎕←⊂'VENICE' VENICE ⊂[1]3 3⍴⍳9 1 2 3 4 5 6 7 8 9

Table 3-3. Mixed Functions (cont.)

Function	Usage
⊃	Monadic – Disclose: Decreases the depth of the argument by 1 and increases the rank. If the axes are not specified, the new axes are inserted after the last axis of the argument. The disclose of a simple array yields the array unchanged. Examples: ρ⎕←⊃⊂'*HI*' *HI* 2 ρ⎕←⊃(1 2) (3 4 5) 1 2 0 3 4 5 2 3 Dyadic – Pick: Select an item from the right argument specified by the path indices in the left argument. Each item of the left argument must be a simple scalar or vector of integer indices which selects an item to be indexed by the next item of the left argument. Example: 2 ⊃7 8 9 8 2 1 (2 1)⊃ 1 ((2 2ρ3 4 5 6) 7) 8 5 ''⊃9 9
≡	Monadic – Depth: Returns a simple non-negative integer scalar indicating the maximum depth of nesting in the right argument. A simple scalar number or character has depth 0. Arrays containing simple scalar numbers or characters have depth 1. Examples: ≡'*A*' 0 ≡1 2 3 1 ≡'*ABC*' (4 (5 6)) 7 3 Dyadic – Equivalence: Returns a simple logical scalar. The result is 1 if the left argument is identical to the right argument, otherwise the result is 0. Arrays are identical if they have the same shape and the same values in all corresponding positions. Empty arrays are identical only if their prototypes are identical. Examples:

Table 3-3. Mixed Functions (cont.)

Function	Usage
	'APPLE'≡'APPLE' 1 'CP6'≡'CPV' 0 9≡,9 0 ''≡⍳0 0
⌹	Monadic – Matrix Inverse: Used to invert matrices. Examples: A←3 3⍴4 2 ¯5 5 ¯4 4 2 2 ¯20 ⎕PP←2 ⌹A 0.17 0.072 ¯0.029 0.26 ¯0.17 ¯0.099 0.043 ¯0.0097 ¯0.063 Dyadic – Matrix Divide: Used for solving systems of linear equations. Examples: A 4 2 ¯5 5 ¯4 4 2 2 ¯20 B←22 ¯7 80 B⌹A 1 ¯1 ¯4
⍉	Monadic – Transpose: Performs row column transposition on its matrix argument. Examples: A 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ⍉A 1 6 11 2 7 12 3 8 13 4 9 14 5 10 15 Dyadic – Transpose: Returns an array similar to the right argument except that the coordinates (dimensions) are changed according to the left argument (that is, the left argument specifies the new position of the original coordinates). Examples: B←2 4 3⍴⍳24 2 1 3⍉B 1 2 3 13 14 15

Table 3-3. Mixed Functions (cont.)	
Function	**Usage**

	4 5 6
	16 17 18
	
	7 8 9
	19 20 21
	
	10 11 12
	22 23 24

⌽	Monadic – Reversal: Reverses the order of the components of a vector, or the components of each each row of a matrix. Examples: *A*←1 2 4 6 ⌽*A* 6 4 2 1 ⌽⍳5 5 4 3 2 1 Dyadic – Rotation: Rotates the items in the right argument as specified by the left argument (i.e. according to the number of places specified in the left argument). Examples: *A*←1 2 4 6 1⌽*A* 2 4 6 1 2⌽*A* 4 6 1 2
⊖	Monadic – Reversal along the first coordinate: Same as ⌽ above except along the first coordinate instead of the last. This is equivalent to ⌽[⎕*IO*]. Example: ⎕←*MAT*←3 4⍴⍳12 1 2 3 4 5 6 7 8 9 10 11 12 ⊖*MAT* 9 10 11 12 5 6 7 8 1 2 3 4 Dyadic – Rotation along the first coordinate: Same as ⌽ above, except along the first coordinate instead of the last. This is equivalent to ⌽[⎕*IO*]. Examples: ⎕←*MAT*←3 4⍴⍳12 1 2 3 4 5 6 7 8 9 10 11 12 1⊖*MAT* 5 6 7 8 9 10 11 12 1 2 3 4

Operator Summary

APL operators usually take functions (primitive, system, or user-defined) and produce a derived function which is then applied to array arguments. The manner in which the function argument is applied to the array arguments distinguishes the various operators.

The letters f and g in the following table represent any functions.

Table 3-4. Operators

<table>
<tr><th>Function</th><th>Usage</th></tr>
<tr><td>f/</td><td>
Monadic – Reduction:

Inserts the APL function specified to the left of the / between each item of the right argument, performs the operation from right to left, and returns a value with one less coordinate than the right argument. Examples:

<pre>
      +/1 2 3 4 5
15
      -/1 2 3 4 5
3
      □←N←3 4⍴⍳12
1 2 3 4
5 6 7 8
9 10 11 12
      +/N
10 26 42
      -/N
¯2 ¯2 ¯2
</pre>

Dyadic – Compression and Replicate:

Suppresses some items of a vector and retains others. Items of the right argument corresponding to a 1 in the left argument are retained while those corresponding to a 0 are dropped. If either argument contains just one item, it applies to all items of the other arguments. Examples:

<pre>
      A←5 7 9 11
      B←'ABCD'
      1 0 1 1/A
5 9 11
      1 0 1 1/B
ACD

      □←MAT←3 4⍴⍳12
1 2 3 4
5 6 7 8
9 10 11 12
      1 0 1 0/MAT
1 3
5 7
9 11
</pre>

Replicate is like compression but this function will replicate items as well as suppress. In this case, the left argument is an integer vector, whose items are greater than or equal to zero. Each item of the left argument indicates the number of times the corresponding item in the right argument is to be replicated. Examples:
</td></tr>
</table>

Table 3–4. Operators (cont.)

Function	Usage
	2/'*APPLE*' *AAPPPPLLEE* 1 2 3/'*WOW*' *WOOWWW* 2 1 0/'*ITS*' *IIT*
f⌿	Monadic – Reduction along the first coordinate: Same as f/ above except reduction occurs along the first coordinate rather than the last (equivalent to f/[⎕*IO*]. Examples: ⎕←*N*←3 4⍴⍳12 1 2 3 4 5 6 7 8 9 10 11 12 +⌿*N* 15 18 21 24 -⌿*N* 5 6 7 8 Dyadic – Compression along the first coordinate: Same as above except that compression or replication is along the the first coordinate instead of the last. Equivalent to /[⎕*IO*]. Examples: ⎕←*MAT*←3 4⍴⍳12 1 2 3 4 5 6 7 8 9 10 11 12 0 1 0⌿*MAT* 5 6 7 8
f.g	Dyadic – Generalized inner product: This operator is a generalized form of the inner product of matrix multiplication. The particular form that corresponds to traditional matrix multiplication is *A*+.×*B*, where the second dimension of matrix *A* is the same as the first dimension of *B*. The result has the same first dimension as *A* and the same second dimension as *B*. In the conventional matrix inner product, each item of the result is the sum of products of items from *A* and *B* (see Section 5 for detailed description). The APL generalized inner product allows different forms such as the sum of equality tests, the maximum of sums, etc. Examples: *A*←2 3⍴⍳6 *B*←3 2⍴-⍳6 *A* 1 2 3 4 5 6 *B* ¯1 ¯2 ¯3 ¯4 ¯5 ¯6 *A*+.×*B*

Table 3-4. Operators (cont.)

Function	Usage
	¯22 ¯28 ¯49 ¯64 *A*+.=*B* 0 0 0 0 *A*⌈.+*B* 0 ¯1 3 2 The general form is *A*f.g*B* where f and g represent any function. *A* and *B* may be vectors, matrices, or higher order arrays, subject to conformability rules described in Section 5.
∘.f	Dyadic – Generalized outer product: This operator is a generalization of matrix outer product, *A*∘.×*B*. The conventional form multiplies each item of *A* by each item of *B*. The shape of the result is the catenation of the shapes of *A* and *B*. In the generalized form, multiplication may be replaced by any APL function. Examples: *A*←¯1+⍳5 *A*∘.+*A* 0 1 2 3 4 1 2 3 4 5 2 3 4 5 6 3 4 5 6 7 4 5 6 7 8 *A*∘.×*A* 0 0 0 0 0 0 1 2 3 4 0 2 4 6 8 0 3 6 9 12 0 4 8 12 16 *A*∘.<*A* 0 1 1 1 1 0 0 1 1 1 0 0 0 1 1 0 0 0 0 1 0 0 0 0 0
f\	Monadic – Scan: Returns value of same shape as argument. For vectors, the i'th result item is formed by taking the first i argument items, placing f between them, and evaluating right to left. For example: +\1 3 5 7 9 1 4 9 16 25 -\3 1 1 5 3 2 3 ¯2 A coordinate specification [*K*] may be used; if omitted, the last coordinate is assumed. +\[1]2 3⍴⍳6 1 2 3 5 7 9

Table 3–4. Operators (cont.)

<table>
<tr><th>Function</th><th>Usage</th></tr>
<tr><td></td><td>
Dyadic – Expansion:
Inserts additional items into an array. For each 0 in the left argument, a prototype item (blank for character, zero for numeric) is inserted in the result, which otherwise is the same as the right argument. Examples:
<pre>
      A←1 2 3 4
      B←'ABCD'
      1 0 1 0 1 0 1\A
1 0 2 0 3 0 4
      1 0 1 0 1 0 1\B
A B C D
      ⎕←M←3 4⍴⎕AV[65+⍳12]
ABCD
EFGH
IJKL
      1 0 1 0 1 0 1\M
A B C D
E F G H
I J K L
</pre>
</td></tr>
<tr><td>f⍀</td><td>
Monadic – Scan along the first coordinates:
Same as f\[⎕IO]. Thus, as above,
<pre>
      +⍀2 3⍴⍳6
1 2 3
5 7 9
</pre>
Dyadic – Expansion along the first coordinate:
Same as \ above, except expansion occurs along the first coordinate rather than the last. This is equivalent to \[⎕IO]. Example:
<pre>
      ⎕←M←3 4⍴⎕AV[65+⍳12]
ABCD
EFGH
IJKL

      1 0 1 0 1⍀M
ABCD

EFGH

IJKL
</pre>
</td></tr>
<tr><td>f¨</td><td>
Monadic – Each:
Returns a value of the same shape as the argument. Each item of the result is formed by applying the monadic function to the corresponding item of the right argument. Examples:
<pre>
      ⍴¨'ABC' 'HAPPY'
3 5
      ⍳¨2 3
1 2 1 2 3
      ⌽¨'XYZ' 'MOOD'
ZYX DOOM
</pre>
</td></tr>
</table>

Table 3-4. Operators (cont.)

Function	Usage
	Dyadic – Each: Returns a value of the same shape as the left and right arguments (a singleton argument is extended to the shape of the higher ranked argument). Each item of the result is formed by applying the dyadic function to the corresponding items of the left and right argument. Examples: 2⍴¨'ABC' 'HAPPY' AB HA 1⌽¨'BCA' (⍳4) CAB 2 3 4 1

Defined Function References

Defined functions are used in much the same way as primitive functions, but defined functions must first be formed by the user instead of being an inherent part of the language. Once a defined function has been formed, or "defined", it is referenced by its assigned name. (Naming conventions are described earlier in this section under Names.) A general discussion of functions is given in this section; for a detailed discussion, see Section 7, Defined Functions.

Like primitive functions, defined functions can have arguments which in turn have attributes of domain, rank, length, and shape (see Functions and Arguments above). Functions are classified as monadic, dyadic, or niladic, according to their number of arguments. A monadic function has one argument to the right of the function name. A dyadic function may have one or two arguments, one to the right of the function name and one optionally to the left. A niladic function has no arguments; the function name is referenced by itself.

The right argument is the value of the largest, complete APL expression immediately to the right of a function. For the example below, *F* is a function whose right argument is 2+⍳3.

```
(F  2+⍳3) 'POUNDS'
```

In this case, the character vector *'POUNDS'* is not included in the argument since the parenthesis splits the example into two distinct expressions.

The left argument is the value of the smallest complete APL expression to the left of a function. In the example below, *D* is a dyadic function whose left argument is (⍳3).

```
2+ (⍳3) D 4
```

In this case, the parenthetical expression (⍳3) is the smallest complete APL expression immediately to the left of *D*. 2+(⍳3) is also an APL expression, but it is larger. Therefore, the above example is interpreted as

```
2+result
```

where "result" is the result supplied by the function reference

```
(⍳3) D 4
```

In addition, any of the classes of defined functions may specify an implicit or explicit result. Thus there are actually six types of defined functions: monadic, dyadic, and niladic each of which may optionally produce a result.

The class is determined by the way a function is defined (that is, the function header), and it affects the way a function is referenced in an expression. Defined functions with explicit results may appear in compound expressions, much like primitive functions. Defined functions without results may appear alone; they cannot appear in compound expressions except as the last function to be executed.

A defined function may reference itself; that is, it may be recursive. A recursive function is one that references itself in the process of its execution.

When a function is invoked, it may complete execution and return a result or it may become suspended or pendent during execution. A suspended function is one in which execution has been stopped before completion (the reasons for stopping execution are given under Suspending Execution in Section 7). A pendent function is usually one that has referenced a suspended function and is unable to complete execution because of the suspended function. Suspended functions are always stopped "between" lines, but a pendent function is stopped in the process of executing a line. A function can be both suspended (stopped at some point) and pendent (in execution at some point). For instance, if a recursive function is stopped after it calls itself, it is suspended (at the stop) and pendent (where it called itself).

Assignment

The following paragraphs define simple assignment, multiple assignment, and indexed assignments.

Simple Assignment

The assignment symbol, denoted by a left-pointing arrow, is used to assign values to named variables or to a system variable. (Some programmers may refer to this symbol as the specification symbol or the replacement symbol, but the term assignment symbol is used throughout this manual.) It is the assignment that causes a variable to be a scalar, a vector, a matrix, or a higher-order array. The assignment of a value or an expression to a quad displays the value. Examples of assignments are shown below:

```
      A←5÷2×4
```

Assigns the value of the expression 5÷2×4 to variable *A*.

```
      B←1 2 3 4 5
```

Indicates that *B* is to be a vector with the values 1, 2, 3, 4, and 5.

```
      B←⍳5
```

Another way of assigning the numbers 1 through 5 to variable *B*. (Assuming an index origin of 1.)

```
      C←2 4⍴⍳8
```

Indicates that *C* is to be a matrix (with two rows and four columns) and that it is to be made up of the values 1 through 8 (assuming an index origin of 1), as shown here:

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

```
      D←2 3⍴5 6 1 2 8 9
```

Indicates that *D* is to be a matrix (with two rows and three columns) and that it is to be made up of the values 5, 6, 1, 2, 8, and 9, as shown here:

```
5  6  1
2  8  9
```

```
      E←D
```

Indicates that the value of *D* is assigned to *E*.

Multiple Assignments

APL allows repeated use of assignment, or multiple assignments, in a single statement. Examples of multiple assignment are shown as follows:

```
          A←5,B←6
          A,B
    5 6 6
          Z←2+Y←2+X←5
          X,Y,Z
    5 7 9
          ⎕←C←2 3 4 5
    2 3 4 5
```

Vector Assignment

This notation may be used to assign each item of a vector to a name in a list of names. In this case, the specification symbol (←) is preceded by the list of names enclosed in parentheses. The specification symbol must be followed by an APL expression which produces a vector having the same length as the number of names.

Examples:

```
      (A B C)←1 2 3
      A ⋄ B ⋄ C
1
2
3

      (NAME ADDRESS)←'JOE WHO'  '21 CENTURY BLVD, LOS ANGELES'
      NAME
JOE WHO
      ρ⎕←ADDRESS
21 CENTURY BLVD, LOS ANGELES
28
```

Indexed Assignment

One or more items of an already established array may be assigned new values. This is done by placing the variable name and the index designation(s) to the left of the assignment symbol, and the new value(s) to the right, as shown below (these examples all assume an index origin of 1):

```
          ⎕←A←1 5 4 3 2
    1 5 4 3 2
          A[1 2]←2 3
          A
    2 3 4 3 2
          A[]←0
          A
    0 0 0 0 0
          ⎕←B←2 3ρι6
    1 2 3
    4 5 6
          B[1;2]←4
          B
    1 4 3
    4 5 6
          B[;]←0
          B
    0 0 0
    0 0 0
```

Selective Assignment

This operation permits selected elements of a named array to be given new values while leaving the shape and the unselected elements unchanged. Bracket indexing or use of the selection functions are used to select the array elements to be changed. The selection functions that are used with selective specification are ravel (monadic ,), reshape (monadic ⍴), take (dyadic ↑), drop (↓), first (↑), transpose (⍉), reversal (monadic ⌽), rotate (dyadic ⌽), compression (/), pick (dyadic ⊃) and disclose (monadic ⊃).

The result of the selection expression must be a subset or re-arrangement (or both) of selected element locations. Only those selected locations receive the new value. The value being assigned must have the same shape as the selection expression after skipping all dimensions of length 1 in both.

Examples:

```
      N←5 5⍴ 1 2 4 3
      ((4=,N)/,N)←0             ⍝ REPLACE ALL 4S WITH 0
      N
1 2 0 3 1
2 0 3 1 2
0 3 1 2 0
3 1 2 0 3
1 2 0 3 1

      (1 1⍉N)←55                ⍝ REPLACE DIAGONAL
      N
55  2  0  3  1
 2 55  3  1  2
 0  3 55  2  0
 3  1  2 55  3
 1  2  0  3 55
```

Input/Output

This subsection describes how the user can enter input and display output.

Input/Output Devices

The CP-6 APL system gives the user a choice of five input/output methods:

- APL/ASCII terminal input/output: a terminal with either bit paired or typewriter paired APL/ASCII character transmission codes.
- ASCII terminal input/output.
- Batch input/output.
- File input/output.
- Blind input/output.

The input/output described in this section refers to terminals with the APL character set.

General Input/Output

After logging on to CP-6 and invoking APL, the user is in immediate execution mode and can enter input whenever the carriage or cursor is indented six spaces. The fundamental item of input to APL is the line. A line is a collection of characters that does not include the carriage return. Striking the RETURN key completes a line, and APL attempts to interpret it and perhaps output data. An incomplete line can be corrected as described in Section 2. User input and computer output are easily distinguished at the terminal; computer output usually begins at the left margin while user input is usually indented six spaces from the left margin. An input line is limited to 390 characters in length, not counting the carriage return (overstrikes count as single characters).

Types of Input

CP-6 APL acknowledges four kinds of input: direct, evaluated, quote quad, and blind. Direct input occurs when APL is not executing the user's program, evaluated input results from quad-input execution, quote-quad input results from quote-quad execution, and blind input results from quad-0 through quad-9 execution. Direct input, evaluated input, and quote-quad input are described below and are considered to exist only after input translation and current-line editing. Blind input is covered in Section 16.

Direct Input

Direct input is entered during execution mode. APL is ready to accept direct input when it skips to a new line and indents six spaces. Evaluation of direct input occurs immediately, and the response is either printed at the left margin (if the input was a non-assignment statement) or assigned to a variable (if the input was an assignment statement). Examples of direct input follow:

```
      5÷2×4
0.625
      A+A←5
10
      ⎕←B←3 4⍴⍳12
1  2  3  4
5  6  7  8
9 10 11 12
```

Evaluated Input

The quad symbol ⎕ can be used as an argument in a statement, to denote that input is desired. When APL encounters the quad during statement execution it halts execution and requests input by printing the symbols ⎕: at the left margin. A response of any valid APL expression causes execution to continue, using the value obtained in response to the quad symbol. Examples:

```
      8÷⎕
⎕:
      2
4
      5×⎕
⎕:
      1 2 3 4
5 10 15 20
```

If the quad symbol is built into an input loop, the user can terminate the input requested by entering the symbol → (not followed by an argument). Simply entering nothing and pressing the RETURN key is not sufficient to terminate the input request; it will merely cause the ⎕: to reappear at the left margin. An example of escaping from an input request is shown below:

```
      ∇CUBE;A
[1]   LOOP:A←⎕
[2]    A←A×A×A
[3]    A
[4]    →LOOP
    ∇

      CUBE
⎕:
      3
27
⎕:
      4
64
⎕:
      5
125
⎕:
      →
```

Entering any of the following system commands will terminate an input request: *)CLEAR*, *)LOAD*, *)OFF*, *)END*, *)SIC*, or *)CONTINUE*. Entering other system commands merely causes the ⎕: to reappear after the command is executed.

Functions can be defined during evaluated input. This is similar to function definition during normal (direct) input except that at the conclusion of the definition, APL re-requests evaluated input. This is to be expected since when APL originally requested evaluated input it needed a value, and defining a function provides no value. This enhancement is not limited to just providing definition capability. The full range of function definition mode features are available during evaluated input:

- Creating a new function
- Revising an existing non-pendent function (If a function makes an evaluated input request, the function becomes pendent. Therefore, that function cannot be opened during the evaluated input request): inserting a line, deleting a line, replacing a line, and editing characters of a line.
- Displaying one or more lines of the open function.

Entering an *)SI* or *)SINL* command in response to an input request will cause the state indicator to contain a ⎕. For example:

```
      10÷⎕
⎕:
      )SI
⎕
⎕:
      2
5
```

Quote-Quad Input

The quote-quad symbol ⍞ (except when to the left of an assignment arrow) denotes literal input. When APL encounters this symbol during statement execution, it awaits user input (nothing is printed to prompt for input). Literal character strings are entered without beginning and ending quote symbols, and a quote within a string is represented by one quote. Quote-quad input always produces a vector result. To terminate a request for literal input without having any value associated with the variable being requested, press the BREAK key twice.

Note that if the request for literal input is initiated from within an executing function and a double break is entered, execution of the defined function is suspended at that point. Examples of quote-quad input are:

```
      A←⍞

      ⍴A
0
      B←⍞
QUOTES AREN'T NEEDED
      B
QUOTES AREN'T NEEDED

      X←'CALIFORNIA'∊⍞
ABCDEFGHIJKLMN
      X
1 1 1 1 1 0 0 1 1 1
```

Output

As previously mentioned, the display of most computer output begins at the left margin. Important output characteristics are described below.

1. Width of line. The user can change the number of characters displayed on a line to any number from 32 to 390 via the *)WIDTH* system command (see Section 8), or the ⎕*PW* system variable (see Section 11). Output processing always assumes that the left and right margin stops are placed full left and full right.

2. Fractional number. A fractional number is displayed with one leading zero to the left of the decimal point, even if the number was entered without zero. Examples of fractional numbers are:

```
      .2+.4
0.6
      2÷3
0.6666666667
      .123
0.123
```

3. Exponential notation. APL usually uses exponential form for printing numbers less than *1E*¯5, or greater than *1EN* where *N* is the value of the ⎕*PP* system variable. Decimal form is used for other cases. Numbers printed in exponential form have a magnitude between one and ten followed by an appropriate exponent.

   When an array is displayed, some numbers may be printed in exponential form and some in decimal form, depending on the size of each number. Numbers in a vector are printed with one space between each number, as shown below:

```
1234567.89 1234567890 1.23456789E10
```

   When a matrix is displayed, each column of numbers is printed all in exponential form or all in decimal form. One number requiring exponential form in a column will cause all the numbers in that column to be printed in exponential form. One column of blanks separates columns of numbers. Numbers in a matrix are printed with decimal points aligned, as shown below:

```
      A
 0.0100003 1.2345E12 ¯1.99032
12.3456703 3.0000E0   7.76767676

      A*11
1.000330050E¯22 1.014850423E133      ¯1941.565195
1.015456727E12  1.771470000E5    6211587288
```

4. Significant digits. CP-6 APL carries out all calculations to approximately 18 significant digits, and displays the result rounded off to the value of ⎕*PP* digits. Any trailing zeros are suppressed in the display. Examples are shown below:

```
      4÷3
1.333333333
      5÷2
2.5
```

The user can use the *)DIGITS* system command (see Section 8) or the ⎕*PP* system variable to change the number of significant digits displayed, to a number ranging from 1 to 20. Examples are shown below:

```
      ⎕PP←4
      4÷3
1.333
      5÷2
2.5
```

5. Comparison Tolerance. The arithmetic functions (addition, subtraction, multiplication, and division) are implemented in the computer as functions which represent real numbers through a set of discrete numbers. In CP-6 APL, calculations are carried out to approximately 18 decimal digits. Comparison tolerance is provided by APL to partly disguise the fact that only 18 digits of precision are available. The default value of comparison tolerance in a clear workspace is 1*E*¯13 which causes the equals function to return 1 if the numbers being compared are equal in the first 13 digits. An example of comparison tolerance in comparison is:

```
      1=1+¯2E¯13 ¯9E¯14 0 9E¯14 2E¯13
0 1 1 1 0
```

6. Numeric and character vectors. Numeric vectors are displayed with one blank between items, while character vectors are displayed with no blanks between items, as shown:

```
      2+⍳6
3 4 5 6 7 8
      'ABCXYZ'
ABCXYZ
```

If an array contains both numeric and character scalar values, a trailing blank column is included after each numeric column (except the last column).

```
      1 'A' 2
1 A2
```

7. Arrays of two or more dimensions. The components of a two-dimensional array (i.e., a matrix) are displayed in a rectangular arrangement. The components of an array of more than two dimensions (i.e., a higher-order array) are displayed as a set of rectangles. Character arrays of two or more dimensions are displayed with no spaces between columns. In addition, arrays of more than two dimensions are displayed with extra blank lines separating planes. Examples are shown below:

```
      3 5ρ¯2+ι15
¯1  0  1  2  3
 4  5  6  7  8
 9 10 11 12 13
      2 3 4ρ4+ι24
 5  6  7  8
 9 10 11 12
13 14 15 16

17 18 19 20
21 22 23 24
25 26 27 28
      3 4ρ'NOWISTHETIME'
NOWI
STHE
TIME
      2 2 5ρ'ABCDEF GHIJKL MNOPQR'
ABCDE
F GHI

JKL M
NOPQR
```

8. Simple. An APL array is simple if every item of the array is either a scalar character or a scalar number.

9. Nested Arrays. An array is nested if it is not simple. That is, an array is nested if an item of the array contains another APL array of rank greater than 0. Nested arrays, like other APL arrays, are displayed with columns aligned. The column width is determined by the widest formatted representation of the items in the column.

   The space required to display non-simple items is controlled by the system variable ⎕PS. The column width for a non-simple item may be stated as the width of the formatted value plus the value (|¯1↑⎕PS). The row depth for a non-simple item can be stated as the number of rows required to display the value plus the value (|1↑¯2↑⎕PS).

   The first two items in ⎕PS control the placement of the arrays within the column and row. The first item controls the vertical placement of the formatted array and the second item controls the columnar placement of the formatted array. The first item of ⎕PS can be ¯1 (top), 0 (center), or 1 (bottom). The second item of ⎕PS can be ¯1 (left), 0 (center), or 1 (right).

   The last two items of ⎕PS can be negative to indicate that a vertical bar or box be drawn around the border of the array. The magnitude of the value must be greater than 1 for the box to be drawn.

   The following is an example of displaying nested arrays:

```
      ⎕PS←0 0 ¯3 ¯3  ⍝ CENTER AND DRAW BOXES
      A←2 2ρι4  ⋄ B←3 4ρ'ABCDEFGHIJKL'
      C←2 3ρA B 7 'Z' A B
      C
+---+ +-----+
|1 2| |ABCD|  7
|3 4| |EFGH|
+---+ |IJKL|
      +-----+

      +---+  +-----+
 Z    |1 2|  |ABCD|
      |3 4|  |EFGH|
      +---+  |IJKL|
             +-----+
      ⎕PS←¯1 1 0 2  ⍝ DEFAULT VALUE
      C
1 2 ABCD 7
3 4 EFGH
    IJKL
Z    1 2 ABCD
     3 4 EFGH
         IJKL
```

10. Prototypes. Every APL array contains a prototype which is the type of the first item of the array. For an array whose first item is a simple scalar number, the prototype is 0; for an array whose first item is a simple scalar character, the prototype is a blank. For all arrays, the prototype has the same structure (shape and depth) as the first item and contains zeroes where the corresponding item is numeric, and blanks where the corresponding item is character.

11. Empty arrays. An empty array (an array of no components) can take the form of a vector or an array of two or more dimensions. An empty array produces no display (just another prompt for input). An empty vector (also known as a null vector) can be entered in one of the following ways: ⍳0 or '' or 0⍴0. Similarly, examples of entering empty arrays of two or more dimensions are 0 2⍴4 and 0 0 0⍴0. The display of an empty vector and an empty matrix are shown below:

    ```
          ⍳0
          0 2⍴6
          2+2
    4
    ```

    Note that an empty numeric vector is represented by the expression ⍳0 and any empty character vector is represented by the expression ''. These expressions cannot always be used interchangeably because their prototypes differ. An example is in their use as the right argument in an expansion operation:

    ```
          0\''

          0\⍳0
    0
    ```

    Empty vectors are useful in initializing vectors, in branching, and in the limiting cases of some algorithms.

    Note that the use of an empty array as the argument of a scalar function will result in an empty array:

    ```
          34+⍴0
          0≠2 0⍴5
    ```

12. Blind output. Blind output (see in Section 16) is output as one record of character (literal) data.

13. Stopping a display. The user can stop display of output by pressing the BREAK key.

14. Quad output. When ⎕ appears immediately to the left of an assignment arrow, the value of the expression to the right of the arrow is output. Example:

    ```
          ⎕←A←2+3
    5
    ```

15. Bare output. Normal output includes a concluding carriage return in order that the succeeding entry (whether it is input or output) will begin at the first position on the following line. Bare output, denoted by expressions of the form ⍞←X, does not include a carriage return if the expression is followed either by another expression denoting bare output or character input (of the form X←⍞). For example:

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

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

The carriage returns normally caused by the width setting (⎕PW) are still present in bare output.

Because any expression of the form ⍞←X entered at the keyboard (rather than being executed within a defined function) is followed by another keyboard entry, (concluded by a carriage return), its effect is indistinguishable from the effect of the corresponding normal output.

Section 4

Expression Evaluation

Order of Evaluation

The following subsections describe the order in which APL evaluates expressions.

Right to Left

APL evaluates expressions from right to left, not from left to right as in most written languages. Each function or assignment symbol in an expression operates on the entire expression to the right of it, with the rightmost expression evaluated first, then the next rightmost, and so on. In illustration, notice the following expression:

```
      20×4+5÷2
130
```

In this expression the result of 5÷2 is added to 4, and the result of that is multiplied by 20, thereby yielding the value 130.

Precedence of Functions

Unlike most programming languages (and unlike common algebraic usage) no APL function has precedence over another function. A division operation, for example, is not performed before an adjacent addition unless, of course, the division appears to the right of the addition. Note that in the example cited above, the conventional algebraic function hierarchy would have treated the expression as equivalent to (20×4) + (5÷2), which would have resulted in the value 82.5.

Parentheses

Parentheses can be used in an expression to depart from the right-to-left rule for function execution or left-to-right order for operator execution. They are used just as they are in mathematics for grouping. APL evaluates everything within a pair of parentheses (from right to left) before evaluating the expression of which they are a part. There must be an equal number of left and right parentheses. The beginning APL user may find parentheses convenient to avoid confusion over the difference between APL and conventional algebraic notation.

Some examples of the use of parentheses are shown below:

```
      (3+⍳5)×2+1
12 15 18 21 24
      ((6÷2)×5×4)÷3+12
4
      6÷2×5×4÷3+12
2.25
      (20×4)(+)(5÷2)
82.5
```

Precedence of Operators

Operators have higher precedence than functions. They may be monadic or dyadic (but not both); they always produce a function which may be monadic, dyadic or both. The left operand of an operator is the expression to the left of the operator up to a function (or array) with an array or function to its left. The right operand of a dyadic operator is the first function or array to its right. Monadic operators have their only argument on their left.

Unlike functions, operators are permitted to have arguments that are functions.

Operators and their arguments combine to produce functions (called "derived functions") which are then executed like all other APL functions. In fact, the derived function that is produced by an operator may be used as an argument to another operator.

```
      A←↑∘.+/ (0 100)(0 10 20)(1 2 3 4)
      A
  1   2   3   4
 11  12  13  14
 21  22  23  24

101 102 103 104
111 112 113 114
121 122 123 124
```

In this example, the plus—outer—product reduction is performed on the vector argument to produce the scalar enclosed matrix (which is subsequently disclosed by the first function). Notice that the + is the argument to the outer product operator ∘. and that this derived function (called plus outer product or ∘.+) is the argument to the reduction operator (/).

Value of a Variable versus its Name

When APL encounters a name, it obtains the associated value immediately. This value becomes an argument, and the argument will not change value even if the named variable is assigned a new value. The following example illustrates this evaluation procedure:

```
      (K←2)+K←1
3
```

The *K* to the right of the plus sign was evaluated to the argument having, at that time, value 1. This argument did not change even through *K*'s value changed before the addition was completed.

Default Output

Default output occurs when a non—assignment statement is evaluated. That is, the result is displayed instead of being stored in memory. For example, 2×4 gives default output:

```
      2×4
8
```

Default output is killed by assignment. For example, the expression *A*←2×4 prints no output at the terminal.

```
      A←2×4
```

Instead, the value 8 is assigned to variable *A* and stored in computer memory.

When a compound statement (Section 6) includes both non—assignment and assignment expressions, the non—assignment expressions produce output while the assignment expressions do not. Some examples are:

```
      4 ◊ 5
4
5
      4 ◊ 'A' ◊ 5
4
A
5
      4+2 ◊ A←5+2 ◊ 4+3
6
7
      X←⍳5 ◊ Y←2+4
```

Errors and Breaks

If the user discovers an error in a statement before the RETURN key is pressed, the user can RUBOUT to the error and retype the rest of the line as described in Section 2. (On all terminals, the standard CP-6 input line editing mechanism is applicable. See the CP-6 Programmer Reference Manual (CE40)). An example (using the RUBOUT key) is:

```
      A←5×B←8×\<R>              (<R> indicates <ESC> R.)
      A←5×B←8÷4
      A
10
```

If the user has entered a line and APL detects an error or double break during statement execution, execution of the statement is terminated. If the statement in execution contains multiple assignments or is a compound statement, the assignments and expressions to the right of the termination point (denoted by a caret) will be completed. The current expression and any expressions to the left of the termination point will usually not be completed. If a dyadic operator or function is indicated, however, its left argument expression (possibly containing assignments) will have been completed before the function or operator was invoked. Examples are shown below (it is assumed that sidetracking, see Section 10, is not applicable in these examples).

```
      C←4÷(D←0)×Z←5
DOMAIN ERR
      C←4÷(D←0)×Z←5
         ^

      C
UNDEFINED
      C
      ^
      D
0
      Z
5

      A←4÷2⋆.5 ◊ F←0 ◊ E←4÷2+1 ◊ E÷F
DOMAIN ERR
      A←4÷2⋆.5 ◊ F←0 ◊ E←4÷2+1 ◊ E÷F
                                  ^
      E
1.333333333
      F
0
```

In both of these examples the user has attempted to divide by zero, thus producing a *DOMAIN ERR* message. In the first example the error is detected before variable *C* is assigned a value, so *C* remains *UNDEFINED* as shown. In the second example, *E* and *F* had values assigned to them before the error was detected.

If the user has entered a line and APL detects a simple error before any part of the line is executed, APL displays the message *LINESCAN ERR* and a caret at the error point. The user can type <ESC> D to recall the line in error and edit it to correct the problem. For example:

```
      A←234 + ( ) ×⎕⋆3
LINESCAN ERR   ^
      <D>
      A←234 + ( )×⎕⋆3
                \⍳3<R>
      A←234 + (⍳3)×⎕⋆3
⎕:
      4
      A
298  362   426
```

Note that the difference between a *LINESCAN ERROR* and a *BAD CHAR* error is that the former involves an error in expression logic or syntax, while the latter involves the typing of an illegal APL character.

Section 5

APL Primitive Functions

A primitive function is a symbol indicating that a basic APL function, such as addition or division, is to be performed. A symbol denoting a primitive function is either a non-alphanumeric character or a combination of such characters. For example, addition is denoted by the + symbol and division is denoted by the ÷ symbol.

Some of the basic primitive functions are "monadic" and others are "dyadic". That is, some require a single argument and others require two. For example, the reciprocal function is monadic (e.g., ÷*A*) and the division function is dyadic (e.g., *A*÷*B*). Most of the symbols denoting functions are used for both monadic and dyadic functions. APL distinguishes between the monadic and dyadic use of any given function by testing for the absence or presence of a left argument.

o Syntax Conventions

Syntax conventions used throughout this section are as follows:

R denotes the result of a function.

← denotes the replacement of any previous value of the symbolic variable to the left of the arrow.

A denotes a left argument.

B denotes a right argument.

M denotes a monadic function.

D denotes a dyadic function.

Following are some examples of the use of these conventions:

R←*M B*   *R*←*A D B*

o Argument Characteristics

In discussing functions, certain argument characteristics will be referenced frequently. The terms used are described below.

Domain – In general, the type of data item such as integer data or floating-point data. For some functions the domain of an argument may be especially restricted (see the example for the circular function later in this section).

Rank – The number of coordinates in an array argument. (A rank of zero indicates a scalar.)

Length – The number of items in a coordinate of an argument.

Shape – The vector made up of the lengths of all coordinates of an argument.

o Domain Tables

In the tables listing the domains of the results for various types of argument data, the following symbology is used:

N denotes numeric data.

C denotes character data.

L	denotes logical data (1 or 0).
I	denotes integer data.
F	denotes floating-point data.
DE	denotes a *DOMAIN ERR*.
RE	denotes a rank error.

Scalar Functions

APL functions vary considerably in how they reference the items of array arguments and in the characteristics (rank and dimensions) of the result compared with those of the arguments. A group of functions called scalar functions follow a common set of rules with respect to the characteristics of the arguments and results. These functions, comprising the arithmetic group, the relational group, and the logical group, are so named because they are defined in terms of scalar arguments. Extensions of scalar functions to array arguments are equivalent to performing item-by-item scalar functions.

If an item of an array contains another APL array, the operation is performed on each item within the nested array repeatedly, until the operation selects a simple scalar numeric or character item. All of the rank, length and domain checks are made at each level of nesting. The shape of the resulting structure follows the rules at each function application level.

o Monadic Scalar Functions

The argument used with a monadic scalar function may have any rank and dimensions. The result has the rank and dimensions of the argument. The domain of the result may differ from the domain of the argument.

o Dyadic Scalar Functions

If the rank and dimensions of the argument used with a dyadic scalar function are the same, the function is performed on corresponding items of the two arguments and the result has the same rank and dimensions. If the arguments have different ranks or dimensions and both contain other than one item, a rank or length error will be reported.

If one argument has multiple items and the other is a scalar or single item array, the function is performed on the single item with each item of the multiple item argument. The result has the rank and dimensions of the multiple item argument. If neither argument has multiple items, the result is given the shape of the higher ranked argument. The shapes of results of scalar functions for various arguments are tabulated below.

		Right Argument							
		S	V1	M1	H1	V	M	H	
	S	S	V1	M1	H1	V	M	H	
	V1	V1	V1	M1	H1	V	M	H	
	M1	M1	M1	M1	H1	V	M	H	
Left Argument	H1	H1	H1	H1	H1	V	M	H	Result
	V	V	V	V	V	V~	RE	RE	
	M	M	M	M	M	RE	M~	RE	
	H	H	H	H	H	RE	RE	H~~	

~ Dimensions of arguments must be identical.
~~Rank and dimensions of arguments must be identical.

where

S	denotes a scalar.
V	denotes a vector.
M	denotes a matrix.
H	denotes a higher order array.
RE	denotes a rank error.
V1	denotes a single item vector.
M1	denotes a single item matrix.
H1	denotes a single item higher order array.

Arithmetic Functions

Each function in the arithmetic group has a monadic and dyadic form. If any argument is in the character domain, a *DOMAIN ERR* is reported. Results are always in the numeric (integer or floating) domain. If during the execution of any function a numeric result exceeds the range of CP-6 APL numbers, a *DOMAIN ERR* is reported.

+ Function (Conjugate, Addition)

- Monadic + is the Conjugate function.

  R←+B

  Domain Table:

B	C	L	I	F
R	C	L	I	F

  Examples:

```
      +5
5
      +(¯3 2 1.1)
¯3 2 1.1
      +0 1 0
0 1 0
```

- Dyadic + is the Addition function.

  R←A+B

  Domain Table:

\B A\	C	L	I	F
C	DE	DE	DE	DE
L	DE	I	I	F
I	DE	I	I/F~	F
F	DE	F	F	F

  ~ The result is floating-point if the value exceeds the integer range.

  Examples:

```
      2 3 1+5 ¯1 0
7 2 1
      2.5+1 2 3
3.5 4.5 5.5
      2.5 3.5+1 2 3
LENGTH ERR
      2.5 3.5+1 2 3
             ^
```

- Function (Negate, Subtraction)

- Monadic - is the Negate function.

  R←-B

  Domain Table:

B	C	L	I	F
R	DE	I	I/F	F

Examples:

```
      -5
¯5
      -(¯3 2 1.1)
3 ¯2 ¯1.1
```

- o Dyadic - is the Subtraction function.

R←A-B

Domain Table:

\B A\	C	L	I	F
C	DE	DE	DE	DE
L	DE	I	I	F
I	DE	I	I/F~	F
F	DE	F	F	F

~ The result is floating-point if the value exceeds the integer range.

Examples:

```
      2 3 1-5 ¯1 0
¯3 4 1
      2.5-1 2 3
1.5 0.5 ¯0.5
      1 2 3-2.5
¯1.5 ¯0.5 0.5
```

× Function (Signum, Multiplication)

- o Monadic × is the Signum function.

R←×B

If *B* is positive, *R* is 1. If *B* is zero, *R* is 0. If *B* is negative, *R* is -1.

Domain Table:

B	C	L	I	F
R	DE	L	I	I

Examples:

```
      ×¯2 3.5 0 .001
¯1 1 0 1
```

- o Dyadic × is the Multiplication function.

R←A×B

Domain Table:

\B A\	C	L	I	F
C	DE	DE	DE	DE
L	DE	L	I	F
I	DE	I	I/F~	F
F	DE	F	F	F

~ The result is floating-point if the value exceeds the integer range.

Examples:

```
      5×1 ¯1 7
5 ¯5 35
      ¯1 2 0×1.5 2.5 3.5
¯1.5 5 0
      2.5 3×1.7 12 .01
LENGTH ERR
      2.5 3×1.7 12 0.01
           ^
```

÷ Function (Reciprocal, Division)

- Monadic ÷ is the Reciprocal function.

  R←÷B

  Domain Table:

B	C	L	I	F
R	DE	F	F	F

  If B is zero, the error *DOMAIN ERR* is reported.

  Examples:

```
      ÷1 2 5
1 0.5 0.2
      ÷.01
100
```

- Dyadic ÷ is the Division function.

  R←A÷B

  Domain Table:

\B A\	C	L	I	F
C	DE	DE	DE	DE
L	DE	I/F~	I/F~	F
I	DE	I/F~	I/F~	F
F	DE	F	F	F

  ~ The quotient is integer if *B* is an exact multiple of *A*; otherwise, it is floating-point.

If *B* is zero and *A* is other than zero, the error *DOMAIN ERR* is reported. If both *B* and *A* are zero, *R* is 1. If *R* exceeds the range of floating-point numbers, *DOMAIN ERR* is reported.

Examples:

```
      7 8 9÷2 10 18
3.5 0.8 0.5
      0÷12
0
      0÷0
1
```

* Function (Exponential, Exponentiation)

- o Monadic * is the Exponential function.

The monadic * is the equivalent of the dyadic form with e (the base of the natural logarithms) supplied as a left argument. The value used for e is approximately 2.71828182845904524.

*R←*B*

Domain Table:

B	C	L	I	F
R	DE	F	F	F

If *B* exceeds 352.1187677244522173, *DOMAIN ERR* is reported. If *B* is less than ¯355.2379300369719713, *R* is 0.

Examples:

```
      *1 .5 0 ¯190
2.718281828 1.648721271 1 3.048234951E¯83
```

- o Dyadic * is the Exponentiation function.

*R←A*B*

Domain Table:

\B A\	DE	L	I	F
C	DE	DE	DE	DE
L	DE	L	I	F
I	DE	I	I/F	F
F	DE	F	F	F

If both *A* and *B* are zero, *R* is 1. If *A* is zero and *B* is less than zero, *DOMAIN ERR* is reported. If *A* is less than zero and *B* is not an integer, *DOMAIN ERR* is reported. If *R* exceeds range of floating-point numbers, *DOMAIN ERR* is reported.

Examples:

```
      0 1 2 ¯2*0 5.3 0.5 3
1 1 1.414213562 ¯8
      ¯2*¯.3
DOMAIN ERR
      ¯2*¯0.3
        ^
```

⍟ Function (Natural Logarithm, Logarithm)

- Monadic ⍟ is the Natural Logarithm (base e) function.

  R←⍟B

  Domain Table

B	C	L	I	F
R	DE	F	F	F

  If *B* is not a positive number, *DOMAIN ERR* is reported.

  Example:

```
      ⍟ 2.718281828459 1 .049787068367893943
1 0 ¯3
```

- Dyadic ⍟ is the Generalized Logarithm (base A) function.

  R←A⍟B

  If *A* or *B* is not a positive number, *DOMAIN ERR* is reported. If *A* is 1 and *B* is other than 1, *DOMAIN ERR* is reported.

  Domain Table:

\B A\	C	L	I	F
C	DE	DE	DE	DE
L	DE	F	F	F
I	DE	F	F	F
F	DE	F	F	F

  Examples:

```
      2 3 16⍟1 27 .25
0 3 ¯0.5
      10⍟10 .1 250
1 ¯1 2.397940009
```

⌈ Function (Ceiling, Maximum)

- Monadic ⌈ is the Ceiling function.

  R←⌈B

  For ⌈, *R* is the algebraically smallest integer greater than *B-⎕CT×1⌈B*. *⎕CT* is *1E¯13* unless it has been reassigned.

  Domain Table:

B	C	L	I	F
R	DE	L	I	I/F~

  ~ The result is floating-point if the value exceeds the integer range.

  Examples:

```
      ⌈ 2.1  2.01  ¯2.01  2.00000000000000001
3 3 ¯2 2
```

- Dyadic ⌈ is the Maximum function.

  R←*A*⌈*B*

  R is the larger value of *A* and *B*.

  Domain Table:

\B A\	C	L	I	F
C	DE	DE	DE	DE
L	DE	L	I	F
I	DE	I	I	F
F	DE	F	F	F

  Examples:

```
      5⌈12
12
      (¯1 5 7)⌈ 5
5 5 7
      ¯1 2 3.5⌈¯3 ¯2 7.1
¯1 2 7.1
```

⌊ Function (Floor, Minimum)

- Monadic ⌊ is the Floor function.

  R←⌊*B*

  ⌊*B* is the largest integer less than *B*+⎕*CT*×1⌈*B*

  Domain Table:

B	C	L	I	F
R	DE	L	I	I/F~

  ~ The result is floating-point if the value exceeds the integer range.

  Examples:

```
      ⌊ 2.9 2.99 ¯2.99 2.99999999999999999
2 2 ¯3 3
```

- Dyadic ⌊ is the Minimum function.

  R←*A*⌊*B*

  R is the smaller value of *A* and *B*.

  Domain Table:

\B A\	C	L	I	F
C	DE	DE	DE	DE
L	DE	L	I	F
I	DE	I	I	F
F	DE	F	F	F

Examples:

```
      5⌊12
5
      5⌊¯1 5 7
¯1 5 5
```

| Function (Absolute Value, Residue)

- Monadic | is the Absolute Value function.

  R←|*B*

  Domain Table:

B	C	L	I	F
R	DE	L	I	F

  Examples:

```
      |¯2.15
2.15
      |¯1 ¯4.3 5 7.2
1 4.3 5 7.2
```

- Dyadic | is the Residue function.

  R←*A*|*B*

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

  Examples:

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

      X←21.824
      .01|X
0.004
```

  The definition of residue can be stated formally as follows:

```
      A|B ←→ B-A×⌊B÷A+A=0
```

  Domain Table:

\B A\	C	L	I	F
C	DE	DE	DE	DE
L	DE	L	I	F
I	DE	I	I	F
F	DE	F	F	F

○ Function (Pi Times, Circular)

- Monadic ○ is the Pi Times function.

  R←○*B*

  The result is 3.14159265358979324 times *B*.

  Domain Table:

B	C	L	I	F
R	DE	F	F	F

  Examples:

```
      ○1
3.141592654
      ○2 .5
6.283185307 1.570796327
```

- Dyadic ○ is the Circular function.

  R←*A*○*B*

  The value of *A* determines the computed function of *B* according to the following convention.

Table 5–1. Circular Functions

A	R	Domain of *B*~	Range of *R*~
–7	Archtanh	1≥|*B*	24.9532985 to ¯24.9532985
–6	Arccosh	(1≤*B*)^*B*≤*MAX*⋆.5~~	3.292722539*E*¯10 to 0
–5	Archsinh	(*MAX*⋆.5)≥|*B*~~	+352.811914905 to 0
–4	*B*×(1-*B*⋆¯2)⋆0.5	1≤|*B*	+352.811914905 to 0
–3	Arctan	*MAX*≥*B*	Pi/2 to Pi/2
–2	Arccos	1≥|*B*	0 to Pi
–1	Arcsin	1≥|*B*	–Pi/2 to Pi/2
0	(1-*B*⋆2)⋆.5	1≥|*B*	0 to 1
1	Sine	4096>|*B*	¯1 to 1
2	Cosine	4096>|*B*	¯1 to 1
3	Tangent	4096>|*B*	approximately ¯6*E*18 to 6*E*18
4	(1+*B*⋆2)⋆.5	(*MAX*⋆.5)≥|*B*	1 to *MAX*⋆.5
5	Sinh	352.811914905≥|*B*	-*MAX* to *MAX*
6	Cosh	352.811914905≥|*B*	_1 to *MAX*
7	Tanh	*MAX*≥|*B*	¯1 to 1

~The domains of *B* and ranges of *R* are narrower than those theoretically possible. The limitations reflect the precision with which real numbers are represented and with which computations are made in the computer.

~~*MAX*=8.379879956*E*152
MAX⋆.5=2.894802231*E*76

For sine, cosine, and tangent functions and their hyperbolic counterparts, *B* is expressed in radians. For the inverse trigonometric functions, the value of *R* is in radians. The domain of the result is always floating-point.

Examples:

```
      1○○2
¯2.064961208E¯18
      0○.4 .5 .6
0.916515139 0.8660254038 0.8
      ¯7○.5
0.5493061443
```

Notice in the first example that the result (the sine of 2×Pi) should actually be zero. The actual result reflects the effect of computing with approximately 18 decimal-place precision.

! Function (Factorial, Binomial)

- Monadic ! is the Generalized Factorial function.

  R←!B

  The result is *B* factorial for non-negative integral value of *B*. If *B* is not an integer, the result is the gamma function of *B*+1.

  Domain Table:

B	C	L	I	F
R	DE	L	I/F	F

  Examples:

```
      !7
5040
      !.66 ¯.75 0
0.9016683712 3.625609908 1
```

- Dyadic ! is the Binomial function.

  R←A!B

  If the arguments are positive integers and *A* is less than or equal to *B*, the result is the number of combinations of *B* things taken *A* at a time. In general, (*A!B*) is:

  R←(!B)÷(!A)×!B-A

  Domain Table:

\B A\	C	L	I	F
C	DE	DE	DE	DE
L	DE	L	I	F
I	DE	I	I/F	F
F	DE	F	F	F

  Examples:

```
      1!2
2
      1.5!2
1.697652726
      1.5!2.5
2.5
      5!52
2598960
```

Relational Functions

The six relational functions are used to compare two values and return a value of 1 if the relation is true or a value of 0 if the relation is false. The truth value can be used in calculations in the same way as any other value of 1 or 0. The relational functions are strictly dyadic, requiring a left argument.

The expressions used below to define the relational functions includes a value *DELTA*. This is a relative tolerance value related to the user—established comparison tolerance in the following way:

```
    DELTA←⎕CT×(|A)⌈|B
```

< Function (Less Than)

o Dyadic < is the Less Than function.

R←A<B

The result is 1 if *(A-B)<-DELTA*, and is 0 otherwise.

Domain Table:

\B A\	C	L	I	F
C	DE	DE	DE	DE
L	DE	L	L	L
I	DE	L	L	L
F	DE	L	L	L

Examples:

```
      2<4.5
1
      1 2 3<3 2 1
1 0 0
```

≤ Function (Less Than or Equal)

o Dyadic ≤ is the Less Than or Equal function.

R←A≤B

The result is 1 if *(A-B)≤DELTA*, and is 0 otherwise.

Domain Table:

\B A\	C	L	I	F
C	DE	DE	DE	DE
L	DE	L	L	L
I	DE	L	L	L
F	DE	L	L	L

Examples:

```
      1≤2
1
      1 2 3≤3 2 1
1 1 0
```

= Function (Equals)

- o Dyadic = is the Equals function.

  R←A=B

  If *A* and *B* are numeric, the result is 1 if (|*A-B*)≤*DELTA*, and is 0 otherwise. If *A* and *B* are characters, *R* is 1 if *A* and *B* are the same, and 0 if they are not. If one argument is character and the other numeric, *R* is 0.

  Domain Table:

\B A\	C	L	I	F
C	L	L	L	L
L	L	L	L	L
I	L	L	L	L
F	L	L	L	L

Examples:

```
      1 2 3=3 2 1
0 1 0
      'THIS'='THAT'
1 1 0 0
      'A'=5
0
```

≥ Function (Greater Than or Equal)

- o Dyadic ≥ is the Greater Than or Equal function.

  R←A≥B

  The result is 1 if (*A-B*)≥-*DELTA*, and is 0 otherwise.

  Domain Table:

\B A\	C	L	I	F
C	DE	DE	DE	DE
L	DE	L	L	L
I	DE	L	L	L
F	DE	L	L	L

Examples:

```
      1≥2
0
      1 2 3≥3 2 1
0 1 1
```

> Function (Greater Than)

- Dyadic > is the Greater Than function.

  R←A>B

  The result is 1 if (*A-B*)>*DELTA*, and is 0 otherwise.

  Domain Table:

\B A\	C	L	I	F
C	DE	DE	DE	DE
L	DE	L	L	L
I	DE	L	L	L
F	DE	L	L	L

  Examples:

```
      2>3.4
0
      1 2 3>3 2 1
0 0 1
```

≠ Function (Not Equal)

- Dyadic ≠ is the Not Equal function.

  R←A≠B

  If *A* and *B* are numeric, the result is 1 if (|*A-B*)>*DELTA*, and is 0 otherwise. If *A* and *B* are characters, *R* is 0 if *A* and *B* are the same, 1 if they are not. If one argument is character and the other numeric, *R* is 1.

  Domain Table:

\B A\	C	L	I	F
C	L	L	L	L
L	L	L	L	L
I	L	L	L	L
F	L	L	L	L

  Examples:

```
      1 2 3≠3 2 1
1 0 1
      'THIS'≠'THAT'
0 0 1 1
      'A'≠5
1
```

Logical Functions

The five logical functions are used to perform logical operations, returning a result of 0 or 1. The first four operations are strictly dyadic, and the last (the "not" operator) is strictly monadic.

∧ Function (And, LCM)

o Dyadic ∧ is the And function.

R←A∧B

For logical values of *A* and *B* (0,1) the result is 1 if *A* and *B* are both 1, and is 0 otherwise. Otherwise, the result is the least common multiple of *A* and *B*.

Domain Table:

\B A\	C	L	I	F
C	DE	DE	DE	DE
L	DE	L	I	F
I	DE	I	I/F	F
F	DE	F	F	F

Examples:

```
      1∧1
1
      (1<2)∧(3=4)
0
      1 1 0 0∧1 0 1 0
1 0 0 0
      3∧2
6
      0.2∧0.7
1.4
```

∨ Function (Or, GCD)

o Dyadic ∨ is the Or function.

R←A∨B

For logical values of *A* and *B* (0,1) the result is 1 if either *A* or *B* are both 1, and is 0 otherwise. Otherwise, the result is the greatest common divisor of *A* and *B*.

The greatest common divisor of two values will always be less than or equal (in magnitude) to each of the values. The result of this function is always non-negative.

Domain Table:

\B A\	C	L	I	F
C	DE	DE	DE	DE
L	DE	L	I	F
I	DE	I	I	F
F	DE	F	F	F

Examples:

```
          1∨1
1
          (1<1)∨(3=4)
0
          1 1 0 0∨1 0 1 0
1 1 1 0

          4∨6
2
```

⍲ Function (Nand)

- Dyadic ⍲ is the Nand function.

  R←A⍲B

  The result is 0 if *A* and *B* are both 1, and is 1 otherwise.

  Domain Table:

\B A\	C	L	I	F
C	DE	DE	DE	DE
L	DE	L	L	L
I	DE	L	L	L
F	DE	L	L	L

A *DOMAIN ERR* results if both *A* and *B* are not equal to either 1 or 0.

Examples:

```
          1⍲1
0
          (1<2)⍲(3=4)
1
          1 1 0 0⍲1 0 1 0
0 1 1 1
```

⍱ Function (Nor)

- Dyadic ⍱ is the Nor function.

  R←A⍱B

  The result is 0 if either *A* or *B*, or both, are 1, and is 1 otherwise.

  Domain Table:

\B A\	C	L	I	F
C	DE	DE	DE	DE
L	DE	L	L	L
I	DE	L	L	L
F	DE	L	L	L

  A *DOMAIN ERR* results if both *A* and *B* are not equal to either 1 or 0.

  Examples:

```
      1⍱1
0
      (1>2)⍱(3=4)
1
      1 1 0 0⍱1 0 1 0
0 0 0 1
```

~ Function (Not)

- Monadic ~ is the Not function.

  R←~B

  The result is 1 if *B* is 0, and is 0 if *B* is 1.

  Domain Table:

B	C	L	I	F
R	DE	L	L	L

  A *DOMAIN ERR* results if *B* is not equal to either 1 or 0.

  Examples:

```
      ~1
0
      ~0
1
      ~(2.5-1.5)
0
```

Mixed Functions

Functions not categorized previously as monadic or dyadic scalar functions are called mixed functions. Rules for shapes and domains of the arguments and results vary and are described for the individual functions.

? Function (Roll, Deal)

o Monadic ? is the Roll function.

R←?*B*

Each item *R*[*I*] of the result is an integer selected pseudorandomly from (⍳ *B*[*I*]). The range of the result depends on the value of the index origin (see the deal operator below). The shape of the result is the same as that of the right argument.

Examples:

```
      ?5
3
      ?2 4 6
2 4 1
      ?3 3 3 3
1 2 3 1
```

o Dyadic ? is the Deal function.

R←*A*?*B*

The result is a vector of integers comprising *A* components pseudorandomly selected from (⍳ *B*) without replacement, preventing the duplication of integers in *R*. The range of the result depends on the index origin. If the index origin is 0, the range is 0 through *B*-1. If the index origin is 1, the range is 1 through *B*.

A may not exceed *B*, and both must be simple numeric items.

Examples:

```
      2?4
4 2
      6?6
2 1 3 5 4 6
      A←10 20 30 40 50 60 70 80
      A[4?8]
70 20 10 40
```

⍳ Function (Index Generator, Index Of)

o Monadic ⍳ is the Index Generator function.

R←⍳ *B*

B must be a single simple numeric item, equal to an integer. The result is a simple integer vector comprising *B* items, beginning with the index origin and incrementing monotonically by 1. The index origin can be changed by assigning a value to ⎕*IO*. If *B* is 0 the result is an empty numeric vector.

Examples:

```
      ⎕←R←⍳4
1 2 3 4
      ⎕IO←0
      ⎕←R←⍳4
0 1 2 3
```

- Dyadic ⍳ is the Index Of function.

  R←A⍳ B

  The value of each item of the result is the smallest index *I* such that *A[I]* is equivalent to the corresponding item in *B*. The left argument must be a vector. The right argument may have any rank. If no match for an item of *B* is found in *A*, that item of the result is set to (⍴*A*)+⎕*IO*. The shape of the result is the same as the shape of the right argument. The result is simple and in the integer domain.

  Note that *A* may be an empty vector and the value of the result depends on whether the index origin is 1 (the default case) or 0. *A* and *B* may be of any domain. Note, however, that if *A* is all character data, for example, and *B* is all numeric, the result will be entirely "no match" values.

  Examples:

```
      2 4 6 8⍳3
5
      'XYZ'⍳'W'
4
      'DOG'⍳'COT'
4 2 4

      'XYZ'  'DOG'⍳'O'  'XYZ'  'X'  'DOG'
3 1 3 2
```

⊂ Function (Enclose)

- Monadic ⊂ is the Enclose function.

  R←⊂B

  R←⊂[K]B

  B may be any APL array. This function increases the depth of *B* by 1 and decreases the rank. If an axis is not specified, the result is a scalar whose only item is the array *B*. If *B* is a simple scalar character or number, the result is *B* unchanged.

  If an axis is specified, all of the axes specified by *K* are enclosed, resulting in an array of rank (⍴⍴*B*)-⍴,*K*, containing items of rank ⍴,*K*. The shape of the result is (⍴*B*)[(~(⍳⍴⍴*B*)∊*K*)/⍴*B*] and the shape of each item of *B* is (⍴*B*)[*K*].

  Examples:

```
      ⎕PS←0 0 ¯3 ¯3
      ⊂6
 6
      ⊂,6
 +-+
 |6|
 +-+
      ⊂'SENATE'
 +------+
 |SENATE|
 +------+
      ⊂[1]2 3⍴'ABCDEF'
+--+ +--+ +--+
|AD| |BE| |CF|
+--+ +--+ +--+
      ⎕PS←¯1 1 0 2
      A←'STEVE'  'MARK'  'TOM'  'BRUCE'
      A[3]←⊂'THOMAS'
      A
STEVE  MARK  THOMAS  BRUCE
```

⊃ Function (Disclose, Pick)

- Monadic ⊃ is the Disclose function.

  R←⊃B

  R←⊃[K]B

  The result is an array whose depth is one less than that of *B* and whose rank has increased by the rank of the non-scalar items of *B*. All of the non-scalar items of *B* must have the same rank although they may vary in shape. If *B* is a simple array then the result is *B*.

  If *B* is a simple scalar, the result is *B*. Otherwise if *B* is a scalar, the result is the array contained in *B*.

  If axes are specified, they indicate where to insert the axes of the items of *B* into the result. When no axes are specified, the new axes are inserted after the axes of *B*. The number of axes specified must equal the rank of the non-scalar items of *B*.

  Examples:

```
      ⍴⎕←⊃'WHO'  'WHAT'  'WHEN'  'WHERE'
WHO
WHAT
WHEN
WHERE
4 5
      ⍴⎕←⊃[1]'STEVE'  'MARK'  'TOM'  'BRUCE'
SMTB
TAOR
ERMU
VK C
E  E
5 4
      ⊃'CP-6'
CP-6
```

- Dyadic ⊃ is the Pick function.

  R←A⊃B

  The result is an item from the (⍴*A*)'th level of nesting in *B* selected by the path specified in *A*. *A* must be a scalar or vector containing only simple integer scalars or vectors.

  The first item of *A* must contain valid indices of *B*. These indices select an item of *B* which is then indexed by the next item of *A* until all items of *A* have been used. The final array is the result of this function.

  If *A* is empty, the result is *B*.

Examples:

```
      3 ⊃'ABCDEFGHIJ'
C
      2 3 (2 1) ⊃ 1 (1 2 (2 2 ⍴1 2 3 4) 4) 3 2 1
3
      2 (⍳0)⊃ 90 91 92
91
```

≡ Function (Depth, Equivalence)

- Monadic ≡ is the Depth function.

  R←≡*B*

  The result is a simple non-negative integer scalar indicating the maximum depth of nesting in *B*. *B* may be any APL array.

  The depth of a simple scalar character or number is defined as 0. Non-scalar arrays containing only depth 0 items have depth 1. All other arrays have a depth of 1+⌈/≡¨*B*.

  A depth greater than 1 indicates that an array is not simple.

  Examples:

```
      ≡29
0
      ≡23 29 31
1
      ≡'ABC'  4  (5 (6 7))
3
      ≡'CABLE'  'CARS'
2
```

- Dyadic ≡ is the Equivalence function.

  R←*A*≡*B*

  The result is a simple logical scalar indicating whether every item of the left argument is equivalent to every item of the right argument. The result is 0 if any item of *A* is not equivalent to the corresponding item of *B*.

  Comparison tolerance is used if corresponding items of *A* and *B* are numeric. Arrays are equivalent if they have the same shape and structure, and if all corresponding values in each structure are equal.

  Empty arrays are equivalent only if their prototypes are also equivalent.

  Examples:

```
      1984≡,1984
0
      'APPLE'≡'PIE'
0
      10 20≡9+1 11
1
      10 (9 8)≡4 (3 2)+6
1
```

, Function (Ravel, Catenation, Lamination)

- Monadic , is the Ravel function.

  R←,B

  The result is a vector comprising the components of the argument *B* in index sequence. The argument can have any shape and dimensions.

  Examples:

```
      B←2 2⍴⍳4
      B
1 2
3 4
      ,B
1 2 3 4
      B[1;1]←⊂B
      B
1 2  2
3 4
  3  4
      ,B
1 2  2 3 4
3 4
      ⎕←C←2 4⍴'LEVELSIX'
LEVE
LSIX
      ,C
LEVELSIX
```

- Dyadic , is the Catenation and Lamination function.

  R←A,[K]B

  The catenation coordinate *K* is acceptable if (⌈*K*)∊⍳(⍴⍴*A*)⌈⍴⍴*B*. The catenation coordinate is ⌈*K*.

  If *A* and *B* are vectors or scalars, the result is a vector comprising all items of *A* followed by all items of *B*.

  Examples:

```
      A←1 2 3
      B←4 5 6
      A,B
1 2 3 4 5 6
      C←'STR'
      D←'AND'
      C,D,A
STRAND1 2 3
```

Catenation

Arguments *A* and *B* are conformable for catenation if:

1. The ranks are equal and all coordinates except the catenation coordinate are equal.

2. The rank of one argument is one less than the other and all coordinates except the catenation coordinate of the higher rank argument are equal to all coordinates of the lower rank argument. The lower rank argument is subsequently treated as if its rank were equal to the other argument and its catenation coordinate length were 1.

3. Either *A* or *B* is a scalar. The scalar argument is subsequently treated as if its shape were equal to the other argument with a catenation coordinate length of 1.

If *A* and *B* have conformable shapes and one or both are of higher rank than vector, catenation joins *A* and *B* along an existing coordinate. If no coordinate is specified, catenation occurs along the last coordinate. Scalar arguments are extended for catenation in this case.

Examples:

```
      ⎕←M←4 7⍴'M'
MMMMMMM
MMMMMMM
MMMMMMM
MMMMMMM
      X←2 7⍴'X'
      Y←'1234567'
      Z←'1234'
      W←'o'
      M,[1]X
MMMMMMM
MMMMMMM
MMMMMMM
MMMMMMM
XXXXXXX
XXXXXXX
      M,[1]Y
MMMMMMM
MMMMMMM
MMMMMMM
MMMMMMM
1234567
      M,Z
MMMMMMM1
MMMMMMM2
MMMMMMM3
MMMMMMM4
      M,[1]W
MMMMMMM
MMMMMMM
MMMMMMM
MMMMMMM
ooooooo
      M,W
MMMMMMMo
MMMMMMMo
MMMMMMMo
MMMMMMMo
```

Lamination

If a non-integer coordinate value is indicated in catenation, and its ceiling is a valid coordinate, the function performed is termed lamination. In this case the variable *A* and *B* are joined on a new coordinate. The length of the new coordinate is always 2.

In the following examples, the index origin is 1. If a coordinate of zero or less, or three or more, were specified, *RANK ERR* would be reported.

Examples:

```
      M,[.5]W
MMMMMMM
MMMMMMM
MMMMMMM
MMMMMMM
```

```
ooooooo
ooooooo
ooooooo
ooooooo
      ρM,[.5]W
2 4 7
      M,[1.5]W
MMMMMMM
ooooooo

MMMMMMM
ooooooo

MMMMMMM
ooooooo

MMMMMMM
ooooooo
      ρM,[1.5]W
4 2 7
      M,[2.5]W
Mo
Mo
Mo
Mo
Mo
Mo
Mo

Mo
Mo
Mo
Mo
Mo
Mo
Mo

Mo
Mo
Mo
Mo
Mo
Mo
Mo

Mo
Mo
Mo
Mo
Mo
Mo
Mo
      ρM
4 7
      ρM,[2.5]W
4 7 2
```

ρ Function (Shape, Reshape)

- Monadic ρ is the Shape function.

  R←ρ*B*

  The result is an integer vector comprising the number of items each index of *B* contains. That is, *R* contains the highest index in each coordinate of *B* in origin 1. Thus, the expression ρρ*B* represents the rank (number of dimensions) of *B*. If *B* is a scalar, ρ*B* results in an empty vector.

  Examples:

```
      B←2 4 6 8
      ρB
4
      C←2 3ρ'PIFFLE'
      ρC
2 3
```

- Dyadic ρ is the Reshape function.

  R←*A*ρ*B*

  The result is an array with the dimensions specified by vector *A* and the contents of *B*. Items of *A* may be positive integers or zero. If any component of *A* is zero, *R* is empty. If *A* is empty, *R* is a scalar. If *B* is empty, the prototype of *B* is used to fill the result. If the reshape requires fewer items than *B* contains, only the required items are in the result. If the result requires more items than *B* contains, *B* is cyclically reused as required. *B* may be of any rank or domain.

  Examples:

```
      2ρ3 4 5 6
3 4
      2 4ρ⍳5
1 2 3 4
5 1 2 3


      3 3ρ'AB'   'CDE'   'FGHI'   'JKLMN'
   AB    CDE  FGHI
JKLMN     AB   CDE
 FGHI  JKLMN    AB
```

⌽ Function (Reversal, Rotation)

- Monadic ⌽ is the Reversal function.

  R←⌽[*K*]*B*

  The result is a reversal along the *K*'th coordinate of *B*. If *K* is omitted, the last coordinate is assumed. (If ⊖ is used instead of ⌽, the first coordinate is assumed).

  Examples:

```
      ⌽'EMIT'
TIME
      ⌽[1]3 3ρ⍳9
7 8 9
4 5 6
```

```
1 2 3
      ⌽3 3⍴⍳9
3 2 1
6 5 4
9 8 7
      ⌽'FOX' 'WOLVERINE' 'DOG' 'CAT'
CAT DOG WOLVERINE FOX
```

- Dyadic ⌽ is the Rotation function.

  R←*A*⌽[*K*]*B*

  The result is a cyclic rotation of *B* by the number of components determined by *A*. If *A* is positive, rotation is to the left; if *A* is negative, rotation is to the right. Rotation is performed along the *K*'th coordinate of *B*. If *K* is omitted, the last coordinate is assumed. (If ⊖ is used instead of ⌽, the first coordinate is assumed).

  Arguments *A* and *B* are conformable for rotation if:

  1. *A* is a scalar or one element vector.

  2. The rank of *A* is one less than the rank of *B* and the shape of *A* is equal to the shape of *B* omitting axis *K*.

  A must be a simple integer array.

  Examples:

```
      3⌽'LEAP'
PLEA
      2⌽3 4⍴⍳12
 3  4 1  2
 7  8 5  6
11 12 9 10
      ¯1⌽3 4⍴⍳12
 4 1  2  3
 8 5  6  7
12 9 10 11
      1⊖ 3 4⍴⍳12
5  6  7  8
9 10 11 12
1  2  3  4
```

⍉ Function (Transpose)

- The monadic Transposition function has the following syntax:

  R←⍉*B*

  The result is an array comprising the items of *B* with the order of all coordinates reversed. For any *B*, (⍴⍉*B*)≡(⌽⍴*B*). If *B* is a matrix, for example, the result is a matrix whose rows are the columns of *B* and whose columns are the rows of *B*. Monadic transpose of a scalar or vector yields *R*←*B*.

  Examples:

```
      ⎕←A←3 5⍴'AGENTVIGORAGONY'
AGENT
VIGOR
AGONY
```

```
      ⍉A
AVA
GIG
EGO
NON
TRY
      ⎕←B←2 3 4⍴(⍳12),100+⍳12
  1   2   3   4
  5   6   7   8
  9  10  11  12

101 102 103 104
105 106 107 108
109 110 111 112
      ⍴⍉B
4 3 2
      ⍉B
 1 101
 5 105
 9 109

 2 102
 6 106
10 110

 3 103
 7 107
11 111

 4 104
 8 108
12 112
```

- The dyadic Transposition function has the following syntax:

  R←A⍉B

  The result is an array similar to B except that the coordinates of B are permuted according to A. The shape of A and B must be related by

  ```
        (⍴A)=⍴⍴B
  ```

  There are two cases of dyadic transposition:

  Case 1:

  A is a permutation of ⍳⍴⍴B (the coordinates of B). A is described as the inverse permutation vector. The A[I]'th component of ⍴R is the I'th component of ⍴B, and thus the A[I]'th coordinate of the result is the I'th coordinate of B.

  Examples:

  ```
        2 1⍉2 3⍴⍳6
  1 4
  2 5
  3 6
        3 2 1⍉2 2 3⍴'EXASPERATION'
  ER
  SI

  XA
  PO

  AT
  EN
  ```

  Case 2:

  A satisfies the relationship (⍳⌈/A)∊A; that is, A is a dense set of the first K coordinates of B, permuted, with some coordinates duplicated. If B is a matrix, the one possible form for A is (1 1), and the result is the principal diagonal of the matrix.

Example:

```
      ⎕←X←3 3⍴'GETEARTRY'
GET
EAR
TRY
      1 1⍉X
GAY
```

If *B* has rank 3 or more, the rule is that the rank of *R* equals the highest value in *A*. If 1<+/*A*[*I*]=*A* and *N*←(*A*[*I*]∊*A*)/⍳⍴*A*, then the *A*[*N*[*I*]]'th coordinate of *R* is made up of those components of *B* whose *N*'th coordinate indices are the same. All other coordinates of the result are structured as in Case 1.

For higher order arrays, the generalized "diagonal" case of dyadic transpose is varied and somewhat difficult to visualize. The examples below show some forms for Case 2:

```
      Z←2 4 4⍴'ABCDEFGHIJKLMNOPQRSTUVWX'
      ⍴⎕←Z
ABCD
EFGH
IJKL
MNOP

QRST
UVWX
ABCD
EFGH
2 4 4
      A←1 1 1⍉Z
      A
AV

      ⍴A
2
      B←1 2 2⍉Z
      B
AFKP
QVCH
      ⍴ B
2 4
      C←2 1 1⍉Z
      C
AQ
FV
KC
PH
      ⍴ C
4 2
      D←2 1 2⍉ Z
      D
AR
EV
IB
MF
      ⍴ D
4 2
      E←1 2 1⍉ Z
      E
AEIM
RVBF
      ⍴E
2 4
      F←2 2 1⍉Z
      F
AU
BV
CW
DX
      ⍴F
4 2
```

```
      G←1 2 2⍉Z
      G
AFKP
QVCH
      ⍴G
2 4
      X←2 3 4 5⍴⍳120
      1 1 1 1⍉X
1 87
      1 1 1 2⍉X
 1  2  3  4  5
86 87 88 89 90
      2 2 2 1⍉X
1 86
2 87
3 88
4 89
5 90
      1 1 2 2⍉X
 1  7 13 19
81 87 93 99
      2 2 1 1⍉X
 1 81
 7 87
13 93
19 99

      1 2 2 3⍉X
  1   2   3   4   5
 26  27  28  29  30
 51  52  53  54  55

 61  62  63  64  65
 86  87  88  89  90
111 112 113 114 115
      3 2 2 1⍉X
 1  61
26  86
51 111

 2  62
27  87
52 112

 3  63
28  88
53 113

 4  64
29  89
54 114

 5  65
30  90
55 115
```

⍋ Function (Grade-up)

o Monadic ⍋ is the Grade-up function.

```
R←⍋B
```

The result is a vector of indices (index origin sensitive) of the first coordinate of *B* ranked in ascending order of magnitude. *B* may be any simple non-scalar array containing only numbers or only character items. Identical components of *B* are ranked in index order.

If *B* is a vector, then the result values are the indices of the individual items of *B*. If *B* is a matrix, the result values are indices of rows of *B*, and the rows are ranked such that a difference in the first column of *B* is more significant than a difference in the remaining columns. This ranking extends to higher ranked arrays by sorting on the first coordinate and treating all other dimensions in ravel order.

Ranking Numeric Arrays

```
      ⍋ 5 10 15 20
1 2 3 4
      ⍋ 3 1 4 1
2 4 1 3
      ⍋ 5 2⍴ 2 8 1 9 2 3 4 2 1 4
5 2 3 1 4
```

Ranking Character Arrays

If B is a character array, then ⍋*B* is treated as *A*⍋*B* where *A* is the default collating sequence shown in Table 5–2. For this default array, difference in case (lower or uppercase) is less significant than a difference in spelling. Also, numeric suffixes sort in numeric order.

Examples:

```
      A←11 3⍴'L10L1 L3 L9 L33L  LX L7 L30LL L6 '
      ⍋A
6 10 7 2 3 11 8 4 1 9 5

      A[⍋A;]
L
LL
LX
L1
L3
L6
L7
L9
L10
L30
L33
```

Table 5-2. Default Collating Sequence Array

```
(10 2 27↑' ',1 2 26⍴⎕AV[65 97∘.+⍳26]),'0123456789',[1.5]' '
ABCDEFGHIJKLMNOPQRSTUVWXYZ0
A̲B̲C̲D̲E̲F̲G̲H̲I̲J̲K̲L̲M̲N̲O̲P̲Q̲R̲S̲T̲U̲V̲W̲X̲Y̲Z̲

                           1


                           2


                           3


                           4


                           5


                           6


                           7


                           8


                           9
```

- Dyadic ⍋ is the Grade-up function.

  R←*A*⍋*B*

  The result is a vector of indices (⎕*IO* sensitive) of the first coordinate of *B* ranked in ascending order of magnitude using the collating sequence specified by the array *A*. *A* and *B* must be simple non-scalar arrays containing only character items.

  The left argument collating sequence array is arranged such that the indices of the first occurrence of each character determines the significance and order for the ranking operation. When two characters differ in their indices along the columnar axis (the last dimension), this difference is more significant than a difference in indices along the row axis or plane axis.

  For example, to sort an array containing letters and underscored letters, a matrix might be used. In this case, if the first row of the matrix contained the letters and the second row contained the underscored letters, the sort would rank a difference in spelling (letters) higher than a difference in case. The result would cause all similarly spelled words to sort together regardless of their case.

  Any characters occurring in *B* but not in *A* are treated as though their index position in *A* is beyond the end of each axis of *A*.

  Examples:

```
      ⎕←A←'A̲B̲C̲D̲E̲F̲G̲H̲I̲J̲K̲L̲M̲N̲O̲P̲',[.5]'ABCDEFGHIJKLMNOP'
A̲B̲C̲D̲E̲F̲G̲H̲I̲J̲K̲L̲M̲N̲O̲P̲
ABCDEFGHIJKLMNOP
      ⎕←B←5 3⍴'AM̲A̲AMAP̲I̲ A̲M̲A̲A̲M̲M̲'
AM̲A̲
AMA
P̲I̲
A̲M̲A̲
A̲M̲M̲
```

```
      A⍋ B
4 1 2 5 3
      'A̲B̲C̲D̲E̲F̲G̲H̲I̲J̲K̲L̲M̲N̲O̲P̲ABCDEFGHIJKLMNOP'⍋ B
4 5 3 1 2
      'A̲AB̲BC̲CD̲DE̲EF̲FG̲GH̲HI̲IJ̲JK̲KL̲LM̲MN̲NO̲OP̲P'⍋ B
4 5 1 2 3
```

⍒ Function (Grade-Down)

o Monadic ⍒ is the Grade-down function.

R←⍒B

The result is a vector of indices (⎕*IO* sensitive) of the first coordinate of *B* ranked in descending order of magnitude. *B* may be any simple non-scalar array containing only numbers or only character items. Identical components of *B* are ranked in index order.

If *B* is a vector, the result values are the indices of the individual items of *B*. If *B* is a matrix, the result values are the indices of rows of *B* and the rows are ranked such that a difference in the first column of *B* is more significant than the remaining columns. This ranking extends to higher ranked arrays by sorting on the first coordinate and treating all other dimensions in ravel order.

Examples:

```
      ⍒ 3 1 4 1 5 9
6 5 3 1 2 4
      ⍒ 7 2⍴ 3 1 1 3 2 7 3 4 6 1 5 7 1 4
5 6 4 1 3 7 2
```

Ranking Character Arrays

If B is a character array, ⍒*B* is treated as *A*⍒*B* where *A* is the default collating sequence shown in Table 5–2. For this default array difference in case (letter or underscored letters) is less significant than a difference in spelling. Also, numeric suffixes sort in numeric order.

Examples:

```
      A←12 3⍴'NFDNS NB PEIQUEONTMANSASALBBC NWTYUK'
      A[⎕←⍒A;]
12 8 5 4 6 11 2 1 3 7 10 9
YUK
SAS
QUE
PEI
ONT
NWT
NS
NFD
NB
MAN
BC
ALB
```

o Dyadic ⍒ is the Grade–down function.

R←A⍒B

The result is a vector of the indices of the first coordinate of *B* arranged such that *B* is ranked in descending order of magnitude using the collating sequence specified by the array *A*. *A* and *B* must be simple non–scalar arrays containing only character items.

The left argument collating sequence is arranged such that the indices of the first occurrence of each character determines the significant and order of the ranking. When two characters differ in their indices along the columnar axis (the last dimension), this difference is more significant than a difference in indices along the row axis or plane axis.

Any characters occurring in *B* but not in *A* are treated as though their index position in *A* is beyond the end of each axis of *A*.

Examples:

```
      A0←'ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ'
      A1←'AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ'
      A2←2 26⍴A0


      B←7 6⍴'TOSOONTRUDGEPHOTO UNDER TOSOONTRUDGETEK   '

      A0⍋B
6 1 7 3 4 2 5
      A1⍋B
4 6 1 7 2 5 3
      A2⍋B
4 6 2 1 5 7 3

      B[A0⍋B;],' ',B[A1⍋B;],' ',B[A2⍋B;]
TRUDGE UNDER  UNDER
TOSOON TRUDGE TRUDGE
TEK    TOSOON TRUDGE
PHOTO  TEK    TOSOON
UNDER  TRUDGE TOSOON
TRUDGE TOSOON TEK
TOSOON PHOTO  PHOTO
```

⊥ Function (Base Value, Decode)

o Dyadic ⊥ is the Base Value function.

R←A⊥B

The argument *A* is referred to as the radix or radix vector. If *A* is a scalar, it is conceptually expanded to a vector. *A* and *B* must be simple and numeric; *R* is numeric.

The argument *A* is used internally to generate a set of weights, *W*, to operate on *B* as follows. Let *I* be the length of *B*. Then:

```
W[I]←1
W[I-1]←A[I]×W[I]
W[I-2][A[I-1]×W[I-1]
W[1]←A[2]×W[2]
```

Note that *A*[1] has no effect on the result.

Example:

```
         A←0  60  60⊥1  2  3
W[3] is 1
W[2] is W[3]×A[3], or 60
W[1] is W[2]×A[2], or 3600
```

The result is formed by *W*+.×*W*:

```
        1×3600   2×60   3×1
        ------   ----   ---

W×B is 3600      120    3
```

R is 3723

If *A* is a vector and *B* is an array, ρ*A* must be the same as the length of the first coordinate of *B*. If *B* is a matrix for example, *B* must have the same number of rows as the length of *A*. Each column of *B* is decoded to provide one item of the result. If *A* is also an array, each row of *A* represents a different radix vector. The shape of *R* is the catenation of the shape of all but the last coordinate of *A* with all but the first coordinate of *B*. (Structure rules for *A*, *B*, and *R* are the same as for inner product.)

Examples:

```
      2⊥1 0 1 1
11
      4⊥3 2 1 0
228
      10⊥9 8 7
987
      1 2 3⊥ 45 67 89
560

      ⍝ K IS A TABLE OF TIMES REPRESENTED IN DAYS (ROW 1),
      ⍝ HOURS (ROW 2), MINUTES (ROW 3), AND SECONDS.

      +K←4 6ρ0 0 0 0 1 11 0 0 0 2 3 13 0 1 16 46 46 46 10,5ρ40
 0  0  0  0  1 11
 0  0  0  2  3 13
 0  1 16 46 46 46
10 40 40 40 40 40

      ⍝ EACH COLUMN OF K REPRESENTS A  TIME VALUE.
      ⍝ IF K IS OPERATED ON BY THE 'BASE VALUE' FUNCTION,
      ⍝ THE RESULT IS A VECTOR OF TIMES IN SECONDS.
      ⍝ THE RADIX VECTOR IS -- 365 24 60 60

      365 24 60 60⊥K
10 100 1000 10000 100000 1000000
```

⊤ Function (Representation, Encode)

o Dyadic ⊤ is the Encode function.

R←*A*⊤*B*

R is a "base *A*" representation of *B*. *R* satisfies the relationship ((×/*A*)|*B*-*A*⊥*R*)=0. *A* and *B* must be simple and numeric, *R* is numeric. Note that the ⊤ and ⊥ functions are inverses (opposites). Note also that since Encode carries out a residue operation, its values are subject to the rules for that function.

If vector *A* contains too few items for *B* to be represented, the most significant digits of the result are truncated. If *A*[1] is 0, any unencodeable portion of *B* will be returned as *R*[1] rather than being truncated. Note that *A* and *B* may be negative or non-integer values. In this case, the result is as well defined but not as intuitively clear as for positive integer values.

B may be an array rather than a scalar, and the shape of the result will be the catenation of the shapes of the arguments. (The structure rules for *R*, *A*, and *B* are the same as for outer product.)

Examples:

```
        ⍝ BINARY REPRESENTATION

        (8⍴2)⊤75
0 1 0 0 1 0 1 1


        ⍝ OCTAL REPRESENTATION

        (3⍴8)⊤75
1 1 3

        ⍝ DECIMAL REPRESENTATION

        (5⍴10)⊤31415
3 1 4 1 5

        ⍝ VARIED UNIT REPRESENTATION
        24 60 60⊤75432
20 57 12

        ⍝ EXAMPLE OF TRUNCATION
        10 10⊤31415
1 5

        ⍝ THE ARGUMENTS FOR REPRESENTATION NEED NOT BE INTEGER
        (8⍴1.5)⊤32.75
1 0.5 1 0 0 0.5 0 1.25

        ⍝ H IS A VECTOR OF TIME VALUES IN SECONDS
        H←10 100 1000 10000 100000 1000000
        ⍝ H CAN BE ENCODED IN TERMS OF DAYS,HOURS,MINUTES AND SECONDS.
        365 24 60 60⊤H
 0  0  0  0  1 11
 0  0  0  2  3 13
 0  1 16 46 46 46
10 40 40 40 40 40


        ⍝ IN THE RESULT, EACH COLUMN REPRESENTS ONE ELEMENT OF H
        ⍝ ROW 1 IS DAYS, ROW 2 IS HOURS, ROW 3 IS MINUTES AND
        ⍝ ROW 4 IS SECONDS.
```

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

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

⍕ Function (Format)

- Monadic ⍕ is the Format function.

  R←⍕ *B*

  The symbol ⍕ (⊤ and ∘ overstruck) defines two format functions which convert numerical arrays to character arrays. The monadic function produces a character array which is identical to the array which would be produced if the argument were merely printed; the difference (and advantage) is that the result is made explicitly available. The monadic format function can also be applied to a character array and will return the same character array. When applied to numeric arrays, however, the shape of the result is the same as the shape of the argument except for the required expansion along the last coordinate, each number being expanded, in general, to several characters. When applied to a nested array, the result is a vector or a matrix.

  Examples:

```
      PTABLE←2=?4 4⍴2
      PTABLE
1 1 0 1
0 0 1 0
1 0 0 0
0 0 0 1
      ⍴PTABLE
4 4
      ⍴⎕←DFORMAT←⍕PTABLE
1 1 0 1
0 0 1 0
1 0 0 0
0 0 0 1
4 7
      ⍕'LITERAL'
LITERAL
```

- Dyadic ⍕ is the Format function.

  R←*A* ⍕ *B*

  The dyadic format function accepts only simple numeric arrays for its right argument, and uses the left argument to provide detailed control over the result. In general, a pair (or pairs) of numbers in the left argument controls one or more columns of the result. The first number of the pair determines the total width of a number field and the second number specifies the desired precision. For decimal form numbers, precision is defined as the number of digits to the right of the decimal point; for scaled form it is defined as the number of digits in the multiplier. The form to be used is defined by the sign of the precision number in the control pair. Negative numbers indicate scaled form. For example:

```
      ⍴⎕←DMATRIX←3 2⍴12.34 ¯34.567 0 12  ¯.26 ¯123.45
12.34  ¯34.567
 0       12
¯0.26 ¯123.45
3 2

      ⍴⎕←12 3⍕DMATRIX
      12.340     ¯34.567
       0.000      12.000
      ¯0.260    ¯123.450
3 24

      ⍴⎕←SCALED←9 ¯2⍕DMATRIX
 1.2E1   ¯3.5E1
 0.0E0    1.2E1
¯2.6E¯1  ¯1.2E2
3 18
```

  A single control number may also be used, and is treated as a number pair with a width indicator of zero. In this event, a field width is chosen such that at least one space will appear between adjacent numbers. For example:

```
      ⍴⎕←2⍕DMATRIX
 12.34  ¯34.57
  0.00   12.00
 ¯0.26 ¯123.45
3 14

      ⍴⎕←¯2⍕DMATRIX
  1.2E1    ¯3.5E1
  0.0E0     1.2E1
 ¯2.6E¯1   ¯1.2E2
3 20
```

Each column of an array can be individually formatted by defining a left argument containing a control pair for each column of the array. For example:

```
      ⍴⎕←0 2 0 2⍕DMATRIX
 12.34  ¯34.57
  0.00   12.00
 ¯0.26 ¯123.45
3 14

      ⍴⎕←6 2 12 ¯3⍕DMATRIX
 12.34  ¯3.46E1
  0.00   1.20E1
 ¯0.26  ¯1.23E2
3 18
```

When applied to an array having a rank greater than two, the format specifications apply to each of the planes defined by the last two coordinates. For example:

```
      MATRIX3D←2=?2 2 5⍴2
      MATRIX3D
1 0 1 1 1
0 0 1 1 0

0 1 0 1 1
0 0 0 0 0

      4 1⍕MATRIX3D
 1.0 0.0 1.0 1.0 1.0
 0.0 0.0 1.0 1.0 0.0

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

Tabular displays which incorporate row and column headings or other information between columns or rows, can be configured using the format function together with extended catenation. For example:

```
      ROWHEADS←4 3⍴'JANAPRJULOCT'
      YEARS←78+⍳5
      TABLE←.001×¯4E5+?4 5⍴8E5

      (' ',[1]ROWHEADS),(2⌽9 0⍕YEARS),[1]9 2⍕TABLE
       79        80        81        82        83
JAN ¯159.97   153.85   269.01   208.60   ¯88.20
APR    8.89  ¯322.64   293.61   297.76   213.28
JUL  254.56    73.44   255.15  ¯134.65   305.28
OCT   52.33     1.25    ¯6.41  ¯234.24  ¯314.15
```

A *DOMAIN ERR* results when the width indicator of the control pair does not specify a size large enough to hold the requested form. The width need not, however, provide for blanks between adjacent numbers.

↑ Function (First, Take)

- Monadic ↑ is the First function.

  R←↑B

  B may be any APL array. The result is the APL array which is within the first item of *B*. If *B* is empty, the result is the prototype value of the array *B*.

  Examples:

```
      ↑10 20 30
10
      ↑'HIYA'
H
      ↑'SASKATOON'  'MOOSE JAW'
SASKATOON
      ↑↑'SASKATOON'  'MOOSE JAW'
S
      ↑↑↑'SASKATOON'  'MOOSE JAW'
S
      ↑0⍴(1 2) 3 (4 5)
0 0
```

- Dyadic ↑ is the Take function.

  R←A↑B

  A must be an integer scalar or vector, and the length of *A* must equal the rank of *B*. (If *B* is a scalar it is treated as though it were a 1 item array whose rank is the length of *A*.) Each item of *A* controls the "take" from a coordinate of *B*. *R* has the same rank as *B*. The shape of *R* is |*A*.

  If *A[I]*≥0, then the *I* th coordinate of *R* is the first *A[I]* items in the *I* th coordinate of *B*. If *A[I]*<0, the last |*A[I]* items are used. If |*A[I]* indicates more items than are present in the coordinate of *B*, *R* is padded with prototype values of *B*.

  Examples:

```
      ¯3↑⍳5
3 4 5
      7↑⍳5
1 2 3 4 5 0 0
      3↑(1 2 3) (4 5)
1 2 3  4 5  0 0 0
      ' '=10↑'OLYMPICS'
0 0 0 0 0 0 0 0 1 1
      +B←2 2 2⍴⍳8
1 2
3 4

5 6
7 8
      1 2 3↑B
1 2 0
3 4 0
```

↓ Function (Drop)

- Dyadic ↓ is the Drop function.

  R←A↓B

  A must be integer scalar or vector, and the length of *A* must equal the rank of *B*. (If *B* is a scalar it is treated as through it were a 1 item array whose rank is the length of *A*.) Each item of *A* controls the "drop" from a coordinate of *B*. *R* has the same rank as *B*. The shape of *R* is 0⌊(ρ*B*)-|*A*. If a dimension in the result thus created would be negative it is set to zero.

  If *A*[*I*]≥0, then the *I*'th coordinate of *R* is all but the first *A*[*I*] items of the *I*'th coordinate of *B*; that is, the first *A*[*I*] items are dropped. If *A*[*I*]<0, the last |*A*[*I*] items of the *I*'th coordinate of *B* are dropped.

  Examples:

```
      ¯3↓⍳5
1 2
      3↓⍳5
4 5
      +B←2 2 2ρ⍳8
1 2
3 4

5 6
7 8
      1 2 2↓B      ⍝ NOTE: RESULT IS AN EMPTY ARRAY
      2 2 1↓B
      1 1 1↓B
8
```

∊ Function (Type, Membership)

- Monadic ∊ is the Type function.

  R←∊B

  The result is an array with the same structure (shape and depth) as *B* with all numbers replaced by zero and all characters replaced by blanks.

  Examples:

```
      ∊1 2 3 4
0 0 0 0
      ' '=∊'CHARACTER'
1 1 1 1 1 1 1 1 1
      ∊1  (2 3)  (4  (5 6)  7)
0  0 0  0  0 0  0
```

- Dyadic ∊ is the Membership function.

  R←A∊B

  If an item of *A* is contained in *B*, the corresponding item of *R* is equal to 1; otherwise, it is 0. The result has the same shape as *A* and is in the logical domain. *B* may have any rank. If *A* and *B* are numeric, ⎕*CT* is used in the equality test.

Examples:

```
      A←'ALPHABET'
      B←'ABCDE'
      C←2 4⍴⍳8
      A∊B
1 0 0 0 1 1 1 0
      1 5 10∊C
1 1 0
      'TWO'  'TEN'∊'ONE'  'TWO'  'THREE'  'FOUR'
1 0

      ⍝   NOTE THAT MEMBERSHIP MAY BE USED WITH NUMERIC VERSUS
      ⍝   TEXT ARGUMENTS, BUT THE RESULT IS ALWAYS ZERO

      A∊C
0 0 0 0 0 0 0 0
      C∊A
0 0 0 0
0 0 0 0
      1 2 3∊'1 2 3'
0 0 0
```

⍎ Function (Execute)

o Monadic ⍎ is the Execute function.

```
R←⍎B
```

B must be a simple scalar or vector. The domain of *B* must be character unless *B* is an empty vector. Ordinarily, the argument *B* will be a small character vector. If *B* contains unbalanced quotes, the error *OPEN QUOTE* is reported.

Once the argument has met the above requirements, the execute function departs from the mold of the other functions. That is, the characters in its argument, if any, are treated as if they were an APL statement to execute.

It is even possible in CP-6 APL to execute system commands. Execute operations can be applied so that an application can create its own variable names, or compose new formulas and evaluate them.

The execute function is a powerful tool. It can, however, be costly in execution time. The cost stems from the translation process when accepting its argument as if freshly input. This translation is repeated each time the same execute operation is performed; a function line, on the other hand, is translated only once regardless of the number of times it is invoked. Thus, 'execute' should be used sparingly in interactive or recursive processes.

As stated previously, the execute function permits formula evaluation, or system command execution in the midst of any APL statement. As with evaluated input, the result of executing an expression is the value resulting from evaluating that expression. The following examples illustrate this:

```
      ⍎'2+2'
4
      ⍎'''AB'''
AB
      3+⍎'2+2'
7
      X←'2+'
      Y←'2'
      3+⍎X,Y
7
```

Executing an empty vector yields an empty (numeric) vector result.

```
      0\⍎'⍳0'
0
      0\⍎''
0
```

There are three important differences between execute in CP-6 APL and execute in most other APL's. These are:

1. System commands may be the object of execute statements in CP-6 APL.

2. Function editing is possible using the execute functions in CP-6 APL.

3. Executing an empty or all blank vector results in an empty numeric vector.

Executing some system commands yields no result. For example: *)OFF*, *)OFF HOLD*, *)CLEAR*, and *)LOAD* yield no display. In CP-6 APL, the "execute" of a system command which produces a display is returned as a character vector. This character vector is directly usable by the program.

The argument to the execute function may contain a number of expressions separated by diamonds. The result of such an argument is the result of the last expression evaluated. For example:

```
      4+⍎'1 ◊ 2 ◊ 3'
1
2
7
```

prints the values 1 and 2, returns 3 as the result of execute, which is added to 4 to print 7. (The diamond separator is described in Section 6 under Compound Statements.)

The execute function can also be used to access function definition mode, but limitations are imposed. A basic limitation exists since only one "statement" (character vector) can be the argument of an execute function.

The result of executing function definition mode operation is an empty vector unless a function display was requested, in which case the text of the display is returned as the result.

When using the execute function, the argument cannot contain unbalanced quotes (the error message *OPEN QUOTE* is issued in such cases).

Error handling is unique in the case of the execute function. After the diagnostic message (such as *DOMAIN ERR*), the path leading to the error is displayed until a normal suspension point is reached. The following example illustrates error handling during an involved execute function.

```
      ∇Z←Y F X
[1]   A←'Y+'
[2]   B←'X'
[3]   C←'⍎A,B'
[4]   Z←100+⍎C
    ∇

      5 F 4
109
      5 F 'FOUR'
DOMAIN ERR
      Y+X
       ^
      ⍎A,B
      ^
F[4]  Z←100+⍎C
            ^
```

⌹ Function (Matrix Inverse, Matrix Divide)

- ⌹ is the Matrix Inverse function.

This function is used to solve systems of linear equations and to invert matrices. The monadic form is equivalent to the dyadic form with an identity matrix as a left argument, and the function can best be explained in terms of the dyadic form. The right argument must be matrix with at least as many rows as columns; that is, 1=(≤/⍴*B*). The first coordinate of the left and right arguments must have the same length; that is, (1↑⍴*A*) = 1↑⍴*B*. A vector argument is treated as though it were a one-column matrix; and a scalar is treated as though it were a one-by-one matrix, in terms of shape requirements. The shape of the result is (⍴*R*) = (1↓⍴*B*),(1↓⍴*A*). For inversion, the shape of the result is (⍴*R*)=(⌽⍴*B*). *R*←*A*⌹*B* produces *R* such that the expression +/(,*A*-*B*+.×*R*)*2 is minimized; that is, *R* indicates the least-squares solution (or solutions) to a system (or systems) of linear equations.

If *B* is a non-singular square matrix, then the minimum is (except for computational round-off errors) zero, and *R* is the solution of a set of simultaneous equations. If, in addition, *A* is an identity matrix, *R* is the inverse of *B* (that is equivalent to *R*←⌹*B*). If *A* is a vector, *R* is the solution to one system of simultaneous equations. If *A* is a matrix, each column of *A* represents the constants for a linear system with coefficient matrix *B*, and each column of *R* is the corresponding solution.

If *B* is non-square, then the minimum of +/(,*A*-*B*+.×*R*)*2 is not generally zero, and *R* represent a solution in the least-squares sense.

If *B* is singular (has fewer linearly independent rows than columns), then a *SING MATRIX* error is reported.

If *B* is non-square and *A* is an identity matrix, the result is the left inverse of *A* and the function is equivalent to *R*←⌹*B*.

Examples:

```
      ⍝   INVERSE OF A SQUARE MATRIX

      ⎕←B←3 3⍴3 1 4 1 5 9 2 6 5
3 1 4
1 5 9
2 6 5
      ⎕←R←⌹B
 0.3222222222  ¯0.2111111111   0.1222222222
¯0.1444444444  ¯0.07777777778  0.2555555556
 0.04444444444  0.1777777778  ¯0.1555555556

      ⎕PP←5

      ⍝   VERIFY THAT THE INNER PRODUCT OF R AND B IS
      ⍝   ESSENTIALLY THE IDENTITY MATRIX.

      R+.×B
 1.0000E0    ¯5.4210E¯19 ¯8.6736E¯19
¯4.3368E¯19  1.0000E0    ¯3.2526E¯18
¯2.7105E¯20 ¯8.1315E¯20   1.0000E0

      ⍝   LEFT INVERSE OF A NONSQUARE MATRIX

      ⎕←B←5 3⍴3 1 4 1 5 8 2 7 4 3 5 9 8 7 6
3 1 4
1 5 8
2 7 4
3 5 9
8 7 6
      ⎕←R←⌹B
 0.074106 ¯0.082157 ¯0.072245 ¯0.015323  0.13129
¯0.10492   0.011612  0.17084  ¯0.048546  0.013386
 0.061261  0.06862  ¯0.073814  0.085902 ¯0.04531
```

```
      ⍝    AGAIN, VERIFY THAT THE INNER PRODUCT OF R AND B IS
      ⍝    VERY CLOSE TO THE IDENTITY MATRIX

      R+.×B
 1.0000E0   1.0842E¯19 ¯1.1926E¯18
¯1.5179E¯18 1.0000E0   ¯1.4095E¯18
 1.9516E¯18 8.6736E¯19  1.0000E0

      ⍝    SOLUTION OF A SINGLE LINEAR SYSTEM
      ⍝          B IS THE COEFFICIENT MATRIX
      ⍝          A IS THE VECTOR OF CONSTANTS

      ⎕←B←3 3⍴3 1 4 1 5 9 2 6 5
3 1 4
1 5 9
2 6 5
      ⎕←A←35 89 79
35 89 79

      ⎕←R←A⌹B
2.1444 8.2111 5.0889

      ⍝    VERIFY THAT B+.×R APPROXIMATELY EQUALS A

      A-B+.×R
5.5511E¯17 1.1102E¯16 8.3267E¯17

      ⍝    SOLUTION OF A SET OF LINEAR SYSTEMS
      ⍝       B IS A COEFFICIENT MATRIX
      ⍝       A IS A MATRIX; EACH COLUMN IS A SET
      ⍝                      OF CONSTANTS FOR B.
      ⍝       EACH COLUMN OF R, WHICH IS A MATRIX, IS THE
      ⍝       SOLUTION FOR THE CORRESPONDING COLUMN OF A.

      ⎕←A←3 2⍴35 36 89 88 79 75
35 36
89 88
79 75

      R←A⌹B
      R
2.1444 2.1889
8.2111 7.1222
5.0889 5.5778

      ⍝    CHECKING...

      A-B+.×R
5.5511E¯17 2.7756E¯17
1.1102E¯16 8.3267E¯17
8.3267E¯17 5.5511E¯17


      ⍝    LEAST-SQUARES SOLUTION

      ⎕←B←6 2⍴1 1 1 2 1 3 1 4 1 5 1 6
1 1
1 2
1 3
1 4
1 5
1 6

      A←12.03 8.78 6.01 3.75 ¯0.31 ¯2.79
      A
12.03 8.78 6.01 3.75 ¯0.31 ¯2.79
      R←A⌹B
      R
14.941 ¯2.9609

      ⍝    THE RESULT GIVES THE INTERCEPT AND SLOPE OF THE INE
      ⍝   THAT IS THE LEAST-SQUARES BEST FIT TO THE POINTS OF A.

      B+.×R
11.98 9.0196 6.0588 3.0979 0.13705 ¯2.8238
      A-B+.×R
0.049524 ¯0.23962 ¯0.048762 0.6521 ¯0.44705 0.03381
```

To find the values of *X*, *Y*, and *Z* in the following linear equations:

```
4X + 2Y - 5Z = 22
5X - 4Y + 4Z = ¯7
2X + 2Y - 20Z = 80
```

assign the values of the coefficients to *A* and the constant vector to *B*, as in :

```
      A
4  2  ¯5
5 ¯4   4
2  2 ¯20

      B
22 ¯7 80
```

and then obtain the solution:

```
      B⌹A
1 ¯1 4
```

Thus in the linear equations provided above, *X* has the value 1, *Y* has the value ¯1 and *Z* has the value ¯4.

Operators

The five operators in CP-6 APL extend functions to arrays. In the following descriptions of these operations, the bracketed value *K* represents that coordinate of the argument array along which the specified operator is to act. If *K* is unspecified, the last coordinate of the array is assumed. The symbols d, f, and g represent any dyadic function, including a primitive function, a system function, a defined function, or a derived function.

Reduction d/ Operator

o Monadic d/ is the Reduction operator.

  R←d/[*K*]*B*

  The result is an array having dimensions equal to that of array *B* except that the *K*'th component is not present. If ⌿ is used instead of /, the first coordinate axis is used.

  For a vector argument, the value of the result is that produced by placing the function d between each pair of adjacent components of vector *B*. A minus reduction results in an alternating sum and a divide reduction results in an alternating product.

  For a scalar or an array comprising a single component along the reduction coordinate, the result has the same value as *B*. For an empty array the result has the value of the identity item of function d as shown in the table below or a *DOMAIN ERR* if no identity exists.

Table 5-3. Identity Values for Scalar Functions

d	Identity Item	Comment
×	1	
+	0	
÷	1	Right identity only.
-	0	Right identity only.
*	1	Right identity only.
|	0	Left identity only.
⍟	None	
○	None	
∨	0	
∧	1	
⍲	None	
⍱	None	
!	1	Left identity only.
⌊	8.379879956E152	
⌈	¯8.379879956E152	
>	0	Right identity only.
≥	1	Right identity only.
<	0	Left identity only.
≤	1	Left identity only.
=	1	
≠	0	

Domain restrictions for function d apply. If the function argument d is not a scalar function, then the result is a possibly nested array. If *B* has more than one item, the domain of the result is the same as indicated in the domain tables for the dyadic scalar functions, or a nested array for all other functions.

Examples:

```
      ⎕←+/2 4 6 8
 20
      ⎕←-/2 4 6 8
 ¯4
      ⎕←!/10
 10
      ≡/'APPLE'  'APPLE'
 1
      ≡/'APPLE'  'PEAR'
 0

      ⎕←A←2 4⍴⍳8
 1 2 3 4
 5 6 7 8
      +/A
 10 26
      +⌿A
 6 8 10 12
      +/+/A
 36


      ⎕PS←0 0 ¯3 ¯3
      ⍴⎕←,/4 5⍴'ONE  TWO  THREEFOUR '
 +-----+ +-----+ +-----+ +-----+
 |ONE  | |TWO  | |THREE| |FOUR |
 +-----+ +-----+ +-----+ +-----+
 4
      ,/ 1 (2 3 4) 5 (6 7 8)
 +---------------+
 |1 2 3 4 5 6 7 8|
 +---------------+
      ρ/2 3
+---+
|3 3|
+---+
      ρ⎕←ρ/2 3ρ4
+-------+ +-------+
|4 4 4 4| |4 4 4 4|
+-------+ +-------+
2
      ⎕PS←¯1 1 0 2

      B←2 3 4ρι24
      B
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24
      +/B
10 26 42
58 74 90
      +/[2]B
15 18 21 24
51 54 57 60
      +/+/B
78 222
      +/+/+/B
300

      C←3 4ρ1 1 1 0 1 1 0 0 1 0 0 0
      C
1 1 1 0
1 1 0 0
1 0 0 0
      ∧/C
0 0 0
      ∧⌿C
1 0 0 0
```

Compression A/ Operator (Replicate)

- Monadic *A/* is the Compression operator.

  R←A/[K]B

  The result includes all items in *B* that correspond to a 1 in *A*. Those corresponding to a 0 are suppressed. If either argument is scalar, it is applied to all items of the other argument.

  Compression is performed along the *K*'th coordinate of *B*. If *K* is omitted, the last coordinate is assumed. (If ⌿ is used instead of /, the first coordinate is assumed.)

  A may be a simple logical scalar or vector, and *B* may be of any rank or domain. If *A* consists of more than one item, its length must be the same as that of the coordinate of *B* being compressed.

  Examples:

```
      B←2 2ρι4
      1 0/B
1
3
```

The compression operator may be used in a test and branch situation. In this case, when the left argument has a value 1, a branch is made to the statement indicated by the right argument. If the left argument has the value 0, a branch is not taken and execution proceeds with the next statement. For example, the statement:

```
      →(2>3)/END ⋄ 'NO BRANCH'
NO BRANCH
```

falls through to the next statement; whereas

```
      →(3>2)/END ⋄ 'BRANCH'
```

causes a branch to the statement labeled *END*.

- The Replicate operator.

  ```
  R←A/[K]B
  ```

  Like compression, the result includes *A*[*I*] copies of each *B*[*I*]. That is, if *A*[*I*]=0 then the corresponding item of *B* is suppressed, if *A*[*I*]=2 then the corresponding item of *B* appears twice and so on.

  Replication follows all the rules of compression except that *A* may be an integer scalar or vector of non-negative values.

  Examples:

  ```
        B←2 2⍴⍳4
        2 3/B
  1 1 2 2 2
  3 3 4 4 4
  ```

Scan d\ Operator

- Monadic d\ is the Scan operator.

  ```
  R←d\[K]B
  ```

  The result has the same shape as that of *B*. If ⍀ is used instead of \, the first coordinate axis is used.

  For a vector argument, the result is a vector of the same length with values as follows:

  ```
  R[1]←B[1]
  R[2]←B[1] d B[2]
  R[3]←B[1] d B[2] d B[3]
    .
    .
    .
  ```

  Thus the last component of the result is equal to d/*B*.

  For a scalar or a one-component array, the result is the same as *B*. For an empty argument, the result will be empty.

  Domain restrictions for function d apply. If *B* has more than one item, the result domain is that indicated in the domain table for d if d is a scalar function; otherwise the result is a nested array.

  Examples:

  ```
        +\1 3 5 7
  1 4 9 16
        +\¯5 0 7 0 1
  ¯5 ¯5 2 2 3
        -\3 9 5 1
  3 ¯6 ¯1 ¯2
        ×\1 2 3 4 5
  ```

```
1 2 6 24 120
      ≤\7 9 5 ¯4
7 1 0 0
      ,\'ABCD'
A  AB  ABC  ABCD
      =\'AA'
A 1
```

Scan generalizes to higher ranked arguments in the same way reduction does, by doing the operation along the *K*'th coordinate as shown by the example below:

```
      B←2 3 4⍴⍳24
      +⍀B
 1  2  3  4
 5  6  7  8
 9 10 11 12

14 16 18 20
22 24 26 28
30 32 34 36
      +\[2]B
 1  2  3  4
 6  8 10 12
15 18 21 24

13 14 15 16
30 32 34 36
51 54 57 60
      +\B
 1  3  6 10
 5 11 18 26
 9 19 30 42

13 27 42 58
17 35 54 74
21 43 66 90
```

Expansion A\ Operator

- Monadic *A*\ is the Expansion operator.

  R←*A*\[*K*]*B*

  A must be a vector of 1's and 0's and must include the same number of 1's as the length of the coordinate to be expanded. *B* may be of any rank and domain. Expansion occurs along the *K*'th coordinate of *B*. If *K* is omitted, the last coordinate is assumed. If ⍀ is used instead of \, the first coordinate is assumed. Thus, the difference between \ and ⍀ is

  R←*A**B*   expands along the last coordinate of *B*.

  R←*A*⍀*B*   expands along the first coordinate of *B*.

  Expansion consists of extending the length of the affected coordinate of *B* by insertion of prototype values in positions indicated by zeros in the argument *A*. The process is best described by example. The prototype for a simple numeric array is 0. The prototype of a simple character array is ' '. In general, the prototype of an array *B* is (⊂∊⊃*B*).

```
      1 0 1 0 1\⍳3
1 0 2 0 3
      1 0 1\(1 2) (3 4 5)
1 2  0 0  3 4 5

      ⍝   THE FOLLOWING EXAMPLES SHOW EXPANSION ON EACH OF THE
      ⍝   COORDINATES OF A RANK 3 ARRAY.
```

```
          B←2 2 2⍴⍳8
          1 0 1⍀B
1 2
3 4

0 0
0 0

5 6
7 8
          1 0 1\[2]B
1 2
0 0
3 4

5 6
0 0
7 8
          1 0 1\B
1 0 2
3 0 4

5 0 6
7 0 8

          A←2 2 2⍴'ABCDEFGH'
          1 0 1⍀A
AB
CD




EF
GH
          1 0 1\[2]A
AB

CD

EF

GH
          1 0 1\[3]A
A B
C D

E F
G H
```

Inner Product f.g Operator

o Dyadic f.g is the Inner Product operator.

R←*A* f.g *B*

The result is an array having shape equal to all except the last dimension of array *A* catenated with all except the first dimension of array *B*. If the function g is a scalar function, the length of the last dimension of *A* must be the same as that of the first dimension of *B*, or one of those lengths must be 1. The domain of the result is indicated by the functions f and g. Functions f and g may be any dyadic functions. For example, *R*←*A*+.×*B* gives the conventional matrix inner product.

For vector arguments, the result is:

f/*A* g *B*

Examples:

```
      3 4+.×5 6
39
      +/3 4×5 6
39
      1 2 3+.×4 5 6
32
      +/1 2 3×4 5 6
32
      1 0 1 0+.∧1 1 0 0
1
      1 0 1 0+.∨ 1 1 0 0
3
```

If *A* is a vector and *B* is a matrix, the *I*'th component of the result is:

f/*A* g *B*[;*I*]

Example:

```
      A←2 4
      B←2 4ρ3 2 6 8 5 4 9 4
      B
3 2 6 8
5 4 9 4
      A+.×B
26 20 48 32
      B[;1]
3 5
      B[;2]
2 4
      B[;3]
6 9
      B[;4]
8 4
      +/A×3 5
26
      +/A×2 4
20
      +/A×6 9
48
      +/A×B[;4]
32
      1 2 3+.!3 3ρι9
42 68 102
```

If *A* is a matrix and *B* a vector, the *I*'th component of the result is:

f/*A*[*I*;] g *B*

Example:

```
      C←1 2 3 4
      B+.×C
57 56
      B[1;]
3 2 6 8
      B[2;]
5 4 9 4
      +/B[1;]×C
57
      +/B[2;]×C
56
```

For matrix arguments, the *I*;*J*'th component of the result is:

f/*A*[*I*;] g *B*[;*J*]

Example:

```
      (2 4⍴ ⍳ 8)+.×4 2⍴ ⍳ 8
 50  60
114 140
      X←3 3⍴'CANDIDATE'
      Y←3 3⍴'DRAMATIZE'
      X∧.=Y
0 0 0
0 0 0
0 0 1
      X
CAN
DID
ATE
      Y
DRA
MAT
IZE
      X∨.=Y
0 1 0
1 0 0
0 0 1
      X∧.≠Y
1 0 1
0 1 1
1 1 0
```

Inner product also applies to higher order arrays. For the example below, the arguments are each three dimensional and the result has four dimensions. The I;J;K;Lth item of the result is: +/A[I;J;]×B[;K;L].

```
      A←2 2 3⍴⍳12
      A
 1  2  3
 4  5  6

 7  8  9
10 11 12
      ⎕←B←3 2 2⍴12+⍳12
13 14
15 16

17 18
19 20

21 22
23 24
      A+.×B
110 116
122 128

263 278
293 308


416 440
464 488

569 602
635 668
      +/A[1;1;]×B[;1;1]
110
      +/A[1;1;]×B[;1;2]
116
      +/A[1;2;]×B[;2;2]
308
```

Outer Product ∘.d Operator

- Dyadic ∘.d is the Outer Product operator.

  R←*A* ∘.d *B*

  The result is an array having a shape equal to the shape of *A* catenated with the shape of *B*. The dyadic function d is performed for each item of *A* with respect to all items of *B*. The domain of the result is determined by the rules for the function d.

  For vector arguments, the I;Jth component of the result is:

  A[*I*] d *B*[*J*]

  Example:

```
      1 2 3∘.×1 2 3 4
1 2 3  4
2 4 6  8
3 6 9 12
      1 (2 3)∘.×1 (2 3) 4
  1  2 3     4
2 3  4 9  8 12
      'ABC'∘.,'D'  'DEF'  (2 2⍴'DEFG')
AD  ADEF  ADE
          AFG
BD  BDEF  BDE
          BFG
CD  CDEF  CDE
          CFG
```

  Outer product is valid for arguments of higher rank. If, for example, *A* has a rank 3 and *B* has rank 2, the items of the result are defined by:

  R[*I*;*J*;*K*;*L*;*M*] ←→ *A*[*I*;*J*;*K*] d *B*[*L*;*M*]

Each Operator

- Monadic d¨ is the Each operator.

  R←d¨*B*

  The result is an array with the same shape as *B*. Each item of the result contains the result of applying the monadic function d to the corresponding item of *B*. d may be any monadic function including a monadic primitive function, a monadic system function, a monadic defined function, or a monadic derived function.

  Examples:

```
      ⍴¨'CENTURY'  'DECADE'  (1972 1974 1976 1979 1980)
7  6  5

      ⌽¨'NOW'  'POOL'  'ON'
WON  LOOP  NO

      ⍳¨2 3
1 2  1 2 3

      ⍎¨'1+1'  'PW←80'  '4×3+⍳2'
2 80  16 20
```

- Dyadic d¨ is the Each operator.

  R←*A* d¨ *B*

  The result is an array where each item of the result contains the result of applying the dyadic function d to the corresponding items of *A* and *B*.

A *RANK ERR* is reported if both *A* and *B* are not singletons and their ranks differ. A *LENGTH ERR* is reported if both *A* and *B* are not singletons and the lengths of *A* and *B* are not the same. If *A* or *B* is a singleton, it is reshaped to the rank of the higher ranked argument before performing the Each operation.

d may be any dyadic function, including a dyadic primitive function, a dyadic system function, or a dyadic defined function.

Examples:

```
      1,¨10 20 30 40
1 10  1 20  1 30  1 40

      2⌽¨'CENTURY'  'DECADE'  (1972 1974 1976 1979 1980)
NTURYCE  CADEDE  1976 1979 1980 1972 1974
```

Section 6

APL Statements

As mentioned in Section 2, each completed line of input is classified as either a statement or a system command. Statements specify the operations to be performed by APL, such as calculations, branching, and assignment of values or expressions. System commands (treated in Section 8) are concerned with the mechanical aspects of the system, such as logging off and saving, loading, and deleting workspaces. Statements can be entered when the system is in either execution mode or function definition mode. The user indicates the end of a statement by pressing the RETURN key. In execution mode, the computer then executes the operations contained in the statement. In definition mode, the computer stores the statements until the function is invoked. Blanks may appear anywhere in a statement except embedded within a number or a name. In general, an APL statement cannot be continued on another line. A character constant, however, may include one or more carriage returns, thus allowing multi-line statements.

When a character constant is being entered and APL detects a carriage return before receiving the closing quote, it automatically types a closing quote at the beginning of the next line. The assumption is that the user may simply have forgotten the closing quote. If that is not the case, the user may delete the closing quote and continue the text constant.

```
      A←'LONG VECTOR, CLOSING QUOTE FORGOTTEN
'
      A
LONG VECTOR, CLOSING QUOTE FORGOTTEN
      A←LONG VECTOR, CLOSING QUOTE NOT FORGOTTEN,
'
VECTOR CONTINUED ON SECOND LINE'
      A
LONG VECTOR, CLOSING QUOTE NOT FORGOTTEN,
VECTOR CONTINUED ON SECOND LINE
```

In this example, note that APL automatically provides the closing quote in the first specification of *A*. In the second specification, the user cancels APL's action and continues the character constant on the second line.

For all practical purposes there are four kinds of statements in CP-6 APL: comment, branch, assignment and non-assignment, and compound.

Comment Statements

To enter a comment statement, the user types the symbol ⍝ at the beginning of a line and follows it with a comment. The ⍝ symbol is produced by typing a ∩ symbol (upper shift C) and overstriking it with a ∘ symbol (upper shift J). This symbol signals APL that the line is a comment and is not to be executed. Any valid APL characters may be included in a comment; invalid APL characters produce an error message. If a comment extends over several lines, each line must begin with the ⍝ symbol. Some examples of comments are shown below:

```
      ⍝  ROOM AREA ROUTINE.
      ⍝
      ⍝
      ⍝  EACH LINE OF A MULTIPLE-LINE
      ⍝  COMMENT MUST BEGIN WITH A ⍝.
```

A comment statement can be entered as a direct input line (during execution mode) or it can be entered as part of a defined function. If a comment statement is entered as a direct input line, it is not retained in the workspace. If a comment statement is used in a function definition, however, the statement will have a line number, will occupy workspace, and will be displayed like any other function line. Function definition mode cannot be closed on a comment line, because the closing ∇ symbol will be treated as just another symbol in the comment. An example of a comment in a function definition is shown below:

```
      ∇A←H TRIAREA B
[1] ⍝  CALCULATES AREA OF TRIANGLE..
[2]    A←H×B÷2
[3]    ∇
```

In CP-6 APL, any executable statement may include a comment to its right. Everything to the left of a ⍝ character is considered executable. Everything to the right is considered comment. Some examples are:

```
[10]  COST←HOURS×RATE  ⍝  COST FOR STRAIGHT¯TIME LABOR.
[15]  OCOST←1.5×HOURS×RATE  ⍝  COST FOR OVERTIME LABOR.
```

When functions are displayed, comment lines are highlighted by indenting them one space less to the left of executable lines.

Branch Statements

Branch statements are generally used within defined functions to alter the sequential execution of statements. Another form of branch statement, covered later, is the branch arrow that is not followed by an expression. A branch arrow by itself can be used to terminate execution of a suspended function and the functions that invoked it, thus effectively clearing the state indicator to the next suspension (if any). This application of the branch arrow is described in Section 7. A branch statement has the general form

```
→exp
```

where exp stands for an integer value. The value determines the line number of the statement to be executed next, as follows:

1. If the value is a line number within the current function, that line is executed next. Thus the statement

   ```
   [5] →(2>A)×3
   ```

   where *A* has a value of zero, causes a branch to line 3 of the current function. (The value 3 is derived as follows: the expression (2>*A*) returns a value of 1; and this value is multiplied by 3.)

2. If the value is a line number outside the function being executed, then execution of that function terminates. For example, the statement

   ```
   [4]  →0
   ```

   indicates a branch to line 0, which is outside the function. Since functions begin with line 1, branching to line 0 is an effective way to exit a function.

3. If the value is an empty vector, then no branch occurs and the next sequential line is executed. If there are no more lines, execution of the function is terminated. An empty vector can be created in any of the following ways:

   ```
   0/S
   0⍴S
   0↑S
   ```

   where 0 may be the result of a comparison expression, and S represents a line number. (If the result of the comparison statement is 1 instead of 0, the next line executed is the one indicated by the line number.) Substituting the comparison expression *A*=4, which produces a value of 0 or 1, and line 2 in the above expressions illustrates the simplicity of this type of branching:

```
[5]  →(A=4)/2
[5]  →(A=4)ρ2
[5]  →(A=4)↑2
```

In each case if the value of *A* equals 4 (that is, the comparison expression returns a 1), then line 2 is executed next. If A is any other value, then the comparison expression returns a 0, yielding an empty vector, and line 6 will be executed next if it exists; otherwise execution of the function terminates.

The expression indicating the line numbers can be a scalar or a vector. In other words, the user can specify branching to one line, to one of two lines, or to one of any number of lines. Branching to one line is described above. Branching to one of two lines can take either of the following forms:

```
→(S1,S2)[1+X OP Y]
→((X OP Y),~X OP Y)/S1,S2
```

where

S1 is the line number to be branched to if the comparison expression yields a 0.

S2 is the line number to be branched to if the comparison expression yields a 1.

X OP Y is a comparison expression; *X* and *Y* are the values to be compared, and *OP* is any of the following functions: <, ≤, =, ≥, >, ≠, ∨, ∧, ⍱, ⍲, ∊, or ≡.

Both of these forms cause a branch to the first line number if the comparison yields 0, or to the second line number if the comparison yields 1. In illustration, the second form is entered in a defined function, and then executed with values for *X* and *Y*.

```
      ∇X F Y
[1]   →((X<Y),~X<Y)/A1,A2
[2]   A1:'STEP A1'
[3]   →0
[4]   A2:'STEP A2'
[5]   →0
[6]   ∇

      1 F 2
STEP A1
      2 F 1
STEP A2
```

Clearly the second form can be expanded to include more line numbers. Similarly, a branch to one of several statements can also take the form:

```
→I⌽V
```

where

I is a counter.

⌽ is the rotation function.

V is a vector of line numbers, the first of which must be a positive integer or zero.

In this case the branch function selects statement *I⌽V* as the next one to be executed. The following illustration shows how this branch function is carried out (see line number 3):

```
      ∇NUMB I
[1]   →(I≥4)/2 ◊ 'LOW' ◊ →0
[2]   →(I≤6)/3 ◊ 'HIGH' ◊ →0
[3]   →(I-4)⌽4 5 6
[4]   'FOUR' ◊ →0
[5]   'FIVE' ◊ →0
[6]   'SIX' ◊ →0∇
```

```
      NUMB  3
LOW
      NUMB  4
FOUR
      NUMB 5
FIVE
      NUMB 6
SIX
      NUMB 7
HIGH
```

See Figure 6–1 for a summary of common branch function formats that can be used; APL also offers many other forms of branching.

```
Branch to line S or to next line:

   →(X  OP  Y)/S
   →(X  OP  Y)ρS
   →(X  OP  Y)↑S

Branch to line S1 or line S2:

   →(S1,S2)[1+X OP Y]
   →((X OP Y),~X OP Y)/S1,S2

Branch to one of several lines:

   →((X OP Y),(X OP Y),X OP Y)/S1,S2,S3
   →IΦV
   →(S1 S2 S3)[I]
```

Figure 6–1. Summary of Common Formats for Branching

Statement Labels

Instead of referencing a line number in a branch statement, a statement label can be assigned to the branch destination. Referencing that label will obtain the current line number of the line. To assign a label to a line, precede the first statement with a variable name and a colon, as shown:

```
[5]   END:A←B÷2
```

The label *END* can now be used in a branch statement to transfer execution to this statement. For example, the statement

```
[3]   →(A<1)/END
```

will cause a branch to line 5 if *A* is less than 1, or a branch to line 4 if *A* is 1 or more.

The value of a label is the line number with which it is associated at the close of function definition. If new lines are inserted via function editing (see Section 7), then the values of the labels are automatically respecified at the closing of the function definition. The value of a label cannot be respecified by an assignment; any attempt to do so will produce a *SYNTAX ERR* message.

Like local variables (Section 3), the integer values of labels in one function can be accessed in other functions invoked by the function.

Use of a statement label in a branch statement is preferable to use of a line number, since function editing may change the original line number. If any lines are inserted or deleted during function editing, all lines will be renumbered at the close of a function definition mode. For example, consider the following statement which specifies a branch to line 5.

```
[3]  →5
```

If two new statements are inserted between lines 3 and 4, the old line 5 is renumbered as line 7 at the close of function definition. However, the branch statement will still cause a branch to statement 5 instead of line 7 as now desired. This problem can be avoided if labels are used instead of statement numbers as branch points. (See Changing Suspended Functions in Section 7 for other considerations about labels.)

When labeled lines are displayed within a function, they are highlighted by indenting them one space less than usual.

Assignment and Non-assignment Statements

An assignment statement assigns the result of an expression or a value to a variable name. It has the general form:

```
      name ← expression
```

where name can be any variable name and expression can be any APL expression. Three examples of assignment statements are

```
      B←6
      A←B÷2
      Z←(B<1)+3×5
```

A non-assignment statement is similar to an assignment statement except that it does not have the assignment arrow and the variable name to the left of it; however, a non-assignment statement can contain embedded assignments. Examples are:

```
      B÷2
3
      (B<1)+3×5
15
      2×4+A←2
12
      +A←1
1
```

Notice the differences between assignment and non-assignment statements: (1) execution of an assignment statement ends on the assignment, and (2) an assignment statement produces no display, while a non-assignment statement displays the resulting value of the statement.

Compound Statements

Using diamonds for separation, all of the preceding kinds of statements can be combined in "compound" statements. Compound statements have the following characteristics:

1. The statements are evaluated in left-to-right order, with each individual statement evaluated in the normal APL manner. Example:

   ```
   A←2 4 6 ⋄ ρA ⋄ A,1
   ```

   would produce two lines of output, an integer 3 corresponding to the result of the second statement, and a vector 2 4 6 1 corresponding to the third.

2. An assignment statement produces no display. Example:

   ```
         5×4÷2 ⋄ A←4
   10
         5×4÷2 ⋄ +A←4
   10
   4
   ```

3. A comment statement can have no statement to its right. All characters from the comment symbol ⍝ up to the end of the line are considered to be commentary. Example:

   ```
         3⌈2    ⍝ SHOWS ⌈ FUNCTION ⋄ THIS IS STILL A COMMENT.
   3
   ```

4. A branch statement implies no special display. In the no-branch case, statements to the right of the branch will be executed; they are ignored if a branch occurs. This provides conditional execution capability. Example:

   ```
         ∇VERACITY X
   [1]   →(X≠1)/2 ⋄ 'TRUE' ⋄ →0
   [2]   →(X≠0)/3 ⋄ 'FALSE' ⋄ →0
   [3]   'NEITHER TRUE NOR FALSE'
       ∇
         VERACITY 4=2+2
   TRUE
         VERACITY 2+2=4
   NEITHER TRUE NOR FALSE
         2+2=4
   2
         VERACITY (2+2)=4
   TRUE
   ```

5. If the statement is the subject of an execute function or evaluated input request, then the result of the function (or input request) is the result of the last expression executed. For example:

   ```
         A←⍎'⍳5 ⋄ +/⍳5'
   1 2 3 4 5
         A
   15
   ```

Section 7

Defined Functions

As mentioned in Section 3, defined functions are used in the same way as primitive functions. Defined functions must first be formed by the user instead of being an inherent part of the APL language.

User-Defined Functions

The following tasks are handled in function definition mode:

- Creating user-defined functions
- Displaying user-defined functions
- Editing user-defined functions

Once created, most functions can be edited and displayed. Once a locked function is created, however, it cannot be edited or displayed (see "Locking Functions" later in this section). Locked function lines cannot even be displayed for error diagnosis. It is possible, however, to erase a locked function.

User defined functions can be created or modified by function definition mode or by the ⎕*FX* system function. They can be loaded or copied from a library workspace or "packaged" and read or written to a file (see Section 14).

Function definition mode begins when a function is opened and continues until a function is closed or abandoned. (It is possible to close a different function than was originally opened by revising the name of the function.) A function may be "opened" during direct input or evaluated input (see Section 3), and it may be opened briefly during execution (see the Execute Operator, ⍎, Section 5). A function cannot be opened during any other form of input, such as quote-quad input or blind input; and a different existing function cannot be opened while still in function definition mode. Until a function is closed during function definition mode, APL execution is impossible except for system commands (which are executed and do not become part of the function being defined). Most system commands leave the currently open function intact and return the user to definition mode; however, some system commands cause a function definition to be abandoned (see Issuing System Commands later in this section).

Function Definition Mode

A del symbol, ∇, followed by a function name specifies a change from execution mode to function definition mode. A second ∇ symbol ends function definition mode and declares a change back to execution mode. No execution of statements occurs during function definition, and no errors are reported except for linescan errors, character errors, and definition errors. Instead, each statement is stored as part of the function.

Upon entry to definition mode, the editor is selected depending upon the setting of the last)*EDITOR* command. The default function editor is a line-oriented editor similar to the editor provided by other APL systems. A screen editor is also available and the capabilities unique to screen editing are described under the heading Screen Editor later in this section.

Each defined function has a header and a body. The function header is the opening line of a function and declares the name (the identifier used to reference the function) and type of a function. The body of a function is the rest of the function. After the user enters a function header, APL responds with a statement number as follows:

```
      ∇CUBE
[1]
```

The line number [1] signifies that the first line of the function program may be entered. Each line thereafter is numbered sequentially until the function is completed. The statements are stored and are not executed until definition mode is exited and the function named has been referenced.

Syntax of Defined Functions

A defined function can be niladic, monadic, or dyadic; that is, it can have zero, one, or two arguments. In addition, a defined function may return an explicit result or no result. Thus, there are actually six types of defined functions as illustrated by the following table of function header syntax possibilities:

Table 7-1. Function Header Syntax

Function	No Explicit Result	Explicit Result
Niladic function	∇name	∇r ← name
Monadic function	∇name y	∇r ← name y
Dyadic function	∇x name y	∇r ← x name y

where

name is the user-assigned function name.

r is a variable to which the result is returned.

x and y are dummy variable names.

The syntax of the function header affects the way a function can be referenced in a statement; that is, whether the function requires zero, one, or two arguments for execution. Defined functions with explicit results may appear in compound expressions, much like primitive functions. Defined functions without an explicit result must appear alone; they cannot appear in compound expressions except as the last function to be executed. Examples of creation and use of each function type are shown in Table 7-2.

Dyadic defined functions are not strictly dyadic. They may be executed monadically, in which case the left argument will be undefined at execution time. The ⎕*NC* function may be used to test for the presence of the left argument. (If the function is being executed monadically, the name class of the left argument is 0).

The result name in the function header may optionally be enclosed in braces {}. If it is enclosed, then the result of the function execution will not print if it is the primary function on the line (the last function executed).

Table 7-2. Defined Function Examples

Function Type	Header Syntax	Examples
Niladic function with explicit result	∇r ← name	∇*RESULT*←*PI* [1] *RESULT*←○1 [2] ∇ *PI* 3.141592654 ∇*RESULT*←*TRIANGLE* [1] *AREA*←0.5×*BASE*×*HEIGHT* [2] *DIAGONAL*←((*HEIGHT*⋆2)+*BASE*⋆2)⋆0.5 [3] *RESULT*←*AREA*,*DIAGONAL* [4] ∇ *BASE*←5 *HEIGHT*←8 *TRIANGLE* 20 9.433981132
Niladic function with no explicit result	∇name	∇*PI* [1] *X*←○1 [2] *X* [3] ∇ *PI* 3.141592654 ∇*TRIANGLE* [1] *AREA*←0.5×*BASE*×*HEIGHT* [2] *DIAGONAL*←((*HEIGHT*⋆2)+*BASE*⋆2)⋆0.5 [3] '*AREA IS* ',⍕*AREA* [4] '*DIAGONAL IS* ',⍕*DIAGONAL* [5] ∇ *BASE*←5 *HEIGHT*←8 *TRIANGLE* *AREA IS* 20 *DIAGONAL IS* 9.433981132
Monadic function with explicit result	∇r ← name y	∇*RETURN*←*EXPAND INPUT* [1] *RETURN*←((2×⍴*INPUT*)⍴1 0)*INPUT* [2] ∇ *EXPAND* '*COPY COMMAND*' *C O P Y C O M M A N D* ∇*RETURN*←*DESCENDINGSORT INPUT* [1] *RETURN*←*INPUT*[⍒*INPUT*] [2] ∇ *DESCENDINGSORT* ¯5 ¯3 10 5 6 8 10 8 6 5 ¯3 ¯5
Monadic function with no explicit result	∇name y	∇*EXPAND INPUT* [1] *X*←((2×⍴*INPUT*)⍴1 0)*INPUT* [2] *X* [3] ∇ *EXPAND* '*COPY COMMAND*' *C O P Y C O M M A N D* ∇*DESCENDINGSORT INPUT* [1] *X*←*INPUT*[⍒*INPUT*] [2] *X* [3] ∇ *DESCENDINGSORT* ¯5 ¯3 10 5 6 8 1 8 6 5 ¯3 ¯5

Table 7-2. Defined Function Examples (cont.)

Function Type	Header Syntax	Examples
Dyadic function with explicit result	∇r ← x name y	∇RESULT←BASE TRIANGLE HEIGHT [1] AREA←0.5×BASE×HEIGHT [2] DIAGONAL←((HEIGHT*2)+BASE*2)*0.5 [3] RESULT←AREA,DIAGONAL [4] ∇ 5 TRIANGLE 8 20 9.433981132
Dyadic function with no explicit result	∇x name y	∇BASE TRIANGLE HEIGHT [1] AREA←0.5×BASE×HEIGHT [2] DIAGONAL←((HEIGHT*2)+BASE*2)*0.5 [3] 'AREA IS ',⍕AREA [4] 'DIAGONAL IS ',⍕DIAGONAL [5] ∇ 5 TRIANGLE 8 AREA IS 20 DIAGONAL IS 9.433981132 ∇X PLUS Y [1] ANS←X+Y [2] ANS [3] ∇ 2 PLUS 5 10 15 20 7 12 17 22

Variables Local to a Defined Function

The three types of variables that can be local to a defined function are:

- Dummies
- Locals
- Labels

Dummies and locals are named in the function header, while labels are named in the body of the function.

Dummies

Dummies are used in the header of a defined function to indicate the syntax of a function. For example, notice the header of the following simple function (this function calculates the area of a triangle):

```
     ∇A←H TRIAREA B
[1]  A←H×B÷2∇
```

The dummies *A*, *H*, and *B* in the function header indicate that the function named *TRIAREA* returns an explicit result and that the function operates on two arguments which must be furnished by the user. For example, suppose the user calls this function with the statement

```
      AREA←10 TRIAREA 5
```

The dummy *H* in the function is assigned the value 10, and the dummy *B* is assigned the value 5. The result is returned in the dummy *A*, and is finally assigned to the variable *AREA* in the calling statement. Dummies possess values only within the function. That is, the use of *A*, *H*, and *B* as dummies does not affect their use as variables outside the function. If variables *A*, *H*, and *B* have values assigned to them before the function is called, they will have the same values after the function is executed. For example, suppose the variable *A* (with value 21) existed in the workspace before function *TRIAREA* was called. A display of variable *A* after the execution of *TRIAREA* demonstrates that *A* still has the value 21:

```
      A←21
      AREA←10 TRIAREA 5
      AREA
25
      A
21
```

Body of a Function

After the opening statement, in which the user creates the function header, the process of creating a function consists of inputting function statements and, finally, closing function definition. The user is prompted with a function line number each time the system is ready for further input. The process is ended by typing a closing ∇ followed by a RETURN key.

Locals

Locals are variables that retain their values only within the function in which they are defined. While a function is active, its local variables take precedence over any externally defined variables of the same name. A list of a function's local variables is added to the end of the function header, with each variable in the list preceded by a semicolon. For example, the function header

```
      ∇R←A CIRCLE B;X;Y;Z
```

indicates that the function named *CIRCLE* has locals *X*, *Y*, and *Z*. The values for these variables are assigned within the function; if these variables are referenced without having a value assigned within the function, an *UNDEFINED* error will be produced. If variables *X*, *Y*, and *Z* have values assigned to them before the function is called, they will revert to those values after the function has finished execution.

Labels

Function lines may be labeled to allow symbolically controlled branching (if a function is edited, line numbers may change). A labeled line has the form

```
    [n]   name:statement
```

where n is the line number, name is the label, and statement is the content of the line. For example:

```
[4]  ERREXIT:  'ERROR EXIT' ◊ →0
```

In this example, the label *ERREXIT* has the value 4. If an attempt is made to assign a value to *ERREXIT* during function execution, a syntax error message will be reported. If the function is edited and the line number changes to [5], *ERREXIT* will then have the value 5.

Changing Suspended Functions

At the time a function is suspended, its (current) local variables have been determined by APL, and its labels have already been assigned their values. Changing the suspended function does not alter these assignments. Resuming execution of a suspended function causes the determined items to remain in effect, regardless of how the function was altered.

Directives

During function definition mode, editing directives are used to display, modify, and add new lines. A directive may take any one of the following forms:

[1]

Directs APL to a line – here line 1.

[1⎕]

Directs APL to display a line and to stay at that line for further editing – here line 1.

[1-5⎕]

Directs APL to display a range of lines, here lines 1 through 5.

[⎕ 2]
[2-⎕]

Directs APL to display from a line to the end of the function – here beginning at line 2.

[⎕]

Directs APL to display the entire function.

[3-9;/x/]

Directs APL to display lines containing a string – here string "x" in lines 3 through 9.

[1⎕ 6]

Directs APL to edit a line, starting at a given column – here line 1 at column 6.

[1-20;2/x/*S*/y/]

Directs APL to change all occurrences of one string to another string in the specified range of lines – here the second occurrence on each line of the string "x" is changed to string "y" in lines 1 through 20.

The separator *S* may be replaced by the letter *F* (in which case the replacement string will follow the search string), the letter *P* (in which case the replacement string will precede the search string), or the letter *D* (in which case the replacement string may not be specified and the search string will be deleted).

If the occurrence number is omitted, only the first occurrence is replaced. If the occurrence number is 0, then all occurrences on each line are replaced.

[/*X̲*/]

Directs APL to search for the next occurrence of the string 'x' starting at the current line number through the end of the function.

[\X\]

Directs APL to search for the next occurrence of the string 'x' starting at the current line to the beginning of the function.

[∆2]

Directs APL to delete a line – here line 2.

[∆4-9]

Directs APL to delete a range of lines – here lines 4 through 9.

[∆5-8;/x/]

Directs APL to delete lines from a range which contain a string – here lines 5 through 8 containing string "x".

[→]

Directs APL to abandon definition mode and ignore all editing changes made.

[*D*]

In screen editing mode, APL will scroll down. If not in screen editing mode, an error is reported.

[*U*]

In screen editing mode, APL will scroll up. If not in screen editing mode, a *DEFN ERR* is reported.

A directive always starts with a left bracket and ends with a right bracket. With the exception of search or replacement strings, only legal line numbers, a dash, quad, delta, semicolon, slash or backslash are permitted within directives.

A line number is a number in the range 0 through 9999.999 which contains at most three digits after the decimal point. Scientific or *E* notation is not permitted. A line number range contains a dash separating the first line number of the range and the last line number of the range. At least one of the line numbers in a range directive must be specified. If the first number of a range is omitted, it is assumed to be 0. If the second number is omitted, it is assumed to be 9999.999. Examples of legal line numbers are:

```
    0
    123
    0.999
```

A quad appearing within the directive indicates a display directive and a delta appearing immediately following the left bracket indicates a line delete directive.

Directives must be the leftmost items input during function definition mode. Several directives may be used on one line, however, the rightmost directive overrides all directives to the left of it. For example, notice the following portion of a function definition:

```
     ∇FF
[1]  X←Y
[2]  [1] Y←X
[2]  [5] A←B
[6]
```

The [1] directive on the third line overrides the [2] directive to its left and causes the statement on line 1 to be replaced with *Y←X*; notice that the next prompt is [2]. (It should be obvious by now that a function line prompt is a form of directive.) Similarly, the [5] directive on the next line overrides the [2] directive to its left and causes line 5 to become the expression *A←B*. The next line prompt is then [6].

Search and Replacement Strings

Directive strings are delimited by either a slash or backslash. They may contain any characters in the CP-6 APL character set, but if they contain the delimiting character, it must be doubled. For example:

Delimited string	String
/2+*A**0.5/	2+*A**0.5
/*B*\C//*D*/	*B\C*/*D*
/*NAME*/	*NAME*

In the last example, the string NAME would be found in a line containing '1+*NAME*÷2' or in a line containing '1+*VARNAMES*÷2'.

When a search string is specified, the final delimiter may be followed by the character *N*, which is used to indicate that a string match should only be made if the characters before and after the match are not in the permitted set of identifier name characters. This option then allows searches for all occurrences of a specified identifier name. For example:

```
[3]  [0-4;/X/]
[0]  R←A FUN B;EXTRA;X
[1]  EXTRA←'TESTS'
[2]  X←A*B
[3]  EXTRA←EXTRA,⍕,X

[5]  [0-4;/X/N]
[0]  R←A FUN B;EXTRA;X
[2]  X←A*B
[3]  EXTRA←EXTRA,⍕,X
[5]
```

In the first example above, every occurrence of the letter *X* is displayed, but in the second example, only those occurrences of the identifier *X* are displayed. Note that if the identifier *X* appears within quotes, that line is still displayed.

Displaying User-defined Functions

A user-defined function can be displayed in any of the following ways:

- Display all lines of the function.
- Display one line.
- Display a range of lines.
- Display the next line.
- Display lines containing a string.
- Display the next line containing a string.

Displaying All Lines

To display a function, the user opens the function with a del symbol, names the function, and specifies what is to be displayed, all on the same line. The user can then either close the function with another del symbol (if no editing is to be done) or leave the function open for further editing.

If the user wants to display all of a function, function *TRIANGLE* for example, the procedure is as follows:

```
      ∇TRIANGLE [⎕]∇
    ∇ BASE TRIANGLE HEIGHT
[1]   AREA←0.5×BASE×HEIGHT
[2]   DIAGONAL←((HEIGHT*2)+BASE*2)*0.5
[3]   'AREA IS ',⍕AREA
[4]   'DIAGONAL IS ',⍕DIAGONAL
    ∇
```

Displaying One Line

If the user wants to display only one line of a function, say line 3 of function *TRIANGLE*, the procedure is

```
     ∇TRIANGLE[3⎕]∇
[3]  'AREA IS ',⍕AREA
```

Displaying a Range of Lines

If the user wants to display from one line to the end of a function, say from line 2 of a function *TRIANGLE*, the procedure is

```
     ∇TRIANGLE [2-⎕]∇
[2]  DIAGONAL←((HEIGHT*2)+BASE*2)*0.5
[3]  'AREA IS ',⍕AREA
[4]  'DIAGONAL IS ',⍕DIAGONAL
```

If the user wants to display a range of lines, for example, lines 1 through 2 of function *TRIANGLE*, the procedure is:

```
     ∇TRIANGLE[1-2⎕]∇
[1]  AREA←0.5×BASE×HEIGHT
[2]  DIAGONAL←((HEIGHT*2)+BASE*2)*0.5
```

The display of lengthy functions can be stopped at any point by pressing the BREAK key. The user can request the display to start at line 10 and then press the BREAK key after line 15 has been displayed. If the display command is closed with a del symbol, APL is in execution mode after the interruption. If the closing del is omitted, APL is in function definition mode after the interruption.

Notice that the display commands in all of the above examples are closed with a del symbol. This symbol causes control to be returned to execution mode as soon as the display is complete. To remain in function definition mode and edit the function instead, the user merely omits the closing del in the display command. See how the above examples appear without a closing del in each display command.

```
       ∇TRIANGLE [⎕]
     ∇ BASE TRIANGLE HEIGHT
[1]    AREA←0.5×BASE×HEIGHT
[2]    DIAGONAL←((HEIGHT*2)+BASE*2)*0.5
[3]    'AREA IS ',⍕AREA
[4]    'DIAGONAL IS ',⍕DIAGONAL
     ∇
[5]

       ∇TRIANGLE [3⎕]
[3]    'AREA IS ',⍕AREA
[3]

       ∇TRIANGLE [⎕ 2]
[2]    DIAGONAL←((HEIGHT*2)×BASE*2)*0.5
[3]    'AREA IS ',⍕AREA
[4]    'DIAGONAL IS ',⍕DIAGONAL
[5]
```

Notice that after a single-line display, APL reprompts with the same line number; and that after a multiple-line display, APL prompts with the next available line number. The user can then edit the function as described below or can enter another del symbol to close the function. Closing the function definition with a del symbol does not alter the content of that line. For example, the following operation does not change the value of line 3; it will still be *'AREA IS ',⍕AREA*:

```
     ∇TRIANGLE[3⎕]
[3]  'AREA IS ',⍕AREA
[3]  ∇
```

In order to find and display the line following a particular line, enter linefeed after the closing bracket of a simple line number directive. For example, to display the line following 1.5, the procedure is:

```
[4]  [1.5]
[2]  DIAGONAL←((HEIGHT*2)+BASE*2)*0.5
```

The entire function can be displayed one line at a time by entering linefeed after each line is displayed. In summary remember that

```
    [⎕]      displays entire function.

    [2⎕]     displays a single line (here 2).

    [⎕ 2]    displays from a line (here 2) to end of the function.

    [1-2⎕]   displays a range of lines (here 1 through 2)
```

Displaying Lines Containing a String

In order to display all lines containing a particular string of characters, the line range to search is followed by a semicolon, an optional count, and either a slash or backslash delimited string. If the count is present, the line is displayed only if it contains at least count occurrences of the string. When count is not present, 1 is assumed. For example, to display all lines in the function *TRIANGLE* containing the string *BASE*, the procedure is:

```
      ∇TRIANGLE[0-9;/BASE/]
[0]   BASE TRIANGLE HEIGHT
[1]   AREA←0.5×BASE×HEIGHT
[2]   DIAGONAL←((HEIGHT*2)+BASE*2)*0.5
[10]
```

If the search string is for a particular identifier, the closing slash or backslash may be followed by the letter *N* which causes APL to display only those lines in which the string is both preceded and followed by characters that are not legal within a name. For example, if the search is for all occurrences of the identifier *A*, then the directive:

```
    [7-20;/A/N]
```

will not display a line containing *AREA* (unless it also contains the name *A*).

Displaying the Next Occurrence of a String

In order to search for the next occurrence of a string, (either forward or backward), the directive must contain only a search string. If the search string is delimited by a slash, the search begins at the next line through the end of the function. If the search is delimited by a backslash, the search begins at the previous line through to line zero. If the string is found, APL displays that line and issues a prompt for that line. If the string is not found, APL then prompts for the line at which the search ended (either zero or 1+ the last line number in the function). For example, if the search is for the first occurrence of the string D after line 2, the procedure is:

```
      ∇TRIANGLE[2][/D/]
[4]   'DIAGONAL IS ',⍕DIAGONAL
[4]
```

Editing User-defined Functions

Editing of user-defined functions is oriented to line-at-a-time editing capabilities:

- Deleting a line
- Inserting a line
- Replacing a line
- Modifying a line

The first three capabilities can be performed as shown in Table 7-3. The last capability, modifying a line, permits character editing (that is, deletion, insertion, and replacement of characters), adding to a line, and overstriking existing characters on a line. All of these capabilities are detailed below. Column one of table 7-3 states the action to be performed. Column two gives an example of the action within definition mode. Column three gives an example of the same action when exiting definition mode. In both examples the functions are already open.

Table 7-3. Displaying and Editing Defined Functions

Action	Within Definition Mode	Exiting Definition Mode
Display entire function	[2] [⎕] ∇ *F* [1] *A* [2] *B* [3] *C* ∇ [4]	[2] [⎕]∇ ∇ *F* [1] *A* [2] *B* [3] *C* ∇
Display a line	[4] [2⎕] [2] *B* [2]	[4] [2⎕]∇ [2] *B*
Display line and change	[4] [2⎕]*B*←*X*+*Y* [2] *B* [3]	[4] [2⎕]*B*←*X*+*Y*∇ [2] *B*
Display function beginning with specified line	[4] [⎕ 2] [2] *B*←*X*+*Y* [3] *C* ∇ [4]	[4] [⎕ 2]∇ [2] *B*←*X*+*Y* [3] *C* ∇
Delete a line	[∆2] [3]	[∆2]∇
Delete a range of lines	[4] [∆1-2] [3]	[4][∆1-2]∇
Insert a line	[3] [0.5] *X* [0.6]	[3] [0.5] *X*∇
Replace a line	[4] [2] *Z* [3]	[4] [2]*Z*∇
Override a line number	[4] [2] [2]	[4] [2] [2] ∇
Display a range of lines	[4] [1-2⎕] [1] *A* [2] *B* [3]	[4] [1-2⎕]∇ [1] *A* [2] *B*
Find occurrences of a string	[4] [0-4;/*B*/] [2] *B* [4]	[4] [0-4;/*B*/]∇ [2] *B*

Table 7-3. Displaying and Editing Defined Functions (cont.)

Action	Within Definition Mode	Exiting Definition Mode
Find occurrences of identifier	[4] [0-4;/B/N] [2] B [4]	[4] [0-4;/B/N]∇ [2] B
Find next occurrence of a string	[4] [1][/C/] [3] C [3]	[4] [1][/C/]∇ [3] C
Find previous occurrence of a string	[4] [\B\] [2] B [2]	[4] [\B\]∇ [2] B
Abort changes, restore original version of function		[4] [→]
Change all occurrences of a string in a range of lines	[4] [1-3;/A/S/AB/] [1] AB [4]	[4] [1-3;/A/S/AB/]∇ [1] AB
Change function header	[4] [0] F;B [1]	[4] [0] F;B∇
Erase current function		[4])ERASE F
Erase another function	[4])ERASE G [4]	[4])ERASE G [4] ∇

A simple three-line function named *F* has been assumed in the examples in this table (see the first display entry in the table for the original content of function *F*). Note: The example which illustrates changing a function header, adds a local variable to the functional header.

Deleting a Line

A statement in a defined function can be deleted by using the delete directive. A delete directive may specify all of the line numbers to be deleted. For example, to delete line 2 of the following function:

```
      ∇BASE TRIANGLE HEIGHT
[1]     ⍝  THIS FUNCTION CALCULATES AREA AND HEIGHT OF TRIANGLE
[2]     ⍝  BASE AND HEIGHT CANNOT EXCEED 5 AND 15 RESPECTIVELY
[3]     AREA←0.5×BASE×HEIGHT
[4]     DIAGONAL←((HEIGHT*2)+BASE*2)*0.5
[5]     'AREA IS ',⍕AREA
[6]     'DIAGONAL IS ',⍕DIAGONAL
[7]     ∇
```

First, the user opens the function and issues the delete directive:

```
      ∇TRIANGLE[∆2]
```

APL responds with a prompt for line 3.

The user can now either close the function with a del symbol or proceed with further editing (including deleting the next line). (The user can also press the RETURN key if nothing is to be done to the line. APL simply responds with the line number, in this case [3]. A linefeed may be used in place of RETURN in which case APL displays the next line of the function.) A display of the function at this point illustrates that line 2 is deleted:

```
[3]   [⎕]
    ∇ BASE TRIANGLE HEIGHT
[1]   ⍝  THIS FUNCTION CALCULATES AREA AND HEIGHT OF TRIANGLE
[3]   AREA←0.5×BASE×HEIGHT
[4]   DIAGONAL←((HEIGHT*2)+BASE*2)*0.5
[5]   'AREA IS ',⍕AREA
[6]   'DIAGONAL IS ',⍕DIAGONAL
    ∇
[7]
```

The function can now be closed with a del symbol.

```
[7]  ∇
```

Once definition mode is exited, APL renumbers the line in sequential order, as illustrated by another display of the function

```
      ∇TRIANGLE[⎕]∇
    ∇ BASE TRIANGLE HEIGHT
[1]   ⍝  THIS FUNCTION CALCULATES AREA AND HEIGHT OF TRIANGLE
[2]   AREA←0.5×BASE×HEIGHT
[3]   DIAGONAL←((HEIGHT*2)+BASE*2)*0.5
[4]   'AREA IS ',⍕AREA
[5]   'DIAGONAL IS ',⍕DIAGNOAL
    ∇
```

Inserting a Line

A new line can be inserted in a defined function simply by reopening the function and entering the statement as described below. The user reopens the function by typing a del and the function name, to which APL responds by printing the line number of the next statement to be entered. If the new line is to be inserted at the end of the function, the user can now enter the new statement and close the function as shown:

```
      ∇TRIANGLE
[6]   ⍝  THIS FUNCTION IS USED IN ROUTINES 1 AND 2.
[7]   ∇
```

If the new line is to be inserted between two existing lines, however, the user must specify a line number between those two lines. For example, suppose the user wants to add a comment as the first line of function *TRIANGLE* instead of the last line. This can be done as follows:

```
      ∇TRIANGLE
[6]   [0.5]
[0.5]  ⍝  THIS FUNCTION IS USED IN ROUTINES 1 AND 2.
[0.6]
```

Notice the [0.6] prompt in this example. After an insert statement is entered, APL adds 1 to the last place of the number chosen for the insert, and prompts with the new number. (The next prompt after [0.6] will be [0.7]; the next, [0.8]; and so on.) This allows the user to insert several lines.

A display of function *TRIANGLE* illustrates that line [0.5] has been added

```
[0.6]  [⎕]∇
    ∇ BASE TRIANGLE HEIGHT
[0.5] ⍝  THIS FUNCTION IS USED IN ROUTINES 1 AND 2.
[1]   ⍝  THIS FUNCTION CALCULATES AREA AND HEIGHT OF TRIANGLE
[2]    AREA←0.5×BASE×HEIGHT
[3]    DIAGONAL←((HEIGHT*2)+BASE*2)*0.5
[4]    'AREA IS ',⍕AREA
[5]    'DIAGONAL IS ',⍕DIAGONAL
    ∇
```

After the function is closed, APL automatically renumbers the lines, as illustrated by the following display:

```
      ∇TRIANGLE[⎕]∇
    ∇ BASE TRIANGLE HEIGHT
[1] ⍝  THIS FUNCTION IS USED IN ROUTINES 1 AND 2.
[2] ⍝  THIS FUNCTION CALCULATES AREA AND HEIGHT OF TRIANGLE
[3]    AREA←0.5×BASE×HEIGHT
[4]    DIAGONAL←((HEIGHT*2)+BASE*2)*0.5
[5]    'AREA IS ',⍕AREA
[6]    'DIAGONAL IS ',⍕DIAGONAL
    ∇
```

Line Numbers

APL allows the user to type a line number with up to four numbers to the left of the decimal point and up to three numbers to the right. As noted above, after each insert line is entered, APL adds 1 to the last place of the insert. As illustrated in the following portion of a printout, the next prompt after an [.88] insert will be [0.89]; the next, [0.9]; the next, [0.91]; and so on:

```
      ∇F
[7]  [.88]
[0.88]
[0.89]
[0.9]
[0.91]
```

The highest integer line number printed by APL is [9999]; thus the highest possible line number is [9999.999]. If the user is prompted with [9999.999] and enters a legal statement, APL will prompt with the same line number since it cannot go any higher.

Replacing a Line

A line in a defined function can be replaced simply by reopening the function, directing control to the statement that is to be replaced, and entering the desired statement. For example, line 1 of function TRIANGLE is to be replaced with another statement. The user reopens the function by typing a del and the function name and directs control to line 1 by typing that line number in brackets. After the RETURN key is pressed, APL responds to this entry by printing the specified line number at the left margin, as shown:

```
      ∇TRIANGLE[1]
[1]
```

Any statement the user enters at this point will replace what previously existed at that line. Suppose the user now enters the following comment statement:

```
[1]  ⍝  INPUT MUST BE IN FEET
[2]
```

Notice that the next prompt is at line 2. If no more editing is required, the user can close the function by entering another del:

```
[2]  ∇
```

This action has no effect on line 2; it merely closes the function. The following display of function *TRIANGLE* illustrates the change to line 1:

```
      ∇TRIANGLE [⎕]∇
    ∇ BASE TRIANGLE HEIGHT
[1] ⍝  INPUT MUST BE IN FEET
[2] ⍝  THIS FUNCTION CALCULATES AREA AND HEIGHT OF TRIANGLE
[3]    AREA ←0.5×BASE×HEIGHT
[4]    DIAGONAL←((HEIGHT*2)+BASE*2)*0.5
[5]    'AREA IS ',⍕AREA
[6]    'DIAGONAL IS ',⍕DIAGONAL
    ∇
```

Issuing Multiple Directives

APL allows the user to open a function, change a line, and close the function all on one line. For example:

```
     ∇G[1][2]2.2∇
```

In this case the user opens function *G*, issues a directive to line 1 (realizes line 2 was meant), changes the directive to line 2, replaces whatever exists on that line with the value 2.2, and then closes the function. This shortcut operation allows the user to change a function without having to interact extensively with the computer. Another example is shown below:

```
      ∇G[1⎕]1.11∇
[1]   1.1

      ∇G[⎕]∇
   ∇ G
[1]   1.11
[2]   2.2
   ∇
```

The first line requests that line 1 of function *G* be displayed, and the contents of that line changed to the value of 1.11. The display of function *G* shows that line 1 has indeed been changed from 1.1 to 1.11. It should be noted that the user can display one line and change it at the same time, but cannot display an entire function and change something at the same time.

Modifying a Line

As mentioned earlier, modifying a line involves character editing (that is, deletion, insertion, and replacement of characters), adding to a line, and overstriking existing characters in the line. Modifications to a line can be specified by overriding the present line number with the directive:

```
      [n⎕c]
```

where n is the number of the line to be edited (0 for a function header), and c is the column at which to begin editing (the column position is the number of spaces from the left margin). APL will normally display the specified line, and position to the designated column. The editing column specified may be 0, in which case APL displays the line and stops at the end. If the designated line does not fit on a single line, no character editing can be done. In this case, APL simply displays the line and then reprompts the user with the same line number. The following is an example of such a line:

```
[2]   'A
      B'
```

If the typing element is still not in the proper position, the user can backspace, tab or use <CTL-R> to space forward until the desired position is reached.

The line-modifying capabilities of CP-6 APL are identical to those described in the CP-6 Programmer Reference Manual (CE40). In summary, the user may enter escape sequences to successively modify the content of function lines in a manner similar to that afforded direct input.

NOTE: An escape sequence is generated by pressing the <ESC> key once and then the appropriate key for the action desired (e.g., <ESC> followed by R for an escape-R sequence). The CP-6 system prints <R> on the terminal in response to an <ESC> R sequence.

For example, suppose the following function had been previously defined by the user.

```
      ∇ A PLUS B
[1]   'THE SUM OF ',(⍕A),' AND ',(⍕B),' IS ',⍕A+B
      ∇
```

Now the user wants to change the function to perform a multiplication rather than an addition. Suppose the function that will do this is called *TIMES*. To proceed with the example, the user opens the function for editing with

```
      ∇PLUS
```

and APL responds with [2]. The user types [1⎕ 0] to tell APL to display line 1 and remain at the end to await new instructions.

```
[2]  [1⎕ 0]
[1]  'THE SUM OF ',(⍕A),' AND ',(⍕B),' IS ',⍕A+B
```

APL waits at the end of the displayed line. The user presses RUBOUT twice to delete the *B* and then the +, and enters <ESC> followed by R to retype the line. The line appears as:

```
[1]  'THE SUM OF ',(⍕A),' AND ',(⍕B),' IS ',⍕A+B\\<R>
```

The two backslashes indicate the rubouts and <R> indicates the <ESC> R sequence. APL immediately types

```
[1]  'THE SUM OF ',(⍕A),' AND ',(⍕B),' IS ',⍕A
```

and waits after the last *A*. Now the user types ×*B* and enters <ESC> V S to move to the *S* in *SUM*.

```
[1]  'THE SUM OF ',(⍕A),' AND ',(⍕B),' IS ',⍕A×B
```

The terminal now waits at the *S* for more instructions. Now the user presses RUBOUT three times to delete *S* then *U* then *M* enters <ESC> J to switch to CP-6 insert mode types *PRODUCT* and then <ESC> R to show the line.

```
[1]  'THE SUM OF ',(⍕A),' AND ',(⍕B),' IS ',⍕A×B
          \\\PRODUCT<R>
[1]  'THE PRODUCT OF ',(⍕A),' AND ',(⍕B),' IS ',⍕A×B
```

APL now waits for input at the *T* in *PRODUCT*. The user has decided to change the name of the function to *TIMES*. First, the user presses RETURN to tell APL that a new line 1 has been defined. APL responds with [2]. The user rewrites line 0 directly and ends function definition all in one line.

```
[2]  [0] A TIMES B∇
```

Now the user demonstrates the new function.

```
      4 TIMES 6
THE PRODUCT OF 4 AND 6 IS 24
      0 TIMES 9
THE PRODUCT OF 0 AND 9 IS 0
```

Adding Characters to End of Line

To add one or more characters to the end of a line, specify zero as the column at which to begin editing. APL will then display the line unaltered and wait at the end of the line for the user to add something. An example of adding local variables to a function header is shown below:

```
[3]  [0⎕ 0]
[0]  RETURN←FUNC X;A;B
[1]
```

In this case APL typed the header as *RETURN←FUNC X* and waited at the end of the line, and the user typed *;A;B*.

Overstriking a Character

To edit a line and create a legal overstrike, specify zero as the column at which to begin editing. APL will display the line and wait at the end of it; the user can then backspace to the character to be overstruck, and type the second character. An example of overstriking a character is shown below:

```
[8]  [5⎕ 0]
[5]  A←⍞
```

In this case the first line caused statement 5, consisting of the expression *A*←⎕, to be displayed and APL to wait at the end of the line. The user then backspaced to the quad and typed an apostrophe, thus creating the legal overstrike ⍞.

Editing a Line Number

Line numbers may be edited in the same way that the content of a line is edited. One application of editing line numbers is in repeating a statement at several different lines. For example, the following procedure can be used to repeat the contents of line 2 at line 4.1:

```
      ∇G[2⎕ 1]
[2]   A←30÷12×A
```

APL waits under the [. The user presses <CTRL-R> to move under the 2, presses RUBOUT to delete it, enters <ESC> J to switch to insert mode, types 4.1, and enters <ESC> R to see the result.

```
[2]   A←30÷12×A
 \4.1<R>
[4.1] A←30÷12×A
```

When the user now presses RETURN, a new line 4.1 has been defined. The contents of line 2 remain the same; that line was merely copied to line 4.1.

Changing a Function Header

There are four changes the user can make to a function header (that is, to line zero).

1. Change the name of the function. Suppose the user reopens an existing function called *FF*1 and changes only the name of the function to *G*1 as shown below:

   ```
         ∇FF1[0]
   [0]   RETURN←G1 ARG
   [1]
   ```

   This example assumes that *G*1 does not already exist. (If it did, a *DEFN ERR* message would be reported.)

   Changing the function name has no effect on function *FF*1, the function still exists as it did before the reopen. Of course, *FF*1 is no longer the open function, *G*1 is. *G*1 is initially a copy of *FF*1 and any modifications subsequently made while in function definition mode apply only to *G*1. This feature allows synonymous function names as long as only the header is revised. It is possible for a user to make a locked version of an unlocked function in this manner, retaining the unlocked version only until satisfied that the locked version is error-free. Erasing the original function does not affect a synonymous function, nor does subsequent revision of the original. A synonymous function retains the stop and trace vectors supplied with the original function when it was copied.

2. Change the name of the result, change a function with a result to a no-result function, or change a no-result function to a function with a result. The following illustrates the change of function *FF*1's result name from *RETURN* to *R*:

```
      ∇FF1[0]
[0]   R←FF1 ARG
[1]
```

3. Change the name of an argument. An example is shown below, where function *FF1*'s argument is changed to *X*:

```
       ∇FF1[0]
[0]    R←FF1 X
[1]
```

4. Change the names of locals, insert locals, or delete locals. APL does not allow the user to delete a function header. Any attempt to do so will cause APL to print an error message and reprompt the user with line zero. To get rid of the current function, the user must issue an *)ERASE* command.

Screen Editing

CP-6 APL provides a screen-oriented editor for editing defined functions. The screen editor is requested by the)EDITOR system command. In this mode, a portion of the function being edited is always on the screen, and the bottom of the screen typically contains an area for error messages and the output of system commands. To modify a line, position to the character or characters to be changed and enter the appropriate characters to be inserted or replaced. To position the cursor to a particular line, enter <ESC> X and one of the line positioning directives listed below. In order to leave screen editing mode, enter a ∇ character at the end of a line or enter an <ESC> X∇ sequence. Please note that in this section, the sequence "<ESC>" indicates pressing of the escape key on the terminal.

Whenever the cursor moves off a line, APL determines whether the line is changed or if a directive was entered. A directive can be entered at anytime by typing <ESC> X followed by the directive. The line that was erased in order to enter the directive reappears as soon as the directive is acted upon. Display directives are not permitted in screen editing mode (mainly because the function is already being displayed). If the screen does not contain the portion of the function requiring modification, the line to be modified can be reached by entering linefeed characters to get to it, or by entering a bare line number directive such as [6] (which would position the cursor to line 6).

Table 7-4 contains a list of character sequences to perform some common input editing. For a complete list of screen editing input editing sequences, see the CP-6 Programmer Reference Manual (CE40). Table 7-5 contains a list of the directives that may be entered in screen editing mode and their meaning.

Table 7-4. Screen Editing Control Characters

Input	Meaning
<ESC>A	Position one line up.
<ESC>P	Restore line to its contents when the cursor arrived.
<ESC>N	Position to the end of the line.
<ESC>-	Repaint the screen.
<ESC>V?	Position the cursor to the next '?' character.
<ESC>J	Toggle insertion mode.
<ESC>K	Delete characters to the end of the line.
<ESC>X	Delete all characters on the line.
<ESC>*	Remember the characters in the insertion window (or line).
<ESC>:	Copy characters remembered by <ESC>* into line.
<ESC>N<ESC><LF>	Insert a line after the current line.
<ESC><BS>	Join current line to previous or next line.
<ESC><LF>	Split current line by inserting a new line following the current line.
<BS>	Position backward one character.
<CTL-R>	Position forward one character.
<CTL-I>	Position forward to next tab setting.
<tab>	Position forward to next tab setting.

Table 7-5. Screen Editing Directives

Input	Meaning
[n]	Position to line n if directive has nothing following it. Otherwise replace line n with the remaining text.
[/string/]	Position to the next higher line number which contains the string 'string'.
[\string\]	Position to the next lower line number which contains the string 'string'.
[Δn]	Delete line number n.
[Δn-m]	Delete line numbers n through line m.
[Δn-m;/str/]	Delete line numbers n through line m if they contain the string 'str'.
[→]	Abandon screen editing mode ignoring all editing changes.
[n-m;c/x/S/y/]	Replace the c'th occurrence of the string 'x' with the string 'y' in lines n through m.

Issuing System Commands

CP-6 APL allows the user to enter any system command while in function definition mode. Most system commands keep the user in function definition mode, while some system commands (described below) return the user to execution mode or even exit APL. After commands that keep the user in definition mode, APL will prompt with the same line number at which the command was given. For example, suppose the user is at line 5 of a function and wants to find out the names of variables in the workspace:

```
[5]  )VARS
AAA  BAT  DDD
[5]
```

The system commands that exit function definition mode are: *)CLEAR*, *)LOAD*, *)COPY*, *)PCOPY*, *)QLOAD*, *)QCOPY*, *)QPCOPY*, *)CONTINUE*, *)CONTINUE HOLD*, *)OFF*, *)OFF HOLD*, *)END*, *)SAVE*, and an *)ERASE* of the current function. All of these commands force a close of the definition mode as though the user had closed it, but the resulting disposition of that function depends on the command. The *)CLEAR*, *)LOAD*, *)QLOAD*, *)ERASE*, *)OFF*, and *)END* commands cause the function to be discarded; the *)SAVE*, *)COPY*, *)PCOPY*, *)QCOPY*, *)QPCOPY*, *)CONTINUE*, and *)CONTINUE HOLD* commands automatically reopen the function after the command has finished. In the last situation, as soon as the command has finished, APL signals the user of the reopening by printing the function name (with an opening del) and prompting with the next available line number. With the *)CONTINUE* and *)CONTINUE HOLD* commands, of course, the function is not opened until the next APL session. The user should display the function before doing any more editing, since renumbering may have occurred because of the forced close.

Function Execution

APL permits recursive functions (those which reference themselves when they are executed). APL also allows the user to suspend function execution. These topics are discussed in detail below.

Recursive Functions

Recursive functions reference themselves in the body of their definitions. As an example, notice the following function which returns the factorial of its argument:

```
      ∇Z←FAC N
[1]   Z←1 ◊ →(N≤1)/0 ◊ Z←N×FAC N-1∇

      FAC 0
1
      FAC 1
1
      FAC 4
24
```

Suspending Execution

Execution of a function is suspended (stopped) before completion, if any of the following occurs: the BREAK key is pressed, an error is encountered (unless sidetracking occurs, see section 10), or a user-set stop control is reached (see ⎕*STOP*). When a suspension occurs, APL prints the name of the suspended function and the line number at which it was suspended. At this point, APL is in direct execution mode (subject to any ⎕*SA* requirements, see Section 11). Any functions that can be performed in execution mode are applicable during function suspension. As long as a function is suspended, its local variables are active and can be examined and modified.

The user can resume execution of a suspended function by specifying a branch. Entering a branch arrow followed by a RETURN key clears that suspension, while specifying a branch to a particular line number resumes execution at the beginning of that line (that is, at the right end of that line). Branching to a line outside a function's range of line numbers, or zero, terminates the execution of that function.

As a general rule, it is best not to leave a function suspended, because the information about that function occupies workspace which is valuable to the APL user (see State Indicator). In addition, each time the user attempts to execute an already suspended function, even more information about that function is added to computer memory. Thus, if the user has no specific reason to leave a function suspended, it should be cleared before proceeding with the rest of the program. (See also the)*SIC* command in Section 8.)

State Indicator

APL maintains a "state indicator" that gives a list of all suspended and pendent functions (that is, all "active" functions). A suspended function is one where execution is stopped before completion (see Suspending Execution). A function is pendent unless specifically suspended. Most commonly, this is observed when one (pendent) function has called a suspended function. As a rule, suspended functions are stopped between lines, while pendent functions are stopped in the middle of a line. Note, however, when a function is suspended due to an error, the error marker may indicate the middle of the line; nevertheless, the function is stopped between that line and its predecessor. A display of pendent and suspended functions can be obtained via the)*SI* system command, with the most recent active function displayed first.

```
      )SI
Z[2]  *
X[4]  *
Y[3]
Z[2]  *
X[2]
W[5]  *
```

An asterisk after an entry indicates a suspended function; absence of an asterisk indicates a pendent function. The bracketed number after a function name is the number of the next line to be executed. If there are no suspended or pendent functions in the state indicator, no report will result from the)*SI* command. The number of items in the state indicator can be determined by typing the expression ρ⎕*LC*.

Unlike suspended functions, pendent functions cannot be erased, copied over, or edited. As an example, look at the state indicator list shown above. Functions *Z* and *W* can be edited but functions *X* and *Y* cannot. Notice that function *X* is listed as both pendent and suspended; it cannot be edited because it is pendent in one of its states. Also notice that function *Z* has been suspended twice.

There is one instance in which a pendent function will not be listed in the state indicator. Suppose a dyadic function is about to be executed, pending resolution of its left argument. Assume that argument is obtained as the result of some function, say *F*, and *F* is suspended. Then the dyadic function is pendent, because it is ready to execute as soon as *F* is resumed. But the dyadic function is not listed in the state indicator because it has not yet entered a state of execution. Fortunately, this situation is rare and seldom will confuse the user.

The system command)*SINL* lists the contents of the state indicator, including a list of variables local to pendent and suspended functions. Using the command)*SINL* lists the following:

```
      )SINL
Z[2]  *    A    B
X[4]  *    AA
Y[3]
Z[2]  *    A    B
X[2]       AA
W[5]  *
```

As with the)*SI* command, the most recent active function is displayed first. This example indicates that variables *A* and *B* are local to function *Z* and that variable *AA* is local to function *X*. Only the local variables of the most recent active functions can be accessed by the user. Thus, the user can access local variables *A* and *B* of the last invocation of function *Z*, and variable *AA* of the last invocation of *X*. But, the user cannot access local variables *A* and *B* of the first invocation of function *Z* or local variable *AA* of the earlier invocation of function *X* (see *X*[2]).

The user can clear the state indicator by using the branch arrow (that is, →). Each branch arrow clears one suspended function and its associated pendent functions; thus, to clear the entire state indicator, the user enters a branch arrow for each asterisk in the list. For example, the user can clear the previous indicator.

```
      →
      )SINL
X[4]  *    AA
Y[3]
Z[2]  *    A    B
X[2]       AA
W[5]  *
      →
      )SINL
Z[2]  *  A   B
X[2]     AA
W[5]  *
      →
      →
      )SINL
```

The *)SINL* commands in this example show what is left in the state indicator after each branch arrow. The user can also clear the same state indicator by entering four successive branch arrows.

```
      →
      →
      →
      →
      )SINL
```

In this case, the *)SINL* command shows that nothing is left in the state indicator. The easiest way to completely clear the state indicator is to issue a *)SIC* command.

CP-6 APL provides limited protection against *SI DAMAGE*. As an example, suppose the user opens function *F* and modifies the header, changing the function's type (e.g., monadic to dyadic, result to no-result) and then attempts to close function *F*. If *F* is not suspended, the function is closed as usual. If *F* is suspended, APL issues a warning (to the effect that references in the state indicator will be damaged by the change to the header) and requests a response from the user. The user can either order the close to occur with *SI DAMAGE* by typing *YES* followed by a RETURN, or cancel the close in order to revise the function further, hopefully correcting the header. Only a type change requires this protection. It is perfectly permissible to make other changes to the header, such as adding locals or renaming the result or dummy arguments; however, this is seldom advisable (see Changing Suspended Functions above).

Locking Functions

A function can be locked during definition or editing by using an opening or closing ⍫ (∇ overstruck with ~) instead of a ∇. A locked function can be executed, copied, or erased, but it cannot be displayed, suspended, or altered. After a function is locked, any associated trace control or stop control is automatically reset. Examples of locking functions are:

```
      ⍫ HH                 ∇HH              ⍫ HH
[8]   ∇               [8]  ⍫          [8]   ⍫
```

Once locked, if an error exists that is not sidetracked in the function, the error is implicitly sidetracked by APL to the line on which the locked function was invoked and the error report occurs on that line.

System Functions Controlling Defined Functions

CP-6 APL provides system functions which have the ability to create, modify, display, and set or query the attributes of defined functions. This section also introduces the terms namelist and canonical representation which are defined in Section 11 under the heading "Namelist and Canonical Representations". The system functions covered in this section are:

⎕*TRACE*	Set/query function trace attribute
⎕*STOP*	Set/query function stop attribute
⎕*CR*	Obtains function character representation
⎕*FX*	Creates or modifies a function
⎕*AT*	Query function attributes

Each function is discussed in detail below.

⎕*TRACE* System Function (Tracing Execution)

Syntax:

```
R←⎕TRACE F

R←V ⎕TRACE F
```

Parameters:

F   is a namelist containing the name of a displayable defined function.

V   is an integer or vector of integers that specify the line numbers for which execution results are to be displayed. Only the integers that correspond to line numbers in the named function are significant.

R   is an integer vector containing the original trace settings.

Description:

Function execution can be traced by displaying the results of statements (some or all) as execution of the function progresses. When any of the traced line numbers is executed, the result of its statements are printed. If the specified line contains a branch statement, a branch arrow followed by the new line number is printed. Specifying a trace vector of (⍳0) discontinues the trace.

Examples:

```
      (⍳0)⎕TRACE 'FAC'
```

stops trace of function *FAC*.

Below is an example of tracing the execution of a function. Notice that all output resulting from a trace is identified by the function name and line number.

```
      ∇Z←FAC N
[1]   Z←1
[2]   →(N≤1)/0
[3]   Z←N×FAC N-1
[4]   ∇

      1 2 3 ⎕TRACE 'FAC'
      FAC 0
FAC[1]  1
FAC[2]  →0
1
      FAC 1
FAC[1]  1
FAC[2]  →0
1
      FAC 4
FAC[1]  1
FAC[2]  →⍳0
FAC[1]  1
FAC[2]  →⍳0
FAC[1]  1
FAC[2]  →⍳0
FAC[1]  1
FAC[2]  →0
FAC[3]  2
FAC[3]  6
FAC[3]  24
24
      (⍳0) ⎕TRACE 'FAC'
1 2 3
```

The same function written as a compound statement produces the following trace output:

```
      ∇Z←FAC N
[1]   →(N≤Z←1)/0 ◊ Z←N×FAC N-1∇

      1 ⎕TRACE 'FAC'
      FAC 0
FAC[1]  →0
1
      FAC 1
FAC[1]  →0
1
      FAC 4
FAC[1]  →⍳0
FAC[1]  →⍳0
FAC[1]  →⍳0
FAC[1]  →0
FAC[1]  ◊ 2
FAC[1]  ◊ 6
FAC[1]  ◊ 24
24
```

The dyadic ⎕*TRACE* function requires that the right argument contain a valid name or a *DOMAIN ERR* is reported. The explicit result of dyadic ⎕*TRACE* is an integer vector containing the original trace setting of the named function.

Setting a trace vector can also be included as part of a defined function. For example, if the statement 1 ⎕*TRACE 'FAC'* is included within the above function, line 1 will also be traced each time the function is invoked. More complex expressions can be used to produce conditional tracing. In such cases, the condition produces one or more values (line numbers) that are the left argument of ⎕*TRACE*. This generalization also applies to the stop vector described below.

The)*OBSERVE* command, described in Section 8, extends the tracing facility. It permits the user to see not only the final result of a trace command, but every intermediate result occurring as APL executes a traced statement.

The current trace settings may be obtained by the monadic execution of the ⎕*TRACE* system function. In this case, the right argument is the same as in the dyadic usage of ⎕*TRACE* and the result is an integer vector containing the current trace settings. For example:

```
      ⎕TRACE 'FAC'
1
      '' ⎕TRACE 'FAC'
1
      ⎕TRACE 'FAC'
```

Possible Errors:

A *RANK ERR* is reported if:

o the left argument (new trace settings) is not a scalar or vector.

A *DOMAIN ERR* is reported if:

o the left argument is not a simple array containing only integers.

⎕*STOP* System Function (Stopping Execution)

Syntax:

```
      R←⎕STOP F

      R←V ⎕STOP F
```

Parameters:

F is a namelist containing the name of a displayable defined function.

V is an integer or vector of integers that specify the line numbers at which the function is to stop. Of course, only the integers that correspond to line numbers in the named function are significant. If 0 is an item of *V*, the function stops on exit.

R is an integer vector containing the original stop settings.

Description:

A planned suspension of function execution, called a function stop, can be established by setting a stop control vector. This vector is set in the same manner that a trace control vector is set for a function trace.

When each specified line number is reached, APL stops execution and prints the function name, the line number, and optionally the line about to be executed. Function execution is now in a normal suspended state (subject to ⎕*SA* setting), and can be terminated or resumed by appropriate branching (see Suspending Execution). Specifying ⍳0 discontinues the stop control vector; for example, (⍳0) ⎕*STOP* *'FAC'* discontinues any function stops in function *FAC*. The)*REPORT* system command is used to include the APL statements in the stop report.

Examples:

Below is an example of stopping execution of a function named *CIRCLE*:

```
      2 5 ⎕STOP 'CIRCLE'
      CIRCLE
CIRCLE[2]
       Suspension activities
      →2
13
10
30
CIRCLE[5]
```

The explicit result of ⎕*STOP* is an integer vector containing the original stop settings of the named function. Like the trace control vector, the stop control vector can also be used within a defined function to stop execution after a certain number of loops. Editing a line that has a trace or stop control set removes the control for that line. Deleting, copying the function from a saved workspace, or locking a function also deletes trace control and stop control vectors associated with a function.

The current stop settings may be obtained by executing the ⎕*STOP* function monadically. In this case, the right argument is the same as in the dyadic usage of ⎕*STOP* and the result is a simple integer vector of the current stop settings. For example:

```
      STOPS←⎕STOP 'CIRCLE'
      ⍴STOPS
2
      STOPS
2 5
      '' ⎕STOP 'CIRCLE'
2 5
      ⎕STOP 'CIRCLE'
```

Possible Errors:

A *DOMAIN ERR* is reported if:

o the right argument does not contain a valid name.

A *RANK ERR* is reported if:

o the left argument (new stop settings) is not a scalar or vector.

A *DOMAIN ERR* is reported if:

o the left argument is not a simple array containing only integers.

⎕*CR* System Function (Canonical Representation)

Syntax:

```
    R←⎕CR F
```

Parameters:

F is a namelist containing the name of a displayable defined function.

R is a simple character matrix.

Description:

The ⎕*CR* system function is used to obtain the character representation of a defined function. The right argument must be a namelist containing the name of a single defined function. The result is a matrix containing the canonical representation of the function (if it is displayable) or a 0 by 0 matrix if the name is not a defined function or not displayable.

The canonical representation of a function contains the function header in the first row, followed by the function lines in the remaining rows.

Examples:

```
      ⍴R←⎕CR 'FAC'
4  11
      R
R←FAC N
R←1
→(N≤1)/0
R←N×FAC N-1
```

⎕FX System Function (Fix Definition)

Syntax:

```
    R←⎕FX CR
    R←AT ⎕FX CR
```

Parameters:

CR is a simple character matrix (or vector with carriage returns) containing the canonical representation of a defined function.

AT is a scalar or four-item vector containing only the scalar values 1 or 0.

R is a simple character vector containing the name of the function established or an integer scalar row index of *CR*.

Description:

The ⎕FX system function creates a defined function from its canonical form. The right argument must be a character matrix (or vector with carriage returns separating lines). The first row of the matrix must be a valid function header and the remaining rows must be valid function lines. The explicit result of this function is the name of the function that was established, or the integer row index of the line which caused the definition attempt to fail.

Before the function is established, APL makes sure that the name is not currently in use for anything other than a defined function. A *DOMAIN ERR* is reported if the name is in use and not a defined function or if the right argument is not a simple character array. A *RANK ERR* is reported if the right argument is not a scalar, vector or matrix. If the name is currently a local symbol to an active or executing function, then this function will exist as a local function.

When ⎕FX is used dyadically, the left argument must either be a scalar or four-item vector of simple booleans (1's and 0's). The left argument specifies the execution properties of the defined function. The four properties in order are:

1. not displayable
2. not suspendable
3. not interruptable
4. execution errors converted to DOMAIN ERR

If a scalar is used as the left argument, all four properties are set to that value. Setting all of the properties to 1 is the same as locking the function.

Examples:

```
      ⍴R←⎕FX CR←(24↑'R←FAC N'),[0.5]'→(N≤R←1)/0 ◊ R←N×FAC N-1'
3
      R
FAC
      CR
R←FAC N
→(N≤R←1)/0 ◊ R←N×FAC N-1

      ∇FAC[⎕]∇
      R←FAC N
[1]   →(N≤R←1)/0 ◊ R←N×FAC N-1
      ∇
```

⎕*AT* System Function (Function Attributes)

Syntax:

```
    R←I ⎕AT NAMES
```

Parameters:

NAMES is a namelist containing the names of defined functions.

I is the simple scalar integer value 1, 2, 3, or 4.

R is a simple matrix containing the requested function attributes.

Description:

The system function ⎕*AT* returns attributes for each function named in the right argument. When a function is created by function definition or by the ⎕*FX* system function, four attributes specific to the function are defined. The attributes include the valence of the function, the creation time, the execution properties, and the account which created the function.

The right argument of the ⎕*AT* system function must be a name list containing the names of the functions whose attributes are to be returned. The left argument is an integer scalar in the range 1 through 4 whose value determines the attribute to be returned. The result is a matrix (or vector if the namelist is a vector containing one name) with one row for each name in the namelist.

The left argument value and the associated attributes are:

1 – Valences

Three items indicating whether a result may be produced and the number of arguments. The first item is 0 if there is no result, or 1 if there is a result. The second item is 0, 1, or 2 for niladic, monadic or dyadic functions. The third item is 0 and is reserved for future use.

2 – Creation Time

A seven-item vector indicating the time that the function was created. The items are in the following order: year, month, day, hour, minute, second, and millisecond.

3 – Execution Properties

A four-item vector, indicating execution properties of this function. The first item is 1 if the function may not be displayed (⎕*CR* not permitted). The second item is 1 if the function may not be suspended (by double attention or an error). The third item is 1 if the function is not interruptable by a single attention. The fourth item is 1 if any execution error (non-resource) produces a *DOMAIN ERR* report. The action of locking a function sets all but the last of these properties to 1. The dyadic use of the ⎕*FX* system function permits each of these properties to be set independently.

4 – Creator

An eight-item character vector, indicating the account that created (or last modified) this function.

Examples:

```
      ρR←1 ⎕AT 'FAC'
3
      R
1  1  0

      2  ⎕AT 'FAC'
1983  10  11  12  29  59  610
```

Section 8

System Commands

System commands allow the user to control the mechanical aspects of APL, and can be divided into three categories:

1. Workspace Control Commands – commands that affect the state of active and saved workspaces.

2. Inquiry Commands – commands that supply information about the active workspace.

3. Communications Commands – commands that send messages to the computer operator and log the user off APL.

System commands always begin with a right parenthesis and can be entered when the system is in execution mode or definition mode. By using the Execute operator (see Section 5), system commands can be embedded in an APL expression and in a function line. Thus, a system command can be placed under control of such expressions or functions. Only the first four letters of command names are significant. Name characters after the fourth are ignored. Thus *)CLEA* and *)CLEAVAGE* are both interpreted to be the *)CLEAR* command. Note that a blank must separate the command name and any following parameters; for example, *)WIDTH* 30 is not the same as *)WIDTH*30. A number of conventions are used in this section to describe the command formats.

1. Uppercase letters and special symbols must be typed exactly as they appear (except that only the first four letters of a command are required, as noted above).

2. Lowercase letters are employed to indicate where in a command to substitute a name or numerical value. The meanings or the notations in lowercase letters are as follows:

account	User account.
fid	CP-6 file identifier of the form: name.account.password. Name can consist of up to 31 characters. Account and password can consist of up to 8 characters.
fname	Name of a function.
grpname	Name of a group.
list	List of names (of functions, variables, groups), separated by blanks.
message	Actual message to computer operator.
n	An integer value.
objname	Name of function, variable, or group.
string	Any sequence of characters not including a blank or carriage return. If a string includes more than 79 characters, those past the 79th are ignored. Strings are used for range demarcation in certain commands.
vname	Name of a variable.

wsname	A workspace name; can consist of up to 31 characters (letters, underscored letters and numbers) as long as the first character is not a number. It has the same form as fid.

The actual system commands are detailed later in this section, but first it is necessary to describe the concept of a workspace in order to understand how certain commands are used.

Workspace Concept

Each user has a storage area containing control information which can be saved for future use.

Active Workspace

Associated with each user is a storage area in the computer known as an active workspace. This active workspace contains the following:

1. All control information currently applicable to the terminal session.

2. The variables, functions, and groups entered for calculations and still active during the session.

3. A state indicator that keeps track of the names of suspended and pendent functions and at what point they were interrupted.

4. System variables that control several features of APL, such as index origin, seed for random number generation, line width, and number of significant digits (decimal places) printed. These system variables all assume default values when the user first invokes APL, but they can be respecified with system commands, or by assignment.

When APL is invoked, the active workspace is usually clear (that is, there is nothing in it except the default values of the parameters mentioned above in item 4). An active workspace can also be cleared with the system command)*CLEAR*.

Saved Workspace

An active workspace can be saved for future use with the)*SAVE* command. Once a workspace is saved, any user who knows the workspace name can load it as an active workspace using the)*LOAD* command. The workspace's variables, functions, and groups can be copied into an active workspace using the)*COPY* command. The workspace can also be dropped using the)*DROP* command (if file access controls permit). In addition, the names of saved workspaces in an account can be listed with the)*LIB* command.

Continue Workspace

A line disconnect or either of the following commands cause the active workspace to be saved in the logon account:

```
)CONTINUE
)CONTINUE HOLD
```

The CONTINUE workspace is automatically loaded as an active workspace the next time the user invokes APL unless it is directed to load another workspace. In general, the CONTINUE workspace can be used the same as any other named workspace. It can be saved, copied, loaded, etc. However, it should only be used for temporarily saving a workspace, since another *)CONTINUE* command or line disconnect would save another active workspace over what was previously saved. That is, the previous CONTINUE workspace will be overwritten.

Since the CONTINUE workspace is part of the user's logon account, it is subject to the granule restrictions imposed by an installation. If the user's account is near that limit, the CONTINUE workspace may not be saved, and the information in the active workspace may be lost if a line disconnect occurs (see User Accounts). The CONTINUE workspace is saved with its access controls set to restrict access of the workspace to the user who created it.

Initiating an APL Session

APL is invoked with the following IBEX command syntax:

```
!APL [fid1] [{ON|OVER|INTO} [fid2] [,fid3]] [(options)]
```

Parameters:

fid1 is a CP-6 file identifier designating either a workspace to be loaded, or a file containing APL statements to be used as input. In either case, fid1 indicates "source input" (the current setting of M$SI). If fid1 is a workspace file or if fid1 is not specified, then APL input will default to the terminal on-line or the default input device in batch (ME). The APL *)SET INPUT* command may be used to redirect input after entering APL.

ON specifies that if fid3 already exists, the file is not to be overwritten. An error is reported.

OVER specifies that fid3 is to be overwritten even if it currently exists.

INTO specifies that APL output is to be appended to the end of file fid3 (if it exists).

fid2 is the CP-6 file name that is to be used by APL to designate the CONTINUE workspace name (the current setting of M$OU). If not specified, the CONTINUE workspace name defaults to the string 'APL:' followed by the current user's logon name (established when logging onto CP-6). The account used to hold the CONTINUE workspace is always the logon account. APL uses this file identifier in the event of a line disconnect, an uncontrolled error, or a limit exceeded error in batch mode, or if a *)CONTINUE* command is issued.

fid3 is the CP-6 file identifier that specifies the file containing output generated by the APL session (the current M$LO setting). If fid3 is not specified, then APL output will default to the terminal on-line and the line printer in batch. The APL *)SET OUTPUT* command may be used to redirect output after entering APL.

options is the list of APL options to be used for this session separated by commas. The options permitted are *QUIET*, *WS*, and *CPV*. The *QUIET* option invokes APL without the initial version and either *CLEAR WS* or *SAVED* messages being displayed. The *WS* option must be followed by = and a fid which identifies a workspace to be automatically loaded. If the *WS* option is specified, then fid1 must contain the APL statements to be executed. The *CPV* option causes some of the primitive functions in CP-6 APL to perform as their counterparts in CPV APL performed.

User Accounts

Accounts are specified when logging onto CP-6 or when accessing files in accounts other than the default file management account for a user. CP-6 installations impose restrictions on file allocation space (and file access) of file management accounts. When an account is at (or very near) its space limit, other files (or workspaces) in the account may need to be deleted to create or update a file (or workspace). In this event, APL reports the error. The)? command can be used to obtain more information about the error.

Command Processor

The material which follows assumes that the Command Processor in effect when APL is invoked is the CP-6 IBEX processor. If this is not the case, the commands *)CONTINUE*,)!, *)OFF* and *)SET* may operate in a manner other than specified here. In particular, for the transaction processing command processor (TPCP), some of these commands will result in the *BAD COMMAND* error.

System Command Summary

The system commands are detailed below in alphabetic order, and are summarized by category in Table 8-1.

Table 8-1. System Command Summary

Command	Description
Workspace Control Commands	
)CATCH [vname VIA name]	Removes any current catches (i.e., intercepts of assignments to specified variable names) or designates that assignments to vname are to be "caught" (intercepted immediately after the assignment), and that the test function name, a niladic function or character vector, is to be executed.
)CLEAR	Clears active workspace and restores default width, print precision, index origin, comparison tolerance, random number link, etc.
)COPY fid [list]	Copies functions, variables, and groups from saved workspace. Any password must be included, and so must the account if different than the file management account. If list is present, then only those named are copied. If list is not present, all names in fid are copied.

Table 8-1. System Command Summary (cont.)

Command	Description
)DIGITS [n]	Displays the current value of ⎕*PP* (numeric print precision). If n is specified, sets the value of ⎕*PP*, and displays the previous value of ⎕*PP*.
)DROP [fid]	Deletes a saved workspace. If the file identifier is protected with a password, the
)ERASE list	Removes the named objects such as functions, variables, or groups from active workspace.
)GROUP grpname [list]	Groups objects and names the group. If list is not specified, disperses the named group.
)LOAD fid	Moves a copy of the saved workspace into the active workspace. If the file identifier is protected with a password, the password must be specified. Also, if the saved workspace is in another account, that account must be specified.
)OBSERVE	Specifies that the next (direct input) statement and any traced function statements executed are to be "observed". This displays a number of "observations", showing intermediate results as APL interprets those statements.
)ORIGIN [n]	Displays the current value of ⎕*IO* (the index origin). If n is specified, sets the value of ⎕*IO* where n can be 0 or 1, and displays the previous value of ⎕*IO*.
)PCOPY fid [list]	Same as *)COPY*, except that a name is not copied if it already has a value in the active workspace.
)QCOPY fid [list]	Same as *)COPY*, except that the SAVED message is suppressed, i.e., quiet copy.

Table 8-1. System Command Summary (cont.)

Command	Description
)QLOAD fid	Same as *)LOAD*, except that the SAVED message is suppressed, i.e., quiet load.
)QPCOPY fid [list]	Same as *)PCOPY*, except that the *SAVED* message is suppressed, i.e., quiet copy.
)SALVAGE fid [list]	Similar to *)COPY* except objects may be copied from broken workspaces.
)SAVE [fid]	Saves the active workspace. If fid is specified, saves active workspace under the specified name. To save a workspace and protect it with a password, follow the workspace name with two periods and the password name (i.e.,)SAVE wsname..password).
)SEAL [fid]	Saves the current workspace as a sealed 'execute-only' workspace with the designated name.
)WIDTH [n]	Displays the value of *⎕PW* (the current maximum width of output lines). If n is specified, the value of *⎕PW* is changed, and the previous value of *⎕PW* is displayed. The width parameter n can range from 32 to 390.
)WSID [fid]	Displays the file identifier of active workspace. If fid is specified, assigns the file identifier to active workspace, or changes the name if one already exists and displays the old name.
Inquiry and Communication Commands	
)CONTINUE [*ON*|*OFF*|[HOLD][fid]]	Ends terminal session, and saves the active workspace as a CONTINUE workspace. If HOLD is specified, returns control to the CP-6 IBEX command processor. If *OFF* is specified, suppresses automatic generation of CONTINUE workspace file. If *ON* is specified, reinstates such automatic generation. If fid is specified, it overrides the default CONTINUE workspace name.

Table 8-1. System Command Summary (cont.)

Command	Description
)EDITOR [*CP6RR*|*STD*|*SE*]	Displays the current editor. If *CP6RR* is specified, the CP-6 re-read mode of editing APL lines in definition mode is used when a [line ⎕ position] directive is encountered. If *STD* is specified, APL "super-edit" mode of editing APL lines in definition mode is selected. This is the editing method most often available on other APL implementations. If *SE* is specified, APL uses the CP-6 screen editor.
)END	Returns control to CP-6 IBEX.
)ERROR [*BRIEF*|*FULL*|*SUMMARY*]	Displays the current error message information level. If *BRIEF* is specified, the most concise error messages for error displays are selected. If *SUMMARY* is specified, one-line error messages for error displays are selected. If *FULL* is specified, the most informative error messages for error displays are selected (possibly multi-line error messages).
)FNS [string1 [string2]]	Alphabetically lists all defined function names in active workspace. CP-6 APL uses the following collating sequence in the process of alphabetizing: o blank or end of name o digits o alphabetic letters without underlines (*A* through *Z*) o underline o underlined alphabetic letters (*A̲* through *Z̲*) o Δ̲, Δ If string1 is specified, the list of names starts at the first name which is alphabetically equal to or greater than string1. If string2 is specified, the list of names ends before the first name alphabetically greater than string2.
)GO	Resumes execution at the current line.
)GRP name	Lists the names in the specified group.

Table 8-1. System Command Summary (cont.)

Command	Description
)GRPS [string1 [string2]]	Alphabetically lists all group names in active workspace. CP-6 APL uses the following collating sequence in the process of alphabetizing: o blank or end of name o digits o alphabetic letters without underlines (*A* through *Z*) o underlined alphabetic letters (*A* through *Z*) o Δ, ∆ If string1 is specified, the list of names starts at the first name which is alphabetically equal to or greater than string1. If string2 is specified, the list of names ends before the first name alphabetically greater than string2.
)IBEX message	Issues command to IBEX.
)LIB [account]	Lists the names of saved workspaces in an account.
)NMS [string1 [string2]]	Alphabetically lists all of the global names in use and their name class in the active workspace. The string1 and string2 parameters are used in the same way as the *)FNS* command.
)OFF [HOLD]	Ends the terminal session and discards the active workspace. If *HOLD* is specified, control returns to CP-6 IBEX, otherwise the user is logged off of CP-6.
)OPR message	Sends message to computer operator, and waits for a reply.
)OPRN message	Sends message to computer operator.
)QUIT	Same as *)END*.

Table 8-1. System Command Summary (cont.)

Command	Description
)REPORT [FUNC[TION]|*LINE*]	If *FUNCTION* is specified, APL displays the function name and line number when a function is stopped (default). If *LINE* is specified, APL displays function name, line number and the contents of the line when a function is stopped.
)RESET	Completely clears the state indicator. Same as)SIC.
)SET dcb fid	Allows routing of regular output, input and/or 'blind' I/O channels to files or various devices, and specification of formatting options for device output. Analogous to the SET command in CP-6 IBEX.
)SI [*ON*|*OFF*|*CLEA*[R]]	Lists the contents of the state indicator, a list of suspended and pendent functions. If *CLEAR* is specified, clears the entire state indicator. If *OFF* is specified, prevents an error from suspending the function containing the erroneous statement. If *ON* is specified, restores normal state indicator control. If an error occurs in an active function line, APL suspends the function at that line (assuming sidetracking does not occur, see section 10).
)SIC	Completely clears the state indicator. Same as)RESET.
)SIL	Lists contents of the state indicator, a list of suspended and pendent functions, and the contents of lines in execution.
)SINL [*ON*|*OFF*|*CLEA*[R]]	Lists the contents of the state indicator, a list of suspended and pendent functions, and the local variables named by those functions.
)SIV [*ON*|*OFF*|*CLEA*[R]]	Same as)*SINL*.
)STEP [*LINE*|*FUNC*[TION]] [n]	Executes the line indicated by the top entry in the state indicator, and stops before any other line is executed. If the *FUNCTION* parameter is specified, the stop will not count function lines in functions invoked by the line initially put in execution. The *LINE* parameter is the default, and it causes APL to stop before any other line is executed. The n parameter specifies the number of lines to execute before stopping.

Table 8-1. System Command Summary (cont.)

Command	Description
)TERMINAL [INPU[T]|*OUTP*[UT]] [n]	Identifies to APL the input/output devices being used, where n can be any of the following values: 1,13,14 for devices with APL character set 2,3,4,5 for devices with ASCII character set
)VARS [string1 [string2]]	Alphabetically lists all global variable names in active workspace. CP-6 APL uses the following collating sequence in the process of alphabetizing: o blank or end of name o digits o alphabetic letters without underlines (*A* through *Z*) o underlined alphabetic letters ($\underline{A}$ through $\underline{Z}$) o $\underline{\Delta}$, Δ If string1 is specified, the list of names starts at the first name which is alphabetically equal to or greater than string1. If string2 is specified, the list of names ends before the first name alphabetically greater than or equal to string2.
)?	Displays the next highest detailed error message (if any) about the most recent error condition.

)CATCH Intercepting Assignments

Syntax:

)*CATCH* [vname *VIA* name]

Parameters:

vname is the name of the variable (which may be local or global).

name is the name of a function or character vector.

Description:

The *)CATCH* command is primarily a debugging tool. It permits the programmer to intercept each assignment to a specified variable name, immediately after that assignment is completed. The function is defined according to the user's debugging needs. The only restriction is that this name must be a character vector or represent a niladic function with no result. This restriction isolates the name from the statement or statements assigning values to the specified variable. If the name is undefined or does not indicate a character vector or a niladic, no-result function, no error message occurs; the catch is simply ignored. Catches on shared variables are not permitted.

Catches are not saved when a workspace is saved, so loading a workspace does not automatically reinstall catches. The *)CLEAR* command also removes any current catches. The *)CATCH* command without options removes any existing catches. A maximum of two catches can be defined at one time.

Examples:

Suppose the programmer has invoked the following catch,

```
      )CATCH V1 VIA F1
```

then all assignments to the name *V1* cause function *F1* to be called or if *F1* is a character vector, the expression ⍎*F1* is executed. This includes indexed assignments. *F1* is executed regardless of whether *V1* is a local or global variable. The programmer can modify this catch to enter a different test function. For example,

```
      )CATCH V1 VIA FTWO
```

After the above specification, assignments to *V1* cause test function *FTWO* to be called (instead of *F1*).

The programmer can also invoke a second catch. For instance,

```
      )CATCH VAR2 VIA FOTHER
```

The programmer can have both catches enter the same test function as in the next example.

```
      )CATCH VAR2 VIA FSAME
      )CATCH V1 VIA FSAME
```

The programmer cannot, however, invoke a third catch; this attempt produces a *BAD COMMAND* error.

Any current catches can be removed by issuing the command

```
      )CATCH
```

Following this command, the programmer is free to specify one or two new catches.

The simplicity of the *)CATCH* command may obscure its power as a debugging aid. This power is brought to bear by the test expression. A few hypothetical examples are given below to suggest the potential of catch capability.

Using a catch to display values assigned to vname:

```
      )CATCH X VIA SHOWX
      SHOWX←' ''X IS :'' ◊ X ◊ '
```

As long as the catch is in effect, every assignment to *X* will cause the new value of *X* to be displayed. A catch may be used to stop execution when a particular value is assigned to a name. (Assume that *X* is a scalar and 77 is the value of interest.)

```
      )CATCH X VIA CHECK
      ∇CHECK
[1]   →(X≠77)/0
[2]   STOP ⎕STOP 'CHECK'
[3]   STOP:∇
```

As long as this catch is in effect, each assignment to *X* will be tested at line 1 of the CHECK function. If *X* is not 77, line 1 causes CHECK to exit. If *X* receives the value 77, line 2 is executed. Line 2 sets the stop-vector for the CHECK function so that when the line labeled STOP is reached, CHECK will suspend execution.

Using a catch to change the value of vname:

(Note that this does not affect the value used by the statement making an assignment to vname; the catch is isolated.)

```
      )CATCH X VIA CHANGE
      ∇CHANGE;CHANGE
[1]   X←0 ∇
```

As long as this catch is in effect, each assignment to *X* that occurs "outside" the CHANGE function will cause *X* to be set to 0. The assignment at line 1 of the CHANGE function will not be "caught" because calling the function temporarily declares the name CHANGE to be a local variable (shadowing the definition of CHANGE as a test function); see the function header line.

Suppose the following statement is executed with the above catch in effect.

```
      X + 100 + X←55
```

The answer of 155 results in the following way.

1. The value 55 is obtained.

2. *X* is assigned the value 55.

3. The catch occurs.

4. *X* is set to 0 by the *CHANGE* function.

5. Execution of the original statement resumes, undisturbed (so far, at least) by the catch. This means that the value 55 is the right argument of the next addition.

6. 100 plus that argument yields 155.

7. This value, 155, becomes the right argument of the next addition.

8. The value of *X* is obtained; it is now 0.

9. 0 plus 155 yields the final result.

)CLEAR Clearing Workspace

Syntax:

```
)CLEAR
```

Description:

The)*CLEAR* command deletes all groups, functions, variables, and the state indicator from active workspace. Furthermore, it resets the following system variables and workspace attributes to the values in parentheses:

- Random number link (16807). ⎕*RL*
- Comparison tolerance (1*E*¯13). ⎕*CT*
- Index origin (1). ⎕*IO*
- Platen width (terminal dependent). ⎕*PW*
- Significant digits (10). ⎕*PP*
- Workspace identification (*CLEAR WS*).
- Latent expression (''). ⎕*LX*
- Stop action (''). ⎕*SA*
- State indicator control (ON); see also the)*SI* command description.
- Current catches (none), see the)*CATCH* command description.
- Error number (0) , see Sidetracking on Errors and Breaks in Section 10
- Error location (line number 0 and function name an empty character vector). (See also Section 10).

APL responds to this command by printing the message *CLEAR WS*.

Example:

```
      )CLEAR
CLEAR WS
```

)CONTINUE Saving Active Workspace and Leaving APL

Syntax:

```
)CONTINUE  [[ON|OFF|[HOLD][fid]]
```

Parameters:

fid overrides the default CONTINUE workspace name.

HOLD causes APL to exit to IBEX rather than logging off.

ON specifies reinstatement of automatic generation of the CONTINUE workspace file.

OFF specifies suppression of automatic generation of the CONTINUE workspace file.

Description:

The *)CONTINUE* command saves the active workspace in a CONTINUE workspace, and logs the user off. This workspace is automatically loaded the next time the user invokes APL without specifying a workspace. The active workspace is also automatically saved as a CONTINUE workspace if the terminal is accidentally disconnected or other unexpected end of session (LIMIT exceeded, unexpected error, etc.) occurs, unless such automatic action is suppressed (see below).

A successful *)CONTINUE* command will produce a save report (time and date saved) and the CP-6 log off messages. If insufficient room remains in the user's account to save the workspace, APL prints an error message. If this happens, the user must delete some workspaces or other files before any APL workspaces may be saved.

The default CONTINUE workspace name can be overridden at APL invocation time or by appending a fid to the end of either form of this command. See "Initiating an APL Session" for more details on the continue workspace name.

NOTE: If a user's workspace is passworded, the password is retained in the CONTINUE workspace. In this case, the CONTINUE workspace is not automatically loaded the next time the user logs on.

If an account already contains a passworded CONTINUE workspace, any subsequent CONTINUE will fail until the passworded version is deleted. Sealed workspaces cannot be saved with CONTINUE.

If either form of the *)CONTINUE* command is given during function definition mode, the currently open function is closed by APL. When the CONTINUE workspace is loaded later, APL automatically reopens the function and prompts the user to continue function definition. The automatic saving of this workspace can be suppressed by issuing *)CONTINUE OFF*. It can be reinstated by issuing *)CONTINUE ON*.

The CONTINUE workspace can be used almost like any other named workspace. It can be saved, copied, loaded, etc. However, the default name should only be used for temporarily saving a workspace since any CONTINUE workspace can be erased by a new CONTINUE workspace save.

Examples:

```
      )CONTINUE
APL:201GEISERT SAVED 15:33 DEC 15 '84
```

Saves active workspace and ends terminal session after printing save report and CP-6 log off messages.

```
      )CONTINUE HOLD
APL:201GEISERT SAVED 15:31 DEC 15 '84
!
```

Saves active workspace and returns control to command processor after printing save report. IBEX prompts for commands with the ! character.

)COPY Copying from Saved Workspace

Syntax:

)COPY fid [list]

Parameters:

fid is a CP-6 file identifier of a saved workspace.

list is a list of variable names, function names, or group names, separated by blanks.

Description:

The *)COPY* command copies information from a saved workspace into the active workspace. The information can consist of one, several, or all of the functions, global variables, and groups in the saved workspace. If the list parameter is not specified, all of the global names in the saved workspace are copied (except for system variables).

Note that if a workspace is saved with a password, that password must be included in the *)COPY* command. Also, if a workspace is copied from another user's account, the account must be specified in the *)COPY* command.

When a saved workspace is copied, only global functions, global variables, and groups are copied. If copied functions had sidetracks (see Section 10), then these settings also apply in the active workspace. All referents of a copied group are themselves copied into the active workspace. For instance, suppose group *G1* is copied, where *G1* contains *A*, *B*, and *G2* with *G2* being another group containing *X*, *Y*, and *Z*. Then the following are copied into the active workspace: *G1*, *G2*, *A*, *B*, *X*, *Y* and *Z*. The state indicator and system variables are not copied. (Most system variables can be copied by specifically naming them.)

A copy attempt may fail if there is not enough room in the active workspace to hold the items copied. In that case, an error message is displayed and the workspace will contain the same objects it contained before the *)COPY* command was issued. The error message *TOO BIG TO LOAD* is displayed when copying from a different account in which two conditions are met. First, the workspace copied from is large (so large that it could not even be loaded by the current user). Second, the referenced account is allocated more computer memory than is available to the current user's account (memory allocations are specified by the installation manager). This difficulty can be circumvented with the cooperation of the owner of the larger account, who can copy portions of the large workspace, forming one or more smaller workspaces. After this cooperative activity, the current user can copy required objects from those smaller workspaces.

If a *)COPY* command is issued during function definition mode, the currently open function is temporarily closed. When the copy is completed, the function is automatically reopened. The copy may have replaced the current function. If the *)COPY* command names functions that are pendent in the active workspace, they are not replaced. Suspended functions may be replaced and may cause an *SI DAMAGE* error message to be issued. Use of the *)PCOPY* command precludes this possibility.

The *)PCOPY* command is the same as the *)COPY* command except that an object is not copied if the active workspace already contains an object with the same name.

A group of objects can be copied even though the group definition is not copied. This happens if the group name matches the name of a pendent function in the active workspace or if the name matches any object in the case of *)PCOPY*. Alternatively, a group definition may be copied but some of its objects not copied.

Examples:

```
      )COPY GRANOLA.ACCT33.SECRET
GRANOLA SAVED 15:08 DEC 15 '84
```

Copies a saved workspace named GRANOLA, and prints a save report giving the time and date GRANOLA was saved. The workspace GRANOLA is saved with the password SECRET in another user's account (account ACCT33).

```
      )COPY KAWA
KAWA SAVED 15:00 DEC 15 '84
```

Copies an entire saved workspace named KAWA from the user's own account and produces a save report giving the time and date KAWA was saved.

```
      )COPY WS ATMF CHEAP ⎕PP
WS SAVED 13:31 DEC 01 '84
```

Copies a function named *ATMF*, a group named *CHEAP*, and the system variable *⎕PP* from a saved workspace named WS in the user's own account. A save report giving the time and date WS was saved is printed.

```
      )COPY HENRY..SECRET
HENRY SAVED 15:08 DEC 15 '84
```

Copies a saved workspace named HENRY, and prints a save report giving the time and date HENRY was saved. The workspace named HENRY is saved with the password SECRET in the current user's account.

If the *)COPY* command is used to access a workspace sealed by another user, the error message *SEALED WS* is reported.

)DIGITS Specifying Numeric Print Precision

Syntax:

)DIGITS [n]

Parameters:

n indicates the new value for *⎕PP* (the number of significant digits in printed output) which can be any integer number from 1 through 20. APL then prints the previous value of *⎕PP*. If n is not specified, APL prints the current value of *⎕PP*.

Description:

The *)DIGITS* command sets the number of digits in numeric output to a number between 1 and 20 inclusive. The default value in a *CLEAR WS* is 10, which displays a maximum of 10 significant digits. Only numeric output and the result of the monadic ⍕ function are affected by this command; internal calculations are not affected.

Examples:

```
      )DIGITS
IS  10
```

This requests the value of *⎕PP* to be displayed. APL responds with the current value.

```
      )DIGITS 15
WAS 10
```

This sets the value of *⎕PP* to 15. APL responds with the previous value.

```
    4÷9
0.444444444444444
```

Here, the result of a calculation is printed. APL displays the value to 15 significant digits.

```
    )DIGITS 5
WAS 15
```

This sets ⎕*PP* digits to 5, and APL responds with the previous value.

```
    4÷9
0.44444
```

The result of an expression is displayed again, showing 5 significant digits.

The number of significant digits to be output can also be changed by redefining the value of ⎕*PP*.

)DROP Dropping a Saved Workspace

Syntax:

)DROP [fid]

Parameters:

fid is a CP-6 file identifier (omission of fid implies the default CONTINUE workspace) of a saved workspace.

Description:

The *)DROP* command removes a saved workspace. It has two forms, one for removing unprotected workspaces, and another for removing the default CONTINUE workspace. If the workspace is not found, delete access is not available, or the proper password is not provided, APL returns the message *WS NOT FOUND*. If the workspace is deleted, APL returns a message identifying the workspace and the time it was last saved.

Examples:

```
    )DROP GRANOLA..SECRET
GRANOLA SAVED 14:58 DEC 15 '84
```

Removes the workspace GRANOLA with password SECRET, from the user's account.

)EDITOR Selecting the APL Function Editor

Syntax:

)EDITOR [CP6RR|STD|SE]

Parameters:

CP6RR selects the CP-6 re-read mode of editing APL lines in definition mode when a [line ⎕ position] directive is encountered.

STD selects APL "super-edit" mode of editing APL lines in definition mode. This is the editing method most often available on other APL implementations.

SE selects the CP-6 APL screen editor. It is available for most CRTs that may be connected to the CP-6 system. The terminal profile must indicate RETYPOVR=YES and EDITOVR=YES.

Description:

The *)EDITOR* command permits the APL user to choose the edit mode of line editing for the [line ⎕ position] directive. If a mode is not specified, the current editor setting is displayed. CP-6 re-read mode is by far the more powerful line editing technique, but super edit mode is included for compatibility with other APL implementations.

Super edit is a two pass editing method. In this mode, the line is displayed and APL awaits input on the line following at the position specified. Blanks or backspaces may be entered to position. The digits 0 through 9 insert that number of blanks, the letters A through Z insert 5, 10, 15,... blanks. A slash "/" is used to delete characters. A decimal point "." inserts all of the characters following it.

The line is re-displayed with all of the character insertions and deletions and the cursor is positioned at the first insertion position. Now all of the normal CP-6 line editing capabilities are available to modify the line.

Examples:

```
       )EDITOR STD
WAS CP6RR
      ∇FUN
[1] ⍝  THIS IS A TET OF SUPER EDIT
[1] [1⎕ 20]
[1] ⍝  THIS IS A TET OF SUPER EDIT
                   1
[1] ⍝  THIS IS A TEST OF SUPER EDIT
[2] ∇
```

)END Exiting APL

Syntax:

)END

Description:

The *)END* command causes the contents of the active workspace to be discarded, following which control is passed to the process which invoked APL. This is usually IBEX, the CP-6 Command Processor. This command is functionally identical to the *)OFF HOLD* command.

)ERASE Deleting Objects From Active Workspace

Syntax:

)ERASE list

Parameters:

list specifies the names of the global objects (i.e., functions, variables or groups) to be erased. Note that it is the value that is erased; the name may remain in the symbol table.

Description:

The *)ERASE* command deletes one or more named objects (i.e., global functions, global variables, or groups) from the active workspace. If a group is named in the *)ERASE* command, that group definition is erased along with any functions, groups, or variables named in the group. Pendent functions cannot be erased. It is impossible to erase a locked function in a sealed workspace. During function definition, if the function being defined is erased, definition mode is abandoned (equivalent to closing the function and then erasing it).

Examples:

```
      )ERASE MATHFUNCTIONS
```

Erases a group named *MATHFUNCTIONS* and the functions and variables it names. It disperses any group named within the group *MATHFUNCTIONS*.

```
      )ERASE PAYROUTINE GROSS INS
```

Erases a function named *PAYROUTINES* and two variables named *GROSS* and *INS*.

NOTE: The *)ERASE* command will not remove local variables.

)ERROR   Selecting Error Message Information Level

Syntax:

)ERROR [BRIE[F]|FULL|SUMM[ARY]]

Parameters:

BRIEF   selects the most concise error messages for future error displays.

FULL   selects the most informative error messages for future error displays (possibly multi-line error messages).

SUMMARY   selects one-line error messages for future error displays.

Description:

The *)ERROR* command selects the default error message information level. APL error messages are often available in various levels of information. The most concise messages are known as *BRIEF*. This type includes *DOMAIN*, *RANK*, *LENGTH*, and other general messages. These messages often contain sub-divisions which provide information specific to this instance. These sub-divisions are known as *SUMMARY* and *FULL*. *SUMMARY* messages are typically one line and *FULL* messages can contain up to seven lines of error message text. The)? command may be used to obtain additional error information after an error has been reported.

Examples:

```
      )ERROR SUMMARY
WAS BRIEF

      5÷0
DIVISION BY ZERO
      5÷0
       ^
      )ERROR FULL
WAS SUMMARY

      1 2 + 1 2 3
THIS FUNCTION REQUIRES THAT BOTH ARGUMENTS
HAVE THE SAME SHAPE (DIMENSIONS) OR THAT AT
LEAST ONE ARGUMENT IS A SINGLE ELEMENT ARRAY.
      1 2+1 2 3
         ^

      )ERROR BRIEF
WAS FULL

      1 2+1 2 3
LENGTH ERR
      1 2+1 2 3
         ^
```

)FNS Listing Global Function Names

Syntax:

```
)FNS [string1 [string2]]
```

Parameters:

string1 is any sequence of characters not including blank or carriage return.

string2 is any sequence of characters not including blank or carriage return.

Description:

The)*FNS* command alphabetically lists the names of functions in the active workspace. If string1 is specified, all function names that are alphabetically equal to or greater than string1 and are also less than or equal to string2 are displayed. If string1 is not specified, all function names are displayed. Alphabetic ordering is illustrated in the examples. Note particularly the first)*FNS* command since it indicates where each name character lies in alphabetic order.

If a string includes more than 79 characters, those past the 79th are ignored. Strings are only used for range demarcation in an alphabetic ordering.

Examples:

```
      )FNS
F   F0  F1  FF  FX  FXY FE̲  F⍙  F∆  S   T

      )FNS FF
FF  FX  FXY  FE̲  F⍙  F∆  S   T

      )FNS F FF
F   F0  F1  FF

      )FNS FFF FX
FX

      )FNS FXY FXY
FXY

      )FNS A Z
F   F0  F1  FF  FX  FXY FE̲  F⍙  F∆   S   T
```

)GO Resume Execution

Syntax:

```
)GO
```

Description:

The *)GO* command resumes execution of the most recently suspended function at the start of the current line.

)GROUP Creating a Group

Syntax:

```
)GROUP grpname [list]
```

Parameters:

grpname is the name of the group. A group name follows the same formation rules as a variable or function name, except that a group name cannot be the same as a global function or global variable in the active workspace.

list is a list of the names that make up the group, separated by blanks.

Description:

The *)GROUP* command references a group of names, i.e., variables, functions, other groups, or just names collectively. Group definitions can be used in *)ERASE* and *)COPY* commands to facilitate erasing and copying a group of related objects. Names can be added to an already existing group by merely repeating the group name in any of the command forms:

```
      )GROUP grpname grpname list
      )GROUP grpname list grpname
      )GROUP grpname list grpname list
```

A group can be dispersed with the command form

```
      )GROUP grpname
```

This form disperses the group; that is, removes the name references previously associated with grpname. The names and their references are not themselves erased, only the group identity is lost. An *)ERASE* command can be used to remove the group, but the *)ERASE* command removes the group and deletes the group referents (the actual functions or variables) from the active workspace.

Examples:

```
      )GROUP PROB1 COS TAN A B
```

Defines a group named *PROB*1, consisting of the variable and functions named *COS*, *TAN*, *A*, and *B*.

```
      )GROUP PROB1 PROB1 D ST
```

Adds the variable *D* and *ST* to the already existing group named *PROB*1.

```
      )GROUP PROB1
```

Disperses the group named *PROB*1 from the active workspace. The referents of *PROB*1 are not deleted.

Note that the last example disassociates the function and variable names from the group, but does not delete actual functions and variables from the active workspace. The *)GRPS* command can be used to verify that the group named *PROB*1 has been deleted, and the *)FNS* and *)VARS* commands can be used to verify that the named function and variables still remain in the active workspace.

```
      )GRPS
      )FNS
COS TAN
      )VARS
A   B   C   ST
```

Also see the *)GRP* and *)GRPS* commands, which list the members of a group and the names of groups in active workspace respectively.

)GRP Listing Members of a Group

Syntax:

)GRP name

Parameters:

name   is the name of a group.

Description:

The *)GRP* command prints all of the names contained in the specified group.

Examples:

```
      )GROUP G1 A B C
      )GRP G1
A   B   C
      )GROUP G1 G1 D
      )GRP G1
A   B   C   D
      )GROUP G1 X Y Z G1 G2
      )GRP G1
X   Y   Z   A   B   C   D   G2
      )GROUP G2 X A F1
      )GRP G2
X   A   F1
      )GRPS
G1  G2
      )GROUP G1
      )GRPS
G2
      )GRP G1
      )GRP G2
X   A   F1
```

)GRPS Listing Names of Groups

Syntax:

)*GRPS* [string1 [string2]]

Parameters:

string1 is any sequence of characters not including blank or carriage return.

string2 is any sequence of characters not including blank or carriage return.

Description:

The)*GRPS* command alphabetically lists the names of groups in the active workspace. Alphabetic ordering is illustrated in the examples. Note particularly the first)*GRPS* command since it indicates where each name character lies in alphabetic order.

If string1 is specified all group names that are alphabetically equal to or greater than string1 are displayed. If string1 is not specified, all group names are displayed. If a string includes more than 79 characters, those past the 79th are ignored. Strings are only used for range demarcation in alphabetic ordering. If string2 is specified, all group names that are alphabetically equal to or greater than string1, and are also less than or equal to string2 are displayed.

Examples:

```
      )GRPS
G   G0  G1  GG  GH  GHI GG̲  G⍙  GΔ  H
      )GRPS GG
GG  GH  GHI GG̲  G⍙  GΔ  H
      )GRPS G GG
G   G0  G1  GG
      )GRPS  GGG  GH
GH
      )GRPS  GHI  GHI
GHI
      )GRPS  A  Z
G   G0  G1  GG  GH  GHI GG̲  G⍙  GΔ  H
```

)IBEX Issuing CP-6 Commands

Syntax:

)*IBEX* message
)!message

Parameters:

message specifies text of a legal IBEX command.

Description:

The)! command directs a string of characters to the CP-6 Command Processor (IBEX) for further processing.

Examples:

```
      )IBEX DI
USERS = 63
ETMF = 1
90ρ RESPONSE < 100 MSECS
DEC 15 '84  15:10
      )!DI
USERS = 63
ETMF = 1
90ρ RESPONSE < 100 MSECS
DEC 15 '84  15:11
```

)LIB   Listing Names of Saved Workspaces

Syntax:

)*LIB* [account]

Parameters:

account   specifies a CP-6 account name.

Description:

The)*LIB* command lists the names of workspaces saved in an account. If a password was saved with a workspace, the workspace name is listed, but not the password.

Examples:

```
      )LIB
APLQUIZ
APLSIDR
PROB1
PROB2
```

Lists names of saved workspaces in the current user's account.

```
      )LIB REI07207
EDITFILE
FACTOR
PAYROLL
```

Lists names of saved workspaces in another account (account REI07207).

)LOAD Retrieving a Saved Workspace

Syntax:

```
)LOAD fid
```

Parameters:

fid is the CP-6 file identifier of a saved workspace.

Description:

The *)LOAD* command causes a copy of saved workspace to be loaded into the user's active workspace. The saved workspace may be retrieved from a user's own account or another account. Note that if a saved workspace is retrieved from another account, the account must be specified in the *)LOAD* command. Also, if the workspace is saved with a password, that password must be included in the *)LOAD* command. In response to a successful load, APL prints a message giving the time and day that the workspace was saved. If the workspace is not found or if a proper password is not used, APL prints the message *WS NOT FOUND*. After a successful *)LOAD* the expression ⍎⎕*LX* is executed.

If a workspace is saved during function definition mode, the *)LOAD* command causes APL to automatically reopen that function and prompt the user to continue function definition or editing. (The user may choose to close the function immediately.) If *)LOAD* accesses a workspace sealed by another user, the workspace is sealed, prohibiting any form of function editing or display.

Examples:

```
      )LOAD KAWA
KAWA SAVED 15:00 DEC 15 '84
```

Loads workspace *KAWA* into the active workspace and prints a save report. Workspace *KAWA* was previously saved in the current user's account.

```
      )LOAD HENRY..SECRET
HENRY SAVED 15:08 DEC 15 '84
```

Loads workspace *HENRY* into the active workspace and prints a save report. Workspace *HENRY* was previously saved with password SECRET in the current user's account.

```
      )LOAD GRANOLA.TESTAPL.PASSWRD
GRANOLA SAVED 15:08 DEC 15 '84
```

Loads workspace GRANOLA into active workspace and prints a save report. Workspace GRANOLA was previously saved with password PASSWRD in account TESTAPL.

)NMS Displaying Global Names

Syntax:

```
)NMS [string1 [string2]]
```

Parameters:

string1 is any sequence of characters not including blank or carriage return.

string2 is any sequence of characters not including blank or carriage return.

Description:

The *)NMS* command alphabetically lists the global names in the active workspace. Alphabetic ordering is illustrated in the examples. Note particularly the first *)NMS* command since it indicates where each name character lies in alphabetic order.

If string1 is specified all global names that are alphabetically equal to or greater than string1 are displayed. If string1 is not specified, all global names are displayed. If a string includes more than 79 characters, those past the 79th are ignored. Strings are only used for range demarcation in alphabetic ordering. If string2 is specified, all global names that are alphabetically equal to or greater than string1, and are also less than or equal to string2 are displayed.

Examples:

```
      )NMS
A.2 A0.2    A1.2    AA.3    B.3

      )NMS AA
AA.3    B.3

      )NMS A1 AX
A1.2    AA.3
```

)OBSERVE Observing Intermediate Results

Syntax:

```
)OBSERVE
```

Description:

The *)OBSERVE* command observes intermediate results developed by APL as it interprets a statement. This could be thought of as a "super-trace" capability. Following an *)OBSERVE* command, the succeeding statement is observed along with any traced function lines that are encountered. Subsequent direct statements are not observed unless the user precedes each of them by a new *)OBSERVE* command. Thus, an *)OBSERVE* command is short-lived, applicable to only one direct statement. By setting trace-vectors for functions to be encountered during an execution, however, the user can observe arbitrarily selected statements until issuing another direct input line.

While an)*OBSERVE* command is in effect, CP-6 APL displays a series of observations. An observation consists of displaying: the current line being executed, a marker (error caret) beneath some character in that line, and the value resulting at that point in execution (empty results, as usual, cause no value to be displayed). The observation marker often marks the leftmost point reached, so far, during execution of the line; however, when a function yields its results, the marker is placed below the function for clarity. (The only exception is the Execute function in which case the "leftmost" rule applies.)

For "observed" lines, observations occur for:

- Each operator result
- Each function result
- Arguments that have not already been observed on this line
- Indexed arguments

Observations are not made for assignments since the assigned value has already been observed prior to the assignment. Observations are also not made for the full variable when it is used as an indexed variable; this eliminates lengthy displays in cases such as the following sample)*OBSERVE* command.

```
        A←1000ρ'GORP'
        )OBSERVE
        B←A[5]          This is the observed line.
        B←A[5]
            ∧
    5                   Observation of the argument 5.
        B←A[5]
          ∧
    G                   Observation of the indexed argument A[5].
```

Note in the above sample that *A* was not displayed. This is fortunate since its display would produce 1000 characters, most of which contribute nothing to the observe statement.

Usage Notes:

The)*OBSERVE* command has three valuable uses: for debugging, for learning how a calculation is performed, and for developing better APL functions. Its value in debugging is obvious. Suppose a complicated APL statement produces a *LENGTH ERR*. By using the *OBSERVE* command and reissuing that statement, the programmer can view development of values leading up to the error and readily see what caused the problem.

The)*OBSERVE* command can be a tremendous timesaver. When presented with a new APL statement or function, the user can spend a great deal of time analyzing how it accomplishes its result. By observing a sample run, the interpretation path and values can be readily inspected, simplifying analysis greatly. The reader might apply this process to the following function.

```
      ∇PRIMESUPTO N;R;I;J
[1]   (∧⌿(0≠(1+R)∘.|J)∨((R←ι⌊N*0.5)∘.=I))/J←1+I←ιN-1∇
```

This function produces the prime numbers from 2 up to the positive integer specified as its argument. To observe this function, the user might proceed as follows:

```
      1 ⎕TRACE 'PRIMESUPTO'  Set to trace line 1 of the function
      )OBSERVE                 Request observations.
      PRIMESUPTO 15            Call the function.
      .                        About 30 observations are made; then the
      .                       )OBSERVE command "disappears".
      .
```

There are at least two ways in which the)*OBSERVE* command can be used to develop better APL functions. First, redundant calculations are made obvious and the programmer can then eliminate such redundancies. The following function is an inefficient version of the *PRIMESUPTO* function. The user might try observing the function to discover how apparent such redundancies become under the)*OBSERVE* command.

```
      ∇PRIMESUPTOO N
[1]   (∧⌿(0≠(1+⍳⌊N*0.5)∘.|(1+⍳N-1))∨((⍳⌊N*0.5)∘.=(⍳N-1)))/1+⍳N-1∇
```

The *PRIMESUPTOO* function takes considerably more execution time (and produces more observations) than the *PRIMESUPTO* function shown previously.

The fact that the *)OBSERVE* command is useful for developing better APL functions is not obvious. It depends on the creativity and imagination of the user. By viewing the manner in which a calculation is carried out, the creative user may recognize patterns that can be more easily produced by other calculations. In other words, observations can suggest alternate approaches to solving a given problem.

One final note about the *)OBSERVE* command should be presented. Suppose the user suspends execution during an observed run by hitting the break key, for instance. This removes the *)OBSERVE* command. Subsequent execution will not be observed unless the user issues a fresh *)OBSERVE*. As stated earlier, this command is short-lived. (At times its short life can be inconvenient, but considering the voluminous output possible with the *)OBSERVE* command this is more often a convenience.)

)OFF Logging Off

Syntax:

)OFF [HOLD]

Parameters:

HOLD requests APL to return to the calling run unit or IBEX.

Description:

The *)OFF* command discards the active workspace, exits APL, and logs the user off of CP-6 (producing the CP-6 log off message). If *HOLD* is specified, the user is not logged off. NOTE: If APL has been called from a run unit, not directly from IBEX, *)OFF HOLD* returns to the calling run unit.

Examples:

```
      )OFF
CON=00:00:36 EX=00:00:00.36  SRV=00:00:01:98  PMME=235  CHG=.44
```

Logs the user off and displays the CP-6 log-off messages.

```
      )OFF HOLD
!
```

Ends APL communication and returns control to IBEX.

)OPR Communicating with Computer Center Operator

Syntax:

)*OPR* message

Parameters:

message is the actual message to the operator; it cannot exceed 254 characters. Note that the operator's console does not include special APL characters, so messages should be limited to ordinary alphanumeric characters.

Description:

The)*OPR* command allows the user to send messages to the operator in the computer center and requests a reply. APL prints the word *SENT* and enters WAIT mode until the user presses the BREAK key or the operator response is received.

Examples:

```
      )OPR  CP-6 UP SUNDAY?
SENT

YES,FOR A WHILE.
```

Illustrates sending message to the operator and receiving a reply.

)OPRN Communicating with Computer Center Operator

Syntax:

)*OPRN* message

Parameters:

message is the actual message to the operator; it cannot exceed 254 characters. Note that the operator's console does not include special APL characters, so messages should be limited to ordinary alphanumeric characters.

Description:

The)*OPRN* command allows the user to send messages to the computer operator, without waiting for a reply. APL responds to this command with the message *SENT* and then is ready for more input.

Examples:

```
      )OPRN  TRIAL MESSAGE. DON'T REPLY
SENT
```

Illustrates sending a message to the operator, with no reply expected.

)ORIGIN Setting Index Origin

Syntax:

)ORIGIN [n]

Parameters:

n is either 0 or 1.

Description:

The *)ORIGIN* command sets or displays the value of ⎕*IO* (index origin). There are two index origins available, 0 and 1. The functions affected are index of and index generator (⍳), indexing and axis operator ([]), grade up (⍋), grade down (⍒), and random number generation (?).

The *)ORIGIN* command causes APL to set the index origin (the value of ⎕*IO*) and to print a message indicating the previous index origin. If the user does not supply parameter n when issuing this command, the current index origin is displayed. Note that the *)ORIGIN* command affects the active workspace and is saved along with a workspace. The index origin can also be changed by assigning a value to the system variable ⎕*IO*.

Examples:

```
      )ORIGIN
IS  1
      )ORIGIN 0
WAS 1
      )ORIGIN 1
WAS 0
```

)PCOPY Copying from Saved Workspace

Syntax:

)PCOPY fid [list]

Parameters:

fid is a CP-6 file identifier of a saved workspace.

list specifies a list of variable names, function names, or group names, separated by blanks.

Description:

The *)PCOPY* command, the Protected Copy command, is the same as the *)COPY* command except that a name is copied only if the name in the active workspace is undefined (see the *)COPY* command).

)QLOAD,)QCOPY, and)QPCOPY Quiet Commands

Syntax:

```
)QLOAD fid
)QCOPY fid [list]
)QPCOPY fid [list]
```

Parameters:

fid is a CP-6 file identifier of a saved workspace.

list specifies a list of variable names, function names, or group names, separated by blanks.

Description:

The *)QLOAD*, *)QCOPY*, and *)QPCOPY* commands are slight variants of the *)LOAD*, *)COPY*, and *)PCOPY* commands. The *Q* stands for quiet. The SAVED message normally shown at the conclusion of a load or copy is suppressed on a quiet load or copy. No other messages (i.e., error diagnostic) are suppressed by the quiet commands.

Certain APL application programs benefit from the quiet commands, programs that use execute-operations to load or copy without user intervention. The user is unaware that such *)LOAD* or *)COPY* commands are executed, and would be puzzled by SAVED messages.

The quiet commands should be inserted in programs only after the application is well tested. In the event of an error subsequent to a quiet load or copy, it may be difficult to isolate the problem for lack of knowledge about the workspace environment.

)QUIT Leaving APL

Syntax:

```
)QUIT
```

Description:

The *)QUIT* command causes the contents of the active workspace to be discarded, following which control is passed to the process that invoked APL (usually IBEX, the CP-6 Command Processor). The command is identical to the *)END* command.

)REPORT   Selecting the Function Stop Display

Syntax:

)REPORT [FUNC[TION]|LINE]

Parameters:

FUNCTION   sets function stop display to function name and line number (default).

LINE   sets function stop display to function name, line number and contents of the line.

Description:

The *)REPORT* command is used to control the information displayed when a function stop occurs. The default display is the function name followed by the line number in brackets. Specifying the *LINE* parameter causes APL to also display the contents of the line at which execution stopped.

Examples:

```
      )REPORT LINE
WAS FUNCTION
      ∇FUN
[1]   1+1+2
[2]   ⍝  COMMENT
[3]   ∇
      2  1  ⎕STOP 'FUN'
         FUN
FUN[1]  1+1+2
      )REPORT FUNCTION
WAS LINE
      →⎕LC
4
FUN[2]
      →⎕LC
```

)SALVAGE   Copying from Saved Workspace

Syntax:

)SALVAGE fid [list]

Parameters:

fid   is a CP-6 file identifier of a saved workspace.

list   specifies a list of variable names, function names, or group names, separated by blanks.

Description:

The *)SALVAGE* command retrieves information from a workspace which the *)LOAD* or *)COPY* command report as broken. The process is identical to that performed for the *)COPY* command except that the current workspace must be CLEARed prior to issuing the *)SALVAGE* command, and any items in disrepair are not copied.

If the file contents which describes the major structure of the workspace is defective, the *)SALVAGE* command will still be terminated with a broken workspace report.

)SAVE Saving a Workspace

Syntax:

)SAVE [fid]

Parameters:

fid is a CP-6 file identifier. Like other APL names, the workspace name can consist of one or more letters from a to z, or A to Z, or numbers. Unlike other APL names, a workspace name is limited to 31 characters. If omitted, the fid defaults to the current workspace name (set by the *)WSID* command).

Description:

The *)SAVE* command saves a copy of the active workspace. If the active workspace was loaded as a sealed workspace, the workspace cannot be saved by *)SAVE* or *)CONTINUE* commands. Attempts to do so will result in a *BAD FILE REF* error. A word of caution is necessary about using passwords in the *)SAVE* command. If a saved workspace already exists with a given name and password, specifying the same name with a new password in the *)SAVE* command will not change the password. Instead it results in the error message *BAD FILE REF*. The previously passworded workspace must be deleted before a new version can be saved. To delete the old workspace, use the *)DROP* command with the name and password. The workspace is saved provided that file management write access is available to the current user for the fid (expressed or implied).

When a workspace is saved while in the direct input mode (not in evaluated input, function definition, or execute modes), the variable ⎕*LX* will be executed when that workspace is subsequently loaded.

When a workspace is successfully saved, APL prints a save report giving the name of the workspace and the time and date of the save. The *)SAVE* command also updates the current workspace identification, i.e., WSID. The name of the saved workspace along with its password (if any) becomes the WSID for the active workspace. If the workspace cannot be saved because it exceeds the available space in the user account, APL prints an error message. In this case, the user must delete some workspaces or other files from the account before saving any APL workspace.

If a *)SAVE* command is issued during function definition mode, the currently open function is temporarily closed. The saved workspace carries an indication that the function should be reopened on *)LOAD*. After the *)SAVE* command, APL reopens the function and prompts the user to continue function definition or editing.

Examples:

```
    )SAVE GRANOLA..SECRET
GRANOLA SAVED 14:58 DEC 15 '84
```

Saves a copy of the active workspace with specified workspace name and password, and produces a save report.

```
    )SAVE
CONTINUE SAVED 14:59 DEC 15 '84
```

Saves a copy of the active workspace and produces a save report.

```
    )SAVE KAWA
KAWA SAVED 15:00 DEC 15 '84
```

Saves a copy of the active workspace named KAWA and produces a save report.

)SEAL Saving a Sealed Workspace

Syntax:

)SEAL [fid]

Parameters:

fid is a CP-6 file identifier of the workspace to be sealed.

Description:

The *)SEAL* command is identical to the *)SAVE* command, but in addition the workspace is created with READ access to all accounts using APL as the execution vehicle. Thus, the user who creates the workspace has unrestricted access to the it. All other users can only access the workspace with APL. If the save is not successful, *BAD FILE REF*, *FILE SPACE TOO LOW*, or other relevant error messages may be issued and the active workspace will not be sealed.

)SET Changing Assignments of Input/Output Streams

Syntax:

)SET dcb fid

Description:

Refer to the CP-6 Programmer Reference Manual (CE40) for a complete description of the SET command in CP-6 IBEX. Any *)SET* command is passed to IBEX for processing with only a minor change to the DCB designation as noted below.

The inclusion of the *)SET* command within APL permits error action by the user or, if error control is used, in APL functions. If any of the listed strings noted below are detected by APL, the corresponding substituted string is substituted before referral to IBEX:

STRING	REPLACEMENT STRING
INPUT	#1
OUTPUT	#4
⎕0, ⎕1,...,⎕9	F$Q0, F$FQ1,...,F$Q9

User Prompts

If either output or input is diverted from the terminal, the prompts normally issued to the user are omitted. On ASCII terminals with full duplex, the echoing of characters indicates that input is being accepted. The home device for on-line sessions is the terminal. In batch the home devices are the command stream (card reader) for input, and line printer for output.

Echoing of Input

If input is coming from somewhere other than the user's terminal, then APL input (but not blind-input) is echoed to the output setting depending upon the IBEX ECHO setting. If input is echoing and this is not desired then the APL command:

```
      )!DONT ECHO
```

should be issued. Conversely, echoing may be initiated by the APL command.

```
      )!ECHO
```

Errors on Input or Output

If normal input or output is reassigned from the home device by the *)SET* command and an I/O error occurs, the input (or output) setting is returned to the home device(s). This is the user's terminal for on-line users, the card reader and line printer for batch. If error control is not in effect, an I/O error message is then output. If control is in effect for I/O errors, no error message is output. The user's error control function should note that input and output have been restored 'home' from their *)SET* command assignments.

Break Response

If normal input or output is reassigned from the home device by a *)SET* command, it is restored to the home device by a break. If the user has taken break control, the function which manages break control should note that input and output have been restored to the home device and terminal.

)SI Controlling the State Indicator

Syntax:

```
)SI  [ON|OFF|CLEA[R]]
```

Parameters:

CLEA[R] removes every entry in the state indicator. This may free a substantial amount of workspace and is a valuable tool for recovering from *WS FULL* errors.

ON suspends the executing function if an error occurs (the default). This is useful when debugging the workspace since it allows access to local variables for the suspended function. Suspending the function, however, expends a certain amount of the active workspace, and this can be a disadvantage. The *ON* option sets state indicator control for errors that may occur during subsequent execution of functions in the active workspace.

OFF sets state indicator control to avoid suspending a function when an error occurs. Note that the *OFF* setting applies only to errors. It has no influence over execution breaks or stop vectors; these may still cause function suspension. The *OFF* option sets state indicator control for errors that may occur during subsequent execution of functions in the active workspace.

Description:

The *)SI* command displays the contents of the state indicator, which is a list of suspended and pendent functions. For a discussion of the state indicators and suspended and pendent functions, see State Indicators in Section 7.

Examples:

```
      )SI
A[2]  *
XY[5]
B[3]  *
```

The most recently suspended function is listed first. An asterisk after an entry indicates a suspended function; no asterisk indicates a function that is pendent. In the above example, function *A* has been suspended just before line number 2, and function *B* just before line number 3. Function *XY* is pendent because it referenced function *A* at line number 5. If *)SI* is issued when evaluated input is pending, the input request will also be displayed, using the ⎕ character. If the *)SI* command is issued when an 'execute' is pending, the execute state will be indicated, using the ⍎ character.

Errors causing suspended function should be corrected as soon as possible. Suspended functions can be cleared from the state indicator with the branch arrow (→). (Remember that the state indicator with its list of suspended and pendent functions and local variables may take up a lot of workspace.) Each branch arrow clears the most recent suspended function and all pendent functions associated with it. This can be repeated until all suspended and pendent functions have been cleared; that is, until the *)SI* command returns a blank line. Applied to the above example, this would give

```
      →
      )SI
B[3]  *
      →
      )SI
```

A more convenient method for clearing the state indicator is to issue the following command:

```
      )SIC
```

To restore normal state indicator control, the command

```
      )SI ON
```

may be issued. This setting also occurs automatically if a *)CLEAR* command is issued. The *ON* or *OFF* state indicator control is saved when the active workspace is saved, and loaded when the workspace is loaded. Copying does not alter the control of the active workspace.

)SIC Clearing the State Indicator

Syntax:

)SIC

Description:

The *)SIC* command removes every entry in the state indicator. This may free a substantial amount of workspace and is a valuable tool for recovering from *WS FULL* errors.

)SIL Listing the State Indicator Lines

Syntax:

)SIL

Description:

The *)SIL* command lists the same information as the *)SI* command and also lists the contents of the lines that are currently in execution. For pendent functions, it indicates the position within each line at which execution is to be resumed. An * in column 1 indicates a direct input line in execution.

Examples:

```
      )SIL
A[2]  CC←1  20 BB
XY[5] R←100×L(A N)÷100
              ^
*     XY
      ^
B[3] VAL←(PAY≥100)/PAY ◊ PAY←⎕FREAD  1  20
*     B
      ^
```

)SINL Listing the State Indicator

Syntax:

)SINL [ON|OFF|CLEA[R]]

Parameters:

CLEA[R]   removes every entry in the state indicator. This may free a substantial amount of workspace and is a valuable tool for recovering from *WS FULL* errors.

ON   suspends the executing function if an error occurs (the default). This is useful when debugging the workspace since it allows access to local variables for the suspended function. Suspending the function, however, expends a certain amount of the active workspace, and this can be a disadvantage. The *ON* option sets state indicator control for errors that may occur during subsequent execution of functions in the active workspace.

OFF sets state indicator control to avoid suspending a function when an error occurs. Note that the *OFF* setting applies only to errors. It has no influence over execution breaks or stop vectors; these may still cause function suspension. The *OFF* option sets state indicator control for errors that may occur during subsequent execution of functions in the active workspace.

Description:

The *)SINL* command lists the same information as the *)SI* command and additionally lists the local variable names appearing in the suspended and pendent functions. For a discussion of the state indicators and suspended and pendent functions, see State Indicators in Section 7.

Examples:

```
      )SINL
A[2]  *     BB   CC   DD
XY[5]
B[3]  *     PAY  VAL
```

where *BB*, *CC*, and *DD* are local variables appearing in function *A*; and *PAY* and *VAL* are local variables appearing in function *B*.

Errors causing suspended function should be corrected as soon as possible. Suspended functions can be cleared from the state indicator with the branch arrow (→). (Remember that the state indicator with its list of suspended and pendent functions and local variables may take up a lot of workspace.) Each branch arrow clears the most recently suspended function and all pendent functions associated with it. This can be repeated until all suspended and pendent functions have been cleared; that is, until the *)SI* command returns a blank line. Applied to the above example, this would give:

```
      →
      )SINL
B[3]  *       PAY    VAL
      →
      )SI
```

)STEP Single Step Execution

Syntax:

)STEP [LINE|FUNCTION] [n]

Parameters:

n indicates the number of statements to step. This can be any integer from 1 to 99999.

LINE specifies to stop before the next APL line is executed within any function.

FUNCTION specifies to stop before the next line is executed within the current function.

Description:

The *)STEP* command executes the line at the top of the state indicator and stops before another function line is executed. That is in the simplest case where the current line does not call another user function, the line will be executed and execution will halt before executing the next line. A single right bracket and a carriage return on a line also has this effect.

Examples:

```
      )STEP LINE
      FUN 1
FUN[1]
      )STEP
FUN[2]
      )STEP
FUN2[1]
      )STEP
FUN[1]
      )STEP
FUN[2]
      )STEP FUNCTION
FUN[1]
```

)TERMINAL Specifying Input/Output Device

Syntax:

)TERMINAL [INPUT|OUTPUT] [n]

Parameters:

INPUT specifies that only the input translation tables are to be affected.

OUTPUT specifies that only the output translation process is to be affected.

n indicates the device to be assumed by APL and can be any of the values 1, 2, 3, 4, 5, 13, 14.

Description:

The *)TERMINAL* command is used to identify to APL the input/output device being used. This command is not normally needed for users operating on a terminal or submitting batch runs for card input and line printer output. New terminal declarations are acceptable at any time during an APL session, but the user should be aware of the consequences (such as error message discrepancies and input/output translation problems). ⎕*TT* also results in the integer n; this may be useful for APL programs that are sensitive to terminal type. Using *)TERMINAL INPUT* or *)TERMINAL OUTPUT* modifies only the specified (input or output) translation table. This form is useful when APL input or output is diverted to an alternate device by the *)SET* command.

CP-6 supports two types of input/output devices with respect to APL sessions:

- o Those capable of printing the APL character set.
- o Those capable of printing the ASCII-96 character set.

Specifying a terminal of type 1, 13, or 14 indicates the APL character set; types 2, 3, 4, or 5 specify the ASCII set with types 4 and 5, representing underscored letters as lowercase letters. Types 2 and 3 represent underscored letters as the mnemonic combination $U followed by the letter. For types 3 and 5, certain characters (◊, ≤, ≥, ≠, ⌿ and ⍀) are represented via appropriate backspace overstrike combinations.

Examples:

```
      )TERM 5
WAS 1
```

Indicates that a Diablo 1620 terminal (or equivalent) with a non-APL daisy wheel is being used.

```
      )TERM
IS 5
```

Shows that the non-APL Diablo 1620 terminal was most recently declared.

```
      )TERM OUTPUT 4
WAS 5
```

Sets output translation for the line printer, but does not change input translation.

APL recognizes three separate choices for input/output character translation.

Input terminal type is changed by either *)TERM INPUT* n or *)TERM* n. Output terminal type is changed by either *)TERM OUTPUT* n or *)TERM* n.

Usage Notes:

The combination of the *)TERMINAL* and *)SET* commands permits a variety of I/O operations with devices and files. The user should be warned that some choices, particularly changes to the home terminal, can result in difficulties carrying on further terminal communications. In general, *)TERM OUTPUT* 4 should be used for line printer output. Output which is filed and reread by the same user should preferably use home terminal type. If several users with different terminals want to access the file, a common type should be agreed on, probably 1 for APL or 4 for ASCII.

)VARS Listing Global Variable Names

Syntax:

)VARS [string1 [string2]]

Parameters:

string1 is any sequence of characters not including blank or carriage return. If string1 includes more than 79 characters, those past the 79th are ignored.

string2 is any sequence of characters not including blank or carriage return.

Description:

The *)VARS* command alphabetically lists the names of global variables in the active workspace. Strings are used only for range demarcation in alphabetic ordering. If string1 is not specified, all global variable names are displayed. If string2 is specified, global variable names that are alphabetically equal to or greater than string1 and are also less than or equal to string2 are displayed. Alphabetic ordering is illustrated in the example. Note particularly the first *)VARS* command since it indicates where each name lies in alphabetic order.

Examples:

```
      )VARS
A    A0  A1  AA  AB  ABC AA̲  A⍙  AΔ  B

      )VARS AA
AA   AB  ABC AA̲  A⍙  AΔ  B

      )VARS A AA
A    A0  A1  AA

      )VARS AAA AB
AB

      )VARS ABC ABC
ABC

      )VARS A Z
A    A0  A1  AA  AB  ABC AA̲  A⍙  AΔ  B
```

)WIDTH   Setting Line Width

Syntax:

)*WIDTH* [n]

Parameters:

n   is an integer number ranging from 32 to 390.

Description:

The)*WIDTH* command changes or displays the value of ⎕*PW* (Platen Width). This system variable is used to indicate the length of the longest line that APL will output. In a clear workspace the platen width defaults to the platen width when APL was initially invoked, or the closest value acceptable by APL to the initial value. The value of ⎕*PW* is saved with a workspace and is restored when a workspace is loaded.

Examples:

```
      )WIDTH
IS 120
```

Displays the current width of a line output (i.e., 120 printing positions).

```
      )WIDTH 50
WAS 120
```

Changes the width of an output line to 50 print positions. The previous line width setting was 120.

)WSID Identifying the Active Workspace

Syntax:

```
)WSID [fid]
```

Parameters:

fid is the new CP-6 file identifier of the active workspace. If fid is not specified, the fid of the active workspace is displayed. APL responds with a message showing the previous workspace name. This name can be from 1 to 31 characters.

password a password may be specified, but a previous password is never displayed.

Description:

The *)WSID* command allows the user to identify the active workspace or to change its name. The *)WSID* command cannot be used to change the name of a sealed workspace.

Examples:

```
      )WSID
IS JONES
```

Lists the name (JONES) of the active workspace.

```
      )WSID
IS GOSTYLE.ZZZ02MAR
```

Lists the name GOSTYLE and account ZZZ02MAR of active workspace.

```
      )WSID SMITH
WAS GOSTYLE.ZZZ02MAR
```

Changes the name of the active workspace from GOSTYLE to SMITH.

Section 9

Report Formatting

CP-6 APL provides a formatted output capability with the system function ⎕*FMT* in addition to the ⍕ function. ⎕*FMT* utilizes a set of format control phrases that are applied to a list of APL expressions. Each APL expression may evaluate to numeric or character scalars, vectors, or matrixes. The format control phrases, called format specifications, are described in Table 9-1.

Format Specifications

⎕*FMT* recognizes twelve data format codes. Each code is described in the following table.

Table 9-1. Format Specifications

Code	Description
A	Alphanumeric specification.
E	Floating-point with exponent (scientific format).
F	Floating-point to fixed decimal position.
I	Decimal integer.
X	Blank insertion.
T	Column Tabbing.
G	Picture formatting.
⍞ *TEXT* ⍞	Text insertion.
⎕ *TEXT* ⎕	Text insertion.
< *TEXT* >	Text insertion.
⊂ *TEXT* ⊃	Text insertion.
¨ *TEXT* ¨	Text insertion.

Format specifications may be in any of the following forms:

[r] *A*w

[r] *E*w.s

[r] [q] *F*w.d

[r] [q] *I*w

[r] *X*w

[r] ⍞ *TEXT* ⍞

T c

```
[r] [q] G ⍞ TEXT ⍞
```

where

r   is an optional unsigned integer constant indicating the specification is to be repeated r times. When r is omitted, it is taken as 1.

w   is an integer constant indicating the total field width, or number of print positions occupied by the formatted value (or blanks, for *X* type).

s   is an integer constant indicating the number of significant digits to be printed in *E* format; s must be less than w-5.

q   is an optional "qualifier" or "affixture" code used to position and affix characters to the results of *I* and *F* output forms. The available codes and their uses are described later in this section.

d   is an integer constant indicating the number of digits to the right of the decimal point in *F* formats; d must be less than w.

c   is the column number at which the next field will start to be formatted.

Format Specifications versus Data Types

Format *A* may be applied to character data only. Formats *E*, *F*, and *I* may be applied to numeric data only.

Arrays with rank above 2 (matrix) cannot be processed. If a value cannot meaningfully be expressed in the format and field width specified, the field is filled with asterisks.

Format Statement (Left Argument)

A format statement is the left argument of ⎕*FMT*, operating on data values in the right argument. The format statement consists of a character vector made up of one or more format specifications separated by commas. The left argument of ⎕*FMT* must always be a valid format statement. For example,

```
    '3I3,2E8.2,X12,I3' ⎕FMT ...
```

Parenthesis may be used with repeat counts around phrases. For example:

```
    'I5,3(I2,F8.2)' ⎕FMT ...
```

Parenthesis may be nested up to 7 deep within format phrases.

Format Data List (Right Argument)

The right argument of ⎕*FMT* must be a list of APL variables or expressions, separated by semicolons. The expression may represent scalars, vectors, or arrays. For example,

```
     ... ⎕FMT (VARIABLE1;VARIABLE1+VARIABLE2;'SUM')
```

If the list contains only one value, the parentheses may be omitted. The value of each expression must be simple and all numeric or all character.

Operation of ⎕*FMT*

⎕*FMT* uses the format specifications in its left argument (the format statement) to control printing of its data list (right argument) on one or more columns. The syntax is

```
            'format stmt' ⎕FMT expr
    or
            'format stmt' ⎕FMT list
```

The result of executing ⎕*FMT* is one or more "lines" of formatted character data. A line may be as long as workspace allows. In printing, long lines are broken up according to the ⎕*PW* setting. If more than one line is produced (as will be the case if the data list includes vectors or arrays with more than one row) all lines are of the same length. The result, then, is a character matrix.

If ⎕*FMT* is not used within a larger expression, the amount of temporary workspace required is only the length of one line. Thus, formatted output may be used to process output that would overflow available workspace if assigned or used in its entirety. If ⎕*FMT* is used within a larger expression, the result is always a matrix, even if only one line, and space for the full matrix is required. The operation of ⎕*FMT* on various right arguments is described below.

Formatting Scalar Arguments

If the data list consists exclusively of scalars, a single line is created. Format specifications are used in turn to process elements of the data list in left to right order. Blank insertion and text insertion specifications do not "use up" elements of the data list, however. A repetition indicator causes a particular specification to be applied the designated number of times to successive elements of the data list. If there are fewer format specifications (counting repetition indicators) than values to be formatted, the list of format specifications is reused as necessary until the data list is exhausted.

Examples:

```
      'I3,A5,X5' ⎕FMT (100;'A';200;'B')
100    A      200    B

      '5F5.2' ⎕FMT (1;10;100;¯10;¯1)
 1.0010.00**********¯1.00
```

This last example illustrates the use of the repetition indicator. Also note the asterisks indicating that the value 100 and ¯10 would not fit in the specified format.

Formatting Vector Arguments

If the data list includes vector and scalar arguments (or vectors only), the number of lines generated equals the length of the vector with the most elements. Each vector creates a "column" in the resulting character array. The columns of shorter vectors or scalars are extended by blanks.

Examples:

```
      'E11.4' ⎕FMT  3.1 .123 ¯1.234 5678
 3.100E0
 1.230E¯1
¯1.234E0
 5.678E3

      '2I5,A2'⎕FMT (1 2 3 4;10.4 10.6;'ABCDEF')
    1   10 A
    2   11 B
    3      C
    4      D
           E
           F
```

In the last example, note the rounding off of values as required for I format specifications, and also note the different column lengths.

Formatting a Vector on One Line

The normal result for ⎕FMT on vector arguments is columnar formatting, but it is often desirable to create a formatted row for vectors. There are two ways this can be done:

- Ravel the result of ⎕FMT. This method is appropriate if the result contains a single column.

  Examples:

  ```
        ,'A2'⎕FMT 'DOUBLE SPACE'
   D O U B L E   S P A C E

        ,'I5'⎕FMT .14 1.4 14 140 1400
      0    1   14  140 1400
  ```

- Reshape the vector as a 1 by N matrix. (This method uses a property of the operation of ⎕FMT on matrixes, as discussed below.)

  ```
        V←'TRIPLE+SPACE'
        'A3' ⎕FMT (1,ρV)ρ V
    T  R  I  P  L  E  +  S  P  A  C  E
  ```

Formatting Matrix Arguments

If the data list includes a matrix argument, each column of the matrix occupies a column in the formatted output. Each row of the matrix creates an entry on a "line" of output. Note that a 1 by N matrix creates a single row, and an N by 1 matrix creates the same output form as an N element vector.

In essence, ⎕FMT outputs matrixes in the same shape as unformatted output would, but permits control of decimal placement, column positioning, etc.

Examples:

```
      IOTA←3 5ρι15
      'F5.1'⎕FMT IOTA
  1.0  2.0  3.0  4.0  5.0
  6.0  7.0  8.0  9.0 10.0
 11.0 12.0 13.0 14.0 15.0
```

```
      JKL←'JKL'
      PI←3.14
      VECT←1 2 3 4
      MAT←2 2⍴.1 2.0 30 ¯4
      'A1,F6.3,I5,2F6.1' ⎕FMT (JKL;PI;VECT;MAT)
J 3.140    1   0.1   2.0
K          2  30.0  ¯4.0
L          3
           4
```

Picture Format

Picture formatting provides the greatest control over the result of numeric formatting. The syntax of a picture format is:

    [r] [g] G ⍞ TEXT ⍞

The *B*, *C*, *L* and *Z* format qualifiers are not permitted with picture formatting. The text field may contain any text. The two characters, '9' and '*Z*' by default control the formatting of the numeric data. The data may be scaled by *K* qualifiers and then rounded to integer for formatting.

A '9' in the text field selects the corresponding digit from the data. A '*Z*' in the text field selects the corresponding digit from the data only if the digit is not a leading or trailing zero.

As the text is scanned, characters other than 9 or *Z* are copied to the result. If there are leading or trailing *Z* controls, non-special characters are copied into the result only if the last *Z* selected a digit.

Examples:

```
      'K2G<ZZZ,ZZZ DOLLARS, ZZ CENTS>' ⎕FMT 31415.962
 31,415 DOLLARS, 96 CENTS

      'G⍞99/99/99⍞' ⎕FMT 52282
05/22/82
```

Forms of Output Values

The following rules determine spacing and content of output fields for various format specifications.

- Right-justification. For *A*, *I*, and *F* specifications, the value is right-justified in the field and preceded by blanks where appropriate to fill out the field.

- *E* format. The letter *E* always occupied the fourth space from the right in the field. Three spaces are reserved for the exponent value and exponent sign. If less than three spaces are needed, the right-most space or spaces are blank. In this format, there is columnar alignment of the decimal points and letter *E*.

- ⍞ *TEXT* ⍞ format. Characters between the quote-quads (or other text insertion format specifications) are inserted directly into the output line. There are as many insertions as there are lines of output. No data list elements are expended by text insertion.

- Significance of results. The value of ⎕*PP* is ignored in ⎕*FMT* output; a maximum of 20 significant positions are displayed, however. If a format specification requests more than 20 significant digits, digits beyond the eighteenth, and to the left of the decimal point are replaced by zeroes. Excess digits to the right of the decimal point are replaced by blanks.

- Field width. If field widths are too small to hold formatted values according to the specification, the fields are filled with asterisks.

Format Qualifier and Affixture Codes

I and *F* format specifications may be immediately preceded by one or more qualifier or affixture codes.

- Qualifier codes

  B Leaves the field blank if the result would otherwise be zero.

  C Inserts commas between triads of digits in the integer part of the result.

  L Left-justifies the value in the result field.

  Z Fills unused leading positions in the result with zeros (and commas if *C* is also used) instead of blanks.

  *K*n Scaling factor. Before formatting, the data is multiplied by the power of 10 indicated by n. n may be any positive or negative integer.

- Affixture codes

  M< *TEXT* > prefixes negative results with the text instead of the negative sign.

  N< *TEXT* > postfixes negative results with the text.

  P< *TEXT* > prefixes positive results with the text.

  Q< *TEXT* > postfixes positive results with the text.

  R< *TEXT* > presets the field to the text, which is used as many times as necessary to fill the field. The text is replaced in parts of the field filled by the result.

  S< *TEXT* > symbol substitution.

  Note: If *B* and *R* are both specified, *R* overrides *B*.

  Qualifier and affixture codes do not extend field widths. The modified result must fit in the field width specified or asterisks will be substituted.

  N and *Q* affixtures, since they postfix the text, shift results to the left by the number of characters to be postfixed.

Examples:

```
      V←128 0 ¯.25 ¯64 ¯12345.67
      'BF10.1,X2,BI8,X2,CI10,X2,LI9' ⎕FMT (V;V;V;V)
     128.0       128         128  128
                               0  0
      ¯0.3                     0  0
     ¯64.0       ¯64         ¯64  ¯64
  ¯12345.7    ¯12346     ¯12,346  ¯12346

      'ZF10.2,X2,M⍞**⍞F10.1,X2,P<+>I8' ⎕FMT (V;V;V)
0000128.00       128.0      +128
0000000.00         0.0        +0
¯000000.25       **0.3        +0
¯000064.00      **64.0       ¯64
¯012345.67   **12345.7    ¯12346

      'Q<+++>I9,X2,R<*>I8' ⎕FMT (V;V)
   128+++  *****128
     0+++  *******0
     0+++  *******0
      ¯64  *****¯64
   ¯12346  **¯12346
```

Combinations of qualifier and affixture may be used together to provide various output forms as shown below.

```
      'M<¯$>P<$>CF12.2' ⎕FMT (12345.67;¯9.98)
  $12,345.67      ¯$9.98
```

Format Symbol Substitution

⎕*FMT* uses predefined characters in formatting output and interpreting specifications. In some cases, the default characters may not be appropriate. The *S* qualifier allows these defaults to be changed. The default symbols and their applicable format phrases are listed in the following table.

Table 9-2. Default Formatting Symbols

SYMBOL	USES	PHRASE
9	digit select	*G*
Z	zero suppress digit select	*G*
*	field overflow	*FGI*
0	Z qualifier fill/lead zero fill	*FGI*
_	non-significant digit	*FGI*
,	C qualifier character	*FI*
.	decimal point	*F*

The default character can be replaced by first specifying the default character followed by the character to be used in its place.

Examples:

```
      'S<.,,.>CF16.2' ⎕FMT 2718235.49
    2.718.235,49
```

Format Result

The principal use of ⎕*FMT* is to provide lines of formatted output. However, if ⎕*FMT* is used as part of a larger APL expression, the result of executing ⎕*FMT* is a character matrix which may be manipulated and used just as any other character matrix.

Format Error Reports

If the right argument includes an array of higher order than matrix, or the left argument is not a vector, a *RANK ERR* results.

If the left argument is not simple and all character data, and contains no format specifications, or contains a format specification with inconsistent parameters (such as d greater than w, or w = 0), a *DOMAIN ERR* results.

If there is incorrect syntax in the right argument, a *SYNTAX ERR* results. If there is incorrect syntax in the left argument, a *FORMAT SYNTAX ERR* results.

If the line length of the result is too big for the remaining workspace, or ⎕*FMT* is included in a larger expression and the total result exceeds the remaining workspace, *WS FULL* results.

Formatting Aids

In addition to ⎕*FMT*, the following system functions may be used to aid in output formatting. The)*SET* and)*TERMINAL* commands, described in Section 8, may also be used in the overall process of output report generation.

⎕*PGE* Function (Skip to New Output Page)

Syntax:

```
⎕PGE
```

Description:

⎕*PGE* is a niladic function with an empty vector result. When executed, if output is to a printing device, the current page will be ejected. If output is to a unit record type device and ⎕*HDR* has been established by the)*SET* command, a standard header line will also be produced.

⎕*NLS* Function (Number of Lines Remaining)

Syntax:

```
I←⎕NLS
```

Parameters:

I is a simple integer scalar.

Description:

⎕*NLS* is a niladic function with an integer result. If output is to a device with line count applicable, the result is the number of lines remaining to print on the current output page. If not, the result is zero.

⎕*HDR* Function (Set Page Heading)

Syntax:

```
⎕HDR T
```

Parameters:

T is a simple character scalar or vector of maximum length 160.

Description:

This function establishes the output header line which will be displayed at the start of each page if output is set to a printing device. This system function uses a CP-6 facility which does not recognize special APL characters. If special characters or overstrikes are included, ⎕*HDR* may not produce correct headings.

Possible errors:

A *DOMAIN ERR* is reported if:

o *T* is not text or simple.

A *RANK ERR* is reported if:

o *T* is not a scalar or vector.

A *LENGTH ERR* is reported if:

o *T* contains more than 160 items.

⎕*VFC* Function (Set Line Spacing)

Syntax:

```
    ⎕VFC C
```

Parameters:

C is a simple character scalar or one-item vector.

Description:

⎕*VFC* is a monadic function with empty vector result. The right argument must be a single character. When ⎕*VFC* is executed, the character in the right argument becomes the vertical format control character for the next print line. After that line is printed, the default character is restored as the vertical format control character. Refer to the CP-6 Programmer Reference Manual (CE40) for the specific values of vertical format codes.

⎕*XL* Function (Translate Text)

Syntax:

```
      R←A ⎕XL B
```

Parameters:

A is a simple character vector of length less than 513 (unspecified character positions are treated as blanks).

B is a simple character array.

R is the translated result with the same shape as *B*.

Description:

The ⎕*XL* function facilitates special character set translations within APL. The result of the ⎕*XL* function has the same shape as the right argument and consists of a translation of the right argument. The index position in ⎕*AV* of each item of the right argument is used to index the left argument to obtain the corresponding result item. The result is exactly equivalent to:

```
      (513↑A)[⎕AV⍳ B]
```

but requires much less workspace.

This feature is designed to overcome problems encountered in character set differences between various devices. It allows any character mapping, including mapping several characters to the same result character. An example of this use might be to map all 'illegal' characters to some unique character. Another example is as follows:

```
      L←¯1+⍳256
      L[97+⍳26]←64+⍳26
      L←⎕AV[⎕IO+L]
```

This value of *L*, used as left argument of ⎕*XL*, converts lowercase letters in the right argument to similar uppercase letters in the result.

Possible errors:

A *RANK ERR* is reported if:

o *X* is not a vector.

A *LENGTH ERR* is reported if:

o *X* contains more than 512 items.

A *DOMAIN ERR* is reported if:

o any item of *X* is not a scalar character.

A *DOMAIN ERR* is reported if:

o the right argument *T* contains an item which is not a scalar character.

Section 10

Execution Stops

Execution is stopped if any of the following conditions occurs:

1. Execution is completed (a normal stop).
2. Execution break occurs (BREAK key is pressed), and sidetracking does not occur.
3. User input is required (quad or quote-quad input).
4. Stop control line is encountered.
5. Error is encountered, and sidetracking does not occur.

Normal Stop

Execution comes to a normal stop after any action indicated by direct input is completed. It should be noted, however, that a direct input prompt does not necessarily mean that all pending execution is completed. The user can determine whether any execution is pending via the)*SI* command.

Execution Break

An execution break (that is, the BREAK key) can be issued by the user at any time. There may be a short delay until output stops. Sidetracking can be used to gain break control within an APL function; in this case execution does not stop, but is "diverted" (see Sidetracking on Errors and Breaks).

Either a soft break or interrupt may be signalled by pressing the BREAK key. The first break during the execution of a line of APL is a soft break. Execution of the current line continues until the end of the line is reached at which time, the currently executing defined function will suspend. If a second break is sent before the soft break is processed, this signals a hard break or interrupt and the currently executing line is removed from execution. An *INTERRUPT* message is displayed along with the line that was in execution and a caret indicating the position in the line that execution was interrupted. In this case, if the APL line contains "side effects" such as embedded assignments or shared variable accesses, then the line may not be easily restartable.

APL's reaction to break also depends on whether the BREAK key is pressed during execution mode or definition mode. If break is used during (non-function) execution, APL stops any output in progress and skips to the next line and indents six spaces to prompt for new input. If break is used during execution of a defined function, APL displays the function name and the line number being executed.

If break is used during display of a function, APL will exit from function definition mode if a closing del was included in the display command. If the display command did not have a closing del, APL will remain in function definition mode and will prompt with the next line number after the line range being displayed.

Execution breaks are usually not allowed to interrupt the execution of a system command. However, those that produce lengthy display can be stopped:)*FNS*,)*GRPS*,)*LIB*,)*SI*,)*SINL*, and)*VARS*. Break is also used to abort the wait resulting from the)*OPR* system command.

Stop For User Input

Execution may be stopped by an input request in a line. The normal response to a quad or quote-quad input request is a line of input. While quad input is pending, BREAKS are treated as normal execution errors and thus cause the quad input request to be re-issued. If the user's program contains a loop such that the user is repeatedly prompted for input, the user may escape as follows:

1. For quad input, type a branch arrow (→) followed by a RETURN. An example is shown below:

```
          ∇PITIMES[⎕]∇
        ∇ PITIMES
    [1]   ○⎕
    [2]   →1
        ∇
          PITIMES
    ⎕:
          1
    3.141592654
    ⎕:
          ¯1
    ¯3.141592654
    ⎕:
          )SI
    ⎕
    PITIMES[1]
    ⎕:
          →
          )SI
```

   In this example the user has defined a function, *PITIMES*, that repeatedly requests input and provides a result. The first *)SI* command shows that an input request and line 1 of *PITIMES* are pendent. After the →, the input request is no longer repeated. The second *)SI* command shows that the loop has been broken and *PITIMES* is no longer in use.

2. For quote-quad input, press the BREAK key twice to cause an *INTERRUPT* at the point of the quote quad input request.

Stop Control Vector

As described in Section 7 (under Suspending Execution), a stop control vector can be used to specify the exact place a function suspension is to occur. The user can set a stop control vector by executing the ⎕*STOP* system function with the function name enclosed in quotes as the right argument and the line numbers at which the function is to be suspended as the left argument. For example, suppose the user wants to suspend execution of function *HH* at lines 2 and 4; by typing the expression

```
      2 4 ⎕STOP 'HH'
```

APL will then suspend function execution just before each specified line number is executed, print the function name and line number, skip to the next line and indent six spaces to prompt for user input. (See the possible effects of ⎕*SA* described in Section 11.)

```
      HH
HH[2]
```

The user may then operate as desired, in direct input mode with the function suspended, and can resume or terminate function execution at any time. Function execution can be resumed by appropriate branching; for example, an entry of →3 will resume execution of the suspended function at statement 3. Termination can be accomplished by a branch to a non-existent line number (→0 is a convenient choice). The function suspension can also be abandoned by a suspension clear statement, which is a branch arrow without any line number.

A stop control vector can be specified during execution mode, or during function execution as one of the statements of a defined function. To discontinue an active stop control vector, assign an empty vector to that stop control vector; for example, ''⎕*STOP* '*HH*' will turn off the stop control for function *HH*.

Error Stop

As soon as APL detects an error in a statement, execution of that statement is terminated and any partial result is lost, except for assignments that were completed before the error was detected. Unless sidetracking occurs, APL prints a message indicating the type of error, displays the erroneous statement with a caret below the place the error was detected, (see also the discussion of error messages for the Execute operator in Section 5), and prompts for input. (See the possible effects of ⎕*SA* described in Section 11.) The user can then correct the statement. An example of error detection is shown here:

```
      X1←4÷0
DOMAIN ERR
      X1←4÷0
          ^
```

If a statement contains more than one error, only the first (rightmost) one detected by APL will result in an error report. The next error will not be detected until the user has corrected the first error, as illustrated here:

```
      X1←(4÷0)×(2÷0)
DOMAIN ERR
      X1←(4÷0)×(2÷0)
                 ^
      X1←(4÷0)×(2÷1)
DOMAIN ERR
      X1←(4÷0)×(2÷1)
           ^
```

If an error is detected in a statement with multiple specifications, any assignments to the right of the error will be completed, as illustrated here:

```
      B←5 ⋄ 4÷C←B×0
DOMAIN ERR
      B←5 ⋄ 4÷C←B×0
             ^
      B
5
      C
0
```

During function definition some types of errors are detected immediately while other types are not detected until later when the function is executed. Definition errors, and character errors are detected immediately and must be corrected as soon as an error report is printed.

```
      ∇R←B TRI H
[1]   AREA←0.5×B×H
[2]   DIAGONAL←((H*2)B*2)*0.5
[3]   R←AREA;DIAGONAL
[4]  [0.5 TRI CALCULATES AREA AND DIAGONAL OF TRIANGLE
 DEFN ERR ^
[0.5] ⍝  TRI CALCULATES AREA AND DIAGONAL OF TRIANGLE
[0.6] ∇
```

Linescan errors are detected immediately and may be corrected immediately by function editing or its correction may be deferred. All other errors in a defined function are detected when the function is executed. When APL encounters each error during function execution, it suspends execution and prints an error report containing the type of error and the function name and offending line and statement (with a caret marking the place the error was detected). For example, the following error message is produced because a Not function had been entered instead of a multiplication sign:

```
      5 TRI 8
SYNTAX ERR
TRI[3] DIAGONAL←((H*2)~B*2)*0.5
                      ^
```

An error that causes suspended execution can be corrected during the suspension or after termination of execution.

1. To correct an error during suspended execution, the user can follow normal function editing procedures (see Section 7). For example,

   ```
   SYNTAX ERR
   TRI[3] DIAGONAL←((H*2)~B*2)*0.5
                         ^
         ∇TRI[3] DIAGONAL←((H*2)×B*2)*0.5∇
   ```

   After correcting an error, the user can resume execution at the line suspended by specifying a branch to that line number. Thus, the expression →3 will resume execution at line 3 (starting at the right, as usual for APL).

2. To correct an error with termination of execution, the user enters a branch arrow to terminate function execution, edits the function as necessary, and then reexecutes the function. For example:

   ```
   SYNTAX ERR
   TRI[3] DIAGONAL←((H*2)~B*2)*0.5
                         ^
         →
         ∇TRI[3] DIAGONAL←((H*2)×B*2)*0.5∇
         5  TRI 8
   ```

   Each branch arrow removes the most recent suspension from the state indicator list. Thus if several suspensions have occurred since the last suspension clear, more than one branch arrow (suspension clear) will be required to clear the state indicator. A convenient method for clearing the entire state indicator is to issue a)*SIC* command.

Sidetracking On Errors And Breaks

In some APL applications, the programmer would like to bypass APL's standard error and break procedure (for example, to substitute messages or institute corrective actions). Computer-assisted learning programs and commercial business aids are applications where this may be desired. Users of such applications may have little knowledge of APL, and messages such as *DOMAIN ERR* or *WS FULL* frustrate rather than help.

CP-6 APL allows the programmer to overcome this problem through "sidetracking". The term "error control" is also used (with an understanding that break control is included).

Suppose a *DOMAIN ERR* has been detected by APL. With sidetracking, APL searches the state indicator for active functions for which the programmer has decided to sidetrack. If a sidetracking function wants control over *DOMAIN ERR*, then APL sidetracks (branches) to the line in the function specified for *DOMAIN ERR*. If no active function wants such control, then APL issues the standard diagnostic message.

Sidetracking is both flexible and dynamic. Different errors can be sidetracked to distinct lines of a function. Certain sidetracking functions may control some errors while other sidetracking functions control others. The system function ⎕*ERS* allows an APL program to simulate an error (which can be subject to sidetracking) by supplying the error number and optionally the error message. Sidetracking functions can also compete for control of the same error. In this case, the most recently invoked function gets control, and its competing predecessors never become aware that the error occurred. Sidetrack specifications can be changed at will. They can be turned on and off, the error selection can be altered, and the sidetrack branches can be changed; the application program itself can modify sidetracking specifications throughout its execution. This capability permits a simple or comprehensive treatment at the programmer's discretion.

Table 10-1 shows errors that are subject to sidetracking. Errors not listed in the table include *SYSTEM ERR* and *BROKEN WORKSPACE*.

Since recovery from these errors is impossible for an applications program, APL retains exclusive control. See the discussion following Table 10-1 for details concerning certain unique errors.

Associated with each item in Table 10-1 is an error number. Error numbers are informally grouped by common classifications: statement execution errors, input-translation errors, command errors, file input/output errors, etc. Gaps are provided in the error number sequence to accommodate future diagnostics.

The items in Table 10-1 contain four cases in which APL gives up control after displaying an error message:

SI DAMAGE
name *NOT COPIED*
name *NOT FOUND*
name *NOT ERASED*

These cases, in which command processing or function definition is in effect, must reach an orderly conclusion. Therefore, APL unilaterally displays the messages and proceeds to conclusion. (Nevertheless, sidetracking is still possible, and an application program might issue explanatory messages after the APL messages, as one alternative.) As APL proceeds in these four cases, a series of such messages could be displayed (but this would be unusual). APL permits sidetracking only with regard to the latest error known at the conclusion of this kind of processing. Using the execute function, for example, suppose a *)COPY* command occurs while sidetracking is in effect. Suppose also that some object, *X*, is missing from the copied workspace - *X NOT FOUND* is displayed; furthermore, suppose that the data found will not fit in the active workspace. Then the *)COPY* command concludes without copying anything and would ordinarily issue a *TOO BIG* message. Sidetracking would then apply in this example to the *TOO BIG* error and, the *NOT FOUND* error would be "forgotten".

Note in the foregoing example that copying was attempted by means of an execute operation. This was a necessity. A function can obtain sidetracking only while it is actively in execution. Thus command and function definition errors can be sidetracked only when the function (or some function invoked by the sidetracking function) actually executes a command or function definition. Evaluated-input might seem to provide another way in which a function could, indirectly, invoke command or function definition activity. However, for the reason given below, evaluated input is not considered capable of being sidetracked (except while that input has itself invoked a sidetracking function).

The execute function makes it possible to execute via quote-quad input anything that could be entered via evaluated input. Quote-quad input has an advantage from the standpoint of error recovery. The input text can be assigned to a variable before it is executed. Thus, a sidetrack function can analyze this text to determine correct recovery action. Evaluated-input is not susceptible to this analysis; it is immediately interpreted by APL.

Table 10-1. Events Subject to Sidetracking

Error Number	Error Message
1	WS FULL
2	SYNTAX ERR
3	UNDEFINED
4	DOMAIN ERR
5	RANK ERR
6	LENGTH ERR
7	INDEX ERR
8	NO RESULT
10	IMPLICIT ERR
15	SINGULAR MATRIX
16	FORMAT SYNTAX ERR
20	BAD CHAR
21	LINESCAN ERR
22	TRUNCATED INPUT
23	OPEN QUOTE
30	I/O ERR fcg-xxxxx-s
35	DEFN ERR
36	SI DAMAGE
38	NOT CLEAR WS
39	CLEAR WS
40	BAD COMMAND
41	NOT SAVED, THIS WS IS name
42	FILE IN USE
43	BAD FILE REF
44	WS NOT FOUND
45	TOO BIG TOO LOAD
46	TOO BIG
47	TOO MANY SYMBOLS
48	name NOT COPIED
49	name NOT FOUND
50	name NOT ERASED
51	NOT GROUPED
52	SEALED WS
53	OLD WS, MUST EXPORT
55	NOT HELD
59	HOLD ABORTED
61	HOLD DEADLOCK
62	ENQUEUE FULL
68	SV QUOTA EXHAUSTED
69	NO SHARES
70	FILE SPACE TOO LOW
71	FILE I/O ERR fcg-xxxxx-s
72	FILE DAMAGE
73	FILE NAME ERR
74	NOT APL FILE
75	FILE TBL FULL
76	FILE ACCESS ERR
77	FILE TIE ERR
78	PACKSET NOT MOUNTED
79	FILE INDEX ERR
98	NONCE ERR
100	INTERRUPT

⎕*SM* Function (Set/Query Sidetrack Matrix)

Syntax:

```
E←⎕SM F
R←E ⎕SM F
```

Parameters:

F is a namelist containing the name of a displayable defined function.

E is a simple 2–column matrix of integers in which the first column contains line numbers and the second column contains error numbers.

R is an empty numeric array.

Description:

A sidetracking setting resembles setting a stop or trace vector. The function must be defined when the ⎕*SM* function is executed; it could even be a statement in the function.

The dyadic ⎕*SM* function requires that the right argument contain a legal name or *DOMAIN ERR* is reported. The explicit result of dyadic ⎕*SM* is an empty numeric vector. The left argument (sidetrack matrix) must be a matrix or *RANK ERR* is reported. The second dimension of the left argument must be 2 or *LENGTH ERR* is reported. The left argument must be a simple array containing only integers or *DOMAIN ERR* is reported.

In effect, the sidetrack table becomes part of the function's definition and is copied or loaded if the function is copied or loaded. Function editing has no influence on the sidetrack setting. Since the sidetrack table contains line numbers, the following precaution should be observed. If editing a sidetracking function alters the position of a line specified by the sidetrack table, a correct setting must be reissued. This is necessary to ensure that the proper line will be branched to if the sidetrack does take place.

Erasing a sidetracking function erases its sidetrack setting. A sidetrack setting can also be removed by being replaced with an empty matrix, as in the following example:

```
(0 2⍴0)⎕SM 'FUN'
```

When a (non–empty) table is assigned for sidetracking, it consists of one or more rows. Each row contains a pair of integers – a line number and an error number. The line number designates which line of the function is to be sidetracked to (branched to) if the indicated error occurs. The following sample sidetrack setting specifies a branch to line number 9 in case of a *DOMAIN ERR* (error 4 in Table 10–1).

```
(1 2⍴9 4)⎕SM 'FUN'
```

A new sidetrack setting for a function entirely replaces any previous setting.

Examples:

The following example would remove *FUN* 's control over *DOMAIN ERR*.

```
(2 2⍴9 3 9 2)⎕SM 'FUN'
```

In this example, *FUN* sidetracks to line 9 for *UNDEFINED* or *SYNTAX ERR*. This illustrates that a sidetrack table can contain duplicate line numbers; however, it is useless to duplicate an error number in the same table. Only the first such number would be effective.

In the above example, *FUN* sidetracks on only two of the possible errors. If other errors occur, APL handles them in the standard manner unless some other function has specified sidetracking for those errors.

A special error number, 0, exists for sidetracking on all items in Table 10-1 except the break (number 100). In the following example, *FUN* sidetracks:

o to line 8, if a break is detected

o to line 7, if *WS FULL* occurs

o to line 9, for any other error subject to control.

```
    (3 2⍴8 100 7 1 9 0)⎕SM 'FUN'
```

Breaks are sidetracked only if the sidetrack explicitly includes error number 100.

The current sidetrack matrix may be obtained by the monadic execution of the ⎕SM system function. In this case, the right argument is the same as in the dyadic usage, but the result is a simple N-by-2 matrix of integers. For example:

```
      ⎕SM 'FUN'
8 100
7   1
9   0
```

The result of monadic ⎕SM for all names other than a displayable, active defined function is a numeric matrix of shape (0,2).

The following example (assuming origin 1) shows a compact way of setting several different error numbers to the same line. Suppose *ERRLAB* is the label of the desired line and sidetracking is set within the function *FCN* containing *ERRLAB*.

```
    (ERRLAB,[1.5]2 3 5 8 21) ⎕SM 'FCN'
```

sets the indicated errors to sidetrack to *ERRLAB* (see Lamination).

The above examples illustrate how to set sidetracks. This does not imply that the function *FUN* immediately receives control if an error occurs. If *FUN* is not actively in execution, its sidetracking is disregarded. Even if *FUN* is in execution, it may still not be given control. The error may have occurred in evaluated-input, or *FUN* may have called another function which has a competing sidetrack.

Dynamics of Sidetracking

A step-by-step outline reveals significant aspects of sidetracking dynamics. Assume a controllable error or break has occurred and APL is ready to check for sidetracking.

Step 1: APL designates and saves the current error number, replacing any previously recorded error number; the line in execution, the position in this line and the text of the error message are also saved. For the moment, it initializes the error location to be line zero and an empty function name. APL points to the top (latest) entry in the state indicator.

Step 2: The state entry is examined. If it is a pendent function, APL proceeds to Step 3. If it is an execute-operation state, APL points to the next entry and repeats Step 2. Otherwise, sidetracking is not applicable; so APL issues the standard diagnostic.

Step 3: (Pendent function state) The error location is tested. If still initialized (see Step 1), the line number and name of the pendent function are recorded. The function's definition is tested for sidetrack setting. If it features some sidetracking, APL proceeds to Step 4. Otherwise, APL points to the next state indicator entry and repeats Step 2, attempting to find a function with a sidetrack setting.

Step 4: (Sidetrack setting present) The sidetrack table is tested sequentially versus the recorded error number. If a match is found, or the error number is less than 100 or greater than 199 and the table has an error number entry for the error number 100×⌊*N*÷100 (where *N* is the error number being reported) or the table has a zero error number, APL proceeds to Step 5. Otherwise, APL points to the next state entry and repeats step 2, attempting to find a function interested in the current error.

Step 5: If the specified line number is greater than or equal to zero, then APL proceeds to Step 6. Otherwise, the line number is assumed to represent one of the special actions in Table 10–2.

Step 6: (Sidetrack acknowledged) APL removes from the state indicator any entries it bypassed in reaching Step 5. This puts the sidetracking function at the top of the state indicator. APL then branches to the specified line number.

Table 10–2. Sidetracking Special Action Table

Number	Special Action
¯1	The state indicator is cleared to the entry before this function. The current error is then reported at the point of the sidetracking function call.
¯2	This function explicitly requests no sidetracking for the error. The state indicator, however, is to be cleared to this function and this function suspended. (Debugging Aid)
¯3	Similar to ¯2 except that the state indicator is not cleared to the sidetrack function. The function that was in execution when the error occurred is suspended and the error is reported at the point of the error. (Debugging Aid)
¯4	When used in conjunction with an error class control, this allows a function to control all errors but a specific error. As in (2 2⍴ ¯4 1 5 0) ⎕*SM* '*X*' This setting means that all errors other than a workspace full (1) will sidetrack to line 5.
¯5	The APL session is terminated and a CONTINUE workspace is saved, suspending the function that had the error.
¯6	The workspace is cleared.

Considerations after Gaining a Sidetrack

Once APL performs a sidetrack, it has no further interest in handling the break or error. Responsibility falls to the application programmer, depending on the line number dictated and statements supplied for the sidetracking function. Caution is advised.

If a mistake occurs in statements entered via a sidetrack, a new error may confuse the intended recovery procedure. It is possible for that statement to generate the error being considered, leading to the same sidetrack, the same mistake, and so on indefinitely. *WS FULL* can be particularly troublesome. In some cases, the statement reached by the sidetrack will itself cause another *WS FULL*. There is no general solution to this potential problem, but it is a rare difficulty for two reasons. First, intermediate results may be discarded after any error, freeing up sufficient workspace for recovery. Second, more workspace may become available if state indicator entries are removed in reaching the sidetrack (See Step 5 above).

Aids for Sidetrack Users

Eight system functions are of particular interest after an error has occurred. These functions are described next.

⎕*ERN* Function (Error Number)

Syntax:

```
W←⎕ERN
```

Description:

The result is a two-item integer vector, the first item is the latest error number and the second item is the line number of the function in which the error occurred. If the error did not occur in a defined function, the second item is 0.

⎕*ERF* Function (Error Function)

Syntax:

```
F←⎕ERF
```

Description:

The result is a character vector containing the name of the function in which the error occurred. If the error did not occur in a defined function, the result is an empty character vector.

⎕*ERM* Function (Error Message)

Syntax:

```
T←⎕ERM
```

Description:

The result is a character vector containing the text of the latest error message.

⎕*ERL* Function (Error Line)

Syntax:

T←⎕*ERL*

Description:

The result is a character vector containing the text of the line that was in execution when the error occurred.

⎕*ERP* Function (Error Position)

Syntax:

I←⎕*ERP*

Description:

The result is the integer scalar index in ⎕*ERL* of the error pointer. The result value is ⎕*IO* dependent.

⎕*ERX* Function (I/O Error)

Syntax:

T←⎕*ERX*

Description:

The result is a character vector of length 12, containing the latest I/O error information available to APL. It represents one of the error codes given in the CP-6 Host Monitor Services Reference Manual, v.1, (CE74).

⎕*ERH* Function (Error Help)

Syntax:

T←⎕*ERH*

Description:

The result is a character matrix of shape N-by-120 containing the error message text from the system error message file pertaining to the latest CP-6 I/O error information.

Note: APL sometimes expects I/O errors. Thus, the value reported when using ⎕*ERX* does not necessarily indicate an error condition has occurred.

⎕*ERS* Function (Error Simulation)

Syntax:

```
    MESSAGE ⎕ERS I
            ⎕ERS I
```

Parameters:

I is a simple integer scalar in the range 0 through 19999.

MESSAGE is a simple character vector containing an error message.

Description:

The ⎕*ERS* system function initiates an error report under program control that can be subject to sidetracking. The error simulated may be one of APL's standard execution errors such as *DOMAIN ERR* or the specific error message can be supplied. The left argument of ⎕*ERS* may be supplied only for error numbers greater than 499. Simulating error number 0 clears the current error status variables (⎕*ERN*, ⎕*ERF*, ⎕*ERM*, ⎕*ERL*, ⎕*ERP*) to their initial values in a clear workspace.

If ⎕*ERS* is invoked by an active function, the error generated by the execution of ⎕*ERS* occurs in the environment of the line that invoked the currently executing defined function.

Section 11

System Functions and Variables

CP-6 APL provides a complete set of system functions and variables. Each of these system-defined objects have names which begin with a quad (⎕) and are known as distinguished names.

All of the distinguished names are present in an active workspace, and the values of the system variables are saved with the workspace when a)*SAVE* command is issued.

All CP-6 APL system variables have default values in a clear workspace, may be modified at any time by assignment, and are subject to the normal rules of scope. That is, a user-defined function may localize a system variable and subsequently modify it. When the function exits, the value of the system variable will revert to its original value. The following table lists system variables included in CP-6 APL.

Table 11-1. System Variables

NAME	MEANING	VALUE IN CLEAR WS	RANGE
⎕*CT*	Comparison tolerance	1*E*¯13	Non-negative less than 1*E*¯12
⎕*IO*	Index origin	1	0 or 1
⎕*LX*	Latent expression	''	character vector
⎕*PP*	Print precision	10	integer in the range 1 .. 20
⎕*PS*	Positioning and spacing	¯1 1 0 2	a 4-item integer vector. The first two items are integers in the range ¯1 to 1 . The last two items may be any integer value from ¯2*35 to ¯1+2*35.
⎕*PW*	Platen width	device dependent	integer in the range 32 .. 390
⎕*RL*	Random link	16807	integer in the range 1 .. (¯1+2*35)
⎕*SA*	Stop Action	''	'*EXIT*', '*CLEAR*' and ''
⎕*SP*	Session Parameter	''	any array

Each of the APL system variables is described in detail next.

⎕CT Variable (Comparison Tolerance)

The relational functions (< ≤ = > ≥ ≠), monadic ⌈, ⌊ and dyadic ⍳, ≡, and ∊ involve comparisons that are not absolute because of the internal representation of numbers. ⎕CT is used to establish a neighborhood of equality around any particular value. For the relational functions, any number between B+⎕CT×|B and B-⎕CT×|B will be considered equal to B. The default value for ⎕CT is 1E¯13 which is adequate for almost all situations.

Comparison tolerance is used in APL so that the finite precision of the internal representation of numbers can be partly disguised. Computer arithmetic with real numbers can only approximate the result to numbers which are mathematically close to the true result. Meaningful values of ⎕CT are real numbers in the range 0 to 1E¯12 inclusive.

⎕IO Variable (Index Origin)

The ⎕IO variable is used by APL during indexing, the axis operator, ⎕FX, ?, dyadic ⍉, ⍳, ⍋, and ⍒. Its value indicates the index of the first value in a non-empty vector. The only permissible values are zero and one. The default value in a clear workspace is 1.

⎕LX Variable (Latent Expression)

The ⎕LX variable is executed (as in ⍎⎕LX) whenever a workspace is loaded. The default value in a clear workspace is an empty character vector.

⎕PW Variable (Platen Width)

All output is subject to the constraint that at most ⎕PW characters will appear on a line. Additional characters that would have appeared on a particular line are printed indented on the succeeding line. The meaningful values are integers in the range 32 through 390. The default value in a clear workspace is the CP-6 platen width setting when APL was invoked or the closest permissible value to the CP-6 platen setting.

⎕PS Variable (Positioning and Spacing)

The value of the ⎕PS variable controls the display (and monadic format result) of nested arrays. The meaning of the values are described in Section 3 under Output.

⎕*PP* Variable (Print Precision)

The ⎕*PP* variable is used to determine the number of digits to be used in the default display of numeric arrays. The meaningful values are integers in the range 1 to 20. The default value in a clear workspace is 10.

⎕*RL* Variable (Random Link)

This value is used in the ? function and reset after each use such that it cycles through the entire meaningful range. The meaningful values are integers between 0 and 2*35 exclusive.

⎕*SP* Variable (Session Parameter)

When APL is invoked, ⎕*SP* is established with an initial value of an empty character vector. A new value may be specified at any time, by assignment (or a)*COPY* command containing ⎕*SP* in the copy list). The value associated with ⎕*SP* is carried across)*LOAD* and)*CLEAR* commands.

⎕*SA* Variable (Stop Action)

The ⎕*SA* variable defines the action to be taken when a function terminates execution and direct input mode is entered for any reason. The default value of an empty character vector indicates that no action is to be taken. Other valid values of ⎕*SA* are: '*CLEAR*' which causes the workspace to be cleared, and '*EXIT*' which causes APL to issue an)*END* system command. If the value of ⎕*SA* is undefined the state indicator is scanned until a value associated for ⎕*SA* is found and the associated action is taken.

System Functions

System functions are always present in a workspace, and can be used in defined functions. They are niladic, monadic, or dyadic as appropriate and have an explicit result. In many cases, they also have implicit results, in that their execution causes a change in the environment.

Workspace Management Functions

CP-6 APL provides a set of system functions, ⎕*CR*, ⎕*FX*, ⎕*NL*, ⎕*EX*, ⎕*EXG*, ⎕*LOK*, ⎕*NC*, ⎕*NCG*, ⎕*RM*, ⎕*RMG*, ⎕*ST*, ⎕*TR*, ⎕*SM*, ⎕*STOP* and ⎕*TRACE* to aid in user workspace management and information display.

Namelist and Canonical Representations

The introduction of two concepts is useful to describe the argument and results associated with these system functions. A namelist is a character matrix in which each row represents an APL name. As an argument to a system function, a namelist may also consist of a character vector of names separated by blanks.

When a namelist argument is required, a *RANK ERR* occurs if the rank of the argument is greater than 2. A *DOMAIN ERR* occurs if the argument contains an item which is not a character scalar. A *LENGTH ERR* occurs if a row contains more than 262143 columns. A *DOMAIN ERR* is also reported when a system function which only accepts a single name, is provided with other than a single name. Examples of legal namelists are:

```
'A'          ⍝ NAMELIST CONTAINING THE NAME A
'AB'         ⍝ NAMELIST CONTAINING THE NAME AB
2 2⍴'ABCD'   ⍝ NAMELIST CONTAINING THE NAMES AB AND CD
'AB CD'      ⍝ ARGUMENT NAMELIST CONTAINING THE NAME AB AND CD
```

Canonical Representation is a representation of a function as a character matrix. Each row of the matrix represents a line of the function. The first row must consist of a valid function header. Succeeding rows if present must be valid APL statements.

A *RANK ERR* is reported if the rank of the canonical representation is greater than 2. A *LENGTH ERR* occurs if a row contains more than 262143 columns. A *DOMAIN ERR* is reported if the canonical representation contains an item which is not a character scalar. The canonical representation does not contain line numbers as provided by the function definition mode display.

System Functions for Function Definition

CP-6 APL provides system functions to create, modify, and replace defined functions. These functions include ⎕*TRACE*, ⎕*STOP*, ⎕*CR*, ⎕*FX*, and ⎕*AT* discussed in section 7, and ⎕*SM* in section 10 in addition to the following functions.

⎕*LOK* Function (Lock Function)

Syntax:

```
V←⎕LOK N
```

Parameters:

N is a namelist containing the names of user-defined functions in the active workspace.

V is a simple vector containing the integer values 0 or 1.

Description:

The ⎕*LOK* function returns a numeric vector containing 1 if the corresponding name in *N* is now a locked function or 0 otherwise (also see function ⎕*AT* in Section 7). The referenced functions are locked.

⎕*ST* Function (Set/Query Stop)

Syntax:

V←⎕*ST F*

R←*V* ⎕*ST F*

Parameters:

F is a namelist containing the name of a defined function.

V is a simple integer vector.

R is a simple integer vector.

Description:

For monadic ⎕*ST*, if *F* is a namelist containing the name of a defined function, the result *V* is the stop vector associated with that function.

For dyadic ⎕*ST*, if *F* is a namelist containing the name of a displayable defined function, stop control is set on the lines indicated by *V*. The explicit result is an empty vector. Also see the function $QSTOP.

Possible Errors:

A *RANK ERR* is reported if:

o the left argument *V* is not a scalar or vector.

A *DOMAIN ERR* is reported if:

o *V* is not a simple array containing integers.

⎕*TR* Function (Set/Query Trace)

Syntax:

V←⎕*TR F*

V←*V* ⎕*TR F*

Parameters:

F is a namelist containing the name of a defined function.

V is a simple integer vector.

Description:

For monadic ⎕*TR*, if *F* is a namelist containing the name of a displayable defined function, then *V* is the trace vector associated with that function.

For dyadic ⎕*TR*, if *F* is a namelist containing the name of a displayable user defined function, trace control is set on the lines indicated by *V*. The explicit result is an empty vector. Also see the function $QTRACE.

Possible Errors:

A *RANK ERR* is reported if:

o the left argument is not a scalar or vector.

A *DOMAIN ERR* is reported if:

o *V* is not a simple array containing integers.

Workspace Management System Functions

The following functions are used in the management of active workspace.

⎕*AV* Function (Atomic Vector)

Syntax:

```
      X←⎕AV
```

Parameters:

X is a character vector of length 512.

Description:

The ⎕*AV* function returns a character vector containing all of the possible characters in the APL character set. Many of these characters are not used in CP-6 APL to represent printing symbols. The positions of the individual characters differ between implementations of the APL language.

Example:

```
      ⎕AV[65+⍳27]
ABCDEFGHIJKLMNOPQRSTUVWXYZ[
```

⎕CPU Function (CPU Time Used)

Syntax:

```
      I←⎕CPU
```

Parameters:

I is an integer scalar.

Description:

The ⎕CPU function returns an integer scalar containing the CPU execution time used since entering APL. The time returned is in units of milliseconds. Execution times vary widely between APL implementations and the model of CPU upon which the APL is executing.

Example:

```
      T←⎕CPU ⋄ Z←5+⍳1E6 ⋄ ⎕CPU-T
3
```

In this example, the time required by CP-6 APL to execute the expression *Z*←5+⍳1*E*6 has been computed to be 3 milliseconds (or .003 seconds).

⎕CVT Function (Convert)

Syntax:

```
      R←W ⎕CVT J
```

Parameters:

J is a simple array of either all numeric or all character items.

W is a simple integer vector of length two.

Description:

J must be a simple array containing only numeric or only character items. *W* is a simple two–item integer vector controlling the result type and values of *R*. The first item of *W* is one of 1, 2, 3, or 4 meaning one of the following types:

TYPE	INTERNAL BIT LENGTH	MEANING
1	1	Boolean
2	9	Character
3	36	Integer
4	72	Floating Point

The second item of *W* is a value whose magnitude indicates the number of bits of the right argument to use to represent each item of the result. If the second item of *W* is less than zero, then the sign of each item of *R* is negative if the sign bit of a field is 1. A *DOMAIN ERR* occurs if the first item of *W* is not one of 1, 2, 3, or 4, or if *J* is either a nested array or index sequence. A *LENGTH ERR* occurs if the number of bits in the last dimension of *J* are not evenly divisible by the second item of *W*.

Example:

```
      3 9 ⎕CVT 'ABZ'  ⍝ CHARACTER No.
65 66 90
      ⎕AV ⍳ 'ABZ'
66 67 91

      3 3 ⎕CVT 'AZ'  ⍝ CONVERT TO OCTAL
1 0 1 1 3 2

      3 36 ⎕CVT 12345678.015625
1709378160 8589934592
```

⎕*DL* Function (Delay)

Syntax:

```
        I←⎕DL I
```

Parameters:

I   is a simple integer scalar.

Description:

The ⎕*DL* function requires at least *I* seconds to complete. The explicit result is the number of seconds of delay. The delay may be shorter than the number of seconds specified if it is interrupted by a break.

Example:

```
        ⎕DL  6.75
7
        ⎕DL  ¯60     ⍝  NOW BACK UP ONE MINUTE!
0
```

Note that in the above example, requested delays of 6.75 and ¯60 seconds were actually delayed 7 and 0 seconds.

⎕*EX* Function (Expunge)

Syntax:

```
        V←⎕EX N
```

Parameters:

V   is a namelist.

N   is a simple integer vector.

Description:

The ⎕*EX* function erases the user defined objects in namelist *N*, except groups, labels, and active, pendent or suspended functions. The explicit result is a logical vector whose *I*'th item is 1 if the *I*'th name in *N* is now available for use (whether or not it was erased). For non-names or distinguished names or any names not erased, the result is 0.

Example:

```
      A←C←0
      ⎕EX 4 1⍴'AB7C'
1 1 0 1
```

In the above example, the result of ⎕*EX* indicates that the first, second and fourth names are now available for use (they have no active use) and that the third name is not.

⎕*EXG* Function (Expunge Globals)

Syntax:

```
      V←⎕EXG N
```

Parameters:

V is a namelist.

N is a simple integer vector.

Description:

Same as the ⎕*EX* function except that only global referents are affected.

⎕*FI* Function (Fix Input)

Syntax:

```
      V←⎕FI T
```

Parameters:

T is a simple character scalar or vector.

V is a simple numeric vector.

Description:

The ⎕*FI* function returns a numeric vector containing the value of all numeric constants found in *T* that are delimited by blanks. Non-blanks in *T* that are not legal numbers are indicated by a zero value.

Example:

```
      ⎕FI  '22  3X5  ¯100.5  1E999  1E6 0.0'
22 0 ¯100.5 0 1000000 0
```

Possible Errors:

A *RANK ERR* is reported if:

o   *T* is not a character scalar or vector.

A *DOMAIN ERR* is reported if:

o   *T* contains an item that is not a character scalar.

⎕*GRP* Function (Return Group Members)

Syntax:

```
      N←⎕GRP G
```

Parameters:

G   is a namelist containing the name of a group in the active workspace.

N   is a namelist.

Description:

The ⎕*GRP* function returns a a namelist containing the names associated with the group.

Example:

```
      ⎕GRP 'STAT_GROUP'
MEDIAN
MODE
REG
```

In this example, the name *STAT_GROUP* represents an APL group containing the names *MEDIAN*, *MODE* and *REG*.

⎕*IBEX* Function (IBEX Expunge)

Syntax:

```
     R←⎕IBEX N
```

Parameters:

N is a namelist containing the names of IBEX variables.

R is a simple integer vector.

Description:

The ⎕*IBEX* system function is used to expunge IBEX variables. The right argument is a namelist, the result is a simple logical vector containing 1 for every IBEX name that is now available for re-use, or 0 for the corresponding name representing an unavailable name (illegal name).

Example:

```
    ⎕IBEX 'STATUS'
1
```

⎕*IBLET* Function (Set/Query IBEX Variable)

Syntax:

```
     R←⎕IBLET T

     R←T ⎕IBLET T
```

Parameters:

T is a simple character vector.

R is a simple character vector.

Description:

The ⎕*IBLET* system function returns or sets the value of an IBEX variable. The right argument is the text of the name of an IBEX variable and the left argument when present is the value to be assigned to that variable. Monadically, ⎕*IBLET* returns the value of the named variable. Dyadically, the left argument becomes the new value of the variable.

This function can be useful when communicating with IBEX (outside of APL) or with other CP-6 programs. The left and right argument must be a scalar or vector or a *RANK ERR* is reported. A *LENGTH ERR* is reported if the name in the right argument contains more than 31 characters or if the length of the left argument is more than 511. A *DOMAIN ERR* is reported if either the left of right argument contains an item which is not a character scalar, or if the right argument contains an illegal name. An error is also reported if the monadic syntax is used and the named variable does not currently have a value.

Example:

```
      'RATE=12,BALANCE=1246.42' ⎕IBLET 'NAME'

      ⎕IBLET 'NAME'
RATE=12,BALANCE=1246.42

      )!OUTPUT NAME
RATE=12,BALANCE=1246.42
```

⎕*IBNL* Function (IBEX Namelist)

Syntax:

```
      R←⎕IBNL
```

Description:

The ⎕*IBNL* system function is used to return the names of all of the IBEX variables associated with this CP-6 session. The result is a simple character matrix with one IBEX variable name in each row.

Example:

```
     ⎕IBNL
NAME
STATUS
```

⎕*IDLOC* Function (Identifier Location)

Syntax:

```
      R←⎕IDLOC N
```

Parameters:

N is a namelist.

R is a simple integer matrix having one row for each name in *N*.

Description:

The ⎕*IDLOC* system function returns the local and global name classes for each of the names in the namelist at each level in the state indicator. The result contains a row whose length is 1+ρ⎕*LC* for each name. The name classes returned are:

−1	Not local at this level
0	Local but no value
1	Label
2	Variable
3	Function
4	Not Available

Example:

```
      )SINL
F2[2]  *    C   A
F[4]    X   R   D   B   A

      )VARS
A   C

      ⎕IDLOC  'A B C D F'
 2  3  2
¯1  2  0
 2 ¯1  2
¯1  0  0
¯1 ¯1  3
```

The first row of the result of ⎕*IDLOC* contains the values 2 3 2 which indicate that the name *A* is a global variable, a function local to the execution instance of the function *F* and is a variable local to the function execution of *F2*. The second row indicates that the name *B* has no global usage, it is a variable local to the execution of *F* and is localized (but not yet used by) the execution of *F2*.

⎕*LC* Function (Line Chain)

Syntax:

```
      V←⎕LC
```

Parameters:

V   is a simple integer vector.

Description:

The ⎕*LC* function returns an integer vector whose length indicates the number of entries in the state indicator and whose values are the line numbers of the functions in execution or 0 for state entries that are not defined functions. The result is ordered so that the most recently initiated function has the lowest index.

Example:

```
     ∇ F
[1]   ⎕LC
[2]   ⍎'⎕LC'
     ∇

      F
1
0 2
```

In this example, the function *F* displays two lines of output. The first line displayed indicates that line 1 is in execution. The second line displayed indicates that line 2 of *F* is in execution followed by a state entry that is not a defined function (an execute state entry).

⎕*LGT* Function (Logon Time)

Syntax:

```
      I←⎕LGT
```

Parameters:

I   is an integer scalar.

Description:

The ⎕*LGT* function returns an integer scalar whose value is the number of milliseconds that have elapsed between midnight and the time of day that APL was invoked.

Example:

```
      0 60 60 1000⊤⎕←⎕LGT
45000250
12 30 0 250
```

In this example, APL was invoked at 12:30 PM.

⎕*NC* Function (Name Classification)

Syntax:

```
      K←⎕NC N
```

Parameters:

N   is a namelist.

K   is a simple integer vector.

Description:

The ⎕*NC* function returns the type of object represented by each name in namelist *N*. The *I*'th item of *K* corresponds to the *I*'th name in *N*. The value of each item of the result is one of the following:

0	a name without an active referent
1	a label
2	a variable
3	a function
4	other (distinguished name, group name, or not a name.)

Example:

```
      ∇F∇
      A←C←0
      ⎕NC 5 1⍴'AB7CF'
2 0 4 2 3
```

In the above example, the result of ⎕*NC* indicates that the first and fourth names (*A* and *C*) are variables, the second name (*B*) has no current use. The third name is not available (actually 7 is not a legal name), and the fifth name (*F*) is a defined function.

⎕*NCG* Function (Name Correspondence of Global)

Syntax:

```
      K←⎕NCG N
```

Parameters:

N is a namelist.

K is a simple integer vector.

Description:

The same as ⎕*NC* except only Global referents are examined.

⎕*NL* Function (Namelist)

Syntax:

```
      N←⎕NL K

      N←T ⎕NL K
```

Parameters:

K is a simple integer scalar or vector.

N is a namelist.

T is a simple character scalar or vector.

Description:

For monadic ⎕*NL*, *K* is a simple numeric integer scalar or simple numeric vector with items containing the values 1, 2, or 3. The result is a namelist whose rows represent names whose active referents are of each of the indicated classes as defined for ⎕*NC*.

Dyadic ⎕*NL* is the same as the monadic case, except that only names beginning with one of the characters in *T* are included in the result.

Possible Errors:

A *DOMAIN ERR* is reported if:

o *K* contains an item that is not a simple integer value 1, 2 or 3.

A *RANK ERR* is reported if:

o *K* is not a scalar or vector.

Example:

```
      A←AB←C←CX←0
      ∇AFUNCTION∇
      ∇FUNCTION∇

      ⎕NL 3
AFUNCTION
FUNCTION

      'AB' ⎕NL 2 3
A
AB
AFUNCTION
```

⎕ONL Function (Online)

Syntax:

```
       I←⎕ONL
```

Parameters:

I is an integer scalar value.

Description:

The ⎕*ONL* function returns an integer scalar whose value is 1 if the current APL session is in timesharing mode or 0 if the session is in batch processing mode.

This function allows programs to determine whether to provide prompts for input or to allow a timesharing user to make corrective actions in some situations and provide default actions in batch.

Example:

```
      ⎕ONL
1
```

⎕*OVH* Function (Overhead Time)

Syntax:

```
      I←⎕OVH
```

Parameters:

I is an integer scalar.

Description:

The ⎕*OVH* function returns an integer scalar containing the CPU execution time overhead since entering APL. The time returned is in units of milliseconds. Overhead time is defined as CPU time expended while executing within the CP-6 operating system and not while executing within the APL process.

Example:

```
      ⎕OVH ◊ ⎕OVH,⎕OVH,⎕OVH,⎕OVH
227
231 231 231 231
```

In this example, the first line of output indicates that 227 milliseconds of CPU time of overhead have been used since entering APL. In the second line of output, the overhead CPU time is constant since all four values are obtained without incurring any monitor service time. The different values reflect the monitor processing involved in writing the output to the terminal.

⎕*RM* Function (Room)

Syntax:

```
      V←⎕RM N
```

Parameters:

N is a namelist.

V is a simple integer vector.

Description:

The ⎕*RM* function returns an integer vector whose *I*'th value is equal to the number of bytes of workspace occupied by the *I*'th name in *N*.

Example:

```
     ⎕RM ⎕←⎕NL 2 3
A
C
F
16 16 144
```

In this example, the name *A* occupies 16 bytes, *C* occupies 16 bytes, and *F* occupies 144 bytes of workspace. Expunging these names does not necessarily return that amount of workspace because in CP-6 APL values can be shared with other names.

⎕*RMG* Function (Global Room)

Syntax:

```
      V←⎕RMG N
```

Parameters:

N is a namelist.

V is a simple integer vector.

Description:

Like ⎕*RM* except that the size in bytes is of the global referents of the names in *N*.

⎕*SCT* Function (Session Time)

Syntax:

```
      I←⎕SCT
```

Parameters:

I is an integer scalar.

Description:

The ⎕*SCT* function returns an integer scalar whose value is the number of milliseconds that have elapsed since APL was invoked.

Example:

```
      0 60 60 1000⊤⎕←⎕SCT
1625740
0 27 5 740
```

In this example, 27 minutes, 5 seconds, and 740 milliseconds have elapsed since APL was invoked.

⎕*SI* Function (State Indicator)

Syntax:

```
      X←⎕SI
```

Parameters:

X is a character vector.

Description:

The ⎕*SI* function returns a character vector with the same contents as the display of the)*SI* command. Carriage return characters are used to separate each line of the state indicator display.

Example:

```
     ∇ F
[1]    1+1 ⍝         A LINE TO STOP ON
     ∇

      1 ⎕STOP 'F'  ⍝  STOP ON LINE 1

      F            ⍝  SUSPEND F
F[1]

      )SI
F[1]  *

      ⍴⎕←⎕SI
F[1]  *
6

      F            ⍝  SUSPEND F AGAIN
F[1]

      )SI
F[1]  *
F[1]  *

      ⍴⎕←⎕SI
F[1]  *
F[1]  *
13
```

In this example, the function *F* has been suspended by setting a stop on line 1. The result of the ⎕*SI* function contains the same information as displayed by the)*SI* command.

⎕*SITEID* Function (Site ID)

Syntax:

```
      T←⎕SITEID
```

Description:

The ⎕*SITEID* niladic function returns the site-id of the current CP-6 system as a 6 item character vector.

Example:

```
      ⎕SITEID
LX8001
```

⎕*SITENAME* Function (Site Name)

Syntax:

```
      T←⎕SITENAME
```

Description:

The ⎕*SITENAME* niladic function returns the CP-6 site name as a character vector.

Example:

```
      ⎕SITENAME
LADC L66A
```

⎕*STEPCC* Function (Step Condition Codes)

Syntax:

```
      ⎕STEPCC I
```

Parameters:

I is a simple integer scalar.

Description:

The ⎕*STEPCC* function specifies the value for the step condition code when APL exits. The last value specified will be used. This value may be interrogated in IBEX statements that follow the execution of APL.

Example:

```
      ⎕STEPCC 4
```

⎕SYSID Function (Sysid)

Syntax:

```
      I←⎕SYSID
```

Parameters:

I   is a simple integer scalar.

Description:

The ⎕*SYSID* niladic function returns the sysid of the current CP-6 user as a scalar integer. This number is used by the CP-6 system to identify output to devices (like line printer output) and to schedule and run batch jobs.

Example:

```
      ⎕SYSID
38200
```

⎕TS Function (Time Stamp)

Syntax:

```
      V←⎕TS
```

Parameters:

V   is a 7-item integer vector.

Description:

The ⎕*TS* function returns a 7-item integer vector whose individual items are the current year, month, day, hour, minute, second, and millisecond. The actual time returned depends on the setting of the time in the CPU that CP-6 APL is running on.

Example:

```
      ⎕TS
1984 7 1 11 15 15 450
```

In this example, the current date and time is July 1, 1984 at 11:15 AM and 15.450 seconds.

⎕*TT* Function (Terminal Type)

Syntax:

```
    I←⎕TT
```

Parameters:

I is a simple integer scalar.

Description:

The ⎕*TT* function returns an integer scalar whose value indicates the character set that APL uses to output to the terminal (or home device). The value returned by this function reflects the value determined when APL was invoked or the most recent value specified by the)*TERMINAL* system command. The possible values for terminal type are indicated by Table 11-2. When the output device is not capable of producing the full APL character set, ASCII mnemonics (defined in Appendix B) are used to output those characters which are not available. The choice of the mnemonics used for output depend upon whether the terminal type indicates support for lowercase and overstrikes.

Table 11-2. CP-6 APL Terminal Types

Number	Description
1	Full APL character set
2	ASCII character set, upper case only
3	ASCII character set, upper case only, overstrikes
4	ASCII character set, upper and lower case
5	ASCII character set, upper and lower case, overstrikes
13	Full APL character set
14	Full APL character set

Example:

```
      ⎕TT
1
```

⎕*UA* Function (User Account)

Syntax:

```
    T←⎕UA
```

Parameters:

T is a simple character vector of length 8.

Description:

The ⎕*UA* function returns a character vector containing the CP-6 account number under which the APL user is logged in.

Example:

```
      ⍴⎕←⎕UA
123TEST
8
```

⎕*UL* Function (User Load)

Syntax:

```
      I←⎕UL
```

Parameters:

I is a simple integer scalar.

Description:

The ⎕*UL* function returns an integer scalar indicating the number of users that are currently using the CP-6 system.

Example:

```
      ⎕UL
115
```

⎕*VI* Function (Verify Input)

Syntax:

```
      V←⎕VI T
```

Parameters:

T is a simple character scalar or vector.

V is a simple numeric vector.

Description:

The result has the same length as ⎕*FI T*. Each item of *V* is either 1 meaning that the corresponding item in ⎕*FI T* is a valid representation of a number or 0 meaning that the corresponding item of ⎕*FI T* does not represent a number.

Example:

```
      INPUT←'22  3X̲5  ¯100.5  1E999  1E6  0.0'

      ⎕VI  INPUT
1 0 1 0 1 1
      (⎕VI INPUT)/⎕FI INPUT
22 ¯100.5 1000000 0
```

Possible Errors:

A *RANK ERR* is reported if:

- *T* is not a scalar or vector.

A *DOMAIN ERR* is reported if:

- *T* contains an item that is not a character scalar.

⎕*VERSION* Function (Version)

Syntax:

```
      T←⎕VERSION
```

Description:

The ⎕*VERSION* niladic function returns the current version of APL as a character vector.

Example:

```
      ⎕VERSION
D00
```

⎕*WA* Function (Workspace Available)

Syntax:

```
      I←⎕WA
```

Parameters:

I   is an integer scalar.

Description:

The ⎕*WA* function returns an integer scalar whose value indicates the number of unused bytes in the active workspace. This space is available for the storage of data and defined functions.

Example:

```
      ⎕WA
1048360
```

⎕WSID Function (Workspace Identifier)

Syntax:

```
      T←⎕WSID
```

Parameters:

T is a character vector.

Description:

The ⎕WSID function returns a character vector containing the name of the active workspace.

Example:

```
      ⍴⎕←⎕WSID
CLEAR WS
8
      )LOAD APLANAR.X
 APLANAR SAVED 12:18 NOV 10 '84

      ⍴⎕←⎕WSID
APLANAR.X
9
```

Shared Variable System Functions

CP-6 APL provides the ability to share values between users of the system via the shared variable facility. A variable may only be shared by two users although each user can potentially share many (currently 16) variables. When a shared variable is used, it is indistinguishable from any other variable. It may be assigned a value and its value may be referenced. At any time, a shared variable has only one value, that is, the last value assigned by one of the partners.

All variables (including shared variables) have a degree of coupling associated with them which indicates the status of any shares associated with them. The degree of coupling indicates whether a variable is shared or not shared. The degrees of coupling are:

0. this name is not currently a shared variable

1. this name is a shared variable that has been offered but not yet accepted by another user.

2. this name is a shared variable that has been offered and matched (accepted) by another user.

The term processor is often used to describe each of the partners sharing a variable. In this respect, each processor is identified by the account it is logged on to and an optional string of 12 characters that enables multiple users logged onto the same account to simultaneously use the shared variable facility (see ⎕*SVN*). Using the functions ⎕*SVQ* or dyadic ⎕*SVO* before a unique identification has been established results in a *NO SHARES* error being reported.

Namelists for the shared variable functions are slightly different than namelists used with other system functions. For shared variable functions, a vector is treated as a one row matrix. Each row of a shared variable function namelist may contain one or two names. The first (or only) name in each row designates the name to use for the shared variable in the active workspace. The second (or only) name in the row designates the name that is to be matched by the sharing process. This permits a single shared variable user to share the name *A* with many processes where in fact each instance of a share actually references a unique (different) name in the user's workspace.

The following are the shared variable system functions.

⎕*SVC* Function (Shared Variable Controls)

Syntax:

```
R←⎕SVC N

R←C ⎕SVC N
```

Parameters:

N is a namelist.

R is a simple integer matrix of shape N-by-4.

C is a simple logical scalar, one-item vector, four-item vector or N-by-4 matrix.

Description:

For monadic ⎕*SVC*, the explicit result is an array of shape ((¯1↓⍴*N*),4) giving, in each row, the current combined shared-variable access control vector for the corresponding rows of *N*. For a row which does not denote the name of a variable with a degree of coupling of at least one, zeros are given.

For dyadic ⎕*SVC*, the effect is to set the access controls. The explicit result is an array whose shape is ((¯1↓⍴*N*),4) and whose value is the new combined shared variable access control for the corresponding rows of *N*. The ⎕*SVC* function adds the active workspace contribution to the shared variable access control vector. Setting this vector permits two separate processes to coordinate (or synchronize) their use of a shared variable. The access control vector for a shared variable is a vector of four items whose values are 0 or 1 to turn specific controls off or on.

When a process is blocked from accessing or setting a shared variable by the access control vector, it will wait until the variable's state has changed to an unblocked state before proceeding with execution. The positions (in origin 1) of the access control vector and their meanings are:

1. If 1, then once the shared variable's value has been set in the active workspace, the partner (the other sharing process) must reference it before the variable can be set again in the active workspace.

2. If 1, then once the shared variable's value has been set by the partner, the active workspace must reference it before the partner can set the value again.

3. If 1, then once the shared variable's value has been referenced in the active workspace, the partner must set a value before the active workspace can reference it again.

4. If 1, then once the shared variable's value has been referenced by the partner, the active workspace must assign a value before the partner can reference it again.

NOTE: Both sharing partners contributions are combined to obtain the current settings of the access control vector.

Example:

```
          ⎕SVC  3 3⍴'BA BJ SO '
0 0 0 0
1 1 0 0
0 0 1 1

          1  ⎕SVC  3 3⍴'BA BJ SO '
0 0 0 0
1 1 1 1
1 1 1 1

          0 0 1 1  ⎕SVC  2 3⍴'BJ SO '
0 0 1 1
0 0 1 1

          (2 4⍴1 1 0 0 0 0 1 1) ⎕SVC  2 3⍴'BJ SO '
1 1 0 0
0 0 1 1
```

⎕SVO Function (Shared Variable Offer)

Syntax:

```
          R←⎕SVO N

          R←P ⎕SVO N
```

Parameters:

N is a namelist.

R is a simple integer vector.

P is a character scalar vector or matrix which identifies one or (×/¯1↓⍴*N*) accounts.

Description:

For monadic ⎕*SVO*, the explicit result is a numeric vector giving the degree of coupling for each row of *N*: 2 if shared; 1 if there is an unmatched offer to another processor; 0 if not offered.

For each row of *N*, dyadic ⎕*SVO* tenders an offer to the corresponding account if the first (or only) name in that row was not previously offered and is not already in use as the name of an object other than a variable. The explicit result is a vector giving the degree of coupling in effect after the offer for each name or pair. If a second name is used, the second name (surrogate name) is used only for matching offers. An empty vector *P* is used to denote a general offer: an offer to share a variable with any account whose offer otherwise matches.

The left argument must be a vector of length 0 through 20 or an N-by-20 matrix of processor identifications. The first 8 characters of a processor identification is the logon account and the remaining 12 characters are the name specified in the right argument of ⎕*SVN*.

Examples:

```
      ⎕SVO  2 3ρ'A B C  '
0 0
      '905APL' ⎕SVO  2 3ρ'A B C  '
1 1
      ⎕SVO 'A'
1
      ⎕SVO 'A X'
0
      ⎕SVO 'A B'
1
```

The above example demonstrates the offer of two variables (*A* and *C*) to account 905APL. Account 905APL would see the offers of names *B* and *C* from this account.

Possible Errors:

A *DOMAIN ERR* is reported if:

- a row of *N* contains other than one or two variable names.

A *SV QUOTA EXHAUSTED* is reported if:

- more offers were made than the quota allotted by the system.

A *LENGTH ERR* is reported if:

- a processor identification has more than 20 characters.
- the number of processor identifications is not equal to 1 or the number of names being offered.

⎕*SVQ* Function (Shared Variable Query)

Syntax:

```
      R←⎕SVQ P
```

Parameters:

P is a character scalar or vector.

R is a simple character matrix.

Description:

When *P* is non-empty, the result is a character matrix of names offered by account *P* to this user, either explicitly or generally, but not currently shared. If *P* is an empty vector, the result is a vector which identifies any accounts with unmatched offers to share variables with this user.

Example:

```
      ⎕SVQ ''     ⍝ WHO IS OFFERING US SOMETHING
905APL  ARES
ERSTEST
```

In this example, two accounts are found to be offering the current APL user variables to share. In the following example, the shares offered by account 905APL are matched using ⎕SVO.

```
      P←⎕SVQ ''
      P
905APL  ARES
ERSTEST

      N←⎕SVQ  P[1;]
      N
STATUS
DATA

      P[1;] ⎕SVO  N
2 2
```

⎕SVR Function (Shared Variable Retract)

Syntax:

```
      R←⎕SVR N
```

Parameters:

N is a namelist.

R is a simple integer vector.

Description:

Ends sharing of any variables named in *N*. The result is the degree of coupling before retraction (compare ⎕SVO, above). This function may cause a *WS FULL* error to occur obtaining the current value of the names being retracted.

Examples:

```
      ⎕SVO 'A'  ⍝  PRINT 2 IF A IS SHARED.
2

      ⎕SVR 'A'
2
      ⎕SVR 'A'  ⍝  IT IS NOT SHARED NOW.
0
```

⎕*SVS* Function (Shared Variable State)

Syntax:

R←⎕*SVS N*

Parameters:

N is a namelist.

R is a simple integer matrix of shape N-by-4.

Description:

The result is a numeric array of shape ((¯1↓ρ*N*),4) giving in each row the current shared variable state matrix for the names in *N*. For variables that are not currently shared their state is given as all zeros.

Each row of the result of ⎕*SVS* has four possible values:

0 0 0 0 this is not a shared variable

0 0 1 1 value set by one processor and has been referenced by the other.

0 1 0 1 value set by partner, but not yet referenced in the active workspace.

1 0 1 0 value set in active workspace, but not yet referenced by the partner.

Example:

```
      ⎕SVS  'A'
0 0 1 1
      A←5

      ⎕SVS  'A'
1 0 1 0
```

⎕*SVN* Function (Shared Variable Process Name)

Syntax:

I←⎕*SVN T*

Parameters:

T is a simple character scalar or vector.

I is the simple integer scalar containing the value 0 or 1.

Description:

If shared variables are to be used by multiple users on the same CP-6 account, then this function permits each user to uniquely identify their own process. *T* is a vector of up to twelve characters. If a unique identifier is established the result is 1. If there are currently any shares offered by this process or if the value specified in *T* does not create a unique identifier, the result is 0. If successful, this process is uniquely identified by the eight character CP-6 account followed by 12↑*T*.

Example:

```
      ⎕SVN 'BRUCE'
1
      ''⎕SVO 'A'
1
      ⎕SVN 'ME'
0
```

In the example above, the second execution of ⎕*SVN* returned 0 because a name (*A*) was currently shared.

Possible Errors:

A *SV QUOTA EXHAUSTED* error is reported if:

- o an attempt is made to use shared variables before establishing a unique identifier (the default is blanks).

⎕*SC* Function (State Change)

Syntax:

```
      I←⎕SC
```

Description:

The ⎕*SC* function causes execution of the current line to halt until the state of one of this processes shared variables changes or an explicit offer is made to this process. The result is 1 if a unique processor identifier exists and zero if one does not.

Text Editing System Functions

CP-6 APL provides six text editing functions which facilitate the examination and modification of character vectors.

⎕*TIX* Function (Text Index)

Syntax:

R←⎕*TIX T SDV TDV DDV*

Parameters:

T must be a simple character vector containing the string to tokenize.

SDV must be a simple character scalar or vector defining those characters that are token separators.

TDV must be a simple character scalar or vector defining those characters that are single character tokens and token separators.

DDV must be a simple character vector defining those character pairs that create delimited character strings.

R is an N–by–2 integer matrix containing starting positions in the first column and lengths in the second column.

Description:

The ⎕*TIX* function returns an integer matrix of *N* rows and 2 columns. The first column contains the starting index of each token in *T*. The second column contains the corresponding length of each token in *T*.

The definition of a token in *T* is governed by the arguments *SDV*, *TDV*, and *DDV*. The *SDV* items are token separators and are never tokens themselves (for example, blanks are skipped this way). The *TDV* items are token separators and are also single character tokens themselves (like + is in APL). The *DDV* items are token separators and also create a delimited token (like quote strings in APL). Finally, characters not appearing in *SDV*, *TDV*, and *DDV* are a single token when occurring consecutively (like identifiers in APL).

The scan of *T* starts at the first index position and continues until a character from the *SDV*, *TDV*, and *DDV* vectors is found or the last index position of *T* has been examined. The order of evaluation is as follows:

- Characters in *T* which occur in the *SDV* vector are simply skipped over when they are encountered.

- When the character encountered is not in the *SDV*, *DDV*, or *TDV* vectors, the vector *T* is scanned from this point until a character in one of those vectors is found. A new row is added to the result indicating the position of the first character not in *SDV*, *TDV*, or *DDV* and whose length includes the characters up to but not including the character found that are in *SDV*, *TDV*, or *DDV*.

- When the character encountered is in the *TDV* set, a token is added that indicates the character in *TDV* that was encountered.

- When the character encountered is in the *DDV* set, a delimited string token is added to the result. The delimited string is defined by treating the *DDV* vector as an *N* by 2 matrix and using the first character in each row as a delimited string starter and the corresponding second character in the same row as the terminator. If the delimited string starter and terminator are the same character, it may appear within the string by doubling it. All characters between the string start and end are treated as a single token.

  If the delimited string starter and terminator are separate characters, the first terminator character found terminates the delimited string.

- The scan of the vector *T* continues, searching for characters in the *SDV*, *DDV*, and *TDV* sets until the last index of the vector *T* is scanned.

DDV need not be specified if empty, and *DDV* and *TDV* need not be specified if both are empty.

Examples:

The following examples use the *LIST* function to display the tokens returned by the ⎕*TIX* function. It works only in origin 1 and displays one token per row with the "." character indicating characters which are not part of the actual token.

```
     ∇R←A LIST B;J;K
[1]  R←('.',A)[1+(B[;1]∘.+J)×K∘.>J←¯1+⍳⌈/K←B[;2]]
     ∇
```

The first example demonstrates using blanks and commas as delimiters which are not themselves delimiters.

```
       L LIST ⎕TIX (L←'THIS IS A,TEST') ' ,'
THIS
IS..
A...
TEST
```

The next example demonstrates using a dieresis character to indicate a delimited string.

```
        L LIST ⎕TIX (L←'THIS IS ¨A TEST¨¨S¨+WOW,4') ' ' '' '¨¨'
 THIS.......
 IS.........
 ¨A TEST¨¨S¨
 +WOW,4.....
```

Notice that in the above example, the dieresis is doubled within the token to continue the production of the token. The following example demonstrates using token separators which are tokens themselves.

```
        L LIST ⎕TIX (L←'THIS IS ¨A TEST¨¨S¨+WOW,4') ' ' '+,' '¨¨'
 THIS.......
 IS.........
 ¨A TEST¨¨S¨
 +..........
 WOW........
 ,..........
 4..........
```

In the above example, + and , are token separators and appear as tokens. The character ¨ is a delimited token character in the above example.

Possible Errors:

A *DOMAIN ERR* is reported if:

- a terminating character cannot be found in the *T* string.
- a *DDV* or *TDV* character appears more than once in *SDV*, *DDV* or *TDV* vectors.
- a delimited string terminating character is found that is not within a delimited string.
- *T*, *SDV*, *TDV*, or *DDV* contains any item that is not a simple character scalar.

A *RANK ERR* is reported if:

- *SDV* or *TDV* is not a scalar or vector.
- *T* or *DDV* are not vectors.

A *LENGTH ERR* is reported if:

- the length of *DDV* is not a multiple of 2.
- the right argument to ⎕*TIX* contains more than 4 items or fewer than two items.

⎕*TLEX* Function (Text Lexemes)

Syntax:

```
      R←⎕TLEX T SDV TDV DDV
```

Parameters:

T is a simple character vector containing the string to tokenize.

SDV is a simple character scalar or vector defining the token separator characters.

TDV is a simple character scalar or vector defining the single character tokens.

DDV is a simple character vector defining the character pairs that create delimited tokens.

R is a vector of character vectors.

Description:

This function tokenizes the string *T*, returning a vector where each item was a token found in *T*. The tokenization uses the same method as the ⎕*TIX* function.

Example:

```
      ρ ⎕←⎕TLEX 'THIS IS A TEST'   ' '
THIS  IS  A  TEST
4

      ρ ⎕←⎕TLEX 'TOMATOES=FRUIT,SALMON ARE FISHY'   ' '   '=,'
TOMATOES  =  FRUIT  ,  SALMON  ARE  FISHY
7
```

⎕*SSS* Function (Substring Search)

Syntax:

```
      R←⎕SSS T SS
      R←⎕SSS T SS FCOL
      R←⎕SSS T SS FCOL LCOL
```

Parameters:

T is a simple character vector.

SS is a simple character vector.

R is a simple integer vector.

FCOL is a simple integer scalar index of T.

LCOL is a simple integer scalar index of T.

Description:

The result is a vector of the starting indices in *T* of each non-overlapping occurrence of *SS*. If the string *SS* does not occur in *T* then the result is an empty vector.

FCOL and *LCOL* frame the indices of *T* to be searched for occurrences of *SS*. *FCOL* is the first index of *T* and *LCOL* is the last index of *T* that will be searched for an occurrence of *SS*. When not specified *FCOL* defaults to ⎕*IO* and *LCOL* defaults to (⍴*T*)-1+⎕*IO*.

Examples:

TV is a vector of length 100.

R←⎕*SSS TV 'BOB'* provides the starting indices of all occurrences of *'BOB'* in *TV*.

R←⎕*SSS TV 'BOB'* 30 provides the starting indices of *'BOB'* in *TV* from *TV*[30] to the end of *TV*.

R←⎕*SSS TV 'BOB'* 30 60 provides the starting indices of all occurrences of *'BOB'* in *TV* from *TV*[30] to *TV*[60].

The result *R*, *FCOL* and *LCOL* are origin dependent.

Possible Errors:

A *DOMAIN ERR* is reported if:

- the right argument is not a 2, 3, or 4 item list.
- *T* or *SS* is not a character scalar or vector.
- *FCOL* or *LCOL* are not integer indices.

A *LENGTH ERR* is reported if:

- *FCOL* or *LCOL* are not scalars or one-item vectors.

An *INDEX ERR* is reported if:

- *FCOL* or *LCOL* are not valid indices of *T*.

⎕*SSR* Function (String Search and Replace)

Syntax:

```
R←⎕SSR  T SS RS
R←⎕SSR  T SS RS FCOL
R←⎕SSR  T SS RS FCOL LCOL
```

Parameters:

T is a character vector.

RS is a character vector.

SS is a character vector.

R is a character vector.

FCOL is a simple integer scalar index of T.

LCOL is a simple integer scalar index of T.

Description:

The result is a character vector like *T* except that all non-overlapping occurrences of *SS* are replaced by *RS*. *FCOL* and *LCOL* indicate the range of indices of *T* subject to replacement. That is, only occurrences of *SS* in the range *FCOL*+⍳*LCOL*-*FCOL* are replaced.

Examples:

TV is a character vector containing names separated by blanks.

R←⎕*SSS TV* ' ' ⎕*AV*[13+⎕*IO*] replaces all blanks with carriage returns.

R←⎕*SSS TV* ' ' ⎕*AV*[13+⎕*IO*] 20 replaces blanks from *TV*[20] to the end with carriage returns.

R←⎕*SSR TV* ' ' ⎕*AV*[13+⎕*IO*] 20 30 replaces blanks from *TV*[20] through *TV*[30] with carriage returns.

R←⎕*SSR TV* '*BOB*' '*ROBERT*' replaces all occurrences of '*BOB*' with '*ROBERT*'.

R←⎕*SSR TV* '*PIERRE*' '' removes all occurrences of '*PIERRE*' from *TV*.

⎕*SRP* Function (Substring Replace)

Syntax:

```
R←⎕SRP T RS FCOL LCOL
```

Parameters:

T is a character vector.

RS is a character scalar or vector.

FCOL is an index of T.

LCOL is an index of T.

R is a character vector.

Description:

The result is a character vector like *T* with the location *T*[*LCOL*] through *T*[*LCOL*] replaced by *RS*.

Examples:

```
⎕IO←1

TV←'THE PRICE OF PRODUCT-NAME IS'

RS←'WHEATIES'

R←⎕SRP TV RS 14 25

R
THE PRICE OF WHEATIES IS
```

Possible Errors:

A *DOMAIN ERR* is reported if:

- o the right argument is not a four item list.
- o *T* is not a character vector.
- o *RS* is not a character vector or scalar.
- o *LCOL* or *FCOL* are not numeric scalars or 1-item vectors.

An *INDEX ERR* is reported if:

- o *FCOL>LCOL* or if *FCOL* or *LCOL* are not valid indices of T.

⎕*SCP* Function (String Compare)

Syntax:

```
R←⎕SCP (A;B)
```

Parameters:

A is a character vector.

B is a character vector.

Description:

The ⎕*SCP* function returns a two item numeric vector, the first item of which is 0 if *A* is equal to *B*, or 1 if *A* is greater than *B*, or 2 if *A* is less than *B*. The second item of the result is the first index in *A* that *A*[*R*[2]]≠*B*[*R*[2]] or if *A* is equal to *B*, then *R*[2]←¯1. If *A* is longer than *B* and every item of *B* is equal to every item of *A*, then *R*←2 ¯1.

Possible Errors:

A *DOMAIN ERR* is reported if:

- o the right argument is not a two item list.
- o *A* or *B* is not a character vector.

Terminal I/O System Functions

These system functions return information about or control a terminal session.

⎕*TIN* Function (Terminal Input)

Syntax:

I ⎕*TIN* *T*

Parameters:

T is a simple character vector or scalar.

I is a simple integer scalar.

Description:

The right argument must be a character vector which replaces the current terminal re-read line. If the optional left argument is present when the re-read line is recalled, the value of the left argument is used as the column to position to.

⎕*TATTR* Function (Terminal Attributes)

Syntax:

V←⎕*TATTR*

Description:

The ⎕*TATTR* function returns a simple integer vector containing terminal status information. The vector may in a future release be extended to contain additional information. Currently the vector contains:

1 Line speed (CPS)

2 Parity (even=2, odd=1, none=0, one=3, zero=4)

3 Dial-up/hardwired/foreign net (0=dial-up, 1=hardwire, 2=NET)

4 Normal/multi-drop (0=normal, 1=multi)

5 Character set (0=ASCII, 1=bit paired, 2=type paired)

6 Lowercase=1

7 Screen width (characters)

8 Screen height (lines)

9 Blank erases (1=yes, 0=no, 2=not applicable)

10 Scroll (0=no, 1=yes)

11 Wrap (0=no, 1=yes)

12 Retypovr (0=no, 1=yes)

13 Editovr (0=no, 1=yes)

14 Echo (0=no, 1=yes)

⎕*TTIME* Function (Terminal Timeout)

Syntax:

⎕TTIME N

Parameters:

N is a simple integer scalar.

Description:

The ⎕*TTIME* function sets the timeout period in seconds for terminal reads. After a terminal read is issued, the terminal user must complete input in N seconds or an I/O error will be reported. The I/O error is, of course, sidetrackable. The read timeout may be reset by setting the timeout value to 0.

⎕*TECHO* Function (Terminal Echo)

Syntax:

⎕TECHO L

Parameters:

L is the simple integer scalar value 0 or 1.

Description:

If *L* is zero, then terminal reads will not echo. If *L* is one, then characters typed at the terminal will echo.

⎕*TSQZ* Function (Terminal Mnemonic Translation)

Syntax:

R←I ⎕TSQZ V

Parameters:

I is the simple numeric value 0 or 1.

V must be a simple character scalar or vector.

R is a simple character vector.

Description:

If the value of the left argument is 1 then the result of this function is a character vector containing the text in the right argument translated into internal APL text. This function resolves all valid overstrikes and mnemonics into single characters.

If the value of the left argument is 0, then the result of this function is a character vector containing the text in the right argument translated into external ASCII suitable for blind output. This function generates mnemonics for the internal APL characters that are not representable with the currently set terminal type.

This function is designed to aid in the use of blind I/O and APL characters with blind I/O.

⎕*TWINDOW* Function (Terminal Windows)

Syntax:

```
M←⎕TWINDOW
```

Parameters:

M is a matrix of shape N-by-8 containing information about each of the currently defined logical devices that refer to the terminal.

Description:

The result of the ⎕*TWINDOW* function is a matrix which has one row for each logical device that refers to a terminal (device UC). The information returned indicates the positioning and size of each window associated with the device, and whether or not the window can be used to create another window.

Table 11-3 summarizes the contents of the result matrix. Note: A future release of CP-6 APL may return additional information by adding trailing columns to the result of this function.

The logical terminal devices 1, 98, and 99 always start a session referring to the same window. The system command)!*LDEV* may be used to create additional logical devices or to modify the definitions of existing devices. The)*SET* command may be used to direct APL input/output or blind I/O to any logical device. IBEX LDEV command options include the ability to specify a window size and position relative to the window being used to create the new window.

Table 11-3. Window Column Descriptions

Column	Description
1	Contains the logical device number. For example, if this value is 98, then the remaining columns of this row describe the device UC98.
2	Contains the line number on the screen of the top line of the window. The top-most line on the screen is 1.
3	Contains the column number on the screen of the left side of the window. The left-most column on the screen is 1.
4	Contains the number of lines in the window.
5	Contains the width of the window.

Table 11–3. Window Column Descriptions (cont.)

Column	Description
6	Contains the minimum number of lines that must be available for this window. This value limits the number of lines that may be taken from this window to form a new window.
7	Contains the minimum width that must be available for this window. This value limits the number of columns that may be taken from this window to form a new window.
8	Contains either the value 0 indicating that the window is removable, or the value 1 indicating that the window is not removable.

Example:

```
      ρZ←⎕TWINDOW
4 8
      Z
98 13 1 12 80 0 0 0
 1 13 1 12 80 0 0 0
99  7 1  6 80 0 0 0
 5  1 1  6 80 0 0 0
```

In the preceding example, the APL session has 4 logical devices known as UC98, UC01, UC99 and UC05. Logical devices 1 and 98 share the same window on the screen, beginning at row 13, column 1. The window is 12 lines long and 80 columns wide, have no minimum length or width, and are removable.

Section 12

CP-6 APL File I/O

CP-6 APL provides access to all CP-6 file types. Records within these files can be read or written as non-APL records, APL datablock records, or APL component records.

APL datablock records are read and written along with the data type, rank, and dimensions. This permits arrays to be written to a file and later read back as the same array.

APL component records (the default) are read and written with the data type, rank, dimensions, timestamp, and account identifier of the user that wrote the record. In addition to the capability associated with datablock records, there is a system function which operates on this record format to obtain the component information (timestamp and account identifier). Figure 12-1 shows the component record format used by APL.

A datablock record has similar format except that the first nine words of the component record format are omitted.

Non-APL records are typically files created by other CP-6 programs. Non-APL records may be read or written in a number of ways. The easiest method is to treat the record contents as a simple character vector when reading and writing the raveled data. This form excludes APL's internal type, rank, and shape information. Using this mode, datatype conversions are the programmer's responsibility. Other functions (such as ⎕*CVT* and ⍎) are available to aid in the datatype conversions.

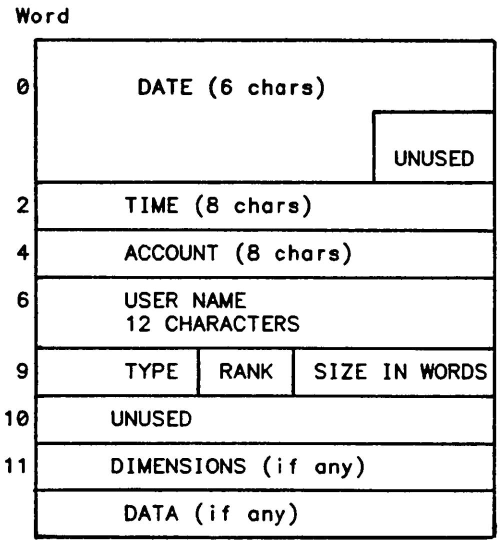
The APL file I/O record types and descriptions are summarized in Table 12-1. The record type numbers indicated in this table are used to indicate the type of record to read or write.

When records are read or written, an encryption seed may be specified to protect the data in the file. If the wrong seed is provided on a read of a component or datablock record, APL informs the user that this is NOT AN APL FILE. If a non-APL record is read with an incorrect seed, the data returned is an encrypted version of the actual data.

Records within a file can be accessed (read or written) sequentially or by record identifier. The record identifier can be specified as an integer number in the range 1 through 134217726 or as a character vector of 1 to 255 characters. Record numbers need not be contiguous; record number 3 can be followed by record number 10099. New records can be inserted in the future and existing records may be deleted (or dropped).

Up to 31 files can be accessed simultaneously. Each file is known to APL by its stream number which is specified when opening (or tying) the file. Stream numbers are integer values in the range 1 through 34359738367. Once a file is opened, it remains open until it is closed using one of the functions ⎕*FCLOSE*, ⎕*FCLEAR*, ⎕*FERASE* or until the APL session ends.

A file stream is not affected by changing the active workspace. In particular the system commands)*LOAD* and)*CLEAR* have no effect upon the files which have been opened.



Word			
0	DATE (6 chars)		
			UNUSED
2	TIME (8 chars)		
4	ACCOUNT (8 chars)		
6	USER NAME 12 CHARACTERS		
9	TYPE	RANK	SIZE IN WORDS
10	UNUSED		
11	DIMENSIONS (if any)		
	DATA (if any)		

Figure 12-1. File I/O Component Record Format

CP-6 file management provides access controls to prevent unauthorized file access. Each file may be passworded and various levels of access are possible once the file is open. For example, READ access permits accounts to be specified that may only read records, WNEW access permits accounts to write new records, UPDATE allows accounts to replace existing records. For more information on file access see the ⎕*FSTAC* and ⎕*FRDAC* functions.

When CP-6 files are created, the system allocates an initial extent and as the file space is used up, the CP-6 system automatically extends the file until the file space limit for the account or packset is used up. Thus, in CP-6 APL there is little need to worry about file size when allocating files.

By default, APL users create keyed files, that is, files whose individual records are identified by a one to 255 item character vector. However, file access within APL is not restricted to this file type. Indexed files (most commonly created by COBOL), relative files, indexed-relational files, random files, unit record files, fixed files, and consecutive files are all accessible from APL. Each of these files have different capabilities (for more information see the CP-6 Host Monitor Services Reference Manual (CE74)).

Table 12-1. File I/O Record Types

Type	Description
1	Component record. The record includes the APL datatype, rank, shape, date, time, account, user name and data in ravel order (default).
2	Datablock record. The record includes the APL datatype, rank, shape and data in ravel order.
3	Data record. Only the actual data is read or written. Reading always returns a character vector.
4	Record field description. If the file was created with a record field description, this is used to read or write the record. The data read or written is always a vector (possibly nested).

File Information Functions

The functions in this group provide information about the files that are currently being accessed.

⎕*FNUMS* Function (Numbers of Open Files)

Syntax:

```
      R←⎕FNUMS
```

Description:

The ⎕*FNUMS* niladic function returns an integer vector containing the stream numbers for the files currently open.

Examples:

```
      ⎕FNUMS
1 314159
```

⎕*FNAMS* Function (Names of Open Files)

Syntax:

```
      R←⎕FNAMS
```

Description:

The ⎕*FNAMS* niladic function returns a character matrix showing the names of files currently open.

Example:

```
      ⎕FNAMS
        *TEST
        TIMING
```

⎕*FID* Function , (File Identifier)

Syntax:

```
      ⎕FID Y
```

Parameters:

Y is a simple integer scalar indicating a file I/O stream that is currently open.

Description:

The ⎕*FID* system function returns the CP-6 file identifier for the file I/O stream specified.

Example:

```
      ⎕FID 1
*TEST

      ⎕FID 314159
TIMINGS.TESTAPL

      ⎕FID¨ 1 314159
*TEST  TIMINGS.TESTAPL
```

Opening, Closing, and Deleting Files

The file functions in this group are used to initiate and terminate access to files.

⎕*FOPEN* Function (Open File)

Syntax:

```
      X ⎕FOPEN T
```

Parameters:

T is a simple integer scalar indicating an available file I/O stream.

X is a simple character vector indicating the name of the file to open or a vector of nested arrays containing a simple text vector indicating the name of the file to open, and optionally a file access matrix, a record field matrix, and a key definition matrix.

Description:

The ⎕*FOPEN* system function is used to initiate file access through a stream. The left argument is the file identifier and the right argument is the stream number to be associated with this file. The number used for the stream number must be an integer and not currently in use as a stream number.

If the left argument is a simple character vector containing a CP-6 file identifier, that file is opened for reading only. In order to create, update, or share a file, options are specified in the left argument following the FID and separated by commas. The options and their meanings are provided in Table 12-2.

Table 12-2. File Open Options

Option	Meaning
UPDATE	Opens file so records can be read and written.
*IN	Opens file for reading only.
CREATE	Creates a file.
OLDFILE	For create, if file already exists use it.
*NEWFILE	For create, even if file already exists create new file.
ERROR	For create, if file already exists report error.
ALL	Share file (multiple updaters).
*NONE	If UPDATE, file open is for exclusive use. If IN, file open shares with other readers only.
SHAREIN	Only readers can share file.
CTG	For create, catalogues file in directory upon open.
SCRATCH	File is not permanent.
*NAMED	File is permanent.
CONSEC	For create, specifies organization of new file.
*KEYED	For create, specifies organization of new file.
RANDOM	For create, specifies organization of new file.
UR	For create, specifies organization of new file.
RELATIVE	For create, specifies organization of new file.
INDEXED	For create, specifies organization of new file.
CG	For create, specifies organization of new file.
IREL	For create, specifies organization of new file.
*DIRECT	
SEQUEN	
LOAD	For create of alternate index file, build indices on file close.
COMP	CP-6 file management will compress records.
REASSIGN	Uses IBEX !SET F$tie for additional options.

*denotes defaults

If one of the option fields is not a valid option and is exactly two characters in length, it is used as the file type. Additional information on the meanings of these options can be found in the CP-6 Host Monitor Services Reference Manual (CE74).

The left argument of the ⎕*FOPEN* system function may also be a vector of nested arrays which permits the specification of file access controls, record-field definitions, and alternate key definitions. The topic named Specialized File Options (later in this section) contains information on this usage.

Examples:

```
    'TEMP' ⎕FOPEN 1

    'COMMON.HISACCT,UPDATE,ALL' ⎕FOPEN 99

    APLSTUFF,DC̲,CREATE,ERROR,CTG'⎕FOPEN 1234

    'DP≠PACK3/'MYSTUFF,UPDATE'⎕FOPEN 31415926
```

⎕*FCLOSE* Function (Closing and Renaming Files)

Syntax:

```
      ⎕FCLOSE TV

      FID ⎕FCLOSE Y
```

Parameters:

TV is a simple integer vector indicating file I/O streams that are currently open.

Y is a simple integer scalar indicating a file I/O stream that is currently open.

FID is a simple character vector indicating the new name or new password by which this file will be known.

Description:

The ⎕*FCLOSE* function closes the specified streams. For monadic ⎕*FCLOSE*, the right argument is a scalar or vector of stream numbers.

A file's name can be changed at closing time using dyadic ⎕*FCLOSE*. The right argument is the stream number. The left argument is the new file identifier.

Renaming a file requires DELF access in the file's access controls. The file's name, password, or both may be changed. The file's access controls (see File Access Controls) may also be modified at close time by specifying an access control matrix as the left argument. In general, the left argument to ⎕*FCLOSE* may contain a fid, access control matrix or a nested vector containing both a fid and an access control matrix.

Example:

```
      ⎕FNUMS
99 1234 31415926

      ⎕FCLOSE 1234 99

      ⎕FNUMS
31415926
```

⎕FERASE Function (Close and Delete File)

Syntax:

```
     ⎕FERASE TV
```

Parameters:

TV is a simple integer vector indicating file I/O streams that are currently open.

Description:

The *⎕FERASE* function closes the specified streams and deletes the files that were opened to them. The right argument is a scalar or vector of stream numbers. Note that once a stream has been closed, referencing it before opening it once again will result in a *FILE TIE ERR*.

Example:

```
      ⎕FNUMS
99 1234 31415926

      ⎕FERASE 1234

      ⎕FNUMS
99 31415926
```

⎕FCLEAR Function (Close All Open Files)

Syntax:

```
     ⎕FCLEAR
```

Description:

The *⎕FCLEAR* system function causes all currently open streams to be closed. It is functionally equivalent to the expression:

```
     ⎕FCLOSE ⎕FNUMS
```

Reading and Writing Records

The functions in this group provide access to records within the file. Records may be accessed either sequentially or directly by specifying record number or key. The absence of the record number or key is used to indicate a sequential operation.

⎕FAPPEND Function (Append Record to File)

Syntax:

```
R←X ⎕FAPPENDR Y

  X ⎕FAPPEND Y
```

Parameters:

X is the APL array that is to be written (appended) to the file.

Y is a vector of up to 3 items. The first item must be a simple integer scalar indicating a file I/O stream that is currently open. The optional second item is a simple integer encryption seed (or a 4-element character vector). The optional third item is a simple integer scalar record type number as described in Table 12-1 (or a record field matrix).

R is the key of the record that was appended.

Description:

The *⎕FAPPEND* function writes the data object to the file at the position of the last record in the file plus the key interval of the file. The key interval can be set or obtained by using the *⎕FKEYINT* function. The key interval for files other than keyed files is always 1. The file must have been open in UPDATE or CREATE mode.

The right argument consists of the stream number, the optional encryption seed, and the optional type of record to be written. Only the stream number is required. If the encryption seed is zero or not present then the record will not be encrypted. If the third item of the right argument is omitted or 1, then an APL component record is written. If it is 2, then an APL datablock is written. If it is 3, then the ravel of the data is written. Finally, if it is 4, the record field description associated with the file is used to format the record before writing. The left argument is any APL array.

⎕FAPPENDR is identical in operation to *⎕FAPPEND*, and additionally returns the numeric key of the record written.

Examples:

```
      'FAR OUT' ⎕FAPPENDR 31415926
1

      'EXTERNAL RECORD TYPE' ⎕FAPPEND 1 0 3

      'ENCRYPTED RECORD' ⎕FAPPEND 1 998
```

⎕*FREAD* Function (Read a Record)

Syntax:

```
R←⎕FREAD Y
```

Parameters:

Y is a vector of 1 to 4 items in length (stream, key, seed, type).

R is the contents of the requested record.

Description:

The ⎕*FREAD* function is used to read records. The right argument contains the stream number and optionally the record number or key, the encryption seed and the record type. If the key is a character vector, the argument must be a nested array with the key as the second item. If the key is an empty vector, then a sequential read is performed. If a record with the specified key does not exist or if a sequential read reaches the end of the file, *FILE INDEX ERR* is reported.

As in ⎕*FAPPEND*, the encryption seed can be non-zero to request encryption and record types 1 (the default), 2, 3, or 4 may be requested. The result is the record with the specified key or the next sequential record if READ access permission has been granted.

Reading when a record type of 3 is specified always results in a character vector result.

Examples:

```
      ⎕FREAD  31415926  1
FAR OUT

      ⎕FREAD  1  2  0  3
EXTERNAL RECORD TYPE

      ⎕FREAD  1  3  998
ENCRYPTED RECORD

      ⎕FREAD  1 'TEXTKEY'
RECORD WITH TEXT KEY
```

Reading Sequentially

A sequential read may be performed by not specifying a key. For example:

```
      ⎕FREAD  5
```

where 5 is the stream number in this case. In order to read a non-APL file sequentially the following expression is used:

```
      ⎕FREAD  9 '' 0 3
```

where 9 is the stream number in this example.

⎕*FWRITE* Function (Write or Replace a Record)

Syntax:

```
X ⎕FWRITE Y
```

Parameters:

Y is a vector of 1 to 4 items in length (stream, key, seed, type).

X is the APL array that is to be written to the file.

Description:

The ⎕*FWRITE* system function causes a record to be written (new or replaced) in the file with the specified key.

The right argument contains the stream number and optionally a record number or key, an encryption seed and a record type.

If the record identifier is a character key, the right argument must be a nested array with the key as the second item. If the key is not specified and this is not an indexed or irel file, then the record last read by this stream is replaced.

A non-zero value for the encryption seed will cause the record to be encrypted before writing it. The same encryption key must be used to subsequently read it.

The record type is 1 for a component record, 2 for a datablock record, 3 for an external record, and 4 for the file's record field definition.

The file must have been opened with either the UPDATE or CREATE options and WNEW or UPDATE permission must be granted.

Examples:

```
'REPLACEMENT' ⎕FWRITE  31415926 1

'TEXT KEY' ⎕FWRITE  1  'OJ SIMPSON'  27165

'EXTERNAL WRITE' ⎕FWRITE  1  3330 0 3
```

⎕*FDROP* Function (Delete Record from File)

Syntax:

```
⎕FDROP Y
```

Parameters:

Y is a vector of length 2 (stream, key).

Description:

The ⎕*FDROP* system function deletes specific records from a file. The right argument identifies the stream and the record number or key of the record to delete.

Examples:

```
      ⎕FDROP 1 2
```

deletes record number 2. The file must be opened in either UPDATE or CREATE mode to use this function and DELR access permission must be granted.

⎕*FRDCI* Function (Return Component Information)

Syntax:

```
      R←⎕FRDCI Y
```

Parameters:

Y is a 1, 2, or 3 item vector (stream, key, seed).

R is a simple character vector of length 36.

Description:

The ⎕*FRDCI* function returns a character vector of 36 items containing the date in the format YYMMDD (e.g., 841030) in the first six items, the time in the format HHMMSSSS (e.g., 12300000 for 12:30 PM) after the blank following the date. The remaining characters are the account and user name fields (see Figure 12-1).

The right argument is the same as for ⎕*FREAD* except that the record type is not specified (this function only works on component records).

If the record was not written as a component record, then the error *NOT AN APL FILE* is reported. For example:

```
      ⎕FRDCI 1 2
840922 16410818MAGAPL  201GONE
```

File Access Controls

The functions in this group set and retrieve the current file access controls for files that are currently open. Access controls may also be set when the file is created by the ⎕*FOPEN* system function, or modified when closing the file by the ⎕*FCLOSE* system function.

File Access Matrix

An APL file access matrix is used to indicate the file access controls. Access controls permit or prevent access to files by users. Permissions are indicated in terms of accounts (which may be wild-carded) and file access permissions granted to those accounts. Table 12-3 contains the file access permissions which are available.

An APL file access matrix is a simple N-by-17 character matrix. The first eight columns contain an account identifier. The ninth column must always be blank. The remaining columns contain either the character 'Y' to permit the corresponding access or the character 'N' to restrict the access.

Example:

```
AC←1 17ρ(8↑'905APL'),' YYYYNNNN'

BC←1 17ρ(8↑'TEST?')',' YNYNNNNN'

CC←AC,[⎕IO]BC
```

In the example, *AC* is a file access matrix which permits the account 905APL to read, delete, update, write new records and see the file name in the file directory (or account).

The file access matrix *BC* permits any account beginning with the characters '*TEST*' to read, write new records, and see the file name in the file directory.

The file access matrix *CC* provides the permissions associated with *AC* and *BC* to their respective accounts.

Table 12-3. CP-6 APL File Access Permissions

Column	Permission	Description
10	READ	can use ⎕*FREAD*, ⎕*FRDCI*, ⎕*FRDAC*, ⎕*FENQ*, ⎕*FDEQ*
11	DELR	can use ⎕*FDROP* to delete records
12	WNEW	can use ⎕*FAPPEND*, ⎕*FWRITE* to write new records
13	UPDATE	can use ⎕*FWRITE* to replace records
14	DELF	can use ⎕*FSTAC*, ⎕*FERASE* or dyadic ⎕*FCLOSE*
15	NOLIST	can use ⎕*FLIB* will not list file name
16	REATTR	can use ⎕*FSTAC*
17	EXEC	not meaningful to APL files

⎕*FRDAC* Function (Return File Access Matrix)

Syntax:

```
R←⎕FRDAC Y
```

Parameters:

Y is a simple integer scalar indicating a file I/O stream that is open.

R is a simple character matrix of shape N-by-17.

Description:

The ⎕*FRDAC* function returns the APL file access matrix for the specified file. Each row of the matrix contains an account identifier (which can be wild-carded) and the corresponding file permissions. The right argument is a stream number. If the file was opened with the create option, then it must also have been opened with the CTG option.

Examples:

```
      ⎕FRDAC  1
 ?         NNNNNNN
```

⎕*FSTAC* Function (Store File Access Controls)

Syntax:

```
      M ⎕FSTAC Y
```

Parameters:

Y is a simple integer scalar indicating a file I/O stream that is open.

M is a file access matrix, that is, a simple character matrix of shape N-by-17.

Description:

When the ⎕*FSTAC* function is executed, the file opened to stream *Y* has its access controls revised to reflect the permissions specified in *M*. The right argument is the stream number of a currently open file. The left argument is a file access matrix.

Examples:

```
      (((9↑ACCOUNT),'YYNNNNYN'),[1]⎕FRDAC 1)⎕FSTAC 1
```

This example will allow the account named in the variable *ACCOUNT* to read and delete records and to change the access permissions of the file.

Coordinating Shared Files

The functions in this group are intended to be used when more than one user is accessing a file, and it is being updated by at least one user. The enqueuing protocol should be agreed upon for all applications using the file; its use is not enforced by the system.

⎕*FENQ* Function (Hold a Record)

Syntax:

R←⎕*FENQ Y*

Parameters:

Y is a vector of length 2 (stream, key).

Description:

After the ⎕*FENQ* function has executed, another user executing ⎕*FENQ* on the same file and resource name will be halted until the user currently holding the resource releases it with the ⎕*FDEQ* function.

The right argument contains the tie number and resource name. The resource name is an integer or character value, most commonly a record key.

⎕*FDEQ* Function (Release Record or File)

Syntax:

R←⎕*FDEQ Y*

Parameters:

Y is a vector of length 2 (stream, key).

Description:

When the ⎕*FDEQ* function is executed, the resource specified is released permitting another user currently waiting for this resource to continue.

The right argument contains the tie number and optional resource name.

If the resource is not specified, then all resources currently held by this user are released for this file.

File Status Functions

The following functions give additional information about a specific currently open file.

⎕*FRKEY* Function (Return Key Values)

Syntax:

```
     R←⎕FRKEY YS
```

Parameters:

YS is a vector of length 2 or 3 (stream, keytype, altkey).

R is the key.

Description:

The ⎕*FRKEY* function returns specific key values depending upon the second item in *YS*. If *YS*[2] is 1, ⎕*FRKEY* returns the key of the first record in this file. If *YS*[2] is 2, ⎕*FRKEY* returns the key of the record most recently read or written. If *YS*[2], is 3, ⎕*FRKEY* returns the key of the last record in the file. If *YS*[2], is 0, ⎕*FRKEY* returns the key of the record last accessed when reading or writing on the specified key index. (For non-IREL files, the difference between 0 and 2 is that file position is maintained independently for every key index and 2 is the key contained in the most recently read or written record. 0 indicates the position at which a sequential read along a particular key index would commence; for IREL files they are equivalent).

The right argument is a simple integer vector of two or three items. The first item is the stream number and the second item is the integer 0, 1, 2 or 3. The third (optional) item is the key index. If the third item is not present, 1 (primary key) is assumed. The key index is not ⎕*IO* dependent.

If the file is currently empty, an empty vector is returned.

For keyed files, a key length of three characters is treated as a numeric key, for all other key lengths the keys are returned as a character vector.

Examples:

```
     FLIM←(⎕FRKEY 1 1),⎕FRKEY 1 3
```

If this is a keyed file with numeric keys or any other file type other than INDEXED or IREL, then this expression results in FLIM being assigned an integer vector of length 2. The first item of FLIM is the record number of the first record in the file, the second item is the record number of the last record in the file. For example:

```
     ⎕FRKEY  1  1  ⋄  ⎕FRKEY  1  2  ⋄  ⎕FRKEY  1  3
20
60
98
     ⍴⎕←⎕FRKEY  4  2  2  ⍝  CURRENT KEY ON SECOND KEY INDEX
SAN FRANCISCO
20
```

⎕*FSIZE* Function (File Size)

Syntax:

```
R←⎕FSIZE Y
```

Parameters:

Y is a simple integer scalar indicating a file I/O stream that is open.

Description:

The ⎕*FSIZE* function returns the number of bytes of storage allocated to the file opened to stream *Y*.

⎕*FKEYINT* Function (Set Key Interval)

Syntax:

```
R←⎕FKEYINT YS
```

Parameters:

YS is a simple 1 or 2 element integer vector.

Description:

The ⎕*FKEYINT* function sets the increment to be used when appending a record to a keyed file. The right argument is a stream number and an optional integer value.

The result is the previous increment or the current increment depending on whether the current increment has been replaced. The default value (after ⎕*FOPEN*) is 1000 for keyed files.

⎕*FKEYS* Function (Return File Keys)

Syntax:

```
R←⎕FKEYS Y
```

Parameters:

Y is a simple integer scalar indicating a file I/O stream that is open.

R is a key definition matrix.

Description:

The ⎕*FKEYS* system function returns the key list for the specified stream.

For INDEXED files, this is a matrix of shape N-by-3 containing the starting position, length, and duplicate indicator. The first column contains the index in character positions of the first character of the key (the first index position is ⎕*IO*). The second column contains the length in characters of the key. The third column contains 1 if the key must be unique or 0 if the duplicate key values are permitted. Each row defines a key, the first row is the primary key (this key must always be unique). Any subsequent rows are alternate keys (may be unique).

For IREL files, this is a matrix of shape N-by-2 containing field numbers in the first column and key unique/key-end flags in the second column. The second column values and their associated meanings are as follows: 0, a non-unique key; 1, a unique key; 2, the last field of a non-unique key; and 3, the last field of a unique key.

For other file types, the result is:

```
      1 3⍴ 0 0 1
```

Example:

An example of ⎕*FKEYS* with an indexed file is:

```
        ⎕FKEYS 4
    10  4  1
    14 20  0
     1  5  0
```

In this example, the file open to stream 4 has 3 keys. The primary key is 4 characters long beginning at position 10. The secondary keys are 20 and 5 characters long and begin at positions 14 and 1 respectively. Only key values for the primary key must be unique.

An example of ⎕*FKEYS* with an IREL file is

```
      ⎕FKEYS 9
3 3
5 0
4 2
8 1
9 3
```

In this example, the file open to stream 9 has 3 keys. The primary key is field 3. The first alternate key is field 5 followed by field 4. The first alternate keys do not have to be unique. The last alternate key is field 8 followed by field 9. The last key values must be unique.

⎕*FCRPT* Function (Set File Encryption Seed)

Syntax:

```
      X ⎕FCRPT Y
```

Parameters:

Y is a simple integer scalar indicating a file I/O stream that is open.

X is a simple integer scalar encryption seed.

Description:

The ⎕*FCRPT* function changes the seed that is used by default for subsequent read and write operations to this stream. The initial default encryption seed is zero when a stream is opened. The right argument is the stream number. The left argument is an encryption seed.

A seed can be specified on the ⎕*FREAD*, ⎕*FWRITE*, ⎕*FAPPEND* and ⎕*FRDCI* functions which overrides this default seed. There is no explicit result to this function.

Library or Account Information

The functions in this group return information about the files in an account.

⎕*FMA* Function (Return File Management Account)

Syntax:

```
R←⎕FMA
```

Description:

The ⎕*FMA* system function returns a character vector of length 8 containing the current default file management account which is used whenever a file identification does not contain an account identifier. This would typically be the same as the current logon account but can be changed by a)!*DIR* command (for example).

⎕*FLIB* Function (Return File Names)

Syntax:

```
R←⎕FLIB A

R←X ⎕FLIB A
```

Parameters:

A is a simple character vector containing an account or a wild-carded file name and account.

R is a simple character matrix of shape N-by-40.

X is a simple character vector of length 2 or a simple character matrix of shape N-by-2.

Description:

The ⎕*FLIB* function returns a character matrix of shape N-by-40 containing names of the files in the specified account. If a wild-carded file name is specified, the result includes only those names which contain all of the characters in the wild-card with any characters (zero or more) in place of the '?' character.

The right argument is either a CP-6 account identifier or a wild-carded file identifier.

An account name is 0 to 8 characters in length. A wild-carded file identifier is 1 to 31 characters in length containing a single '?' character and any other characters, followed by a period, and an (optional) account name.

Monadically, the result is a character matrix of shape N-by-40 where each row contains the account in the first 8 characters (or blanks for the default file management account), a blank, and the file name left justified in the remaining 31 characters.

Dyadically, only those files which have a file type listed in the left argument are returned. If the left argument is empty (a 0-by-2 matrix), all file types are selected. Note that dyadically, ⎕*FLIB* returns a CP-6 FID which can be used directly by the ⎕*FOPEN* system function.

Examples:

```
         ⎕FLIB  'TESTAPL'
TESTAPL  :APLPCF
TESTAPL  :MAIL_CENTRAL
TESTAPL  IDSWS
TESTAPL  GRAFX
TESTAPL  TESTWS

         ⎕FLIB  ':?.TESTAPL'
TESTAPL  :APLPCF
TESTAPL  :MAIL_CENTRAL

         ⎕FLIB  '?WS.TESTAPL'
TESTAPL   IDSWS
TESTAPL   TESTWS


      ⎕FLIB    '*?.'
          *A
          *N
          *S

     'WA' ⎕FLIB  'TESTAPL'  ⍝  WORKSPACES.
IDSWS.TESTAPL
TESTWS.TESTAPL

     ''⎕FLIB  'TESTAPL'     ⍝  ALL FILES.
:APLPCF.TESTAPL
:MAIL_CENTRAL.TESTAPL
IDSWS.TESTAPL
GRAFX.TESTAPL
TESTWS.TESTAPL
```

Possible Errors:

A *RANK ERR* is reported if:

o *A* is not a scalar or vector.

A *DOMAIN ERR* is reported if:

o *A* is not simple or contains an item that is not a character.

o *A* is not a wild-carded file name containing a legal CP-6 account.

- The account name is longer than 8 characters.

Record Field Descriptions

When a file is created, a record field matrix may be specified. This matrix defines the contents of the records in a file in terms of fields. Each field (or row of the record field matrix) defines the datatype, location within the record and the field size.

A record field matrix is a simple integer matrix of shape N-by-4 or N-by-5. The record field matrix column numbers (in origin 1) and their definitions are:

1. Datatype. This is an integer indicating the type of data which is in this field. Table 12-4 contains the numbers of the valid datatypes.

2. Length. For decimal fields, this is the number of digits (minus possible overhead); for floating point and character fields, this is the number of bytes; for integer fields, this is the number of bits. Table 12-4 indicates the rules for the field lengths.

3. Scale. A scale value may be specified for some decimal datatypes. The scale value must be in the range −32 to 31 and it indicates the number of digits after the decimal point. Table 12-4 indicates whether the scale is permitted for each datatype.

4. Vector. Numeric fields may be vectors of numbers. A vector value of 0 indicates that this field is a scalar. A negative vector value indicates a fixed length vector whose length is the absolute value. A vector value greater than zero indicates the field number whose value indicates the length of the vector.

5. Logical Order. This optional value may be used to reorder the physical record contents into a logical order. If all logical order values are 0, then the physical order of fields directly corresponds to the logical order. Otherwise, the logical record corresponds to those fields whose logical order is greater than 0 sorted by increasing logical order number. Logical order values are only used to communicate with other CP-6 processors.

   The physical order of fields in a record is always the order in which the fields are defined. That is, the first physical field is the first row of the record field matrix.

When accessing records in a file, the use of the file's record field matrix may be requested by specifying a record type 4. In this case, APL will automatically perform datatype conversions between the internal APL datatypes and those in the record field matrix datatypes.

When reading records using a record field matrix, the result is a vector with as many items as the record contained (usually the number of fields defined). A *DOMAIN ERR* is reported if the contents of a field are not legal for the datatype indicated by the record field matrix. This occurs if a decimal field (for example) contains a ':' character in a digit position.

When writing records using the record field matrix, the value to write must be a vector that is not longer than the number of fields (rows) in the record field matrix. A *DOMAIN ERR* is reported if:

- a numeric item to be written has a character datatype in the corresponding row of the record field matrix.

- a character item to be written has a numeric datatype in the corresponding row of the record field matrix.

- the value overflows (or underflows) when converted to the type in the corresponding row of the record field matrix.

When creating a file with a record field matrix, the matrix is specified as an item of the left argument of the ⎕*FOPEN* system function. During the execution of the ⎕*FOPEN* function, the system verifies the contents of the record field matrix and reports any inconsistencies as a *DOMAIN ERR* or a *FILE I/O ERR*.

Table 12-4. Record Field Datatypes and Rules

Datatype	Type	Decimal	Size	Scale	Digits	Overhead
Undefined	0		3		0	0
Integer	1		1		1	1
Single Floating	3		3		1	2
Double Floating	4		3		1	2
Packed decimal leading sign	9	YES	2	YES	2	1
Floating packed decimal	10	YES	2		2	3
Fixed length character	21		3		0	0
Varying length character	22		3		0	0
Unsigned integer	24		1		1	0
Packed decimal trailing sign	25	YES	2	YES	2	1
Packed decimal EBCDIC sign	31	YES	2	YES	2	1
Packed decimal unsigned	40	YES	2	YES	2	0
Decimal unsigned	41	YES	3	YES	3	0
Decimal leading sign	42	YES	3	YES	3	1
Decimal trailing sign	43	YES	3	YES	3	1
Leading overpunch sign	44	YES	3	YES	3	0
Trailing overpunch sign	45	YES	3	YES	3	0
Floating decimal	50	YES	3		3	3
Packed decimal leading EBCDIC	51	YES	2	YES	2	1
Date	54	YES	2		4	1
UTS	55		1		5	0
TEXTH	56		3		0	0
Time	57	YES	2		4	1

where

Size rules:

1. The field can be of any length up to 36 bits starting at any bit.

2. The field length is in nibbles and starts on a nibble or character boundary.

3. The field length is in characters and starts on a character boundary.

Digits Rules:

The number of decimal digits in the field is:

```
0.  none
1.  10⍟2*SIZE-OVERHEAD
2.  (⌊(SIZE+⌊SIZE÷9)÷5)-OVERHEAD
3.  ⌊(SIZE÷9)-OVERHEAD
4.  Length must be 16 (nibbles).
5.  Length must be 36 (bits).
```

Note: Decimal fields cannot contain more than 63 digits.

Overhead Rules:

```
0.  0 (none)
1.  1 (bits for integer, digits for decimal)
2.  9 (bits)
3.  2 3[⎕IO+TYPE=10] (digits)
```

⎕FFLDS Function (Return Record Fields)

Syntax:

```
      R←⎕FFLDS Y
```

Parameters:

Y is a simple integer scalar indicating a file I/O stream that is open.

Description:

The ⎕FFLDS system function returns the record field description for the specified file. The result is an integer matrix of shape (*N*,5). If the file tied to the specified stream has no record field list defined, the result is a matrix of shape (0 5).

Example:

```
      ⎕FFLDS 29
 1 36 0 0 0
21 12 0 0 0
10 12 0 0 0
44  6 2 0 0
50 32 0 0 0
22  0 0 0 0
```

In this example, file stream 29 is open to a file whose records contain 6 fields. The field numbers and their corresponding types are:

1. Integer. Values in the range (¯2*35) to (¯1+2*35).

2. Text vector. 12 characters in length.

3. Floating packed decimal. 9 decimal digits in the range ¯1*E*136 to 1*E*136 exclusive. The smallest non-zero numbers in magnitude are ¯1*E*¯128 and 1*E*¯128.

4. Leading overpunched signed decimal. 6 decimal digits (2 after the decimal point). The largest values are 9999.99 and ¯9999.99.

5. Floating decimal. 30 decimal digits in the range ¯1*E*157 to 1*E*157 (exclusive). Note that APL will provide an approximation to these values accurate to the 18 most significant digits.

6. Variable length text vector. The maximum length of this field is 511 characters.

The following examples demonstrate writing records to this file using APL's automatic data conversion capability (record type 4).

```
      X←123 'CHAR(12)' (*1) 12.345 (○1) 'VARYING'

      X ⎕FWRITE 29 1 0 4
      X
123  CHAR(12)  2.718281828 12.345 3.141592654  VARYING
```

In the above example, *X* is written to the file, APL will pad field 2 with blanks to fill it out to 12 characters in length. The value written for fields 3 and 4 will be rounded to the number of digits defined for the field.

```
      ⎕FREAD 29 1 0 4
123  CHAR(12)      2.71828 12.35 3.141592654  VARYING
```

Notice that field 2 has been padded with blanks and that field 4 has been rounded to 2 digits after the decimal point.

Alternate Indexed Files

The following file I/O functions permit the specification of an alternate index: ⎕*FREAD*, ⎕*FRDCI* ⎕*FDROP*, ⎕*FRKEY*. The alternate key list can be obtained via the ⎕*FKEYS* function.

An alternate key is specified as a nested 2-item argument in the position in which the key is found. The first item is the key index number. The second item is the key value. An empty vector for the key value causes a sequential read along the specified alternate key to occur. For example:

```
    ⎕FREAD  9 (3 ('SMITH')) 0 3
```

In this example, stream 9 is read. The third key is searched for the value *'SMITH'* (trailing blanks are supplied by APL if the key is shorter than the key length). If multiple records with that key exist, the first record with that key is returned. Subsequent reads on this stream that do not specify a key index or specify index 0 will use the third key index.

In the following example, all records with the value *'MARTIN'* for the third key index are removed from the file:

```
    ⎕FDROP  9  (3 'MARTIN')
```

If only one particular *'MARTIN'* record is to be deleted, the key value for a unique key (there is always one) must be provided.

For IREL files, the key index must always be provided and the key value must be a vector of field values. For example:

```
      ⎕FREAD 4 (2 (,5)) 0 4
```

In this example, the second key index is used. The key value that will be found is 5.

Specialized File Options

The left argument of the ⎕*FOPEN* and ⎕*FCLOSE* system functions can be:

- a simple character vector containing the CP-6 file identifier and options,
- a vector of arrays, where each item in the vector can be: a character vector with the CP-6 fid, an access control matrix as described in ⎕*FRDAC*, a record field matrix, or an indexed key list as described in ⎕*FKEYS*. The access control matrix and indexed key list are only used for CREATE opens. If OLDFILE is specified and the file exists, the access controls and index keys are not used. The character vector containing the file identifier must appear before the alternate keys in the list of items.

Examples:

To create a file which permits any CP-6 user to read with a 5 character primary key starting at index 1, a 4 character alternate key starting at index position 6, a 20 character alternate key starting at index position 10, and another 20 character alternate key starting at index position 30, the following is entered:

```
        ALTKEYS←4 3ρ 1 5 1  6 4 0  10 200  30 20 0
        AC←1  17ρ'(9↑'?'),'Y',7ρ'N'
        FID←'ALTFILE,CREATE,NEWFILE,INDEXED,LOAD,CTG'
        (FID ALTKEYS AC)⎕FOPEN  1
        ⎕FKEYS 1
 1  5 1
 6  4 0
10 20 0
30 20 0
        ⎕FRDAC 1
?       YNNNNNNN
```

All of the options available through the IBEX !SET command are also available to users wishing to open a file. For information on the SET command, see the CP-6 Programmers Reference Manual (CE40).

To create an INDEXED file in APL with the name INDEX01 and with the key starting in the 73rd character and being 8 characters in length, the following APL expressions can be used:

```
      ⍎')SET F$31 INDEX01,KEYX=72,KEYL=8'

      ',REASSIGN,INDEXED,CREATE,NEWFILE' ⎕FOPEN 31
```

The options that APL permits on the *⎕FOPEN* all have defaults that will override the *SET* options. That is, the *SET* options EXIST=, SHARE=, FUN=, ORG= and ACS= will be ignored on the *)SET* command and so they must be specified on the *⎕FOPEN* if different from APL's default options.

Possible Errors:

A *RANK ERR* is reported if:

o the left argument of *⎕FOPEN* or *⎕FCLOSE* is not a scalar or vector.

A *LENGTH ERR* is reported if:

o the left argument is not simple and it contains zero or more than four items.

A *RANK ERR* is reported if:

o any item of the left argument is not a scalar, vector or matrix.

A *DOMAIN ERR* is reported if:

o any item of the left argument is not simple.

A *LENGTH ERR* is reported if:

o a matrix item of the left argument is not of shape N-by-17 (access control matrix), or an N-by-3 (alternate key matrix) matrix.

Section 13

CP-6 APL I-D-S/II System Functions

CP-6 APL contains system functions which provide access to I-D-S/II databases. All of the COBOL language DML (Data Manipulation Language) statements have equivalent APL functions. In addition to these standard DML functions, the APL interface contains a number of unique functions which may be used to obtain detailed information about the database being accessed.

In order to make use of this facility an APL subschema must be generated. The creation of I-D-S/II databases and subschema generation is achieved through the execution of various I-D-S/II utility programs.

```
R←⎕DBSUB  'sub-schema-name,privacy-lock,SHARE'
```

This function must always be the first I-D-S/II function to be executed during an APL session. It informs I-D-S/II of the name of the subschema to be used and the result of its execution is the name of the associated schema. If the subschema, schema or areas reside in an account other than the current file management account, or if their file management name is different than their schema name or the CP-6 file containing the subschema is passworded, a)SET command is required to direct I-D-S/II to the correct file. For example, if the subschema name is *SSCHFILE* and it resides in the file: *SSCHFILE_7A*.123*TEST*, the schema file name is *SCHFILE* and it resides in the file *SCHFILE*.123*APL*.*PSSWD* and the area to be used in this subschema is *AREA*0 in the CP-6 file *MAGAREA*.123*IDS*. The *)SET* commands required to use this database in APL are:

```
)SET SSCHFILE SSCHFILE_7A.123TEST
)SET SCHFILE SCHFILE.123APL.PSSWD
)SET AREA0 MAGAREA.123IDS
```

The optional positional parameter *SHARE* when present causes all of the items in the database to become automatically shared between the user work area and the APL workspace. This allows the functions ⎕*DBFROM* and ⎕*DBTO* to be performed automatically upon assignment or a reference to a workspace variable with the same name as the corresponding database item. Note however, item names containing underscores will appear in the workspace with deltas replacing the underscores.

The following errors are possible when executing the ⎕*DBSUB* system function:

I-D-S/II LIBRARY NOT AVAILABLE

The alternate shared library named I-D-S/II either does not exist or another alternate library is currently associated. This error may be sidetracked as error number 201.

INVALID SUBSCHEMA

The file indicated as the subschema is not an I-D-S/II subschema file. This error may be sidetracked as error number 202.

FILE TBL FULL

The maximum number of files that an APL user may open are currently open. In order to use I-D-S/II, some files must be closed. This may be sidetracked as error number 203.

SUBSCHEMA NAME ERR

Either the subschema name and privacy key is ill-formed or the subschema name in the file does not match the name supplied. This may be sidetracked as error number 205.

SUBSCHEMA ACCESS ERR

Access to the subschema has been denied because the file does not exist, it is passworded, or the privacy locks do not match. This may be sidetracked as error number 206.

Subschema Information Functions

The following functions give information about subschema names and subschema name types.

⎕DBNAMES Function (List Subschema Names)

Syntax:

R←⎕DBNAMES

Description:

The *⎕DBNAMES* niladic system function returns the names of all of the realms, sets, records, items, and I-D-S/II keywords available through the current subschema. The shape of the result is N by M where N is the number of names and M is the length of the longest name.

⎕DBTYPES Function (Subschema Name Types)

Syntax:

R←⎕DBTYPES

Description:

The *⎕DBTYPES* niladic system function returns a numeric array of shape N-by-6 indicating the attributes of each corresponding name in *⎕DBNAMES*. The first item of each row indicates the type of object represented by that name. The second item of each row contains usage or mode information, the third item of each row contains encoded flags that are type specific, the fourth item of each row contains the index in *⎕DBNAMES* of the object that owns this object, the fifth item of each row contains sub-type information and the sixth item of each row contains the length or the object's order.

Column one contains:

OBJECT TYPE

1	REALM
2	RECORD
3	FIELD
4	SET
5	PARAMETER
6	unused
7	I-D-S/II KEYWORD
8	Record Key

Record Type Information

For records, column two contains the location mode for this record:

0	DIRECT
1	CALC
2	VIA SET
3	SEQUENTIAL
4	INDEXED

Column three contains flags decoded as:

```
F←(35⍴2)⊤⎕DBTYPES[R;3]
```

where (in index origin 1):

R[1]	name is in SUBSCHEMA
R[2]	name understood by I-D-S/II
R[3]	STORE is allowed
R[4]	MODIFY is allowed
R[5]	DELETE is allowed
R[6]	ERASE is allowed
R[7]	variable length
R[8]	in multiple areas
R[9]	has alternate keys
R[10]	an elementary item
R[11]	item is signed
R[12]	item is scaled

Field Type Information

The second column for fields contains usage information. The range of possible values are:

0	DATA ITEM
1	DATA BASE PARAMETER
2	LOCATION MODE DIRECT FIELD
3	AREA-ID FIELD
4	CALC KEY SYNONYM FIELD
5	SCAN KEY SYNONYM FIELD

The third column for fields contains flags that may be decoded with the expression:

```
F←(35⍴2)⊤⎕DBTYPES[F;3]
```

where the meanings are the same as for the record flags.

The fourth column contains the record number in ⎕DBNAMES that this field belongs to. The fifth column contains the data type of this field. Values are:

0	CHARACTER
1	INTEGER DB KEY
2	INTEGER
13	SINGLE PRECISION FLOATING POINT
14	DOUBLE PRECISION FLOATING POINT

The sixth column is used for character data items to indicate the length of the string.

Set Type Information

The second column for sets indicates the mode of this set. Valid values of mode are:

0	CHAIN
1	RECORD ARRAY
2	POINTER ARRAY

The fourth column for sets indicates the index in ⎕*DBNAMES* of the record that owns this set. The fifth column indicates the type of set this is. Values of set type are:

0	USER SET
1	CALC SET
2	PRIMARY KEY
3	SECONDARY KEY

The sixth column indicates how this set is ordered. Values of order are:

0	FIRST
1	LAST
2	NEXT
3	PRIOR
4	SORTED BY KEY

Parameter Type Information

Column four contains the index in ⎕*DBNAMES* of the record related to this database parameter.

I-D-S/II Function Arguments

In the following description of the APL language interface to I-D-S/II, the functions are described in terms of the database objects known to I-D-S/II. The database objects are indicated in APL by either the text of the object's name or by the numeric index of the object's name in ⎕*DBNAMES*.

A series of database objects and keyword names may be indicated by separating the names by blanks in a character vector, as an integer vector of object indices (in ⎕*DBNAMES*), or as a series of object values (including character and/or numeric indices), separated by semi-colons and enclosed in parentheses as in arguments to ⎕*FMT*. This means that all of the following I-D-S/II calls are equivalent:

```
R←⎕DBFIND 'STUREC WITHIN COURSET CURRENT STUNAME'
R←⎕DBFIND ('STUREC';'WITHIN';'COURSET';'CURRENT';'STUNAME')
R←⎕DBFIND ('STUREC WITHIN COURSET CURRENT';'STUNAME')
R←⎕DBFIND (5 42 3 34;'STUNAME')
```

where in the last example, the fifth row of ⎕*DBNAMES* contains the text STUREC, the 42nd row contains WITHIN, etc.

Name and Set Information

The following functions give information about subschema names and sets.

⎕DBANLZ Function (Analyze Subschema Names)

Syntax:

R←⎕DBANLZ A

Description:

The right argument to this function is a simple character vector containing subschema names and keywords. The result is an integer vector containing the index in *⎕DBNAMES* of each name in *A*.

⎕DBOWNER Function (Set Owner)

Syntax:

R←⎕DBOWNER B

Parameters:

B is a character vector containing the name of a set or record. Optionally, it may be the index in *⎕DBNAMES* of the name of a set or record.

Description:

The right argument to this function is either the name of a set or record. The result for a set is the index of the record that owns this set in *⎕DBNAMES*. The result for a record is a vector of the indices of the sets that this record may own.

⎕DBMEMBER Function (Set Member)

Syntax:

R←⎕DBMEMBER B

Parameters:

B is a character vector containing the name of a set or record. Optionally, it may be the index in *⎕DBNAMES* of the name of a set or record.

Description:

The right argument to this function is either the name of a set or record. The result is always an N-by-3 matrix indicating in column one the sets or records that the supplied record or set may be a member of. The second column contains zero if membership is manual or one if membership is automatic. The third column is zero if membership is optional or one if membership is mandatory.

⎕*DBINFORM* Function (Database Register)

Syntax:

```
      R←⎕DBINFORM B
```

Parameters:

B is a simple character vector.

Description:

The ⎕*DBINFORM* function returns the contents of the specified database register. The registername is specified as one of the following:

```
                    DBSTATUS
                    DBREALM
                    DBSET
                    DBRECORD
                    DBPRIVACY
                    DBDATANAME
                    DBKEYNAME
                    DIRECTREFERENCE
```

Accessing Data

The values of record items may be set and obtained by either sharing variables in the workspace with I-D-S/II or by the functions ⎕*DBFORM* and ⎕*DBTO*. Sharing of all database items may be requested when the database is opened (see ⎕*DBSUB*) or by using the Shared Variable system functions. For example, if USERNAME is the name of an item in the database then the expression:

```
      'IDS.:SYS' ⎕SVO 'USERNAME'
2
```

returns 2 indicating that the variable USERNAME in the active workspace may be referenced or assigned is a surrogate for the value in the UWA. Note that erasing USERNAME or retracting USERNAME by the ⎕*SVR* system function will discontinue sharing.

⎕DBFROM Function (Retrieving Data)

Syntax:

R←⎕DBFROM B

Description:

The *⎕DBFROM* function returns the values of the specified items in the UWA (User Work Area). The items in the itemlist must all be of type numeric or all character or a *DOMAIN ERR* will occur. Character values in the result are separated by carriage return characters.

⎕DBTO Function (Storing Data)

Syntax:

R←⎕DBTO (itemlist;valuelist[;valuelist;...])

Parameters:

itemlist is a simple character vector containing the names of I-D-S/II items. Optionally, this may be a simple integer vector containing the indices of items in *⎕DBNAMES*.

valuelist are simple scalar numbers or vectors of numbers to be assigned to the corresponding item named in itemlist. Character values are included as separate items in the list.

Description:

The right argument to this monadic system function is a list, the first item of which is either the character vector of the itemnames or a numeric vector of the name indices in *⎕DBNAMES*. The remaining items in the argument list consist of the values to be moved into the UWA for each corresponding item name. A valuelist consists of character or numeric scalars or vectors. If a character vector is supplied, separate values may be separated by carriage return characters.

Standard I-D-S/II Functions

Brief descriptions of syntax for APL calls on the I-D-S/II data manipulation functions are listed below. For information on their meaning and use, see the I-D-S/II Programmers Reference Manual (CE35).

```
                         setname
R←⎕DBACCEPT  ([FROM] [recordname]  CURRENCY)
                         areaname

                         setname
R←⎕DBACCEPT  ([FROM] [recordname]  REALMNAME)
                         dbkey
```

If the first parameter is a setname or recordname, this function requires that the text of the name be supplied and not the name's index in *⎕DBNAMES*.

```
                                          NEXT
                                          PRIOR
R←⎕DBACCEPT   ([FROM] setname   CURRENCY)
                                          OWNER

R←⎕DBBUFFERS          integer

R←⎕DBCHECK            integer

R←⎕DBCRPT             integer

R←⎕DBCONNECT     ([recordname[TO]]setname )

R←⎕DBDISCONNECT ([recordname[FROM]]setname )

R←⎕DBERASE       ([recordname]  [ALL][MEMBERS] )

                                        dbkey
R←⎕DBFIND        ([recordname]DBKEY[IS]  parametername )


R←⎕DBFIND        (ANY          recordname )

R←⎕DBFIND        (DUPLICATE    recordname )

R←⎕DBFIND        (DUPLICATE[WITHIN]setname[USING]itemlist )

                    [FIRST]
                    [NEXT ]
R←⎕DBFIND        (  [PRIOR] [recordname][WITHIN]setname    )
                    [LAST ]                       realmname
                    [count]
```

If the first parameter is a keyword, this function requires that the text of the name be supplied.

```
                                               [setname   ]
R←⎕DBFIND     (CURRENT [WITHIN] recordname [realmname] )
                                               [keyname   ]

R←⎕DBFIND     (OWNER [WITHIN] setname)

R←⎕DBFIND     (recordname WITHIN setname [CURRENT] [USING] itemlist )

               FIRST
R←⎕DBFIND     (NEXT [recordname] USING keyname)
               recordname

R←⎕DBFINISH   realmnamelist

R←⎕DBGET      recordname

R←⎕DBGET      identifierlist
                             [ONLY        ALL                       ]
R←⎕DBMODIFY   ( [recordname] [INCLUDING   setnamelist [MEMBERSHIP]])

                             [              ALL                      ]
R←⎕DBMODIFY   ( itemlist     [INCLUDING   setnamelist [MEMBERSHIP]])

                    [CONNECT   ]
R←⎕DBPRIVACY ( SET [DISCONNECT] privacykey [setnamelist])
                    [FIND      ]

                          [GET       ]
                          [MODIFY    ]
                          [STORE     ]
R←⎕DBPRIVACY  (RECORD     [FIND      ] privacykey [recordnamelist])
                          [ERASE     ]
                          [CONNECT   ]
                          [DISCONNECT]
                          [GET    ]
```

```
R←⎕DBPRIVACY  (ITEM    [MODIFY ] privacykey [itemlist])

                                             [UPDATE    ] [SHARE    ]
R←⎕DBREADY     ( [realmlist] [USAGE MODE[IS][RETRIEVAL ] [SHAREIN  ] )
                                             [LOAD      ] [NOSHARE  ]
                                                          [SHAREANY ]

                 [REALM      ]
                 [RECORD     ]
R←⎕DBRETAIN    ( [SETS       ] )
                 [setnamelist]

R←⎕DBROLL

R←⎕DBRPTSTATS

R←⎕DBSTATSOFF

R←⎕DBSTATSON

R←⎕DBSTORE     (recordname)

R←⎕DBTRACEOFF    option

R←⎕DBTRACEON     option

                          [EMPTY  ]
R←⎕DBIF        ( setname  [MEMBER ]  )
                          [TENANT ]
```

I-D-S/II Error Reporting and Handling

The execution of I-D-S/II data manipulation functions always cause a code to be returned in the DBSTATUS cell. Values of DBSTATUS greater than zero signify that an exception condition has occurred. Successful completion of the data management function is indicated by a DBSTATUS of zero.

Control over exception conditions may be obtained by APL use procedures.

⎕*DBUSE* Function (Use Procedures)

Syntax:

```
code ⎕DBUSE name
⎕DBUSE name
```

Description:

The ⎕*DBUSE* function returns the name of the use procedure associated with the specified code.

APL use procedures are specified and interrogated by the ⎕*DBUSE* system function. To specify a use procedure, the left argument to ⎕*DBUSE* contains the DBSTATUS value. The right argument contains the name of a niladic function or character vector to be executed when the DBSTATUS value returned by I-D-S/II is equal to the left argument. If an empty vector is supplied for the right argument, the use procedure for the DBSTATUS value in the left argument is deleted. More generally, the left argument to ⎕*DBUSE* may be a vector of status codes, in which case, the specified niladic function becomes the use procedure for all of the status codes specified. If the right argument is empty, use procedures for all of the specified codes are removed. Finally, the left argument may be a namelist, in which case, there must be exactly as many status codes specified as there are names in the use procedure list.

Monadically, ⎕*DBUSE* is used to determine the status codes for which there exists an active use procedure or the names of procedures attached to particular status codes.

Examples:

```
    ⎕DBUSE ⍳0
```

This function returns all of the status codes for which there exists a use procedure.

```
    ⎕DBUSE ⎕DBUSE ⍳0
```

This function returns the names of all of the use procedures.

A status code of zero may be specified for a use procedure in which case that procedure will gain control whenever an exception condition occurs for which there exists no explicitly named use procedure for that code. The use procedure table is of limited size, there will however always be room for 75 use procedures. Whenever the current workspace is cleared or a new workspace is loaded, the use procedure table is lost and must be set up once again.

Section 14

Packages

A package is used to hold the definition of one or more functions, or the value of one or more variables. A package contains a set of names which are distinct from each other. However, the names within a package need not be distinct from names used outside the package or from the name of the package itself. A name must be extracted from the package before it can be referenced in the active workspace. Packages can be written to or read from files, passed as the argument of a defined function, or returned as the result of a defined function. Packages within saved workspaces can be copied.

The package name is displayed by the system command *)VARS* or as the result of ⎕*NL*. The function ⎕*NC* reports the nameclass of the package as "variable", the function ⎕*RM* reports the number of bytes required to store the package and the function ⎕*EX* may be used to expunge a package. The *)ERASE* command can also be used to expunge a package if the package is defined globally.

The following restrictions apply to the use of packages:

1. A package is outside the domain of arithmetic, relational, and structural functions.
2. A package is outside the domain of ⍳, ⊂, and [].
3. A package cannot be:
   - an item of another array.
   - catenated.
   - inserted into an array via indexed assignment.
4. The contents of a package cannot be directly displayed.

The function ⎕*PNAMES* can be used to distinguish a variable that is a package from a variable that is not a package. For example, the expression ⎕*PNAMES X* returns a matrix when *X* is a package and returns an empty vector when *X* is not a package.

Package System Functions

The following system functions are used to create and manipulate packages. When two syntax formats are listed for a function, "Monadic Syntax:" refers to the monadic function and "Dyadic Syntax:" refers to the dyadic function.

⎕*PACK* Function (Package Create)

Syntax:

```
    R←⎕PACK X

    R←Y ⎕PACK X
```

Parameters:

X is a namelist.

R is a package.

Y is a namelist.

Description:

Using monadic ⎕*PACK*, the argument is a namelist which contains the names to be included in the resulting package. The result is a package containing each of the distinct names included in the argument and the object, if any, to which that name refers. If the name in *X* is the name of a shared variable, the value included in the package is the last visible value in the active workspace. (Referring to a shared variable with monadic ⎕*PACK* does not count as a "use" of the shared variable.)

Dyadically, the ⎕*PACK* function packages the value *X* with the name in *Y*. The right argument is any data object (an array or package, but not a function). The left argument is a character vector, scalar or 1-row matrix which contains a single name. The result is a package which contains the name in the left argument and the data in the right argument.

Note:

1. If the same name is included more than once in the right argument *X*, the extra occurrences have no effect.

2. A misspelled name in the argument does not cause an error message and the intended object will be missing from the package.

⎕*PINS* Function (Package Insert)

Syntax:

```
    R←Y ⎕PINS X
```

Parameters:

X is a package.

Y is a package.

R is a package.

Description:

The ⎕*PINS* function inserts the package *X* into the package *Y*. Both *X* and *Y* must be packages. The result is a package which contains all the names along with their referents from package *X*, and, in addition those names that occur in *Y* but do not occur in *X*, and the objects which are referred to in package *Y*.

The resulting package contains all of the names from both of the packages. However, when a name in *X* is the same as a name in *Y*, it's referent from *X* is included as the referent in the result.

Note: The names in a package are reported in arbitrary order. The package resulting from ⎕*PINS* need not have names in the identical order as the names which appear in Y.

⎕*PNAMES* Function (Package Names)

Syntax:

R←⎕*PNAMES X*

Parameters:

X is a package.

R is a namelist.

Description:

The ⎕*PNAMES* function returns the names in a package *X*. The argument is any array or package. The result is an empty vector whenever the argument is not a package, otherwise the result is a character matrix containing the names in the package. The matrix has one row for each name, and the longest name determines the number of columns. Names are left-justified, with following blanks.

Note: APL reports the names within a package in arbitrary order.

⎕*PNC* Function (Package Name Correspondence)

Syntax:

R← ⎕*PNC X*

R←*Y* ⎕*PNC X*

Parameters:

X is a package.

Y specifies an optional argument which is a namelist.

R is a simple integer vector.

Description:

Monadically, *⎕PNC* returns the nameclass of each of the names in the package. Dyadically, *⎕PNC* returns the nameclass of each of the specified names in a package.

The right argument is a package. The left argument is an optional namelist. It is not necessary that a name included in the left argument also be present in the package, or that the names are well formed. The result is an integer vector which indicates the nameclass of package *X* names. Whenever *⎕PNC* is used with a left argument, the result contains one item corresponding to each name in the left argument.

Monadically, the result of the *⎕PNC* function contains one item for each name included within the package *X* in the same order as the names reported by *⎕PNAMES X*, leading to the following identity:

```
    ⎕PNC X ↔ (⎕PNAMES X) ⎕PNC X
```

The class of object to which the name refers is indicated by an item of the result. The following table shows the relationship:

CODE	NAME
−1	No referent
0	Not present in the package
2	Refers to a variable
3	Refers to a function
4	Is not well–formed

The code used in the result of *⎕NC* determines the code in the result of *⎕PNC*. The value 1 cannot appear in the result of *⎕PNC* because a package cannot contain a label. The value −1 does not occur in the result of *⎕NC*.

⎕PVAL Function (Package Value)

Syntax:

```
    R←Y ⎕PVAL X
```

Parameters:

X is a package.

Y is a namelist containing one name.

R is an APL value.

Description:

The *⎕PVAL* function returns the value of the variable named in *Y* from the package *X*. The left argument is a character vector, scalar, or matrix. The referent in *X* must be a variable and the left argument must contain one name. APL reports a *DOMAIN ERR* message when *Y* does not meet those requirements. The result is the value of the variable which is named in *Y*.

⎕*PDEF* Function (Package Definition)

Syntax:

```
⎕PDEF X
Y ⎕PDEF X
```

Parameters:

X is a package.

Y is a namelist.

Description:

Monadically, ⎕*PDEF* defines all of the names in the package right argument. Dyadically, ⎕*PDEF* defines those names in the package right argument that appear in the left argument.

Each name in the left argument must:

- Be well formed.
- Be contained in package *X*.
- Not have an active function which is a visible referent in the workspace. (Its referent may not appear on the state indicator.)

Monadically, the ⎕*PDEF* function defines every name in package *X* in the active workspace with the referent from the package. Dyadically, the ⎕*PDEF* function defines each name in the namelist *Y* in the active workspace with the referent from the package *X*. A name existing in the package without a referent is expunged when ⎕*PDEF* is used to bring the name into the workspace.

Possible Errors:

A *DOMAIN ERR* is reported if:

- any of the names in *Y* do not meet requirements stated in description.

⎕*PPDEF* Function (Protected Package Definition)

Syntax:

```
R← ⎕PPDEF X
R←Y ⎕PPDEF X
```

Parameters:

X is a package.

Y is a namelist.

R is a namelist.

Description:

Monadically, ⎕*PPDEF* defines all of the names in the package right argument that currently have no value. Dyadically, ⎕*PPDEF* defines those names that appear in the package right argument, are named in the left argument, and do not currently have a value. The explicit result for both cases is a namelist containing names of objects not defined because they had values.

⎕*PPDEF* assigns a new meaning to a name with no current use and never expunges or changes an existing name's referent. (It can be compared to the system command)*PCOPY*, except that)*PCOPY* refers to the global and not to the visible meaning of each name.)

When ⎕*PPDEF* is used dyadically, the left argument *Y* is a namelist that specifies the names which are to be defined. Each name in *Y* must be included as a name in package *X*.

When ⎕*PPDEF* is used with two arguments, each name in *Y* (or when ⎕*PPDEF* is used with one argument, each name in *X*) which has no visible referent in the workspace, is given the value which it has in the package.

The result is a namelist containing the names in the package which were not defined because they already exist in the active workspace.

⎕*PSEL* Function (Package Select)

Syntax:

```
R←Y ⎕PSEL X
```

Parameters:

X is a package.

Y is a namelist.

R is a package.

Description:

The ⎕*PSEL* function returns a package containing those names from the package right argument that appear in the namelist left argument. Each name specified in the namelist must be contained within the package.

⎕*PEX* Function (Package Expunge)

Syntax:

R←*Y* ⎕*PEX X*

Parameters:

X is a package.

Y is a namelist.

R is a package.

Description:

The ⎕*PEX* function returns a copy of the package right argument excluding the names specified in the namelist left argument.

⎕*PLOCK* Function (Package Lock)

Syntax:

R←*Y* ⎕*PLOCK X*

R← ⎕*PLOCK X*

Parameters:

X is a package.

Y is an optional namelist.

R is a package.

Description:

Monadically, ⎕*PLOCK* returns a package identical to the right argument except that all of the defined functions are locked. Dyadically, ⎕*PLOCK* returns a package identical to the right argument except that all of the defined functions named in the left argument are locked.

Each of the names must be well formed and must be a name which is present in package *X* and is used there as the name of a function. The function in *X* need not be previously unlocked.

The result is a package which contains the same names with the same referents as package *X*. The exceptions are as follows:

- o When ⎕*PLOCK* has two arguments, the functions which are unlocked in *X* and named in *Y* are locked in the resulting package.
- o When ⎕*PLOCK* is used with one argument, all functions are locked in the resulting package.

Section 15

CP-6 APL Graphics

CP-6 APL provides four system functions which produce device independent graphics output. Additional system functions and variables are also provided which may be used to control the graphics device or the appearance of the graphics output. The CP-6 DIGS Reference Manual (CE72) provides a more detailed definition of the general capabilities and functionality available. This section contains a simple overview of the graphics capability available within APL.

The APL graphics capabilities are sub-divided into the five areas:

- Output Functions
- Segment Primitives
- Attribute Variable
- Viewing Variables
- Control Primitives

The graphics output primitives include the ability to draw lines, draw and optionally fill polygons, draw markers and generate text. Graphics output is subject to the controls provided by the other graphics capabilities and the device on which it is displayed.

All graphics output appears in either a temporary or a retained segment. CP-6 APL provides the ability to create segments, delete segments, rename segments, select segments, and to inquire about the attributes of segments. A retained segment defines an image which is a part of the whole picture displayed on a view surface. Attributes of a retained segment may be dynamically modified, thereby changing the image on the view surface.

The capabilities provided by the graphics attribute variables allow control over the appearance of graphics output primitives. The appearance includes such items as the color, intensity, line widths, marker symbols, and the text font. The graphics viewing variables control the location, size and rotation of graphics output. The viewing variables include such items as the graphics window, the viewport, and the image transformation variables.

The graphics control primitives provide the capabilities required to initiate an APL graphics session, select a graphics device, determine the device capabilities, and define the mappings between certain attribute settings and the corresponding device color or intensity.

The following APL program demonstrates the use of APL graphics. It does all that is necessary within APL to produce graphics output on a graphics terminal.

```
    ∇ HOUSE_EXAMPLE
[1]    ⎕GRINIT ''           ⍝  INITIALIZE GRAPHICS
[2]    '' ⎕GRINITSURF 1     ⍝  PREPARE TERMINAL FOR GRAPHICS
[3]    ⎕GRSURFACE 1         ⍝  SELECT TERMINAL FOR GRAPHICS
[4]    ⎕GRTSEGO             ⍝  OUTPUT INTO TEMPORARY SEGMENT
[5]   ⍝ CREATE OUTLINE OF HOUSE
[6]    HOUSE←7 2ρ 1 6 5 9 9 6 9 1 1 1 1 6 9 6÷10
[7]    ⎕GRLINE HOUSE
[8]   ⍝ DRAW DOOR
[9]    DOOR←4 2ρ 7 1 7 3 8 3 8 1÷10
[10]   ⎕GRLINE DOOR
[11]  ⍝ DRAW SMALL WINDOW
[12]   WINDOW1←5 2ρ 2 2 2 3 3 3 3 2 2 2÷10
[13]   ⎕GRLINE WINDOW1
[14]  ⍝ DRAW LARGE WINDOW
[15]   WINDOW2←5 2ρ 4 2 4 4 6 4 6 2 4 2÷10
[16]   ⎕GRLINE WINDOW2
    ∇
```

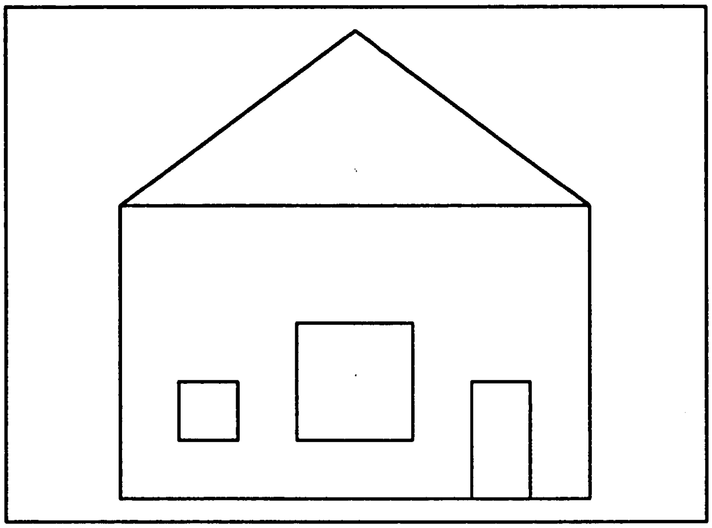


Figure 15-1. Graphics Output Example

This example demonstrates the necessary preparations before graphics output may be created. The ⎕*GRINIT*, ⎕*GRINITSURF* and ⎕*GRSURFACE* functions must all be executed before any graphics output is created. The use of these functions is described under the topic Graphics Control Functions and Variables.

The function ⎕*GRTSEGO* is used to indicate that the following output is temporary (not to be retained across screen clears for example). This function is described under the topic Graphics Segment Functions. A graphics segment (temporary or retained) must also be in use before graphics output may commence.

The function ⎕*GRLINE* is one of the five graphics output functions available in CP-6 APL. This function connects each of the points specified with a line. The type of line drawn may be controlled by variables that are described under the topic Graphics Attribute Variables.

Graphics Output Functions

Five system functions in CP-6 APL are used to produce graphics output. Each of the functions supply a different type of graphics output from a similar graphics argument. The data to be displayed graphically is provided as either an N-by-2 or N-by-3 array. The columns are treated as X, Y, and optional Z components of a graphical position. If the data to be displayed is not a matrix, a *RANK ERR* is reported. If the data to be displayed is not an N-by-2 or N-by-3 array, a *LENGTH ERR* is reported. If the data to be displayed is not a simple numeric array, a *DOMAIN ERR* is reported.

If an APL graphics session has not been initiated by executing the ⎕*GRINIT* system function, or if a device has not been initialized and selected, or there is not a currently open segment, a *DOMAIN ERR* is reported. (Entering a ')?' command indicates which error has occurred).

⎕*GRLINE* Function (Draw Line)

Syntax:

```
    ⎕GRLINE DATA
```

Parameters:

DATA is a simple array of shape N-by-2 or N-by-3 containing only scalar numbers.

Description:

The ⎕*GRLINE* function defines the simplest form of graphical output. A line is drawn connecting each of the positions in the data array provided.

Example:

```
    HOUSE←7 2⍴.6 .1 .9 .5 .6 .9 .1 .9 .1 .1 .6 .1 .6 .9
    ⎕GRLINE HOUSE
```

This example creates a simple stick house on the currently selected view surface.

Possible Errors:

A *RANK ERR* is reported if:

- the right argument is not a matrix

A *LENGTH ERR* is reported if:

- the right argument is not an N-by-2 or N-by-3 matrix

A *DOMAIN ERR* is reported if:

- the right argument is not simple or all numeric
- APL graphics is not initialized
- there is not an open segment
- the current value of ⎕*GRLI* is not supported
- the current value of ⎕*GRLS* is not supported
- the current value of ⎕*GRLW* is not supported
- the current value of ⎕*GRPEN* is not supported

⎕*GRMARK* Function (Draw Marker Symbols)

Syntax:

```
    ⎕GRMARK DATA
```

Parameters:

DATA is a simple array of shape N-by-2 or N-by-3 containing only scalar numbers.

Description:

The ⎕*GRMARK* function produces a marker symbol (as selected by the ⎕*GRMARKER* system variable). at each of the positions indicated by the data array.

Example:

```
      CURVE←7 2ρ0 0 .2 .1 .3 .4 .5 .5 .7 .4 .8 .1 .9
      ⎕GRMARK CURVE
```

Possible Errors:

A *RANK ERR* is reported if:

o the right argument is not a matrix

A *LENGTH ERR* is reported if:

o the right argument is not an N-by-2 or N-by-3 matrix

A *DOMAIN ERR* is reported if:

o the right argument is not simple or not all numeric
o APL graphics is not initialized
o there is not an open segment
o the current value of ⎕*GRLI* is not supported
o the current value of ⎕*GRMARKER* is not supported.
o the current value of ⎕*GRPEN* is not supported

⎕*GRPOLYGON* Function (Draw Polygon)

Syntax:

```
     ⎕GRPOLYGON DATA
```

Parameters:

DATA is a simple array of shape N-by-2 or N-by-3 containing only scalar numbers.

Description:

The ⎕*GRPOLYGON* function produces a polygon whose vertices are the positions provided in the data array. The appearance of the polygon can be controlled through assignment to the attribute variables.

Example:

```
      PARALLELS←4 2ρ.2 .3 .4 .7 .8 .7 .6 .3
      ⎕GRPOLYGON PARALLELS
```

Possible Errors:

A *RANK ERR* is reported if:

o the right argument is not a matrix

A *LENGTH ERR* is reported if:

o the right argument is not an N-by-2 or N-by-3 matrix

A *DOMAIN ERR* is reported if:

o the right argument is not simple or not all numeric
o APL graphics is not initialized
o there is not an open segment
o the current value of ⎕*GRPEN* is not supported
o the vertices are not coplanar
o the current value of ⎕*GRFILL* is not supported
o the current value of ⎕*GRVERTEX* is not supported
o the length of ⎕*GRVERTEX* is not equal to the first dimension of the right argument
o the current value of ⎕*GRLI* is not supported

⎕*GRDRAW* Function (Draw Picture)

Syntax:

```
    I ⎕GRDRAW DATA
```

Parameters:

I is the integer scalar value 0, 1, or 2.

DATA is a simple numeric array of shape N-by-3 or N-by-4. The first column contains the values 0 or 1 only. The last 2 or 3 columns the X, Y or X, Y, and Z world coordinates.

Description:

The ⎕*GRDRAW* function provides a mechanism to define a sequence of strokes or polygons. A new stroke begins when a row of the data in the right argument contains the value 1 in the first column. A stroke ends when all positions in the remaining rows have been connected or before the next row which contains a 1 in the first column.

If *I* is 0, each stroke is joined by a line. If *I* is 1, each point is marked. If *I* is 2, each stroke is treated as a polygon.

Example:

In the following example, the picture of the house created in the example at the beginning of this section is produced by a single use of the ⎕*GRDRAW* function rather than executing the function ⎕*GRLINE* four separate times.

```
    ∇ DRAW_EXAMPLE
[1]   H←((7↑1),HOUSE),[1]((4↑1),DOOR),[1](5↑1),WINDOW2
[2]   0 ⎕GRDRAW H
    ∇
```

Possible Errors:

A *RANK ERR* is reported if:

- the left argument is not a matrix.
- the left argument is not a scalar.

A *LENGTH ERR* is reported if:

- the right argument is not an N-by-3 or N-by-4 matrix.
- the left argument is not a single value.

A *DOMAIN ERR* is reported if:

- the right argument is not simple or all numeric.
- the left argument is not simple or not numeric.
- APL graphics is not initialized
- there is not an open segment
- the current value of ⎕*GRLI* is not supported
- the current value of ⎕*GRPEN* is not supported
- the current value of ⎕*GRLS* is not supported
- the current value of ⎕*GRLW* is not supported.
- the current value of ⎕*GRMARKER* is not supported
- the current value of ⎕*GRFILL* is not supported.
- the current value of ⎕*GRVERTEX* is not supported.
- the length of ⎕*GRVERTEX* is not equal to the number of points in the stroke.
- the vertices are not coplanar.

⎕*GRTEXT* Function (Draw Text)

Syntax:

```
POSITION ⎕GRTEXT STRING
```

Parameters:

POSITION is a simple numeric vector of 2 or 3 items which indicate a graphical position.

STRING is a simple character vector containing the characters to be displayed on the graphics device.

Description:

The ⎕*GRTEXT* function causes the character vector to appear on the graphics device starting at the position indicated. The appearance of the text depends on the current definition of the text graphics attribute variables.

Example:

```
.125 .25 ⎕GRTEXT 'HERMOSA'
```

This example displays the text '*HERMOSA*' at the world coordinate position (0.125,9.25).

Possible Errors:

A *RANK ERR* is reported if:

- the left argument is not a vector
- the right argument is not a vector or scalar

A *LENGTH ERR* is reported if:

- the left argument does not contain 2 items

A *DOMAIN ERR* is reported if:

- the right argument is not simple and all character scalars
- the left argument is not simple and all numeric
- APL graphics is not initialized
- there is not an open segment
- the current value of ⎕*GRPEN* is not supported
- the current value of ⎕*GRTEXT* is not supported
- the current value of ⎕*GRFONT* is not supported
- the string contains illegal characters
- the vectors established by ⎕*GRCHPLANE* and ⎕*GRCHUP* are parallel
- the string contains more than 256 characters
- ⎕*GRCHPLANE* and ⎕*GRCHUP* matrix cannot be inverted

⎕*GRWORLDC* Function (Map to World Coordinates)

Syntax:

```
      DATA0←⎕GRWORLDC DATA1
```

Parameters:

DATA0 is a simple array of shape N-by-2 or N-by-3 containing only scalar numbers.

DATA1 is a simple array of shape N-by-2 or N-by-3 containing only scalar numbers.

Description:

The ⎕*GRWORLDC* function returns the world coordinates associated with the normalized device coordinates supplied as the right argument.

Example:

```
      ⎕GRWORLDC 4 2⍴.2 .3 .4 .7 .8 .7 .6 .3
0.2 0.3
0.4 0.7
0.8 0.7
0.6 0.3
```

Since the default image transformation is an identity matrix, the positions supplied to this function are mapped to themselves.

Possible Errors:

A *RANK ERR* is reported if:

- the right argument is not a matrix

A *LENGTH ERR* is reported if:

- the right argument is not an N–by–2 or N–by–3 matrix

A *DOMAIN ERR* is reported if:

- a specified NDC position is outside the current viewport
- the world coordinate transformation is not invertible
- the view plane normal and view up direction are parallel
- APL graphics is not initialized
- the projection and view parameters are inconsistent
- the clipping planes are inconsistent

⎕*GRNDC* Function (Map to NDC)

Syntax:

```
      DATA0←⎕GRNDC DATA1
```

Parameters:

DATA0 is a simple array of shape N–by–2 or N–by–3 containing only scalar numbers.

DATA1 is a simple array of shape N–by–2 or N–by–3 containing only scalar numbers.

Description:

The ⎕*GRNDC* function returns the normalized device coordinates associated with the world coordinates supplied as the right argument.

Example:

```
      ⎕GRNDC 4 2⍴.2 .3 .4 .7 .8 .7 .6 .3
0.2 0.3
0.4 0.7
0.8 0.7
0.6 0.3
```

Possible Errors:

A *RANK ERR* is reported if:

- the right argument is not a matrix

A *LENGTH ERR* is reported if:

- the right argument is not an N–by–2 or N–by–3 matrix

A *DOMAIN ERR* is reported if:

- a specified world position is outside the current window and clipping is enabled
- the view plane normal and view up direction are parallel
- APL graphics is not initialized
- the projection and view parameters are inconsistent

⎕*GRTEXTX* Function (Inquire Text Extent)

Syntax:

```
      POS0←POS1 ⎕GRTEXTX STRING
```

Parameters:

POS0 is a simple numeric vector of 2 or 3 items which indicate a graphical position in world coordinates.

POS1 is a simple numeric vector of 2 or 3 items indicating a graphical position in world coordinates.

STRING is a simple character vector containing the characters to be displayed on the graphics device.

Description:

The ⎕*GRTEXTX* function is used to return the extent of the specified character string on the specified view surface, if the character string is drawn, unjustified, beginning at the position indicated by the left argument.

Example:

```
      .125 .25 ⎕GRTEXT 'HERMOSA BEACH'
0.13 0
```

Possible Errors:

A *RANK ERR* is reported if:

- the left argument is not a vector
- the right argument is not a vector or scalar

A *LENGTH ERR* is reported if:

- the left argument does not contain 2 items

A *DOMAIN ERR* is reported if:

- the right argument is not simple and all character scalars
- the left argument is not simple and all numeric
- APL graphics is not initialized
- there is not an open segment
- the current value of ⎕*GRFONT* is not supported
- the string contains illegal characters
- the vectors established by ⎕*GRCHPLANE* and ⎕*GRCHUP* are parallel
- the string contains more than 256 characters
- the ⎕*GRCHPLANE* and ⎕*GRCHUP* matrix cannot be inverted

⎕*GRCP* Function (Current Position)

Syntax:

```
      POS←⎕GRCP
```

Parameters:

POS is a simple numeric vector of 2 or 3 items which indicate a graphical position in world coordinates.

Description:

The ⎕*GRCP* niladic function is used to return the current drawing position in world coordinates.

Example:

```
      ⎕GRCP
0.125 0.25
```

Possible Errors:

A *DOMAIN ERR* is reported if:

o APL graphics is not initialized

Graphics Segment Functions

All graphics output occurs in graphical segments. The segments partition the output primitives such that the output of each primitive occurs in one and only one segment and each segment contains only graphic primitive output. Two types of segments may be used: retained segments and temporary segments.

Graphic output directed to a temporary segment exists for the life of the current frame. Output directed to a retained segment may appear in multiple frames depending on its visibility attribute. Many retained segments may be used to represent the image on the view surface. Retained segments are identified by their names which must be an integer number in the range 1 to 65535.

⎕*GRSEGOPEN* Function (Create a Retained Segment)

Syntax:

```
      ⎕GRSEGOPEN SEG
```

Parameters:

SEG is a simple numeric scalar containing the number of the segment to be created.

Description:

The ⎕*GRSEGOPEN* function creates a new empty retained segment. The dynamic attributes for the new segment are determined by the current attribute values for the retained segment dynamic attributes. The set of currently selected view surfaces is recorded with the newly created retained segment. Throughout the life of the retained segment, the image it defines appears on each view surface in this list.

The newly created segment becomes the currently open segment. Subsequent execution of graphics output functions are recorded in this retained segment. If the retained segments visibility attribute is visible, the execution of graphics output functions also results in new information appearing on the view surfaces selected by the segment. While a segment is open, the viewing parameters may not be altered and view surfaces may not be selected or deselected.

Example:

```
    ⎕GRSEGOPEN 314
```

Possible Errors:

A *LENGTH ERR* is reported if:

o the right argument does not contain exactly one item

A *DOMAIN ERR* is reported if:

o the right argument is not simple or numeric
o there is no currently selected view surface
o an open segment already exists
o the specified retained segment already exists
o there is an illegal image transformation
o the view plane normal and view up direction are parallel
o the viewing parameters are inconsistent
o the clipping parameters are inconsistent
o APL graphics is not initialized

⎕*GRSEGCLOSE* Function (Close Retained Segment)

Syntax:

```
    ⎕GRSEGCLOSE
```

Description:

The ⎕*GRSEGCLOSE* function closes currently open retained segments. Output primitives can no longer be executed. Closing a retained segment has no effect on its visibility or other segment attributes.

Example:

```
      ⎕GRSEGCLOSE
```

Possible Errors:

A *DOMAIN ERR* is reported if:

- no open retained segment exists
- APL graphics is not initialized

⎕*GRSEGDEL* Function (Delete Retained Segment)

Syntax:

```
      ⎕GRSEGDEL SEGS
```

Parameters:

SEGS is a simple integer vector containing the numbers of the segments to be deleted.

Description:

The ⎕*GRSEGDEL* function deletes the specified retained segment. If the retained segment's visibility attribute is visible, its image is removed from each view surface on which it appears. After a retained segment is deleted, it is as if the segment had never existed.

Example:

```
      ⎕GRSEGDEL 314 1
```

This example causes segments 314 and 1 to be deleted.

Possible Errors:

A *RANK ERR* is reported if:

- the right argument is not a scalar or vector

A *DOMAIN ERR* is reported if:

- the right argument is not simple or does not have integer values
- APL graphics is not initialized
- the specified retained segment does not exist

⎕*GRSEGREN* Function (Rename Retained Segment)

Syntax:

```
⎕GRSEGREN SEG2
```

Parameters:

SEG2 is a simple integer vector containing 2 items which are the current segment number and the new number by which that segment is to be known.

Description:

The ⎕*GRSEGREN* function renames the specified segment. The original retained segment name can no longer be referenced. This function has no visible effect.

Example:

```
⎕GRSEGREN 2 314
```

This example changes the name (segment number) of segment 2 to number 314.

Possible Errors:

A *RANK ERR* is reported if:

- the right argument is not a scalar or vector

A *LENGTH ERR* is reported if:

- the right argument is not of length 2

A *DOMAIN ERR* is reported if:

- the right argument is not simple or not near integer
- the first item of the right argument is not a retained segment number
- the second item of the right argument is a currently existing segment number
- APL graphics is not initialized

⎕*GRSEGSURFS* Function (Inquire Segment Surfaces)

Syntax:

```
SURFS←⎕GRSEGSURFS SEG
```

Parameters:

SEG is a simple numeric scalar containing the number of an existing segment.

SURFS is a simple integer vector of view surface numbers.

Description:

The *⎕GRSEGSURFS* function returns the number of the view surfaces which were selected when the retained segment was created.

Example:

```
      ⎕GRSEGSURFS 314
1
```

This example demonstrates that graphics output to segment 314 appears on view surface 1.

Possible Errors:

A *LENGTH ERR* is reported if:

o the right argument does not contain exactly one item

A *DOMAIN ERR* is reported if:

o the right argument is not simple or near integer
o the specified segment does not exist
o APL graphics is not initialized

⎕GRSEGS Function (Inquire Retained Segment Names)

Syntax:

```
      SEGS←⎕GRSEGS
```

Parameters:

SEGS is a simple integer vector result.

Description:

The *⎕GRSEGS* function returns the numbers of all of the current retained segments.

Example:

```
      ⎕GRSEGS
3 314
```

This example displays the number of current retained segments, in this case there are currently two retained segments numbered 3 and 314.

Possible Errors:

A *DOMAIN ERR* is reported if:

o APL graphics is not initialized

⎕GRSEGCURR Function (Inquire Open Segment)

Syntax:

```
     SEG←⎕GRSEGCURR
```

Parameters:

SEG a simple integer scalar result.

Description:

The result of the *⎕GRSEGCURR* function is 0 if there is not a currently open retained segment, or it is the number of the currently open retained segment.

⎕GRTSEGO Function (Create Temporary Segment)

Syntax:

```
⎕GRTSEGO
```

Description:

The *⎕GRTSEGO* creates an open temporary segment. Subsequent execution of output primitives result in information appearing on the currently selected view surfaces. While a temporary segment is open, the viewing parameters may not be altered and view surfaces may not be selected or deselected.

Example:

```
     ⎕GRTSEGCO
```

Possible Errors:

A *DOMAIN ERR* is reported if:

- APL graphics is not initialized
- the set of selected view surfaces is empty
- an open segment already exists
- the viewing parameters are inconsistent
- the clipping parameters are inconsistent

⎕*GRTSEGC* Function (Close Temporary Segment)

Syntax:

```
      ⎕GRTSEGC
```

Description:

The ⎕*GRTSEGC* function closes the currently open temporary segment. Output primitives may no longer be executed.

Example:

```
      ⎕GRTSEGC
```

Possible Errors:

A *DOMAIN ERR* is reported if:

- o APL graphics is not initialized
- o there is no open temporary segment

⎕*GRTSEG* Function (Inquire Open Temporary Segment)

Syntax:

```
      LOGL←⎕GRTSEG
```

Parameters:

LOGL is a scalar logical value 0 or 1.

Description:

The result of executing the ⎕*GRTSEG* function is the value 1 if there is a temporary segment currently open, otherwise the result is 0.

Example:

```
      ⎕GRTSEG
0
```

Possible Errors:

A *DOMAIN ERR* is reported if:

- o APL graphics is not initialized

⎕*GRSEGVISIBILITY* Function (Segment Visibility)

Syntax:

```
     LOGL0←LOGL1 ⎕GRSEGVISIBILITY SEGS
     LOGL0←      ⎕GRSEGVISIBILITY SEGS
```

Parameters:

*LOGL*0 is a simple logical vector.

SEGS is a simple integer vector containing the numbers of retained segments.

*LOGL*1 is a simple logical vector of the same length as *SEGS*. Or, it is a single logical value to be used for each item in *SEGS*.

Description:

The ⎕*GRSEGVISIBILITY* function is used to modify or inquire about the current value of the segment visibility attribute for each segment number in the right argument. If the left argument is not provided, the result has the same length as the right argument. In this case, each result value is 1 if the segment visibility attribute is '*VISIBLE*' or 0 if the segment visibility attribute is '*INVISIBLE*'.

If the left argument is provided, the values specified by the left argument are used as the new segment visibility dynamic attribute for each of the segments specified in the right argument; the result is an empty vector. The value 1 is used to set the segment visibility to '*VISIBLE*' and the value 0 is used to set the segment visibility to '*INVISIBLE*'.

Example:

```
      1 ⎕GRSEGVISIBILITY 3 314
      ⎕GRSEGVISIBILITY 3 314
1 1
```

Possible Errors:

A *RANK ERR* is reported if:

- the left or right arguments are not vectors or scalars

A *LENGTH ERR* is reported if:

- the left argument is not the same length as the right argument and the left argument is not a single item.

A *DOMAIN ERR* is reported if:

- the right argument contains an item that is not a simple integer value
- the left argument contains an invalid value
- the right argument contains a number which is not the number of a retained segment
- APL graphics is not initialized

⎕*GRSEGHIGHLIGHT* Function (Segment Highlight)

Syntax:

```
     LOGL0←LOGL1 ⎕GRSEGHIGHLIGHT  SEGS
     LOGL0←      ⎕GRSEGHIGHLIGHT SEGS
```

Parameters:

LOGL0 is a simple logical vector.

SEGS is a simple integer vector containing the numbers of retained segments.

LOGL1 is a simple logical vector of the same length as *SEGS*. Or, it is a single logical value to be used for each item in *SEGS*.

Description:

The ⎕*GRSEGHIGHLIGHT* function is used to modify or inquire on the current value of the segment highlighting attribute for each segment number in the right argument. If the left argument is not provided, the result has the same length as the right argument. In this case, each result value is 1 if the segment highlighting attribute is '*HIGHLIGHTED*', or 0 if the segment highlighting attribute is '*NON-HIGHLIGHTED*'.

If the left argument is provided, the values specified by the left argument are used as the new segment highlighting dynamic attribute for each of the segments specified in the right argument; the result is an empty vector. The value 1 is used to set the segment highlighting to '*HIGHLIGHTED*', and the value 0 is used to set the segment highlighting to '*NON-HIGHLIGHTED*'.

Example:

```
      1 ⎕GRSEGHIGHLIGHT 3 314
      ⎕GRSEGHIGHLIGHT 3 314
1 1
```

Possible Errors:

A *RANK ERR* is reported if:

- the left or right arguments are not vectors or scalars

A *LENGTH ERR* is reported if:

- the left argument is not the same length as the right argument and the left argument is not a single item

A *DOMAIN ERR* is reported if:

- the right argument contains an item that is not a simple integer value
- the left argument contains an invalid value
- the right argument contains a number which is not the number of a retained segment

⎕*GRVISIBILITY* Variable (Set/Inquire Visibility)

Syntax:

```
    ⎕GRVISIBILITY←LOGL
```

Parameters:

LOGL is either a scalar numeric value 0 or 1. Or, it is a simple vector containing the characters *'VISIBLE'* or *'INVISIBLE'*.

Description:

The ⎕*GRVISIBILITY* system variable is used to define the default segment visibility attribute. This value is used when a new segment is created by the ⎕*GRTSEGO* or ⎕*GRSEGOPEN* system functions.

Example:

```
      ⎕GRVISIBILITY←1
      ⎕GRVISIBILITY
1
```

Possible Errors:

A *RANK ERR* is reported if:

o a character value assigned is not a scalar or vector

A *LENGTH ERR* is reported if:

o a numeric value assigned contains more than one item

A *DOMAIN ERR* is reported if:

o the value assigned is not simple.
o a numeric value assigned is not 0 or 1
o a character value assigned is not a legal keyword

⎕*GRHIGHLIGHT* Variable (Set/Inquire Highlighting)

Syntax:

```
    ⎕GRHIGHLIGHT←LOGL
```

Parameters:

LOGL is either a scalar numeric value 0 or 1. Or, it is a simple vector containing the characters *'HIGHLIGHTED'* or *'NON-HIGHLIGHTED'*.

Description:

The ⎕*GRHIGHLIGHT* system variable is used to define the default segment highlighting attribute. This value is used when a new segment is created by the ⎕*GRTSEGO* or ⎕*GRSEGOPEN* system functions.

Example:

```
      ⎕GRHIGHLIGHT←1
      ⎕GRHIGHLIGHT
1
```

Possible Errors:

A *RANK ERR* is reported if:

- a character value assigned is not a scalar or vector

A *LENGTH ERR* is reported if:

- a numeric value assigned contains more than one item

A *DOMAIN ERR* is reported if:

- the value assigned is not simple.
- a numeric value assigned is not 0 or 1
- a character value assigned is not a legal keyword

Graphics Attribute Variables

The graphics output functions use the current values of the graphics attribute system variables to determine the actual appearance of the graphics output. These system variables may be referenced to obtain their current value, assigned new values, or localized within functions. When localized, these system variables maintain their previous value unless reassigned.

Values assigned to these attributes affect future use of their associated graphics output functions.

The assignment of these attribute variables can result in a *DOMAIN ERR* report from APL for the following reasons:

- the attribute value is invalid
- APL graphics is not initialized

Other errors can include *RANK ERR*, *LENGTH ERR*, and *DOMAIN ERR* when requirements set out by the parameter definition for the variable are not met.

⎕*GRMARKER* Variable (Marker Symbol)

Syntax:

```
      ⎕GRMARKER←KEYWORD
```

Parameters:

KEYWORD is a scalar integer value which indicates the number of the desired marker symbol.

Description:

The ⎕*GRMARKER* variable is used by the ⎕*GRMARK* system function to specify a marker symbol. The valid values are integers in the range 1 through the device driver defined maximum. The values 1 through 5 always produce the symbols: dot, star, capital letter O and capital letter X. Legal values greater than 5 produce symbols determined by the device. The default value is 1.

Example:

```
      ⎕GRMARKER←3
      ⎕GRMARKER
3
```

⎕*GRPINS* Variable (Polygon Interior Style)

Syntax:

```
      ⎕GRPINS←KEYWORD
```

Parameters:

KEYWORD is a simple character vector containing one of the keyword values: *'PLAIN'*, *'SHADED'*, or *'PATTERNED'*.

Description:

The ⎕*GRPINS* variable is used by the ⎕*GRPOLYGON* system function to determine the method for filling the image of the interior of a visible polygon. The default value is *'PLAIN'*.

Example:

```
      ⎕GRPINS←'SHADED'
      ⎕GRPINS
SHADED
```

□*GRPES* Variable (Polygon Edge Style)

Syntax:

```
      □GRPES←KEYWORD
```

Parameters:

KEYWORD is a simple character vector containing one of the keyword values: *'SOLID'* or *'INTERIOR'*.

Description:

The □*GRPES* variable is used by the □*GRPOLYGON* system function to determine the method for forming the image of the border (edges) of a visible polygon. The default value is *'SOLID'*.

Example:

```
      □GRPES←'SOLID'
      □GRPES
SOLID
```

□*GRLW* Variable (Line Width)

Syntax:

```
      □GRLW←N
```

Parameters:

N specifies the relative width of the image of a visible line.

Description:

The □*GRLW* variable specifies the relative width of the image of a visible line generated by the □*GRLINE* system function. The specified value must be in the range from 0 to 1 inclusive. The default value is 0.

Example:

```
      □GRLW←.0625
      □GRLW
0.0625
```

⎕GRLI Variable (Line Index)

Syntax:

```
      ⎕GRLI←I
```

Parameters:

I   is a simple integer scalar value.

Description:

The ⎕GRLI variable specifies the index used to select the color or intensity of the image of a visible line, marker, or polygon edge (whose ⎕GRPES is 'SOLID'). This value only applies to the system functions ⎕GRLINE, ⎕GRMARKER, and ⎕GRPOLYGON. The default value is 1.

Example:

```
      ⎕GRLI←2
      ⎕GRLI
2
```

⎕GRLS Variable (Line Style)

Syntax:

```
      ⎕GRLS←I
```

Parameters:

I   is a scalar integer value.

Description:

The ⎕GRLS variable is used by the ⎕GRLINE function to determine the style of the image of a visible line (e.g., solid, dashed). The default value is 1 which corresponds to a solid line.

Example:

```
      ⎕GRLS←4
      ⎕GRLS
4
```

⎕*GRPEN* Variable (Pen)

Syntax:

```
      ⎕GRPEN←I
```

Parameters:

I is a scalar integer value.

Description:

The ⎕*GRPEN* variable may be used by all output primitives to distinguish the image of their output. The particular values of ⎕*GRLI*, ⎕*GRLS*, ⎕*GRLW*, and ⎕*GRTEXTI* which correspond to a particular pen value are implementation and device dependent. The default value of 0 corresponds to the current settings or ⎕*GRLI*, ⎕*GRLS*, ⎕*GRLW*, and ⎕*GRTEXTI*. All other ⎕*GRPEN* values override these values.

Example:

```
      ⎕GRPEN←2
      ⎕GRPEN
2
```

⎕*GRFONT* Variable (Font)

Syntax:

```
      ⎕GRFONT←I
```

Parameters:

I is a scalar integer value.

Description:

The ⎕*GRFONT* variable is used by the ⎕*GRTEXT* function to specify the style of a visible character. Values range from 1 to a device dependent maximum. The default value of 1 corresponds to ASCII.

Example:

```
      ⎕GRFONT←29        ⍝ SELECT APL FONT
      ⎕GRFONT
29
```

⎕*GRTEXTI* Variable (Text Index)

Syntax:

```
      ⎕GRTEXTI←I
```

Parameters:

I is a scalar integer value.

Description:

The ⎕*GRTEXTI* variable is used by the ⎕*GRTEXT* function to select the color or intensity of the images of a visible character. The default value is 1.

Example:

```
     ⎕GRTEXTI←3
     ⎕GRTEXTI
3
```

⎕*GRCHSIZE* Variable (Character Size)

Syntax:

```
      ⎕GRCHSIZE←NN
```

Parameters:

NN is a simple two item numeric vector.

Description:

The ⎕*GRCHSIZE* variable is used by the ⎕*GRTEXT* function to select the desired size (in world coordinate units) of a character. The pair of values is used to select both the width and height of a character. The values must be in the range of 0 to 1 inclusive. The default value is (0.01 0.01).

Example:

```
      ⎕GRCHSIZE←.0625 .0625
      ⎕GRCHSIZE
0.625 0.625
```

⎕*GRCHPLANE* Variable (Character Plane)

Syntax:

```
     ⎕GRCHPLANE←NNN
```

Parameters:

NNN   is a simple numeric vector of length 3.

Description:

The ⎕*GRCHPLANE* variable is used by the ⎕*GRTEXT* function to select the orientation in the world coordinate system of the plane on which the characters will appear. The three values contain an X, Y, and Z component. The default value is (0 0 ¯1).

Example:

```
     ⎕GRCHPLANE←0 2 1
     ⎕GRCHPLANE
0 2 1
```

⎕*GRCHUP* Variable (Character Up)

Syntax:

```
     ⎕GRCHUP←NNN
```

Parameters:

NNN   is a simple numeric vector of length 3.

Description:

The ⎕*GRCHUP* variable is used by the ⎕*GRTEXT* system function selects the principal up direction in the plane on which the characters will appear. The component of ⎕*GRCHUP* perpendicular to ⎕*GRCHPLANE* points up. The value is made up of (X, Y, Z) components. The default value is (0 1 0).

Example:

```
     ⎕GRCHUP←1 .25
     ⎕GRCHUP
1 0.25
```

⎕*GRCHPATH* Variable (Character Path)

Syntax:

```
      ⎕GRCHPATH←KEYWORD
```

Parameters:

KEYWORD is a simple character vector containing one of the keyword values: *'RIGHT'*, *'LEFT'*, *'UP'*, or *'DOWN'*.

Description:

The ⎕*GRCHPATH* variable is used by the ⎕*GRTEXT* function to determine the direction within the plane on which the characters will appear. The default value is *'RIGHT'*.

Example:

```
      ⎕GRCHPATH←'DOWN'
      ⎕GRCHPATH
DOWN
```

⎕*GRCHSPACE* Variable (Character Space)

Syntax:

```
      ⎕GRCHSPACE←N
```

Parameters:

N is a scalar numeric value in the range 0 to 1.

Description:

The ⎕*GRCHSPACE* variable is used by the ⎕*GRTEXT* system function to determine additional spacing between adjacent characters in a string. The value represents the fraction of the ⎕*GRCHSIZE* attribute. The default value is 0.

Example:

```
      ⎕GRCHSPACE←.03125
      ⎕GRCHSPACE
0.03125
```

⎕*GRCHJUST* Variable (Character Justification)

Syntax:

```
    ⎕GRCHJUST←KEYWORDS
```

Parameters:

KEYWORDS is a simple character vector containing two keywords separated by a blank. The first keyword value must be either *'OFF'*, *'LEFT'*, *'RIGHT'*, or *'CENTER'*. The second keyword value must be either *'OFF'*, *'TOP'*, *'BOTTOM'*, or *'CENTER'*.

Description:

The ⎕*GRCHJUST* variable is used by the ⎕*GRTEXT* system function to determine the mode of string justification. It has two components specifying the horizontal and vertical justifications. The default value is *'OFF OFF'*.

Example:

```
    ⎕GRCHJUST←'LEFT TOP'
    ⎕GRCHJUST
LEFT TOP
```

⎕*GRCHPREC* Variable (Character Precision)

Syntax:

```
    ⎕GRCHPREC←KEYWORD
```

Parameters:

KEYWORD is a simple character vector containing one of the keyword values: *'STRING'*, *'CHARACTER'*, or *'STROKE'*.

Description:

The ⎕*GRCHPREC* is used by the ⎕*GRTEXT* system function to determine the precision of the appearance of text output. The default value is *'STRING'*.

Example:

```
    ⎕GRCHPREC←'STROKE'
    ⎕GRCHPREC
STROKE
```

⎕*GRFILL* Variable (Fill Index)

Syntax:

```
     ⎕GRFILL←I
```

Parameters:

I is a simple integer scalar value.

Description:

The ⎕*GRFILL* variable specifies the index used to select the color or intensity used to fill the image of a visible polygon.

Example:

```
      ⎕GRFILL←3
      ⎕GRFILL
3
```

⎕*GRVERTEX* Variable (Vertex Indices)

Syntax:

```
     ⎕GRVERTEX←IV
```

Parameters:

IV is a simple integer vector containing at least 3 values.

Description:

The ⎕*GRVERTEX* variable is used by the ⎕*GRPOLYGON* function to specify the set of index values used to shade the image of a visible polygon (whose ⎕*GRPINS* is '*SHADED*'). The default value is (1 1 1).

Example:

```
      ⎕GRVERTEX←3 2 1 4 5
      ⎕GRVERTEX
3 2 1 4 5
```

Graphics Viewing Variables

APL graphics provides system variables to define the viewing operations and the coordinate transformations to be made when graphics output is generated. These system variables may be referenced to obtain their current value, assigned to modify their value or localized within user-defined functions. The most recently specified values of the viewing parameters (or their defaults) are used to determine the viewing.

Viewing parameters may not be specified while a graphics segment is open (or being created). If the termination of a defined function causes a viewing variable to be surfaced while a segment is open, then the surfaced value is lost.

The valid values of a viewing variable may depend on whether the graphics system is initialized as 2D or 3D. At graphics initialization time, all values of graphics viewing variables are respecified and set to their defaults.

⎕*GRWINDOW* Variable (Window)

Syntax:

```
      ⎕GRWINDOW←NNNN
```

Parameters:

NNNN is a simple numeric vector of length 4.

Description:

The ⎕*GRWINDOW* variable specifies a rectangle in world coordinates. The values define the X-minimum, X-maximum, Y-minimum, and Y-maximum extents of the window. The default window is (0 1 0 1).

Example:

```
      ⎕GRWINDOW←0 .75 .25 1
      ⎕GRWINDOW
0 0.75 0.25 1
```

Possible Errors:

A *RANK ERR* is reported if:

o the assigned value is not a vector.

A *LENGTH ERR* is reported if:

o the assigned value is not of length 4

A *DOMAIN ERR* is reported if:

o the assigned value is not simple or all numeric
o a segment is currently open
o the first item is greater than the second or the third item is greater than the fourth
o APL graphics is not initialized

⎕*GRUP* Variable (View Up)

Syntax:

```
      ⎕GRUP←N23
```

Parameters:

N23 is a simple numeric vector of 2 items for 2D graphics or 3 items for 3D graphics.

Description:

The ⎕*GRUP* variable defines the world coordinate up direction so that the world coordinate Y–axis need not be upright on the view surface. The default value for 2D graphics is (0 1) and (0 1 0) for 3D graphics.

Example:

```
      ⎕GRUP←1 0
      ⎕GRUP
1 0
```

Possible Errors:

A *RANK ERR* is reported if:

- the assigned value is not a vector

A *LENGTH ERR* is reported if:

- the assigned value is not of length 2 or 3

A *DOMAIN ERR* is reported if:

- the assigned value is not simple or all numeric
- APL graphics is not initialized
- a segment is currently open
- all assigned values are zero

⎕*GRSPACE* Variable (NDC Space)

Syntax:

```
      ⎕GRSPACE←N23
```

Parameters:

N23 is a simple numeric vector of 2 items for 2D graphics or 3 items for 3D graphics.

Description:

The *⎕GRSPACE* variable defines the size of the normalized device coordinate space which can be addressed on the view surface of all display devices and within which viewports are specified. All values must be in the range 0 to 1 inclusive and at least one value must be 1. This value may be specified at most once per initialization. The default value for 2D is (1 1) and (1 1 0) for 3D graphics.

Example:

```
      ⎕GRSPACE
1 1
```

Possible Errors:

A *RANK ERR* is reported if:

o   the assigned value is not a vector

A *LENGTH ERR* is reported if:

o   the assigned value is not 2 or 3 items in length

A *DOMAIN ERR* is reported if:

o   the assigned value is not simple or not all numeric
o   APL graphics is not initialized
o   the NDC space is already established
o   a value is not in the range of 0 to 1
o   neither value is 1
o   the first or second value is 0

⎕GRVIEWPORT Variable (Viewport)

Syntax:

```
      ⎕GRVIEWPORT←NNNN
```

Parameters:

NNNN   is a simple numeric vector of length 4.

Description:

The *⎕GRVIEWPORT* variable specifies a rectangle in normalized device coordinate space. The viewport cannot exceed the bounds defined for the variable *⎕GRSPACE*. The values define (in order) the X—minimum, X—maximum, Y—minimum, and Y—maximum limits of the viewport. The default value of this variable is the value of *⎕GRSPACE*.

Example:

```
      ⎕GRVIEWPORT←.125 .875 0 1
      ⎕GRVIEWPORT
0.125 0.875 0 1
```

Possible Errors:

A *RANK ERR* is reported if:

- the assigned value is not a vector

A *LENGTH ERR* is reported if:

- the assigned value is not of length 4

A *DOMAIN ERR* is reported if:

- the assigned value is not simple or all numeric
- APL graphics is not initialized
- a segment is open
- the first item is greater than the second item or the third item is greater than the fourth item
- the viewport is outside of NDC space

⎕*GRVREFPT* Variable (View Reference Point)

Syntax:

```
    ⎕GRVREFPT←NNN
```

Parameters:

NNN is a simple numeric vector of length 3.

Description:

The ⎕*GRVREFPT* variable specifies the view reference point in world coordinates. The value specified defines (in order) the X, Y, and Z location of the reference point. The default value is (0 0 0). This value is only used in 3D viewing.

Example:

```
      ⎕GRVREFPT←1 2 3
      ⎕GRVREFPT
1 2 3
```

Possible Errors:

A *RANK ERR* is reported if:

- the assigned value is not a vector

A *LENGTH ERR* is reported if:

- the assigned value is not 3 items in length

A *DOMAIN ERR* is reported if:

- the assigned value is not simple or all numeric
- APL graphics is not initialized
- a segment is open

⎕*GRVPLNORM* Variable (View Plane Normal)

Syntax:

```
      ⎕GRVPLNORM←NNN
```

Parameters:

NNN is a simple numeric vector of length 3.

Description:

The ⎕*GRVPLNORM* variable specifies the view plane normal. The value specified determines a vector in world coordinates relative to the view reference point. The value specified defines (in order) the X, Y, and Z location of the view plane normal. The default value is (0 0 -1). This value is only used in 3D viewing.

Example:

```
      ⎕GRVPLNORM←1 0 1
      ⎕GRVPLNORM
1 0 1
```

Possible Errors:

A *RANK ERR* is reported if:

o the assigned value is not a vector

A *LENGTH ERR* is reported if:

o the assigned value is not of length 3

A *DOMAIN ERR* is reported if:

o the assigned value is not simple or all numeric
o APL graphics is not initialized
o all three values are zero

⎕*GRVPLNDIS* Variable (View Plane Distance)

Syntax:

```
      ⎕GRVPLNDIS←N
```

Parameters:

N is a simple numeric scalar value.

Description:

The ⎕*GRVPLNDIS* variable specifies the distance of the view plane. The default value is 0 which signifies that the view plane is located at the view reference point.

Example:

```
      ⎕GRVPLNDIS←2
      ⎕GRVPLNDIS
2
```

Possible Errors:

A *LENGTH ERR* is reported if:

o the assigned value is not exactly one item

A *DOMAIN ERR* is reported if:

o the assigned value is not simple or numeric
o APL graphics is not initialized
o a segment is open

⎕*GRVDEPTH* Variable (View Depth)

Syntax:

```
      ⎕GRVDEPTH←NN
```

Parameters:

NN is a simple numeric vector of length 2.

Description:

The ⎕*GRVDEPTH* variable specifies the clipping depth planes but does not affect whether depth clipping is performed. The first item is the front distance from the view reference point. The second item is the back distance from the view reference point. The default value is (0 -1).

Example:

```
      ⎕GRVDEPTH←¯2 2
      ⎕GRVDEPTH
¯2 2
```

Possible Errors:

A *RANK ERR* is reported if:

o the assigned value is not a vector

A *LENGTH ERR* is reported if:

o the assigned value is not of length 2

A *DOMAIN ERR* is reported if:

- the assigned value is not simple or all numeric
- APL graphics is not initialized
- a segment is open
- the first item is greater than or equal to the second

⎕*GRPROJECTION* Variable (Projection Type)

Syntax:

```
      ⎕GRPROJECTION←PROJ
```

Parameters:

PROJ is a 4 item vector. The projection keyword is an enclosed character vector representing the first item. The X, Y, and Z positions in world coordinates are the final 3 items. The projection keyword value must be either '*PARALLEL*' or '*PERSPECITIVE*'.

Description:

The ⎕*GRPROJECTION* variable specifies the type of projection used in the 3D viewing operation. If a parallel projection is specified, the world coordinate position indicates the lines are parallel to a line through the view reference point and this point. If a perspective projection is selected, the world coordinate position specifies the center of the projection. The default value is ('*PARALLEL*' 0 0 1).

Example:

```
      ⎕GRPROJECTION←'PERSPECTIVE' 0 0 2
      ⎕GRPROJECTION
PERSPECTIVE 0 0 2
```

Possible Errors:

A *RANK ERR* is reported if:

- the assigned value is not a vector
- the first item of the assigned value is not a scalar or a vector

A *LENGTH ERR* is reported if:

- the assigned value is not of length 4

A *DOMAIN ERR* is reported if:

- the first item of the assigned value is not character or the remaining items not scalar numbers
- the first item of the assigned value is not a valid keyword
- a segment is open
- the assigned value is ('*PARALLEL*' 0 0 0)
- APL graphics is not initialized

Window Clipping Variables

APL automatically provides a window clipping capability if clipping is enabled. The clipping is controlled by specifying values for the three clipping variables: *⎕GRCLIP*, *⎕GRFCLIP*, and *⎕GRBCLIP*.

⎕GRCLIP Variable (Window Clipping)

Syntax:

```
      ⎕GRCLIP←L
```

Parameters:

L specifies a value which can be the simple numeric scalar 0 or 1, or the keyword value *'ON'* or *'OFF'*.

Description:

The *⎕GRCLIP* variable specifies whether window clipping is enabled. The values *'ON'* or 1 may be used to turn window clipping on. The values *'OFF'* or 0 may be used to turn window clipping off. When referenced, the value is 0 or 1. The default value is 1.

Example:

```
      ⎕GRCLIP←'OFF'
      ⎕GRCLIP
0
```

Possible Errors:

A *RANK ERR* is reported if:

o the character value assigned is not a scalar or vector

A *LENGTH ERR* is reported if:

o the numeric value assigned is not a singleton

A *DOMAIN ERR* is reported if:

o a character value assigned is not a clipping keyword
o a numeric value being assigned is not 0 or 1.
o APL graphics is not initialized
o a segment is open

⎕*GRFCLIP* Variable (Front Plane Clipping)

Syntax:

```
      ⎕GRFCLIP←L
```

Parameters:

L specifies a value which can be the simple numeric scalar 0 or 1, or the keyword value *'ON'* or *'OFF'*.

Description:

The ⎕*GRFCLIP* variable specifies whether front plane clipping is enabled. The values *'ON'* or 1 may be used to turn front plane clipping on. The values *'OFF'* or 0 may be used to turn front plane clipping off. When referenced, the value is 0 or 1. The default value is 0.

Example:

```
      ⎕GRFCLIP←0
      ⎕GRFCLIP
0
```

Possible Errors:

A *RANK ERR* is reported if:

- the character value assigned is not a scalar or vector

A *LENGTH ERR* is reported if:

- the numeric value assigned is not a singleton

A *DOMAIN ERR* is reported if:

- a character value assigned is not a clipping keyword
- a numeric value being assigned is not 0 or 1
- APL graphics is not initialized
- a segment is open

⎕*GRBCLIP* Variable (Back Plane Clipping)

Syntax:

```
      ⎕GRBCLIP←L
```

Parameters:

L specifies a value which can be the simple numeric scalar 0 or 1, or the keyword value *'ON'* or *'OFF'*.

Description:

The *⎕GRBCLIP* variable specifies whether back plane clipping is enabled. The values *'ON'* or 1 may be used to turn back plane clipping on. The values *'OFF'* or 0 may be used to turn back plane clipping off. When referenced, the value is 0 or 1. The default value is 0.

Example:

```
      ⎕GRBCLIP←1
      ⎕GRBCLIP
1
```

Possible Errors:

A *RANK ERR* is reported if:

o the character value assigned is not a scalar or vector

A *LENGTH ERR* is reported if:

o the numeric value assigned is not a singleton

A *DOMAIN ERR* is reported if:

o a character value assigned is not a clipping keyword
o a numeric value being assigned is not 0 or 1
o APL graphics is not initialized
o a segment is open

⎕GRCOORD Variable (Coordinate System Type)

Syntax:

```
      ⎕GRCOORD←KEYWORD
```

Parameters:

KEYWORD is a simple character vector containing one of the keyword values: *'LEFT'* or *'RIGHT'*.

Description:

The *⎕GRCOORD* variable is used to specify whether the world coordinate system is left-handed or right-handed. The default value is *'RIGHT'*.

Example:

```
      ⎕GRCOORD
RIGHT
```

Possible Errors:

A *RANK ERR* is reported if:

- the character value assigned is not a scalar or vector

A *LENGTH ERR* is reported if:

- the numeric value assigned is not a singleton

A *DOMAIN ERR* is reported if:

- a character value assigned is not a clipping keyword
- a numeric value being assigned is not 0 or 1
- APL graphics is not initialized
- a segment is open

⎕*GRWORLD* Variable (World Transformation)

Syntax:

```
      ⎕GRWORLD←MAT
```

Parameters:

MAT is a simple numeric matrix of shape (3 3) for 2D transformations or shape (4 4) for 3D transformations.

Description:

The ⎕*GRWORLD* variable is used to transform world coordinate positions. This matrix is used in a matrix multiplication which can effect scaling, translation, and rotation of the position.

Example:

```
      ⎕GRWORLD
1 0 0
0 1 0
0 0 1
```

Possible Errors:

A *RANK ERR* is reported if:

- the value assigned is not a matrix

A *LENGTH ERR* is reported if:

- the value assigned is not shape 3-by-3 or shape 4-by-4

A *DOMAIN ERR* is reported if:

- the value assigned is not simple and all numeric
- APL graphics is not initialized

Graphics Control Functions and Variables

CP-6 APL provides several functions and variables which control the picture generation process. These capabilities are provided for:

- Initiating and terminating APL graphics
- View surface (device) control
- Picture change control
- Frame control
- Color specification
- Color and intensity binding
- Pixel array definition

Before graphics output can occur, APL graphics must be initialized and a view surface must be initialized and selected.

⎕*GRINIT* Function (Initialize APL Graphics)

Syntax:

```
    ⎕GRINIT STRING
```

Parameters:

STRING is a simple character vector containing initialization keywords separated by blanks or a single comma. Any keyword that is all blank is defaulted.

Description:

The ⎕*GRINIT* function is used to initialize APL graphics. The right argument may contain up to four keywords which define (in order) the graphics output level, the input level, the number of dimensions, and the hidden surface removal level. The values for output level are: *'BASIC'*, *'BUFFERED'*, *'DYNAMIC-A'*, *'DYNAMIC-B'*, and *'DYNAMIC-C'*. The value for the input level is: *'NONE'*. The values for the number of dimensions are: *'2D'* or *'3D'*. The value for the hidden surface removal level is: *'NONE'*.

This function must be executed before any other graphics function is executed or any graphics variable is referenced or assigned. It causes all graphics variables to be initialized to their default values.

Example:

```
    ⎕GRINIT 'BUFFERED,,2D'
```

In this example, APL graphics is initialized with the output level *'BUFFERED'*, the input level *'NONE'*, a dimensionality of *'2D'*, and a hidden surface removal level of *'NONE'*. The default right argument (if an empty or all blank value is specified) is *'BUFFERED,NONE,2D,NONE'*.

Possible Errors:

A *RANK ERR* is reported if:

- the right argument is not a vector or scalar

A *DOMAIN ERR* is reported if:

- o Graphics is already initialized
- o the output level is not supported
- o the input level is not supported
- o the dimension level is not supported
- o the hidden surface level is not supported
- o an undefined keyword is provided
- o too many keywords are provided
- o APL graphics capability is not available

⎕*GRDONE* Function (Terminate APL Graphics)

Syntax:

```
⎕GRDONE
```

Description:

The ⎕*GRDONE* function closes any open segments, terminates all initialized view surfaces, and releases all resources used by APL graphics. After this function is executed, APL graphics may be reinitialized by executing the ⎕*GRINIT* function.

Example:

```
⎕GRDONE
```

Possible Errors:

A *DOMAIN ERR* is reported if:

- o APL graphics is not initialized

⎕*GRINITSURF* Function (Initialize View Surface)

Syntax:

```
STRING ⎕GRINITSURF SURF
```

Parameters:

STRING is a simple character vector containing options separated by commas.

SURF is a simple integer scalar indicating a view surface number.

Description:

The ⎕*GRINITSURF* function obtains access to the specified view surface and clears it. The right argument is a view surface number. The left argument specifies attributes of the view surface. The left argument is composed of the keyword '*INTENSITY*' or '*COLOR*', followed by the CP-6 fid indicating the location of the view surface (default is '*ME*'). This is followed by the CP-6 device profile name (default is current profile if online or super-graphics device if batch). Each of these parameters is separated by blanks or commas. The default left argument (if it is empty) is '*INTENSITY,ME*'.

Example:

```
        'INTENSITY' ⎕GRINITSURF 1
```

In this case, the view surface is cleared and the device is prepared for graphics output on the terminal.

Possible Errors:

A *RANK ERR* is reported if:

- the left argument is not a scalar or vector

A *LENGTH ERR* is reported if:

- the right argument contains more than one item

A *DOMAIN ERR* is reported if:

- the view surface is already initialized
- all possible view surfaces are initialized
- an unknown keyword value is specified
- color is not provided on this surface
- the left argument is not all character scalar items
- the right argument is not an integer scalar
- too many keywords in the left argument

⎕*GRTERMSURF* Function (Terminate View Surface)

Syntax:

```
      ⎕GRTERMSURF SURFS
```

Parameters:

SURFS is a simple integer vector of view surface numbers.

Description:

The ⎕*GRTERMSURF* function terminates access to the view surface. Segments whose images appear only on this view surface are deleted.

Example:

```
      ⎕GRTERMSURF 1
```

In this case, the view surface number 1 is terminated.

Possible Errors:

A *RANK ERR* is reported if:

- the right argument is not a scalar or vector

A *DOMAIN ERR* is reported if:

- the view surface is not initialized
- APL graphics is not initialized
- an item of the right argument is not an integer

⎕*GRCAPABILITIES* Function (Inquire Capabilities)

Syntax:

CAP←⎕GRCAPABILITIES SURF

Parameters:

SURF is a simple integer scalar indicating a view surface number.

CAP is a 6–item nested array indicating the device capabilities.

Description:

The ⎕*GRCAPABILITIES* function returns the device capabilities associated with the specified view surface. Each item of the result indicates attributes of the device. They are separated into 6 classes of information which are:

1 the highest initialization levels supported by the device.

2 an indication of whether the device is physically there or is a pseudo device.

3 the size of the view surface (in centimeters).

4 the level of support available for the primitive attributes.

5 an indication of how the segment attributes are supported.

6 an indication of the effect of batching functions.

The result is a 6 item vector containing:

1. LEVELS

   A 3 item nested vector containing three character vectors:

   1 highest output level supported
   2 highest dimension level supported
   3 highest hidden surface level supported

2. PHYSICAL

   A character vector containing the value 'PSEUDO' or 'REAL'.

3. SIZES

   A 6 item numeric vector containing:

   1–2 width and height of view surface in centimeters
   3–4 width and height of NDC space area in centimeters
   5–6 horizontal and vertical resolution per centimeter

4. PRIMITIVE_ATTRIBUTES

   A 22 item numeric vector containing:

   1 Line index color count
   2 Line index intensity count
   3 Line index global color count
   4 Line index global intensity count
   5 Line index intensity hard/soft
   6 Line index color type
   7 Line index intensity type
   8 Line style hardware count

9 Line style software count
10 Line width count
11 Line width hard/soft
12 Line width minimum NDC
13 Line width maximum NDC
14 PEN hardware count
15 PEN software count
16 FONT count
17 Charsize count
18 Charsize hard/soft
19 Charsize minimum size in NDC
20 Charsize maximum size in NDC
21 Marker hardware count
22 Marker software count

5. SEGMENT_ATTRIBUTES

A vector containing two character vectors:

1 highlighting support
2 image transformation support

6. BATCHING

A character vector indicating the strategy used by the device for batching of updates.

Example:

```
      ρCAP←⎕GRCAPABILITIES 1
6
```

Possible Errors:

A *LENGTH ERR* is reported if:

- the right argument is not a single value

A *DOMAIN ERR* is reported if:

- the right argument value is not an integer
- the right argument value is not a valid view surface
- APL Graphics is not initialized

⎕GRSURFACE Function (Select View Surface)

Syntax:

```
      ⎕GRSURFACE SURFS
```

Parameters:

SURFS is a simple integer vector of view surface numbers.

Description:

The *⎕GRSURFACE* function adds the specified view surfaces to the list of currently selected view surfaces. When a subsequent segment is created, the graphics output will appear only on those surfaces that are currently selected.

Example:

```
      ⎕GRSURFACE 1
```

This example selects view surface 1.

Possible Errors:

A *RANK ERR* is reported if:

- o the right argument is not a scalar or vector

A *DOMAIN ERR* is reported if:

- o the right argument values are not integer
- o a segment is currently open
- o the specified view surface is not initialized
- o the specified view surface has been selected
- o APL Graphics is not initialized

⎕GRUNSURFACE Function (Deselect View Surface)

Syntax:

```
      ⎕GRUNSURFACE SURFS
```

Parameters:

SURFS is a simple integer vector of view surface numbers.

Description:

The *⎕GRUNSURFACE* function removes the view surface specified from the set of selected view surfaces. Subsequent segment creations and *⎕GRFRAME* function execution will not affect this view surface until it is reselected.

Example:

```
      ⎕GRUNSURFACE 1
```

This example deselects view surface 1.

Possible Errors:

A *RANK ERR* is reported if:

- o the right argument is not a scalar or vector

A *DOMAIN ERR* is reported if:

- o the right argument contains a non-integer value
- o a segment is open
- o the view surface has not been selected
- o APL Graphics is not initialized

⎕*GRSURFACES* Function (Inquire Selected Surfaces)

Syntax:

```
      SURFS←⎕GRSURFACES
```

Parameters:

SURFS is a simple integer vector of view surface numbers.

Description:

The ⎕*GRSURFACE* function returns a vector containing the numbers of the selected view surfaces.

Example:

```
      ⎕GRSURFACES
1
```

In this example, the result is the vector 1 indicating that view surface 1 has been selected.

Possible Errors:

A *DOMAIN ERR* is reported if:

- APL graphics is not initialized

⎕*GRIMMVISIBILITY* Function (Immediate Visibility)

Syntax:

```
      ⎕GRIMMVISIBILITY LOGL
```

Parameters:

LOGL is the simple scalar number 0 or 1. Optionally, the keyword '*ON*' or '*OFF*' may be used.

Description:

Specifying the value 1 as the right argument causes all delayed visible picture changes to take effect unless they are deferred as a result of batching.

Example:

```
      ⎕GRIMMVISIBILITY 1
```

Possible Errors:

A *RANK ERR* is reported if:

- o the right argument is character and is not a scalar or vector

A *LENGTH ERR* is reported if:

- o the right argument is numeric and contains more than one item

A *DOMAIN ERR* is reported if:

- o the right argument is not the number 0 or 1 or the character vector is not *'ON'* or *'OFF'*.
- o APL graphics is not initialized

⎕*GRCURRENT* Function (Make Picture Current)

Syntax:

```
⎕GRCURRENT
```

Description:

The ⎕*GRCURRENT* function has no effect if immediate visibility is 1. If immediate visibility is 0, all delayed visible picture changes take effect subject to batching.

Example:

```
⎕GRCURRENT
```

Possible Errors:

A *DOMAIN ERR* is reported if:

- o APL graphics is not initialized

⎕*GRBATCH* Function (Control Batching of Updates)

Syntax:

```
⎕GRBATCH LOGL
```

Parameters:

LOGL is the simple numeric scalar value 0 or 1.

Description:

If the right argument is 1, a batch of updates is started. The end of the batch of updates is indicated by executing the ⎕*GRBATCH* function with a right argument value of 0. While batching of updates is in effect, visible picture changes are deferred.

Example:

```
      ⎕GRBATCH 1
      ⎕GRLINE 7 2ρ .1 .6 .5 .9 .6 .9 .1 .9 .1 .1 .6 .1 .6 .9
      ⎕GRBATCH 0
```

Possible Errors:

A *LENGTH ERR* is reported if:

o the right argument contains more than one item

A *DOMAIN ERR* is reported if:

o the right argument value is not 0 or 1
o APL graphics is not initialized
o the right argument value is 1 and already batching updates
o the right argument value is 0 and not batching updates

⎕*GRCSTATUS* Function (Inquire Control Status)

Syntax:

```
       LOGL2←⎕GRCSTATUS
```

Parameters:

LOGL2 is a simple vector of 2 items containing the values 0 or 1.

Description:

The ⎕*GRSTATUS* function indicates the current status of immediacy and batching of updates. The first item is 1 if updates are immediately visible, otherwise it is 0. The second item is 1 if within a batch of updates, otherwise it is 0.

Example:

```
       ⎕GRCSTATUS
1 0
```

This result indicates that immediate visibility is in effect and that no batching of updates is being performed.

Possible Errors:

A *DOMAIN ERR* is reported if:

o APL graphics is not initialized

⎕*GRFRAME* Function (New Frame)

Syntax:

```
      ⎕GRFRAME
```

Description:

The ⎕*GRFRAME* function causes a new frame action to occur. The result on each affected view surface is that the surface is erased and all visible retained segments are redrawn.

Possible Errors:

A *DOMAIN ERR* is reported if:

- the set of currently selected view surfaces is empty
- APL graphics is not initialized

⎕*GRCOLMODEL* Function (Color Model)

Syntax:

```
      ⎕GRCOLMODEL KEY
```

Parameters:

KEY is a simple character vector containing the keyword value '*HLS*', '*RGB*' or ''.

Description:

The ⎕*GRCOLMODEL* function is used to establish or inquire about the current color model. If the right argument is empty or contains only blanks, the result is the current color model type ('*RGB*' or '*HLS*'). If the right argument contains non-blank characters, they are used to specify the color model. The color model may be established once after APL graphics is initialized before any view surfaces are initialized.

Example:

```
      ⎕GRCOLMODEL ''
HLS
```

This example demonstrates obtaining the current color model.

Possible Errors:

A *RANK ERR* is reported if:

- the right argument is not a scalar or vector.

A *DOMAIN ERR* is reported if:

- the keyword specified is not '*RGB*' or '*HLS*'
- a keyword is specified and a view surface has been initialized
- APL graphics is not initialized

⎕*GRCOLINDEX* Function (Set/Inquire Color Indices)

Syntax:

```
     COLS←COLS ⎕GRCOLINDEX SURF
     COLS←     ⎕GRCOLINDEX SURF
```

Parameters:

SURF is a simple integer scalar indicating a view surface number.

COLS is a simple N-by-3 numeric array whose values range from 0 to 1.

Description:

Dyadically, the ⎕*GRCOLINDEX* function sets all of the color entries for the specified view surface to the color components specified by the left argument and returns an empty vector. Monadically, the ⎕*GRCOLINDEX* function returns the currently defined color entries for the specified view surface.

Example:

```
      (4 3⍴ 0 0 1 0 1 0 1 0 0 1 1 1) ⎕GRCOLINDEX 1
      ⎕GRCOLINDEX 1
0 0 1
0 1 0
1 0 0
1 1 1
```

Possible Errors:

A *RANK ERR* is reported if:

- the left argument is not a matrix

A *LENGTH ERR* is reported if:

- the right argument is not exactly one item
- the second dimension of the left argument is not 3

A *DOMAIN ERR* is reported if:

- the right argument is not an integer
- the left argument is not numeric
- the specified view surface is not initialized
- the view surface is not of type '*COLOR*'
- too many indices are specified
- one or more of the color parameters is invalid
- APL graphics is not initialized

⎕*GRINTINDEX* Function (Set/Inquire Intensity Indices)

Syntax:

```
      INTS←INTS ⎕GRINTINDEX SURF
      INTS←     ⎕GRINTINDEX SURF
```

Parameters:

SURF is a simple integer scalar indicating a view surface number.

INTS is a simple numeric vector whose values range from 0 to 1.

Description:

Dyadically, the ⎕*GRINTINDEX* function sets all of the intensity entries for the specified view surface to the intensity values specified by the left argument and returns an empty vector. Monadically, the ⎕*GRINTINDEX* function returns the currently defined intensity entries for the specified view surface.

Example:

```
     .1 .2 .3 .4 .5 .6 .7 ⎕GRINTINDEX 1
     ⎕GRCOLINDEX 1
0.1 0.2 0.3 0.4 0.5 0.6 0.7
```

Possible Errors:

A *RANK ERR* is reported if:

- the left argument is not a vector or scalar

A *LENGTH ERR* is reported if:

- the right argument is not exactly one item

A *DOMAIN ERR* is reported if:

- the right argument is not an integer
- the left argument is not numeric
- the specified view surface is not initialized
- the view surface is not of type '*INTENSITY*'
- too many indices are specified
- one or more of the intensity values are invalid
- APL graphics is not initialized

⎕*GRBACKGROUND* Variable (Background Index)

Syntax:

```
    ⎕GRBACKGROUND←I
```

Parameters:

I is a simple integer scalar.

Description:

The ⎕*GRBACKGROUND* variable controls the current background index. During a new-frame action, the background is set to the color or intensity specified by the value of this variable.

Example:

```
      ⎕GRBACKGROUND←1
      ⎕GRBACKGROUND
1
```

Possible Errors:

A *LENGTH ERR* is reported if:

o the value being assigned contains more than one item

A *DOMAIN ERR* is reported if:

o the specified index value is not supported by a view surface
o APL graphics is not initialized

⎕*GRPIXEL* Variable (Pixel Array)

Syntax:

```
    ⎕GRPIXEL←MAT
```

Parameters:

MAT is a simple integer matrix.

Description:

The ⎕*GRPIXEL* variable specifies the pixel array that is used when a polygon is drawn, and the ⎕*GRPINS* variable has the value '*PATTERNED*'.

Example:

```
      ⎕GRPIXEL←2 2⍴ 2 3 1 4
      ⎕GRPIXEL
2 3
1 4
```

Possible Errors:

A *RANK ERR* is reported if:

- the assigned value is not a matrix

A *DOMAIN ERR* is reported if:

- the matrix is too large
- the matrix is empty
- APL graphics is not initialized

⎕*GRPIXELORG* Variable (Pixel Pattern Origin)

Syntax:

```
      ⎕GRPIXELORG←XY
```

Parameters:

XY is a simple 2 item vector containing a position in normalized device coordinate space. The default value is 0 0.

Description:

The ⎕*GRPIXELORG* variable is used to specify the origin for the transfer of the pixel array to the view surface. The origin specifies the position of the item in the lower left-hand corner of the current pixel array.

Example:

```
      ⎕GRPIXELORG←0.25 0.125
      ⎕GRPIXELORG
0.25 0.125
```

Possible Errors:

A *RANK ERR* is reported if:

- the assigned value is not a vector

A *LENGTH ERR* is reported if:

- the assigned value is not exactly 2 items

A *DOMAIN ERR* is reported if:

- the value specified is outside of NDC space
- the value specified is not numeric
- APL graphics is not initialized

Section 16
Blind I/O

Blind I/O is a capability which is of use in a number of specialized cases. The major uses of blind I/O include:

- Sending or receiving data to CP-6 devices or files without undergoing any translation or validity checking by APL.
- Exercising more control over a CP-6 device than is possible with normal APL input and output.
- Accessing multiple input and output streams (or devices) simultaneously.
- Creating a terminal independent CP-6 FORM which permits reading, writing and clearing specified fields in a screen-oriented fashion.

For example, blind input permits the entry of overstrikes (or any other characters) which would result in a *BAD CHAR* error if normal APL input were used. Blind output may be used to output special character sequences (including ASCII control characters) to perform special device functions such as controlling a plotter.

Using Blind I/O

APL provides ten DCBs - F$Q0 through F$Q9 to be used for blind I/O, but performs no special set-up on them. It is assumed that the DCB will be assigned to devices or files, using the)*SET* command (section 8). If a)*SET* command has not been issued, the blind I/O streams default to the standard APL input (and output) streams (the CP-6 ME device, which is the terminal if online or the card-reader if batch).

Within APL, the characters 0⃞ through 9⃞ (quad overstruck with 0 through 9) supplement the quad and quote-quad characters. They are used to access the DCBs when blind input or output is desired.

There is no limit on the size of a record input via blind I/O. Input from blind DCBs creates a character vector result. If the data actually contains logic values, integer values, or floating point values, then the ⎕*CVT* function may be used to correct the data type after input.

Blind output may only be used to output simple APL arrays. It should be noted, however, that large output records routed to physical devices with maximum length constraints will be truncated on output. In particular, records output to the user's console should be limited to 511 bytes, and records output to a line printer to 132 bytes. Note also that blind output of non-character data to a printing device may lead to unpredictable results.

APL bypasses all of its translation sequences (overstrike resolution and mnemonic substitution) for blind input. If an end-of-file condition is encountered by a blind-input request, APL returns an empty numeric vector result.

Blind I/O on a Device

In the following examples, ⎕2 is assigned to the user's console (if the session is online) after calling APL, as follows:

```
    )SET ⎕2 ME,ORG=TERMINAL,FUN=UPDATE
```

Quad-2 may then used for blind input and for blind output to the terminal. In the example below, blind input functions much the same as a quote-quad input, since the terminal itself is the input device.

```
      A←⎕2
NOW IS THE TIME FOR ALL GOOD MEN.
      A
NOW IS THE TIME FOR ALL GOOD MEN.
```

Blind input can be used to input illegal overstrike characters, which cannot be done with quote-quad input. Note, however, that the characters entered appear in the result, including backspaces, and that overstrikes are not mapped into single APL characters.

The examples below illustrate blind output to the terminal. Only simple arrays containing all character or all numeric data may be written with blind I/O. Note that the data to be output was specified as a literal. When the RETURN key is struck, the data is output the terminal exactly as it was input.

```
      ⎕2←'1234567890+×QWERTYUIOP→ASDFGHJKL[]ZXCVBNM,./'
1234567890+×QWERTYUIOP→ASDFGHJKL[]ZXCVBNM,./

      ⎕2←'¨¯<≤=≥>≠∨∧−÷?⍵∊⍴~↑↓⍳○⋆→⍺⌈⌊-∇∆∘''⎕()⊂⊃∩∪⊥⊤|;:\'
¨¯<≤=≥>≠∨∧−÷?⍵∊⍴~↑↓⍳○⋆→⍺⌈⌊-∇∆∘'⎕()⊂⊃∩∪⊥⊤|;:\

      ⎕2←'ASDF'
ASDF
```

Note that the ⎕BTRANS function may be used to request transparent input. Transparent input from a terminal (with ORG=TERMINAL specified on the SET command) should only be specified when all of the control characters entered are of interest to the APL program. In this mode, the terminal read is not terminated until the number of characters specified by the read (511 by default, or see ⎕BSIZE) have been received. The command:

```
      )SET ⎕1 ME,FUN=CREATE
```

is used to set ⎕1 to the terminal (or lineprinter in batch). In this case, unit record oriented functions may be performed on the stream. This includes the ability to specify a page heading on the SET command and to use the ⎕BLINES function to determine the number of lines per page and the number of lines remaining on the current page. The ⎕BVFC function may be used to indicate that the first character of each output line is to be used to control line spacing.

The ⎕TSQZ system function may be used to map legal APL overstrikes and mnemonics into their internal representations or to map the internal APL characters into mnemonics and overstrikes appropriate for the current APL terminal (as indicated by the current terminal type).

Accessing Files with Blind I/O

In the following examples, ⍞ is assigned to a test input file which is built using CP-6 EDIT:

```
!EDIT
EDIT C00 HERE
*BUILD BLINDIN
   1.000 BLINDIN, RECORD 1,
   2.000 RECORD 2. TEST BACKSPACING.
   3.000 LAST RECORD
   4.000
*END
```

Record 2 of the file contains a series of blanks and backspaces such that the total number of characters in the record is considerably more than the example shows.

After APL is called:

```
    )SET ⍞ BLINDIN,FUN=IN
```

In the next example, blind input is used to input records from the file. Note that an attempt to use blind I/O to access the non-existent fourth record results in an empty integer vector.

```
    A←⍞
    B←⍞
    C←⍞
    D←⍞
```

A, B, C, and D now contain the data from the file records, as shown below. Note that the length of B reflects the blanks and backspace characters that were part of the file record.

```
      ⍴A
18
      A
BLINDIN, RECORD 1.

      ⍴B
81
      B
RECORD 2. TEST BACKSPACING.

      ⍴C
15
      C
LAST RECORD.

      ⍴D
0
```

When blind output to a file is used, records are output as character data — scalars, vectors, and arrays are written without any sort of header data.

The blind I/O system functions ⎕*BPRECORD*, ⎕*BPFILE*, and ⎕*BREW* may be used to position to a specific record within the file. The system functions ⎕*BSEED* and ⎕*BREWRITE* may be used to set record encryption or or to indicate that the last record read is to be replaced.

When writing records to a keyed, random or relative file, the record identifier (number or key) may be specified by assigning a nested array whose first item is a simple non-negative scalar integer less than 134217726 or a simple character vector. The second item is a non-scalar array to be written.

For example:

```
      ⍞←5000 'EDIT KEY FIVE'
      ⍞←'TEXT KEY' 'RECORD ASSOCIATED WITH KEY:''TEXT KEY'''
```

In the first example, the integer key 5000 is used to write the record. In the second statement, the character key 'TEXT KEY' is used to write the record. Note that blind I/O does not permit records to be read by their key as does APL File I/O (see Section 12).

Blind I/O System Functions

CP-6 APL allows options to be specified for reading and writing with blind I/O. These capabilities include specification and interrogation of VFC, TRANSPARENCY and BINARY modes, setting the encryption seed for reading and writing, and setting the size of the record to read.

The right argument to the blind I/O system functions must be a scalar or vector or a *RANK ERR* is reported. If a vector argument is provided, there must be one or two items or a *LENGTH ERR* is reported. The right argument must be simple and contain only scalar integers or a *DOMAIN ERR* is reported.

When a scalar or one-item vector argument is provided, the current status of an I/O option is returned. When a two item vector argument is provided, the I/O option is set for a subsequent read or write. The first item of the argument is the blind I/O stream number to be affected.

⎕*BBIN* Function (Set and Query Binary Mode)

Syntax:

```
I←⎕BBIN N,L
```

Parameters:

N is a simple integer scalar representing the blind I/O channel number.

L is an optional simple integer scalar value 0 or 1.

I is a simple integer scalar value 0 or 1.

Description:

The ⎕*BBIN* function sets or resets binary mode. If *L* is equal to one, then subsequent reads and writes through channel N will be in binary mode. If *L* is zero, then subsequent reads and writes will not specify binary mode. If *L* is not present, the result indicates whether the last operation was BINARY.

When querying binary mode, this function indicates whether the last record read or written was with BINARY.

⎕*BSIZE* Function (Read Size)

Syntax:

```
I←⎕BSIZE N,I
```

Parameters:

N is a simple integer scalar representing the blind I/O channel number.

I is an optional simple integer scalar value greater than or equal to 0.

Description:

When a blind I/O stream is *SET* or opened for the first time, APL determines a default read size that is sufficient for any record in the file. This function is used to override the default input record size. The integer value *I* is subsequently used for the record size. If *I* is zero, then APL reverts back to the default read size for the stream. If *I* is not specified in the right argument, the result is the current default read record size; otherwise, an empty vector is returned.

⎕*BVFC* Function (Set and Query VFC)

Syntax:

```
I←⎕BVFC N,L
```

Parameters:

N is a simple integer scalar representing the blind I/O channel number.

L is an optional simple integer scalar value 0 or 1.

I is the simple integer scalar value 0 or 1.

Description:

If *L* is not present, then this function indicates whether the last record read was originally written with VFC.

If *L* is one, subsequent writes through channel *N* will specify VFC. If *L* is zero, then subsequent writes will not specify VFC.

⎕*BTRANS* Function (Set and Query Transparency)

Syntax:

```
I←⎕BTRANS N,L
```

Parameters:

N is a simple integer scalar representing the blind I/O channel number.

I is the simple integer scalar value 0 or 1.

L is an optional simple integer scalar value 0 or 1.

Description:

If *L* is not present, then this function indicates whether the last record read or written was with transparency.

If *L* is one, subsequent blind I/O reads and writes through channel *N* will specify TRANSPARENCY. If *L* is zero, then subsequent writes will not specify transparency.

⎕*BLINES* Function (Lines Remaining)

Syntax:

```
I←⎕BLINES N
```

Parameters:

N is a simple integer scalar representing the blind I/O channel number.

I is a simple 2-item integer vector.

Description:

This function returns a two item integer vector containing the number of lines per page, and the number of lines currently remaining on the page printed through channel *N*.

⎕*BKEY* Function (Return Key)

Syntax:

```
K←⎕BKEY N
```

Parameters:

N   is a simple integer scalar representing the blind I/O channel number.

K   is a record key returned as an integer or character vector.

Description:

The result is the key of the next record to be read, or, if the file is not keyed or indexed, the record number of the next record to be read. Three byte keys and record numbers are returned as integers, all other keys are returned as character vectors. This function operates on tape or disk files only.

⎕*BPRECORD* Function (Position Record)

Syntax:

    K←⎕BPRECORD N,I

Parameters:

N   is a simple integer scalar representing the blind I/O channel number.

I   is a simple integer scalar.

K   is a record key returned as an integer or character vector.

Description:

If *I* is positive, the file or device is positioned *I* records ahead. If *I* is negative, the file or device is backspaced *I* records. The result is the key of the record positioned to (or record number for sequential files).

⎕*BPFILE* Function (Position File)

Syntax:

    ⎕BPFILE N,L

Parameters:

N   is a simple integer scalar representing the blind I/O channel number.

L   is a simple integer scalar.

Description:

If *L* is zero, then the file or device is positioned to the beginning of the file. If *L* is one, then the file or device is positioned to the end of the file.

⎕*BREW* Function (Rewind)

Syntax:

  ⎕*BREW N*

Parameters:

N is a simple integer scalar representing the blind I/O channel number.

Description:

The file associated with stream N is positioned to the beginning of file or the device associated with stream *N* is rewound.

⎕*BREWRITE* Function (Rewrite Record)

Syntax:

  ⎕*BREWRITE N,L*

Parameters:

N is a simple integer scalar representing the blind I/O channel number.

L is a simple integer scalar.

Description:

If *L* is one, then subsequent writes will specify the REWRITE option. If *L* is zero, then the REWRITE option is not specified. If *L* is not specified, the result is the current setting of rewrite for this stream.

⎕*BSEED* Function (Encryption Seed)

Syntax:

  ⎕*BSEED N,I*

Parameters:

N is a simple integer scalar representing the blind I/O channel number.

I is a simple integer scalar.

Description:

The integer value *I* is used on all succeeding reads and writes as the encryption seed for channel *N*.

⎕*BRR* Function (Re-Read Mode)

Syntax:

R←⎕*BRR IV*

Parameters:

IV is a simple 1 or 2-item integer vector containing as the first item a blind I/O stream number. The second item must be the value 0 or 1.

R is a simple integer scalar containing the previous re-read setting.

Description:

The ⎕*BRR* function permits the specification of re-read on blind input for each blind I/O stream. This function may be used in conjunction with the ⎕*TTIN* system function which sets the re-read line. If the second item of *IV* is not present, then the result indicates whether re-read will be specified on the next read. If the second item of *IV* is 1, re-read will be specified on the next read. If the second item of *IV* is 0, then re-read will not be specified on the next read.

⎕*BRS* Function (Record Size)

Syntax:

R←⎕*BRS I*

Parameters:

I is a simple integer scalar containing a blind I/O stream number.

R is a simple integer scalar representing the value of the ARS# field of the DCB associated with the specified stream.

Description:

The ⎕*BRS* system function is used to return the setting of the DCB field F$DCB.ARS#. This is intended to be used with screen edit access mode which provides information via this DCB field.

⎕*BKR* Function (Key Returned)

Syntax:

```
R←⎕BKR I
```

Parameters:

I is a simple integer scalar whose value is used to indicate a blind I/O stream to be affected.

R is a simple character vector.

Description:

The ⎕*BKR* system function is used to return the key specified by the most recent M$READ or M$WRITE associated with a blind I/O stream whose ORG is SE or FORM.

⎕*BCLOSE* Function (Close Blind I/O Channel)

Syntax:

```
⎕BCLOSE N

⎕BCLOSE N,L
```

Parameters:

N is a simple integer scalar representing a blind I/O channel number.

L is a simple integer scalar value 0 or 1.

Description:

The ⎕*BCLOSE* function closes the specified blind I/O channel. If *L* is 1 or not specified, the channel is closed with *SAVE*. If *L* is 0, the channel is closed with *RELEASE*. The *RELEASE* option is used to delete files or make windows created by a)*SET* command disappear.

If this function is not executed, an automatic close is performed whenever a)*SET* command is issued to a channel, or when the APL session ends.

⎕*BPAGE* Function (Skip to New Page)

Syntax:

> ⎕*BPAGE N*
>
> ⎕*BPAGE N*,*L*

Parameters:

N is a simple integer scalar representing a blind I/O channel number.

L is a simple integer scalar value 0 or 1.

Description:

The ⎕*BPAGE* function is used to eject the current page of a unit record device.

If the blind I/O channel is open to the terminal with ORG=FORM, the argument *L* is used to control the display. For screens, if *L*=0, then this function causes the screen to be updated. If the device is not a screen and *L*=0, then nothing is printed. If *L*=1, then the screen is updated or the form is printed.

⎕*BDELREC* Function (Delete Record)

Syntax:

> ⎕*BDELREC N*
>
> ⎕*BDELREC N*,*K*1
>
> ⎕*BDELREC N*,*K*1,*K*2

Parameters:

N is a simple integer scalar representing the blind I/O channel number.

*K*1 is a simple integer scalar.

*K*2 is a simple integer scalar.

Description:

The ⎕*BDELREC* function is used to delete a record (or records) from a file. If *K*1 is not specified, the last record read or written is deleted. Otherwise, *K*1 indicates the key of the record to delete. If *K*2 is specified, all records between *K*1 and *K*2 (inclusive) are deleted.

⎕*UNSET* Function (Unset DCB)

Syntax:

```
V←⎕UNSET N
```

Parameters:

N is a simple integer scalar representing the blind I/O channel number or a simple character scalar or vector.

Description:

If *N* is an integer channel number, the result is a character string containing the fid and)*SET* options for this blind I/O channel. If *N* is a character vector, the result is the setting for the DCB named.

Forms Mode

Forms mode is a terminal independent method of defining a screen (form) consisting of a number of fields and accessing specific fields for the purpose of reading, writing and erasing. In forms mode, a field is a variable length string of characters which are on a specific line, start in a specific column and occupy N columns (where N is the length of the field). Multiple fields may appear on the same line, but fields may not overlap. Forms mode permits form definition, field selection, field input, field output and selective erasure.

When a blind I/O channel is open with *ORG=FORM*, the result of a read and the value to write is a matrix of two columns. The first column is an integer key or field number. The second column contains enclosed character vectors.

When forms mode is required, the blind I/O channel that is used should be)*SET* to the terminal with the option *ORG=FORM*. If a CRT terminal is in use, a window for the form should also be defined with the WWIDTH= and WLENGTH= options. Next, the fields for that form must be defined by executing the ⎕*BFLD* system function. This system function defines the position, length, initial contents, and attributes of the fields on the screen. Attributes and contents of currently defined screen fields can be modified by the ⎕*BMFLD* system function.

Before reading from an ORG=FORM blind I/O channel, the fields that are to be input must be selected by the ⎕*BSFLD* system function. Only those input fields that are currently selected may have values input for them. The ⎕*BRFLD* system function is used to release currently selected fields. The ⎕*BXFLD* system function selectively erases fields.

Finally, the ⎕*BPAGE* system function is used to make changes to the form visible. Normally, when a field value is written or an attribute is changed, the changes do not appear until a read is issued to the form. The ⎕*BPAGE* system function is used to force changes to appear. The right argument of ⎕*BPAGE* may be a 1 or 2-item integer vector. The first item is the blind I/O stream and the second item is 0 or 1 to force changes to the current screen or (if the device is not a screen) 1 to force a copy of the form to be displayed on the terminal. If the terminal device is not a screen, then changes to the form are not displayed unless the second item of the right argument is 1 or is omitted.

Field Definition Matrix

The form is initially defined using the ⎕*BFLD* system function. After definition, the attributes and values of the fields can be modified by the ⎕*BMFLD* system function. The left argument for both of these functions must be a field definition matrix.

This matrix must have at least four columns and no more than seven columns. The first six columns always contain integer values; the seventh column must always contain character vectors. Each row of the matrix defines a field.

Field Definition Matrix Columns

1. Field Number. This number is used to refer to the field when selecting, erasing, modifying or writing. Field numbers are integer scalars in the range 0 through 65535. When modifying a field definition, a field number value of −1 is used to modify the definition of all currently selected fields.

2. Row Number. This number locates the row on the screen in which a field will appear. Row numbers are integer scalars in the range 1 through 254. When modifying a field definition, a value of −1 must be specified for the row number.

3. Column Number. This number locates the column on the screen in which a field begins. Column numbers are integer scalars in the range 1 through 254. When modifying a field definition, a value of −1 must be specified for the column number.

4. Length. This number determines the number of character positions that the field takes up. Field lengths are integer scalars in the range 1 through 254. When modifying a field definition, a value of −1 must be specified for the field length.

5. Field Rendition Attributes. Rendition attributes are scalar integers whose values are the sums of the inclusion values listed in table 16−1. The default attributes are obtained (when defining a field) by using the value −1 or 0. When modifying a field definition, a value of −1 for the rendition attribute is used to indicate no change to the current rendition attributes. The value 0 indicates the default attributes.

Table 16−1. Blind I/O Field Rendition Attributes

Value	Description
1	reserved for future use
2	reverse video
4	fast blink
8	slow blink
16	underscore
32	decreased intensity
64	increased intensity
128	hidden

6. Field Input Attributes. Field input attributes are scalar integers whose values are the sums of the inclusion values listed in table 16−2. The default input attributes are obtained by using the value −1 or 0. When modifying a field definition, a value of −1 for the input attribute is used to indicate no change to the current input attributes.

Table 16–2. Blind I/O Field Input Attributes

Value	Description
1	reserved for future use
2	constant
4	input required
8	protected
16	letters permitted
32	numbers permitted
64	graphic characters permitted
128	protect and guard field
256	all characters permitted

Attributes that are not supported by the device in use are not available.

7. Value. Field values are simple character vectors. When modifying a field definition, an empty vector is used to indicate that the current field value is not to be changed.

⎕*BFLD* Function (Field Definition)

Syntax:

```
M ⎕BFLD I
```

Parameters:

M is a field definition matrix (see below).

I is a simple integer scalar representing a valid blind I/O channel.

Description:

The ⎕*BFLD* system function defines the fields indicated by the left argument field definition matrix. The field definition matrix contains definitions of the location and size of each field, input attributes, rendition attributes, and the current (or initial) value of the field.

Examples:

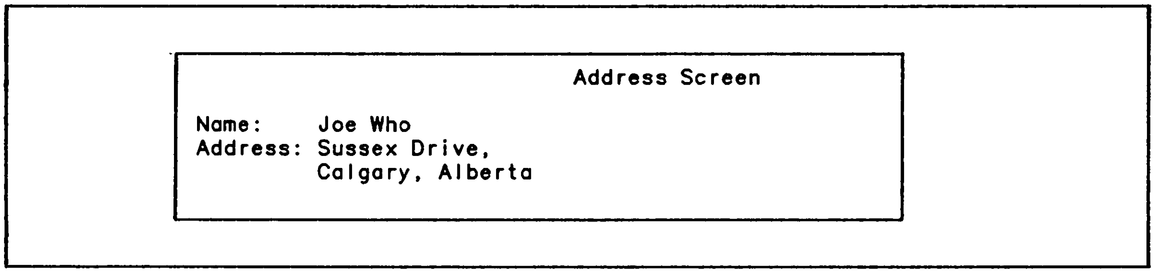
The following example demonstrates the creation of a form for entry of names and addresses. Figure 16–1 shows the screen image that results from the form definition example.

```
      FIELDS←2 7⍴1 1 30 15 0 0 'ADDRESS SCREEN' 2 3 2 5 0 0 'NAME:'
      FIELDS←FIELDS,[1]3 3 11 35 0 0 'JOE WHO'
      FIELDS←FIELDS,[1]4 4 2 8 0 0 'ADDRESS:'
      FIELDS←FIELDS,[1]5 4 11 35 0 0 'SUSSEX DRIVE,'
      FIELDS←FIELDS,[1]6 5 11 35 0 0 'CALGARY, ALBERTA'

      ⍝    DISPLAY THE FIELD DEFINITION MATRIX
      FIELDS
1 1 30 15 0 0    ADDRESS SCREEN
2 3  2  5 0 0             NAME:
3 3 11 35 0 0           JOE WHO
4 4  2  8 0 0          ADDRESS:
5 4 11 35 0 0     SUSSEX DRIVE,
6 5 11 35 0 0  CALGARY, ALBERTA

      ⍝    SET THE BLIND I/O CHANNEL TO BE USED
      )SET ⌷ UC05,ORG=FORM,FUN=UPDATE
```

```
FIELDS ⎕BFLD 1
```



```
                    Address Screen

Name:     Joe Who
Address:  Sussex Drive,
          Calgary, Alberta
```

Figure 16–1. Forms Mode Screen Display

Possible Errors:

A *RANK ERR* is reported if:

- the left argument is not a matrix or vector.
- the seventh column of the left argument does not contain scalars or vectors.

A *LENGTH ERR* is reported if:

- the right argument contains more than one item.
- the left argument contains more than 7 or fewer than 4 columns.

A *DOMAIN ERR* is reported if:

- the right argument is not a simple scalar integer in the range 0 through 9 inclusive.
- the first six columns of the left argument are not scalar integers.
- the seventh column of the left argument is not simple character vectors or scalars.
- the first column of the left argument contains a value (field number) greater than 65535 or less than 0.
- the second column of the left argument contains a value (row number) greater than 254 or less than 0.
- the third column of the left argument contains a value (column number) greater than 254 or less than 0.
- the fourth column of the left argument contains a value (field length) greater than 254 or less than 0.
- the fifth column of the left argument contains a value (field rendition) greater than 1023 or less than −1.
- the sixth column of the left argument contains a value (input attributes) greater than 1023 or less than −1.

An *I/O ERR* is reported if:

- the field definition is not consistent with CP-6 requirements.

⎕BMFLD Function (Modify Field)

Syntax:

```
M ⎕BMFLD I
```

Parameters:

M is a field definition matrix.

I is a simple integer scalar representing a valid blind I/O channel.

Description:

The left argument is a field definition matrix as described for the *⎕BFLD* system function.

Because the field location and length may not be modified, the value specified for them must be 0. A value of −1 for either the field rendition attributes or the field input attributes is used to indicate no change in this attribute. An empty vector is used to indicate no change in the current field value.

Examples:

In the following example, fields 3, 5, and 6 of the current form are selected and their attributes are modified to reverse video and input required:

```
      3 5 6 ⎕BSFLD 1      ⍝  SELECT OUR INPUT FIELDS


      ⍝     THE FOLLOWING EXECUTION OF THE ⎕BMFLD SYSTEM
      ⍝     FUNCTION WILL CAUSE ALL CURRENTLY SELECTED
      ⍝     FIELDS TO BE MODIFIED TO THE RENDITION ATTRIBUTE
      ⍝     OF REVERSE VIDEO AND THEIR INPUT ATTRIBUTES TO
      ⍝     INPUT REQUIRED.

      ¯1 ¯1 ¯1 ¯1 2 4 ⎕BMFLD 1
```

Possible Errors:

A *RANK ERR* is reported if:

- the left argument is not a matrix or vector.
- the seventh column of the left argument does not contain scalars or vectors.

A *LENGTH ERR* is reported if:

- the right argument contains more than one item.
- the left argument contains more than 7 or fewer than 4 columns.

A *DOMAIN ERR* is reported if:

- the right argument is not a simple scalar integer in the range 0 through 9 inclusive.
- the first six columns of the left argument are not scalar integers.
- the seventh column of the left argument is not simple character vectors or scalars.
- the first column of the left argument contains a value (field number) greater than 65535 or less than −1.
- the second column of the left argument contains a value (row number) other than −1.
- the third column of the left argument contains a value (column number) other than

-1.

- o the fourth column of the left argument contains a value (field length) other than -1.
- o the fifth column of the left argument contains a value (field rendition) greater than 1023 or less than -1.
- o the sixth column of the left argument contains a value (input attributes) greater than 1023 or less than -1.

An *I/O ERR* is reported if:

- o the field definition is not consistent with CP-6 requirements.

⎕*BSFLD* Function (Select Field)

Syntax:

IV ⎕*BSFLD* *I*

Parameters:

IV is a simple integer vector of field numbers.

I is a simple integer scalar representing a valid blind I/O channel.

Description:

The ⎕*BSFLD* function is used to select fields that are to be affected by subsequent field operations. For example,in order to input a field value, it must have been selected previous to the read. A field number of -1 is used to select all currently defined fields.

Examples:

In the following example, fields 3, 5, and 6 are selected:

```
3 5 6 ⎕BSFLD 1
```

Possible Errors:

A *RANK ERR* is reported if:

- o the left argument is not a vector or scalar.

A *LENGTH ERR* is reported if:

- o the right argument contains more than one item.

A *DOMAIN ERR* is reported if:

- o the right argument is not a simple integer value representing a valid blind I/O channel (0 through 9 inclusive).
- o the left argument is not a simple integer vector of valid field numbers or -1.

An *I/O ERR* is reported if:

- o the operation is not consistent with CP-6 requirements.

⎕*BRFLD* Function (Release Field)

Syntax:

```
IV ⎕BRFLD I
```

Parameters:

IV is a simple integer vector of field numbers.

I is a simple integer scalar representing a valid blind I/O channel.

Description:

The ⎕*BRFLD* system function is used to release (or deselect) a previously selected screen field. A field number of −1 is used to release all currently selected fields.

Examples:

In the following example, fields 5 and 6 are deselected:

```
5 6 ⎕BRFLD 1
```

Possible Errors:

A *RANK ERR* is reported if:

- the left argument is not a vector or scalar.

A *LENGTH ERR* is reported if:

- the right argument contains more than one item.

A *DOMAIN ERR* is reported if:

- the right argument is not a simple integer value representing a valid blind I/O channel (0 through 9 inclusive).
- the left argument is not a simple integer vector of valid field numbers or −1.

An *I/O ERR* is reported if:

- the operation is not consistent with CP-6 requirements.

⎕*BXFLD* Function (Expunge Field)

Syntax:

```
IV0 ⎕BXFLD IV1
```

Parameters:

*IV*0 is a simple integer vector of field numbers.

*IV*1 is a simple integer vector of length two, containing a valid blind I/O channel number and an erase level code value of 0, 1, 2, or 3.

Description:

The ⎕*BXFLD* system function is used to erase (or expunge) the specified fields. The second item of the right argument controls the extent of the erase. A value of 0 erases input fields, 1 erases input fields and protected fields, 2 erases input fields and constant fields, and 3 erases input fields, constant fields and protected fields. A field number of −1 is used to erase all currently selected fields.

Examples:

In the following example, fields 3, 5, and 6 are erased:

```
      3 5 6 ⎕BXFLD 1 1
```

Possible Errors:

A *RANK ERR* is reported if:

o the left argument is not a vector or scalar.

A *LENGTH ERR* is reported if:

o the right argument contains more than two items.

A *DOMAIN ERR* is reported if:

o the right argument is not a simple integer value representing a valid blind I/O channel (0 through 9 inclusive) followed by a simple integer value in the range 0 through 3 inclusive.

o the left argument is not a simple integer vector of valid field numbers or −1.

An *I/O ERR* is reported if:

o the operation is not consistent with CP-6 requirements.

Appendix A

CP-6 APL Parameters

This appendix defines the limits that apply to the CP-6 APL implementation and references to (or definitions of) the implementation-defined algorithms. This information may be useful in comparing CP-6 APL to other implementations and in determining whether an existing APL application can be run on CP-6 APL.

Arithmetic Limits

Largest positive number:	8.3798799562141231863E152
Largest negative number:	¯8.3798799562141231872E152
Largest counting numbers:	(2*60) (¯2*60)
Maximum exponent field width:	5
Integer tolerance value:	⎕CT
Digits in full print precision:	20

Array Limits

Maximum rank:	62
Maximum index:	34359738367
Maximum number of items:	34359738367

System Variables

The valid value range for the named system variables is:

Name	Minimum Value	Maximum Value	Domain
⎕CT	0	1E¯12	(0..1E¯12)
⎕IO	0	1	integer 0 or 1
⎕LX			character vector
⎕RL	1	34359738367	integers in range
⎕PP	1	20	Integers in range

Implementation Defined System Variables

Name	Minimum Value	Maximum Value	Domain
⎕*PW*	32	390	integers in range
⎕*PS*	¯1	1	integer; for first two items
	¯2*35	¯1+2*35	integer; for last two items

Defined Functions

Maximum number of lines:	65535
Maximum function definition prompt:	9999.999

APL Input and Output

Direct input prompt:	(6⍴' ')
Quad input prompt:	⎕:
Function definition prompt:	[n]
Quote quad output limit:	none

Miscellaneous Limits

Maximum number of shared variables:	16 + unlimited IDS/II shares
Identifier length limit:	79
Account identification:	8-item character vector
Workspace name length limit	31 characters
Shared variable general offer	20⍴' '

File System

Maximum number of simultaneous file ties:	31
File name length limit:	31 characters
File account name limit:	8 characters

Trigonometric and Hyperbolic Algorithms

	CP-6 LIBRARY	HART***
Cosine:	XPE_9DCOS	3346
Sine:	XPE_9DSIN	3346
Tangent:	XPE_9DTAN	4286
Inverse Cosine:	XPE_9DACOS	4904
Inverse Sine:	XPE_9DASIN	4904
Inverse Tangent:	XPE_9DATAN	4904
Hyperbolic Cosine:	XPE_9DCOSH	1067**
Hyperbolic Sine:	XPE_9DSINH	1986
Hyperbolic Tangent:	XPE_9DTANH	1067**
Inverse Hyperbolic Cosine:	XPE_9DACOSH	2705*
Inverse Hyperbolic Sine:	XPE_9DASINH	2705*
Inverse Hyperbolic Tangent:	XPE_9DATANH	2705*

*   The standard logarithmic formula equivalents are used to evaluate these functions.

** The standard exponential formula equivalents are used to evaluate these functions.

*** Algorithm number from Computer Approximations, Hart, J. F., et al, Robert C. Krieger Publishing Company, Huntington, N.Y., 1978.

Numeric Algorithms

	CP-6 LIBRARY	HART
Exponential:	XPE_9DEXP	1067
Gamma-function:	Chebyshev Approximations to the Gamma Function by Helmut Werner and Robert Collinge. Also Hart 5422 when overflow would otherwise occur.	
Modulo:	XPE_9DMOD	
Natural Logarithm:	XPE_9DLOG	2705
Power:	XPE_9PWRII	
	XPE_9PWRDI	
	XPE_9PWRDD	2705
Matrix Divide:	Golub/Businger algorithm with Powell/Reid strategies for scaling and row interchanging except for square matrices where Gaussian reduction with partial pivoting is used.	

Semi Numeric Algorithms

Pseudo-random Number Generation

```
      ∇ R←ROLL N;X
[1]     →(N≥2*35)/BIG
[2]     R←⎕IO+⌊N×ROLLEM÷2*31
[3]     →0
[4]    BIG:X←ROLLEM+(2*31)×⌊ROLLEM÷2
[5]     R←⎕IO+⌊N×X÷2*61
      ∇

      ∇ R←ROLLEM
[1]     QRL←R←⌊(¯1+2*31)|QRL×65539
      ∇
```

Deal Function

```
      ∇ R←A DEAL B;I
[1]     R←⍳0
[2]     R←⍳ B ◊ I←⎕IO
[3]     B←⌽?⌽B-⍳A
[4]   L:→(I≥A)/X
[5]     R[I+B[I],0]←R[I+0,B[I]]
[6]     → L,I←I+1
[7]   X:R←A↑R
[8]     →0
      ∇
```

CP-6 Dependent Algorithms

Input Conversion: XPN_7NS0TOI,
XPN_7NS0TOD

Output Conversion: XPN_7ITONSS

Current time: M$TIME *

Trace display:

```
fun[n] value
fun[n] ◊ value
fun[n] →n
```

Function display:

```
    ∇ fun
[1]   line
    ∇
```

Next Definition Line: 99999.999⍳*CURR*+*INCRLAST*

where *CURR* is the current line number and *INCRLAST* is 1 or the value associated with the last digit position entered that overrode a previously prompted line number.

Read Keyboard: CP-6 M$READ monitor service *

Plus: DPS8 instructions: DFAD, ADQ **

Minus: DPS8 instructions: DFSB, SBQ **

Times: DPS8 instructions: DFMP, MPY **

Divide: DPS8 instructions: DFDV, DFDI, DIV **

Time-Stamp: A seven item integer vector containing the current time as: year, month, day, hour, minute, second, and millisecond as returned by M$TIME. *

Pi-Times: The closest hardware approximation to pi (to 19 digits) times the argument value.

* See CP-6 Host Monitor Services Reference (CE74, CE75)
** See DPS8 Assembly Instructions (DH03)

Array Representation

Arrays in CP-6 APL occupy a minimum of 16 bytes of workspace, where a byte contains 9 bits of information. The total size of the array depends upon the rank and the total number of items in the array.

Character arrays are stored one character in each byte of memory providing 512 possible characters. Numeric arrays are represented in one of four different methods depending, upon the value being represented and the method used to generate the value. Logical arrays are used to represent arrays containing only the values 0 and 1 using one bit, with up to 9 values packed in each byte. Integer arrays are used to represent the integer values ¯34359738368 to 34359738367 using four bytes for each value. Index sequences are used to represent the result of the index generator function. This representation always occupies 24 bytes of workspace. Finally, floating point arrays are used to represent all other numeric values using 8 bytes of workspace for each value.

Nested arrays are used to represent heterogeneous arrays, and arrays with items which are themselves APL arrays. Each item of a nested array occupies 4 bytes of workspace where each item contains either a simple scalar value or a pointer to another APL array.

Consistent Extensions to the ISO APL Standard

CP-6 APL provides many extensions over the ISO and ANSI APL standards. Some of the extensions are minor (in that almost all APL implementations provide the same extension) while some provide capabilities not generally available in other APL implementations. The use of these extensions in CP-6 APL will make a program non-conformant with the APL Standard. The following is a summary of the extensions that are found in CP-6 APL.

Nested Arrays

CP-6 APL provides a nested array capability where any item of an array may contain another APL array (as a scalar item). The introduction of nested arrays has extended the domain of all scalar functions to nested arrays, of all structural mixed functions to nested arrays, of all operators to nested arrays and the monadic format function creates a display form for a nested array.

In addition to providing nested arrays, CP-6 APL arrays may also contain items of differing types (mixed character and numeric).

Additional Primitive Functions

CP-6 APL provides eight primitive functions which are not present in the APL standard. They are: ≡ (equivalence), monadic ↑ (first), ⊃ (disclose and pick), ⊂ (enclose), monadic ∊ (type), dyadic ⍋ (grade-up), dyadic ⍒ (grade-down), ⌶ (I-beam), and ⍑ (T-bar).

Extensions to Primitive Functions

CP-6 APL provides many extensions to the existing primitive functions in addition to those noted under the nested array datatype. Some of these extensions are common extensions to APL made in other implementations.

The dyadic ∧ (and) function has been extended to permit any numeric value as arguments (as opposed to only 0 and 1). This extension returns the Least-Common-Multiple of the two values.

The dyadic ∨ (or) function has been extended to permit any numeric value as arguments (as opposed to only 0 and 1). This extension returns the Greatest-Common-Divisor of the two values.

The monadic ⍋ (grade-up) and monadic ⍒ (grade-down) functions have been extended to sort character data and to sort arrays of any rank.

The left argument of the dyadic / (compress) function may contain positive integers less than the maximum index limit. This new function is known as replicate because it replicates the right argument values the number of times specified by the left argument values.

The dyadic , (join) function does not report a *DOMAIN ERR* when the types of the left and right argument are different. The nested array extensions permit arrays to contain both character and numeric items. If both arguments are empty, the result type is the prototype of the right argument.

The functions which access the system variable ⎕*CT* (comparison tolerance) will only attempt to access the value if the internal CP-6 APL datatype needs to have ⎕*CT* applied.

The dyadic ⍴ (reshape) function does not report a *DOMAIN ERR* if the right argument is empty. Instead, it fills the resulting array with prototype values (if they are needed).

The monadic ⍎ (execute) function always returns a value if the execution of the statement is successful. If the statement does not provide a result, execute returns an empty numeric vector. The line to execute may also contain system commands or function-definition-mode commands.

The monadic + (conjugate) function does not report a *DOMAIN ERR* if it is provided with an argument of type character. It returns the character value unchanged.

The []← (indexed assignment) function does not report a *DOMAIN ERR* when the type of the assigned value is not the same as the type of the name being assigned. The result is an array containing data of both types (numeric and character).

If the left argument of the \ (expand), / (compress and replicate) or ⍴ (reshape) functions is a singleton, it is treated as a scalar.

If either argument of the dyadic ? (deal) function is a singleton, it is treated as a scalar.

If the right argument of the monadic ⍳ function is a singleton, it is treated as a scalar.

Additional Primitive Operators

CP-6 APL provides an operator which is not present in the ISO or ANSI APL standards. It is the ¨ (each) operator.

Extensions to Primitive Operators

In addition to the extensions noted under the section on nested arrays, CP-6 APL contains an extension to the \ (scan) operator. A *DOMAIN ERR* is not reported if the type of the result of applying the function differs from the type of the argument.

All of the operators in CP-6 APL accept any function as an argument, not just a scalar function.

Additional System Functions

CP-6 APL provides many system functions in addition to those defined in the APL standard. These new system functions include:

⎕*AT* (set/query function attributes)
⎕*CVT* (convert datatypes)
⎕*ERS* (signal error)
⎕*EXG* (expunge globals)
⎕*FI* (fix input)
⎕*FMT* (format)
⎕*GRP* (group names)
⎕*HDR* (output heading)
⎕*IBEX* (IBEX expunge)
⎕*IBLET* (IBEX let)
⎕*IBNL* (IBEX names)
⎕*IDLOC* (identifier locations)
⎕*LOK* (function lock)
⎕*NCG* (nameclass globals)
⎕*RM* (room)
⎕*RMG* (room globals)
⎕*SC* (shared variable state change)
⎕*SCP* (string compare)
⎕*SITEID* (system siteid)
⎕*SITENAME* (system name)
⎕*SM* (set/query function sidetrack)
⎕*SRP* (substring replace)
⎕*SSR* (substring search and replace)
⎕*SSS* (substring search)
⎕*ST* (set/query function stop)
⎕*STEPCC* (step condition codes)
⎕*SVN* (shared variable user name)
⎕*SVS* (shared variable state)
⎕*SYSID* (user sysid)
⎕*TATTR* (terminal attributes)
⎕*TECHO* (terminal echo)
⎕*TIN* (terminal input prompt)
⎕*TIX* (text index)
⎕*TLEX* (text lexemes)
⎕*TR* (set/query function trace)
⎕*TTIME* (terminal timeout)
⎕*VERSION* (APL version)
⎕*VI* (verify input)
⎕*XL* (translate)

In addition to these system functions, this manual includes discussions of APL File I/O (section 12), APL I-D-S/II Interface (section 13), Packages (section 14), Graphics (section 15), and Blind I/O (section 16) which contains documentation for more CP-6 APL system functions.

Extensions to System Functions

The ⎕*DL*, ⎕*NL*, ⎕*STOP* and ⎕*TRACE* system functions treat a singleton right argument as a scalar.

Namelist arguments (identifier-row in the APL standard) to system functions may contain more than one name per row (separated by blanks or carriage returns). In this case, the result depends upon the number of identifiers found in the argument, not the number of rows in the namelist.

The dyadic ⎕*STOP* (set function stops) and dyadic ⎕*TRACE* (set function trace) system functions permit stopping and tracing line 0.

The ⎕*FX* system function permits its right argument to be a vector with carriage return characters separating lines. This function may also be used dyadically to create a function with specific execution properties.

Extensions to Defined Functions

Dyadic defined functions may be executed monadically.

The result name in the function header may be enclosed in brace brackets to turn off "output potential".

Additional System Variables

CP-6 APL provides the following system variables (which are not in the APL standard): ⎕*PW* (platen width), ⎕*PS* (positioning and spacing).

Extensions to System Variables

The value assigned to the system variables ⎕*IO* (index origin), ⎕*PP* (print precision) and ⎕*CT* (comparison tolerance) may be a singleton of any rank.

Additional System Commands

CP-6 APL contains many system commands in addition to those defined by the APL standard. (See section 8 for complete descriptions of these commands.)

The additional commands supplied in CP-6 APL are: *)CATCH*, *)CONTINUE*, *)DIGITS*, *)EDITOR*, *)END*, *)ERROR*, *)GO*, *)GROUP)GRP*, *)GRPS*, *)IBEX*, *)LINK*, *)NMS*, *)OBSERVE*, *)OFF*, *)OPR*, *)OPRN*, *)ORIGIN*, *)PCOPY*, *)QCOPY*, *)QLOAD*, *)QPCOPY*, *)QUIT*, *)REPORT*, *)RESET*, *)SALVAGE*, *)SEAL*, *)SET*, *)SIL*, *)SIV*, *)STEP*, *)TERMINAL*, and *)WIDTH*.

All of the system command names can be abbreviated to the first four characters.

Extensions to System Commands

A CP-6 APL workspace identifier may contain additional characters to those defined in the APL standard. The characters are: $, :, _, and -. The workspace identifier may be followed by a period, an 8 character account identifier and optionally followed by another period and an 8 character password.

The *)COPY* command copies groups (see section 8) and more than one name may be specified in the copy list.

The *)DROP* command with no workspace identifier drops the workspace created by the *)CONTINUE* command (see section 8).

The *)ERASE* command permits more than one name to be specified to be erased.

The *)FNS* and *)VARS* commands permit two names to be specified which are used to delimit the start and end of the list of names to display.

The *)LIB* command permits an account name to be specified.

The *)SI* and *)SINL* commands indicate when an execute or quad entry is found in the state indicator. These commands also permit the keyword options *ON*, *OFF* and *CLEAR* to be specified.

Miscellaneous Extensions

The quote quad output prompt may be formed by assignment to ⍞ multiple times. The value assigned to ⍞ is permitted to be any APL array. If an APL statement begins with the left pointing arrow (assignment arrow), default output that would have been generated by the statement is not displayed.

During ⎕ input, defined functions may be modified or created, and system commands may be issued.

When an identifier exceeds the identifier length limit, an error is not reported. Instead, the name is truncated to the limit.

Vector notation may be used to create nested arrays.

Vector assignment may be used to assign the values of a vector to a list of identifier names.

Selective assignment may be used to modify items of an array.

Appendix B

CP-6 APL Character Set

The appendix describes the atomic vector for CP-6 APL. The CP-6 APL atomic vector (⎕*AV*) is a character vector of length 512. The last 256 character positions of this vector are not assigned any meaning and are not permitted to appear in APL expressions.

In the following table each of the first 256 elements of ⎕*AV* are described in terms of their position, the ascii characters to enter on non-APL terminals, the APL character, the APL characters that form the overstrike and the name of the character. Some of these characters are unassigned and do not have a meaning.

APL characters which have no corresponding character in ASCII may be entered as mnemonics. Mnemonics are introduced by the $ character and are followed by 1, 2 or 3 characters which are mapped into the internal APL character during input processing. When the introducer is not followed by a defined mnemonic, the characters entered are passed through the APL input routine. This permits the $ character to be entered in normal input. A true dollar sign can be entered where it could be interpreted as a mnemonic by doubling it. That is, $$ is always a single dollar sign.

Table B-1. CP-6 APL Character Set

Index	ASCII	APL	Overstrike	Name	Index	ASCII	APL	Overstrike	Name
0	NUL				64	@	α		alpha
1	SOH				65	A	A		
2	STX				66	B	B		
3	ETX				67	C	C		
4	EOT				68	D	D		
5	ENQ				69	E	E		
6	ACK				70	F	F		
7	BEL				71	G	G		
8	BS				72	H	H		
9	HT				73	I	I		
10	LF				74	J	J		
11	VT				75	K	K		
12	FF				76	L	L		
13	CR				77	M	M		
14	SO				78	N	N		
15	SI				79	O	O		
16	DLE				80	P	P		
17	DC1				81	Q	Q		
18	DC2				82	R	R		
19	DC3				83	S	S		
20	DC4				84	T	T		
21	NAK				85	U	U		
22	SYN				86	V	V		
23	ETB				87	W	W		
24	CAN				88	X	X		
25	EM				89	Y	Y		
26	SUB				90	Z	Z		
27	ESC				91	[	[		left bracket
28	FS				92	\	\		back slash
29	GS				93	]	]		right bracket
30	RS				94	$TAK	↑		take
31	US				95	$_	_		underbar
32	BL			blank	96	$ENC	⊂		enclose
33	!	!	'.	bang	97	a	A̲	A_	
34	$"	¨		diaeresis	98	b	B̲	B_	
35	$NE	≠		not equal	99	c	C̲	C_	
36	$	$		dollars	100	d	D̲	D_	
37	$R	ρ		rho	101	e	E̲	E_	
38	$CAP	∩		cap	102	f	F̲	F_	
39	'	'		quote	103	g	G̲	G_	
40	(	(		left paren	104	h	H̲	H_	
41	)	)		right paren	105	i	I̲	I_	
42	*	⋆		star	106	j	J̲	J_	
43	+	+		plus	107	k	K̲	K_	
44	,	,		comma	108	l	L̲	L_	
45	$–	−		minus	109	m	M̲	M_	
46	.	.		dot	110	n	N̲	N_	
47	/	/		slash	111	o	O̲	O_	
48	0	0		zero	112	p	P̲	P_	
49	1	1		one	113	q	Q̲	Q_	
50	2	2		two	114	r	R̲	R_	
51	3	3		three	115	s	S̲	S_	
52	4	4		four	116	t	T̲	T_	
53	5	5		five	117	u	U̲	U_	
54	6	6		six	118	v	V̲	V_	
55	7	7		seven	119	w	W̲	W_	
56	8	8		eight	120	x	X̲	X_	
57	9	9		nine	121	y	Y̲	Y_	
58	:	:		colon	122	z	Z̲	Z_	
59	;	;		semicolon	123	{	{		left brace
60	<	<		less	124	|	|		stile
61	=	=		equal	125	}	}		right brace
62	>	>		greater	126	~	~		not
63	?	?		query	127	DEL			

Table B-1. CP-6 APL Character Set (cont.)

Index	ASCII	APL	Overstrike	Name	Index	ASCII	APL	Overstrike	Name
128					192	$MIN	⌊		floor
129					193	$E	∊		epsilon
130					194				
131					195	$DLT	∆		delta
132					196	$I	⍳		iota
133					197	#	×		times
134					198	%	÷		divide
135					199	$MAX	⌈		ceiling
136					200	$DRP	↓		drop
137					201				
138					202	$W	⍵		omega
139					203	$DSC	⊃		disclose
140					204	&	∧		and
141					205	"	∇		del
142					206	-	¯		over bar
143					207	$LE	≤		less equal
144					208	$GE	≥		greater equal
145					209	$OR	∨		or
146					210	$DMD	⋄		diamond
147					211	$LTK	⊢		left tack
148					212	$RTK	⊣		right tack
149					213	$Q	⎕		quad
150					214	$O	○		circle
151					215	$GO	→		right arrow
152					216	_	←		left arrow
153					217	$DCD	⊥		decode
154					218	$ECD	⊤		encode
155					219	$COM	⍝	∩∘	lamp
156					220	$EQV	≡	=_	equivalent
157					221	$NQV	≢	≠_	inequivalent
158					222	$FDI	⍸	⍳_	find index
159					223	$FND	⍷	∊_	find
160					224				
161	$SC	∘		jot	225				
162	$RD1	⌿	/-	slash bar	226				
163	$CUP	∪		cup	227				
164	$XP1	⍀	\-	backslash bar	228				
165	$GD	⍒	∇|	gradedown	229				
166	$LOK	⍫	∇~	lock	230				
167	$XEC	⍎	⊥∘	execute	231				
168	$FMT	⍕	⊤∘	format	232				
169	$QQ	⍞	'⎕	quote-quad	233				
170	$LOG	⍟	○⋆	log	234				
171	$RV1	⊖	○-	rotate first	235				
172	$MDV	⌹	÷⎕	matrix divide	236				
173	$TBR	⍑	⊤¯	t-bar	237				
174	$IB	⌶	⊤⊥	ibeam	238				
175	$UDL	⍙	∆_	delta underbar	239				
176	$TPS	⍉	○\	transpose	240				
177	$GU	⍋	∆|	grade up	241				
178	$NND	⍲	∧~	nand	242				
179	$NOR	⍱	∨~	nor	243				
180	$REV	⌽	○|	reverse	244				
181	$Q0	0⃞	0⎕	quad-zero	245				
182	$Q1	1⃞	1⎕	quad-one	246				
183	$Q2	2⃞	2⎕	quad-two	247				
184	$Q3	3⃞	3⎕	quad-three	248				
185	$Q4	4⃞	4⎕	quad-four	249				
186	$Q5	5⃞	5⎕	quad-five	250				
187	$Q6	6⃞	6⎕	quad-six	251				
188	$Q7	7⃞	7⎕	quad-seven	252				
189	$Q8	8⃞	8⎕	quad-eight	253				
190	$Q9	9⃞	9⎕	quad-nine	254				
191					255				

Appendix C
Error Messages

Table C-1 is an alphabetic listing of possible APL error messages. The first column contains the message and the second column contains explanatory details and recovery procedures where appropriate. The effects of error detection on APL processing are described in more detail in Section 10.

Table C-1. Error Messages

Message	Description
name NOT COPIED	The item has the same name as a pendent function in the active workspace.
name NOT ERASED	The item name in an)*ERASE* command was not erased because it was a pendent function.
name NOT FOUND	The item named in a)*COPY* command was not found (the item may have been a local variable).
ABORTED BY BRK OR CTRL-Y	An enqueue request has been aborted by the user (pertains to shared files).
BAD CHAR	A bad input character was detected. This is usually the result of a transmission error or the input of an illegal overstrike. In the case of nonstandard I/O devices, the message can also indicate the input of a character which is "illegal" for that device.
BAD COMMAND	An improper command construct was detected.
BAD FILE REF	A bad reference to an existing file name was made during a)*SAVE* command. This could occur, for example, if the workspace name specified in the)*SAVE* command referenced some existing workspace that was protected by a password. The)*SAVE* command should be respecified using a different workspace name. This message will also result on a)*CONTINUE* command if a passworded CONTINUE workspace already exists.

Table C-1. Error Messages (cont.)

Message	Description
BROKEN WORKSPACE	Damaged workspace reported during loading. It may be possible to copy specific items from the broken workspace with the)*SALVAGE* command.
DEADLOCK	An enqueue request has been made (pertaining to shared files) which, if honored, would create a deadlock stopping further activity of two or more users.
DEFN ERR	This message is output for any sort of error in function definition, such as misplaced del symbol (∇), improper syntax of header editing, or an attempt to edit a pendent function.
DOMAIN ERR	The indicated argument is of the wrong type or out of the proper range for the specific function or for the other argument. Examples are character data input for a numeric operation, or numbers input for a logical operation which do not reduce to 0 or 1. See the domain tables in Section 5 for examples of acceptable types of argument data for each APL function.
ENQ FULL	The CP-6 Enqueue tables are full.
FILE ACCESS ERR	This file I/O error often means a password is missing or is incorrect.
FILE DAMAGE	This file I/O error indicates some damage to the file contents was discovered, but not necessarily to every record or component in the file. Recovery is often possible by copying undamaged material to a new file, replacing damaged items.
FILE IN USE	The file named in a)*SAVE* command is currently in use, i.e., another user may be simultaneously executing a load of that file. Since this situation is a momentary timing conflict, the user should retry the command after a short wait. This type of timing conflict may also occur when using file I/O.

Table C-1. Error Messages (cont.)

Message	Description
FILE INDEX ERR	This file I/O error may mean that an index (record identifier, sometimes called a key) is incorrect, or an attempt has been made to read beyond the limits of a file.
FILE I/O ERR nnn-xxxxx-s	This is a general file I/O error message. It indicates errors detected by the monitor and corresponds to I/O error codes shown in the CP-6 Programmer Reference Manual (CE40).
FILE NAME ERR	This file I/O error may mean that a file identifier is improperly formatted, an attempt has been made to use a file that does not exist, or an attempt has been made to create a file that already exists.
FILE SPACE TOO LOW	Either the user's or the packset's file space limit has been reached. This can occur when workspaces are being saved or during file I/O operations. Recovery is usually possible; the user drops unneeded files from his account and retries the aborted statement.
FILE TBL FULL	This file I/O error means that the maximum permissible number of files have been "tied" (designated).
FILE TIE ERR	This file I/O error may mean either that the file has not yet been opened (designated as an input or output stream), or that the file being opened has already been opened, or that an attempt has been made to write into a file owned by another user.
FORMAT SYNTAX ERR	A syntax error was detected in the left argument of a ⎕*FMT* expression. See Section 9 for an explanation of correct syntax.
I/O ERR	This message indicates that an irrecoverable system I/O error occurred and an error exit has been made from APL. A system I/O error should be reported to the user's field representative along with the conditions under which it occurred (see also *SYS ERR*).

Table C-1. Error Messages (cont.)

Message	Description
I/O ERR nnn-xxxxx-s	If blind I/O was being used, this message indicates that the requested blind read or write could not be executed for some reason. The user may retry the I/O or otherwise continue operation. The error codes are described in the CP-6 Programmer Reference Manual (CE40).
INDEX ERR	The index subscript specified in an expression is out of the range of the particular array to which it is applied. For example, if A is a four-item vector, the expression *A*[6] would generate an *INDEX ERR* since the requested sixth item does not exist.
LENGTH ERR	The length(s) of the indicated argument(s) are not conformable or are incorrect for the function used. For example, the expression 9 7 8 + 5 3 results in a *LENGTH ERR* because the two vectors do not have the same number of items.
LINESCAN ERR	An obvious error in form (leading right bracket, misplaced colon line ending with a function, etc.) was detected in the scan of a line input for execution or function definition. No part of the line is executed. In function definition mode, the line is entered as part of the function and may then be replaced or edited.
NO RESULT	A defined function that generated no result was used in a context that requires a result.
NO SHARES	Another user logged onto the same account is using shared variables. Create a unique account identifier with the ⎕*SVN* system function. If the shared variable administrator is not available or has become unavailable, this message is reported and the APL user may no longer access the variables he may have previously shared.
NOT APL FILE	This file I/O error means that a component read failed because it did not have the structure required by APL.

Table C-1. Error Messages (cont.)

Message	Description
NOT GROUPED	The group name specified in a *)GROUP* or *)GRP* command references an existing item which is not a group. A different group name must be used.
NOT HELD	The ⎕*FDEQ* system function specified a file and resource that is not currently enqueued.
NOT SAVED, THIS WS IS name	If "name" = CLEAR WS, either there is nothing to save or the *)SAVE* command did not specify a name for the saved workspace. Otherwise, the *)SAVE* command named an existing saved workspace and the active workspace name is different. Change the active workspace name or drop the saved workspace.
OPEN QUOTE	The Execute function has been used on an argument that has an odd number of quotes before the end of the line (or first embedded carriage return).
RANK ERR	The rank of the indicated argument is incompatible with the function or with that of the other argument.
SEALED WS	An attempt was made to save a sealed workspace.
SI DAMAGE	A suspended function has been erased or replaced, and the state indicator has been modified to delete all references to it from its active list. This may occur in function definition, or upon execution of an *)ERASE* or *)COPY* command.
SI DAMAGE WILL RESULT: PROCEED?	This warning message is output in function definition when the header of an existing active function is changed. It indicates that references to this function in the state indicator list will be damaged if the header change is implemented. In order to avoid SI damage, the user may restore the header to the old form or change the function name in the header before closing the function. The user types YES in reply to the warning message.

Table C-1. Error Messages (cont.)

Message	Description
SING. MATRIX	The right argument of a matrix divide operation (⌹) is a singular matrix, i.e., it had no inverse.
SV QUOTA EXHAUSTED	The APL user may share up to 16 variables at any one time. This message is reported if the user attempts to share another variable.
SYNTAX ERR	Improper syntax was detected in the executed line. Examples of improper syntax are unbalanced parenthesis or an attempt to assign a value to a label.
SYSTEM ERR	An irrecoverable system error of indeterminate origin has occurred and an error exit has been made from APL. If APL is reaccessed, it should operate correctly unless the conditions which led to the *SYSTEM ERR* recur. Please report these problems to Honeywell.
TOO BIG	A)*COPY* command refers to more material than would fit in the current workspace; no items were copied.
TOO BIG TO LOAD	The workspace specified in a)*LOAD* command was saved by a user with larger memory allocation than the present user, and there is insufficient space for the workspace to be loaded (in some cases it cannot even be copied). See also the description of the)*COPY* command, Section 8.
TRUNCATED INPUT	The input line was too long.
UNDEFINED	The indicated symbol has not been assigned a value.
WS FULL	The active workspace is full. This may occur during execution, in function definition, or because of a)*GROUP* command. Depending on the particular situation, the user may choose to use an)*ERASE* command to erase unneeded objects from the workspace, clear the state indicator, or)*CLEAR* the entire workspace in order to free up space.

Table C-1. Error Messages (cont.)

Message	Description
WS NOT FOUND	The workspace file specified in a)*LOAD* or a)*COPY* command was not found.

Appendix D

CP-V Compatible Workspace Functions

CP-6 APL provides a set of intrinsic functions, *∆CR*, *∆WM*, and *∆TE* to aid in conversion from CP-V to CP-6. This appendix is provided for CP-V conversion purposes only. CP-6 APL provides more powerful system functions to perform these tasks.

Canonical Representation

The *∆CR* intrinsic function converts user functions to character form, creates user-defined functions, and locks existing functions.

Function to Text

R←1 *∆CR A*

If *A* is not a character vector representing a valid name in APL, *DOMAIN ERR* is reported.

If *A* contains a name which does not represent a user-defined function in the dynamic environment, *DEFN ERR* is reported.

If no error is indicated, *R* is a character vector consisting of lines of the defined function with embedded carriage returns as separators.

Text to Function

R←2 *∆CR LL*

If *LL* is not a linelist, *DOMAIN ERR* results.

DEFN ERR is reported if the 'header' line is not in the proper format for a function or if the function name has an active referent which is not a user function.

If no errors occur, a defined function, with the name specified by *LL*, is created.

R is a character vector indicating the name of the function created.

Locking Function

R←3 *∆CR NL*

NL must be a namelist. For each name in *NL*, if the current referent is a function, it is locked. If not, the name is included in *R*.

R is a character vector consisting of any names in *NL* which were not current function names.

Intrinsic To Text

R←4 Δ*CR A*

R is a character vector containing the name of the intrinsic mentioned in *A*, an assignment arrow, and the particular intrinsic definition statement that defined the named intrinsic.

Workspace Management

The workspace management function, Δ*WM*, is a dyadic intrinsic function providing a variety of operations described below.

Expunge, Local (Active)

R←1 Δ*WM NL*

NL must be a namelist. The active referents of names found in *NL* are erased. *R* is a namelist of any names for which referents were found but not erased.

Expunge, Global

R←2 Δ*WN NL*

Same as 1 Δ*WM NL* except that only global referents of names are erased.

List Workspace Named Items

R←3 Δ*WM I*

The value of *I* must be an integer from 1 to 6. *R* is a character vector with carriage returns separating the names. The entities named depend on *I*.

I	Category Listed
1	Labels.
2	Active variables.
3	Active functions.
4	Groups.
5	Global variables.
6	Global functions.

List Elements of a Group

R←4 Δ*WM A*

A must be a character vector containing one name. *R* is a character vector with names of the members of group *A*.

List Workspace Parameters

R←5 Δ*WM I*

The value of *I* must be an integer from 1 to 8. *R* depends on the value of *I*.

I	*R*
1	WSID as character vector.
2	State indicator as character vector with embedded line feeds.
3	Origin as integer.
4	Width as integer.
5	Digits as integer.
7	Symbol table size.
8	Number of symbols still available.

Identify Local Use of Names

R←6 Δ*WM NL*

The namelist *NL* is scanned for current use of the names. *R* is a numeric vector. Values are as indicated.

0	No current referent.
1	Logical variable.
2	Character variable.
3	Integer variable.
4	Real variable.
5	Index sequence.
7	Label.
8	User-defined function, niladic, no result.
9	User-defined function, niladic, with result.
10	User-defined function, monadic, no result.
11	User-defined function, monadic, with result.
12	User-defined function, dyadic, no result.
13	User-defined function, dyadic, with result.
14	Intrinsic function, dyadic.
15	Intrinsic function, monadic.
16	Intrinsic function, niladic.
17	Group.

Identify Global Use of Names

R←7 Δ*WN NL*

Similar to 6 Δ*WM* except that global use of names is indicated.

List Storage Requirements for Named Active Items

R←8 Δ*WM NL*

NL is a namelist. *R* is a numeric vector. Each item of *R* is the number of bytes of workspace occupied by the active referent.

List Storage Requirements for Named Global Items

R←9 Δ*WM NL*

NL is a namelist. *R* is a numeric vector. Each item of *R* is the number of bytes of workspace occupied by the global referent of the corresponding name.

Text Editing

The character editing function, Δ*TE*, provides five capabilities, described below, to facilitate the examination and modification of character variables in APL.

Text Index Function

R←1 Δ*TE L*

L is a 'list' with two items.

L←1 Δ*TE* (*TV*;*DV*)

TV may be any character vector.

DV is a character scalar or vector of 'delimiters'.

R is an N-by-2 numeric matrix. Each row contains the index and length of a string of non-delimiter characters in *TV*. The values of column 1 of *R* are ⎕*IO* dependent.

Substring Search

R←2 ΔTE L

L must be a list with 2, 3, or 4 items.

R←2 ΔTE (TV;SS)

TV may be any character vector.

SS may be any character scalar or vector not longer than TV.

L←2 ΔTE (TV;SS;FCOL)

FCOL may be any integer scalar value such that FCOL is less than or equal to the highest index value of TV. FCOL indicates the first column in TV at which search is to start or

L←2 ΔTE (TV;SS;FCOL;LCOL)

LCOL may be any integer scalar value less than or equal to the highest index value of TV and greater than or equal to FCOL. LCOL is the last column of TV involved in the search.

R is a numeric vector with the beginning indexes of non-overlapping occurrences of SS in TV, starting at position FCOL and ending at LCOL.

Substring Search and Replacement

R←3 ΔTE L

L must be a list of 3, 4, or 5 items.

L←3 ΔTE (TV;SS;RS)

TV may be a character scalar or vector.

SS may be a character scalar or vector not longer than TV.

RS may be any character scalar or vector.

R is a character vector formed by replaced occurrences of SS, in TV, by RS. Replacement is on a non-overlap basis. Or

L←3 ΔTE (TV;SS;RS;FCOL)

FCOL may be any integer scalar value such that FCOL is less than or equal to the highest index value of TV. FCOL may also be null.

L←3 ΔTE (TV;SS;RS;FCOL;LCOL)

LCOL may be any integer scalar value less than or equal to the highest value of TV and greater than or equal to FCOL.

Substring Replacement (Without Search)

R←4 ΔTE L

L is a list with 4 items.

L←4 ΔTE (TV;RS;FCOL;LCOL)

TV must be a non-empty character vector.

RS must be a character vector or scalar. It may be empty.

FCOL must be an integer scalar representing a valid index of TV.

LCOL must be an integer scalar representing a valid index of TV. LCOL must be greater than or equal to FCOL.

R is formed by replacing that portion of *TV* bounded by *FCOL* and *LCOL* by the string *RS*. If *RS* is empty, this constitutes deletion of a specified subset of *TV*.

String Comparison

R←5 Δ*TE L*

L must be a list with two items:

L←5 Δ*TE* (*A*;*B*)

A and *B* must be character vectors or character scalars.

R is a two-item numeric vector describing the comparison of *A* and *B*. Comparison is based on the ASCII collating sequence as modified to support the CP-6 APL character set.

The first item of *R* indicates which item of *L* should be first in left to right sorted order.

> 0 means the character vectors are identical. 1 means *A* should sort first. 2 means *B* should sort first.

The second item of *R* indicates the lowest position *I* at which *A*[*I*] and *B*[*I*] differ.

If *A* and *B* are identical, the second item of *R* is −1. Thus *R* is 0 ¯1.

If *B* is longer than *A* but *A*[*I*] = *B*[*I*] that is *B* differs from *A* only by being longer, then *A* is considered first in sorting order and *R* is 1 ¯1.

If *A* is longer than *B*, but each *B*[*I*] = *A*[*I*] then *R* is 2 −1.

T-bar Functions

The T-bar function, ⍑ (the encode character, ⊤, overstruck by the negative sign, ¯) is provided for certain system interfaces.

One use of T-bar is the character generator function. It converts integer data into corresponding character data, and thus allows the user to generate special characters, possibly unrecognized by APL. The integer n corresponds to the nth character in the table of APL Codes. This is equivalent to indexing ⎕*AV*.

To generate the nth character, the following form is used.

```
2⍑N
```

The left argument must be the scalar integer 2; this designates that the T-bar function is to be used for character generation. The right argument may have any shape, but its domain must be integer, with values between 0 and 511. The result has a shape identical to the right argument, but is character data.

File Input/Output

CP-6 APL provides more functionality than is available with these functions through the system functions discussed in section 12. The file intrinsic is:

```
A fname B
```

where

A the I/O operation number (ranging from 1 to 29).

B is the argument applicable to the I/O operation.

Opening and Creating Files

Following are the forms for the set of functions required to establish parameters prior to opening a stream to a file.

o Establishing "file number":

    1 fname *B*

where *B* is a positive integer specifying the file number to be used for subsequent file operations.

o Establishing file name:

    2 fname *B*

where *B* is a character vector specifying the file name for the currently set file number.

o Establishing or resetting account:

    2 fname *B*

where *B* is either zero or a character vector specifying the account for the currently set file number.

o Establishing or resetting password:

    4 fname *B*

where *B* is either zero or a character vector specifying the password for the currently set file number.

o Establishing file identification as a single primitive:

    21 fname fid

where fid is a character vector specifying a CP-6 file identifier in the same format permitted for system commands such as *)LOAD*.

o Assigning serial numbers for pack set utilization:

    20 fname *B*

where *B* is a character vector of up to 6 characters, or the numeric value 0.

o Opening stream in indicated mode:

    5 fname *B*

If *B* is an integer specifying the mode of DCB for the currently set file number, as follows:

1 indicates FUN=IN,DISP=NAMED,EXIST=OLDFILE.
2 indicates FUN=CREATE,DISP=NAMED,EXIST=NEWFILE.
4 indicates FUN=UPDATE,DISP=NAMED,EXIST=OLDFILE.
8 indicates FUN=CREATE,DISP=SCRATCH,EXIST=NEWFILE.
17 indicates FUN=CREATE,DISP=NAMED,EXIST=ERROR,TEST=YES.
20 indicates FUN=UPDATE,DISP=NAMED,EXIST=OLDFILE,SHARE=ALL.

Closing Files

- o Closing and saving the file for indicated file numbers:

```
6 fname B
```

where *B* is an integer specifying the file number.

- o Closing and releasing the file for indicated file number:

```
7 fname B
```

where the argument *B* is the same as above. (This form is used to delete files.)

Maintaining Key Range and Current Key Value

When files are created by APL or accessed in other than sequential mode, primitives are provided to find the key range of an existing file. When a file is opened in CREATE mode, values for the 'first component' and 'last component' are initialized to empty vectors and updated when the first record is written.

- o Return the value of a designated key for the currently set file number:

```
8 fname B
```

where *B* is 1, 2, or 3, specifying which key the value is to be returned for (the key returned will be that for the currently open file, if any, of the most recently referenced file number):

1 indicates that the value of the first key in the file.
2 indicates that the value of the current key is to be returned.
3 indicates that the value of the highest key is to be returned.

- o Setting the value of the current key for the currently set file number:

```
9 fname B
```

where *B* is an integer or character vector specifying the value for the current key.

Writing APL Records

- o Writing a record containing the value of an expression:

```
10 fname expression
```

The currently set key value and file number are used.

- o Writing a component:

```
11 fname expression
```

The record contains the time, date, and the user's account and name, and the expression value. The currently set key value and file number are used.

Writing Non-APL Records

Data records may be written that do not retain the APL internal attributes of 'shape' and other internal reference data.

```
22 fname B
```

where *B* is any APL expression. The data represented by *B* is written as a single record in ravel order. If *B* is a logical vector the length is rounded up to a multiple of 9 bits.

Reading APL Records

- Reading a data record:

  12 fname *B*

where *B* is an integer specifying the size of the data record in bytes. The data record is read using current key and file number.

- Reading a component datablock:

  13 fname *B*

where *B* is an integer specifying the key value.

- Reading a component user/time stamp:

  14 fname *B*

where *B* is an integer specifying the key value. The identification record is returned as a character vector with the following format:

DATE	bb	TIME	ACCT	UNAME
1 6	7 8	9 16	17 24	25 36

Reading Non-APL Records

  23 fname *B*

Reads a non-APL record using currently set key and file number. *B* is an integer specifying the record size in bytes. The result is a character vector.

Deleting Records Or Components

- Deleting a specified record:

  15 fname *B*

  16 fname *B*

where *B* is an integer specifying the key value. The current file number is used in deleting the record.

Sequential Access to Existing APL Files

  17 fname *B*

where *B* is an integer specifying the size of the record in bytes. Records are read sequentially, using the current file number. If an integer of zero is specified, the record is accessed but data is not read, regardless of actual record size.

  13 fname 0

This is similar to "13 fname *B*" except that it reads the next record. If it is not a component record, records are skipped until a component record is reached. At end of read, the current key is set to that of the last record read. If no component record is found, an error is reported.

  14 fname 0

This is similar to "14 fname *B*" except that it skips forward to next component record. The current key is updated. If no component record is found, an error is reported.

Sequential Access to Non-APL Files

24 fname *B*

where *B* is an integer specifying the size of the record in bytes. Records are read sequentially, using the current file number. Operation is analogous to 23 fname *B* except that the read is sequential rather than keyed.

Converting Data Types

Primitives 23 and 24, for reading non-APL records, create character vector results.

o Convert character vector to logical vector.

25 fname *B*

where *B* is a character vector. The result is a logic vector consisting of the actual data in *B*.

o Convert character vector to integer vector.

26 fname *B*

where *B* is a character vector. The length must be a multiple of 4. The result is an integer vector consisting of the actual data in *B*.

o Convert character vector to real vector.

27 fname *B*

where *B* is a character vector. The length must be a multiple of 8. The result is a numeric vector consisting of the actual data in *B*.

Controlling Access to Shared Files

The following features are provided to permit the user to lock out records of a file for purposes of reading without other intervening updates or completing an update without interference.

o Locking out a record.

28 fname *B*

B is a key value. Causes the designated record to be enqueued for exclusive use.

o Releasing a locked record.

29 fname *B*

B is a key value.

Listing File Names and Numbers

These operations may be used in functions designed to list file components by number, with or without contents of the records.

o File names in a specified account

18 fname *B*

where *B* is a character vector specifying a user account. Result is a character matrix. Each row has account in columns 1 through 8 and file name in columns 10 through 40.

o Names or numbers of currently open files

19 fname *B*

where *B* is an integer specifying the structure of the result as follows:

1. indicates a character matrix with names of currently open files, one file per row.

2. indicates a numeric vector with the currently open file numbers.

Appendix E

Honeywell CP-6 APL Summary

Scalar Primitive Functions

All scalar functions are applied item-by-item on all operands at all levels of nesting. A scalar or single item array may be used as an argument of a scalar dyadic function and its value is applied to all items of the other argument.

Table E-1. Scalar Monadic Functions

Form	Description
+*Y*	Conjugate of *Y* (*Y*)
-*Y*	Negate *Y* (0-*Y*)
×*Y*	Sign of *Y* (¯1, 0, 1)
÷*Y*	Reciprocal of *Y* (1÷*Y*)
***Y**	e to the *Y*'th power
⌈*Y*	Ceiling of *Y* (round up)
⌊*Y*	Floor of *Y* (round down)
|*Y*	Absolute value of *Y*
⍟*Y*	Natural logarithm of *Y*
!*Y*	Factorial of *Y* (Gamma of *Y*+1)
○*Y*	Pi times *Y*

Table E-2. Scalar Dyadic Functions

Form	Description
X+*Y*	Add *Y* to *X*
X-*Y*	Subtract *Y* from *X*
X×*Y*	Multiply *Y* by *X*
X÷*Y*	Divide *Y* into *X*
*X***Y*	*X* raised to the power *Y*
X⌈*Y*	Maximum of *X* and *Y*
X⌊*Y*	Minimum of *X* and *Y*
X|*Y*	*X* residue of *Y* (remainder of *Y*÷*X*)
X∧*Y*	Least common multiple of *X* and *Y* (and)
X∨*Y*	Greatest common divisor of *X* and *Y* (or)
X!*Y*	Binomial coefficient. Number of combinations of *Y* things taken *X* at a time
X⍟*Y*	Base *X* log of *Y*
X○*Y*	Circular functions: 0○*Y* (1-*Y**2)*0.5 1○*Y* sine *Y* ¯1○*Y* arcsin *Y* 2○*Y* cosine *Y* ¯2○*Y* arccos *Y* 3○*Y* tangent *Y* ¯3○*Y* arctan *Y* 4○*Y* (1+*Y**2)*0.5 ¯4○*Y* *Y*×(1-*Y**¯2)*0.5 5○*Y* sinh *Y* ¯5○*Y* arcsinh *Y* 6○*Y* cosh *Y* ¯6○*Y* arccosh *Y* 7○*Y* tanh *Y* ¯7○*Y* arctanh *Y*

The relational and logical functions obey the rules of scalar conformability and return 0 if the condition is false, and 1 if true.

Table E-3. Relational and Logical Functions

Form	Description
X<Y	X less than Y
X≤Y	X less than or equal to Y
X>Y	X greater than Y
X≥Y	X greater than or equal to Y
X=Y	X equal to Y
X≠Y	X not equal to Y
	The following functions operate on arguments which are 0 or 1.
X∧Y	X and Y (1 if both X and Y are 1)
X∨Y	X or Y (1 if either X or Y is 1).
X⍲Y	X nand Y (not both X and Y)
X⍱Y	X nor Y (neither X nor Y)
~Y	not Y

Mixed Functions

Table E-4. Mixed Functions

Form	Description
XρY	Reshape Y to dimensions X.
ρY	Shape of Y.
XιY	Index of first occurrence of Y within X.
ιY	First Y consecutive integers from index origin.
X∊Y	1 if X occurs in Y, otherwise 0.
∊Y	Type of Y.
X≡Y	1 if X and Y are identical, otherwise 0.
≡Y	Maximum nesting depth of Y.
X⊤Y	Representation of Y in number system X.
X⊥Y	Value of Y in number system X.
X?Y	X integers selected randomly without repetition from ιY (deal).
?Y	A random integer from ιY
X⌽Y	Y rotated along last dimension by X.
X⊖Y	Y rotated along first dimension by X.
X⌽[N]Y	Y rotated along the N'th dimension by X.
⌽Y	Y reversed along last dimension.
⊖Y	Y reversed along first dimension.
⌽[N]Y	Y reversed along the N'th dimension.
X⍉Y	Transpose of Y by coordinates in X.
⍉Y	Transpose of Y (by reversing all coordinates).
X,Y	X joined to Y along the last coordinate.
X,[N]Y	If N is an integer, X is joined to Y along the N'th coordinate of X; otherwise, X and Y are joined (laminated) along the new ⌈N coordinate.
,Y	Ravel of Y (make Y a vector).
X↑Y	Take the first X items from Y (X>0) or take the last (|X) items from Y (X<0).
↑Y	Disclose the first item from Y.
X↓Y	Drop the first X items from Y (X>0) or drop the last (|X) items from Y (X<0).
X⍋Y	The indices of Y select items (or rows) of Y in increasing order of magnitude using the collating sequence X.
⍋Y	The indices of Y select items of Y in increasing order of magnitude.
X⍒Y	The indices of Y select items (or rows) of Y in decreasing order of magnitude using the collating sequence X.
⍒Y	The indices of Y which select items of Y in decreasing order of magnitude.
⊂Y	Enclose of array Y (make a nested scalar).
⊂[N]Y	Enclose along selected axes of Y.
⊃Y	Disclose Y, by decreasing depth and increasing rank.
⊃[N]Y	Disclose Y, inserting new axes at N.
X⊃Y	Select item from array Y at depth (ρX).
⌹Y	Inverse of matrix Y.
X⌹Y	Matrix division (least squares fit).
X⍕Y	Format Y according to specifications in X.
⍕Y	Format of Y.
⍎Y	Evaluate APL expression contained in Y.

Primitive Operators

In the following examples, f and g stand for any dyadic function and h stands for a monadic function.

Table E-5. Operators

Form	Description
f/*Y*	reduction along the last dimension of *Y*
f/[*N*]*Y*	reduction along the *N*'th dimension of *Y*
f⌿*Y*	reduction along the first dimension of *Y*
f\Y	scan along the last dimension of *Y*
f\\[*N*]*Y*	scan along the *N*'th dimension of *Y*
f⍀*Y*	scan along the first dimension of *Y*
X/*Y*	replication along the last dimension of *Y*
X/[*N*]*Y*	replication along the *N*'th dimension of *Y*
X⌿*Y*	replication along the first dimension of *Y*
X\Y	expansion along the last dimension of *Y*
X\\[*N*]*Y*	expansion along the *N*'th dimension of *Y*
X⍀*Y*	expansion along the first dimension of *Y*
X f.g *Y*	inner product of *X* and *Y*
X ∘.g *Y*	outer product of *X* and *Y*
X f¨ *Y*	apply function f to each item of *X* and *Y*
h¨ *Y*	apply function h to each item of *Y*

System Variables

Table E-6. System Variables

Name	Description
⎕*AV*	Atomic vector. The full CP-6 APL character set.
⎕*CT*	Comparison tolerance. Used in numeric comparisons.
⎕*IO*	Index origin. Used in indexing, ⍳, ⍋.
⎕*LC*	Line counter. Vector of lines in execution.
⎕*LX*	Latent expression. Executed after)*LOAD*.
⎕*PP*	Print precision. Maximum digits in numeric output.
⎕*PS*	Positioning spacing. Control nested array display.
⎕*PW*	Print width. Maximum width of output lines.
⎕*RL*	Random link. Seed for random number generator.
⎕*SA*	Stop action. Control entry into direct input mode.
⎕*SP*	Session parameter. Variable saved across)*LOAD*'s.
⎕*TS*	Time stamp. Year,month,day,hour,min,sec,millisec.
⎕*TT*	Terminal type.
⎕*UL*	User load. Number of users logged onto system.
⎕*WA*	Workspace available. Measured in bytes.

Special Symbols

Table E-7. Special Symbols

Form	Description
( )	Parentheses. Expressions may be of any complexity and are evaluated from right to left except as indicated by parentheses.
A[*X*]	Indexing. Returns an array of shape (⍴*X*).
A[*X*]←*Y*	Indexed assignment. The elements of *A* selected by the indices *X* are replaced by *Y*.
(*A* f *B*)←*Y*	Selective assignment. The elements of *B* selected by the expression (*A* f *B*) are replaced by *Y*. The function f may be one of the following dyadic functions: ⍉, ↑, ↓, ⌽, ⊖, ⍴, /, or ⊃. The function f may also be one of the following monadic functions: ⌽, ⊖, ⍉ or , (ravel).
(*A* *B*)←*Y*	Vector assignment. *A* is assigned the first value in the vector *Y* and *B* is assigned the second.
→*X*	Branch. If *X* is an empty vector, execution continues. If the first item of *X* is 0 or beyond the range of statement numbers, execution of the function terminates.
→	Terminates execution of function and related pendent functions.
⎕←*X*	Quad output. Prints the value of *X*.
⍞←*X*	Bare output. Prints *X* without terminating carriage return.
X←⎕	Quad input. The input expression is evaluated and assigned to *X*.
'*XYZ*'	Character vector of 3 elements: *XYZ*.
⍝	Lamp. Characters to the right of this symbol are treated as commentary.
∇	Del. Enter or exit function definition mode
⍫	Lock function.
X:	Label. *X* is a line label within a function.
;	Semicolon. Index separator.
⋄	Diamond. Statement separator.

Function Definition

A ∇ preceding the name of a defined function is used to enter definition mode. In definition mode, entries are held and saved in a function body for later execution. Each entry in definition mode is preceded by a prompt containing a line number in brackets.

Table E-8. Function Header Syntax

Valence	No Result	Explicit Result
Niladic	∇ fname	∇r← fname
Monadic	∇ fname b	∇r← fname b
Dyadic	∇a fname b	∇r←a fname b

Table E-9. Directive Summary

Entry	Description
[⎕]	Display the entire function.
[n⎕]	Display line n.
[n–m⎕]	Display lines n through line m.
[n⎕p]	Edit line n.
[n–m;/st/]	Display lines containing 'st' in lines n–m.
[n–m;/st/S/rt/]	Change string 'st' to 'rt' in lines n–m.
[∆n]	Delete line n.
[∆n m]	Delete lines n and m.
[∆n–m]	Delete lines n through line m.
[∆n–m;/st/]	Delete lines n–m which contain 'st'.
[/ST/]	Find next occurrence of string 'ST'.
[\ST\]	Find previous occurrence of string 'ST'.

Defined Function Controls

Table E-10. Defined Function Controls

Name	Description
R←I ⎕AT F	Return function attributes (1=valence, 2=create time,3=properties,4=creator).
M← ⎕CR F	Return function's canonical representation.
N← ⎕FX M	Fix canonical representation, return name.
N←A ⎕FX M	Like ⎕*FX* but also set execution attributes. *A* is a 4–item logical vector controlling attributes: displayable, suspendable, interruptable, errorable.
E← ⎕SM F	Return function's sidetrack matrix.
E←E ⎕SM F	Set function's sidetrack matrix.
V← ⎕STOP F	Return function's stop vector.
V←V ⎕STOP F	Set function's stop vector.
V← ⎕TRACE F	Return function's trace vector.
V←V ⎕TRACE F	Set function's trace vector.

Sidetracking on Errors and Interrupts

The ⎕*SM* system function is used to set and obtain the current sidetrack settings for a defined function. The optional left argument is a sidetrack matrix of shape (*N*,2) where each row contains a line number in the first column to indicate where execution will resume when the error number in the second column occurs. The right argument is a namelist containing the name of the defined function whose sidetrack matrix is to be set. If the optional left argument is not present, the result is the sidetrack matrix of the function named.

Table E-11. Error Numbers

Num	Message	Num	Message
0	all errors	46	*TOO BIG*
1	*WS FULL*	48	name *NOT COPIED*
2	*SYNTAX ERR*	49	name *NOT FOUND*
3	*UNDEFINED*	50	name *NOT ERASED*
4	*DOMAIN ERR*	51	*NOT GROUPED*
5	*RANK ERR*	52	*SEALED WS*
6	*LENGTH ERR*	53	*OLD WS, MUST EXPORT*
7	*INDEX ERR*	55	*NOT HELD*
8	*NO RESULT*	56	*ALREADY HELD*
10	*IMPLICIT ERR*	57	*NO SHARES*
11	*LIMIT ERR*	59	*HOLD ABORTED*
15	*SINGULAR MATRIX*	61	*HOLD DEADLOCK*
16	*FORMAT SYNTAX ERR*	62	*ENQ FULL*
20	*BAD CHAR*	68	*SV QUTA EXHAUSTED*
21	*LINESCAN ERR*	70	*FILE SPACE TOO LOW*
22	*TRUNCATED INPUT*	71	*FILE I/O ERROR* fcg–Mxxxx–s
23	*OPEN QUOTE*	72	*FILE DAMAGE*
30	*I/O ERR* fcg–Mxxxx–s	73	*FILE NAME ERR*
35	*DEFN ERR*	74	*NOT APL FILE*
36	*SI DAMAGE*	75	*FILE TBL FULL*
40	*BAD COMMAND*	76	*FILE ACCESS ERR*
41	*NOT SAVED, THIS WS IS*	77	*FILE TIE ERR*
42	*FILE IN USE*	78	*PACKSET NOT MOUNTED*
43	*BAD FILE REF*	79	*FILE INDEX ERR*
44	*WS NOT FOUND*	80	*PACKAGE TOO BIG*
45	*TOO BIG TO LOAD*	100	*INTERRUPT*

Error Control Functions

Table E-12. Error Control Functions

Name	Description
T← ⎕*ERF*	Name of function involved in recent error.
T← ⎕*ERH*	Description of most recent I/O error.
T← ⎕*ERL*	Line executing most recent error.
T← ⎕*ERM*	Error message for most recent error.
W← ⎕*ERN*	Error number and line number of error.
I← ⎕*ERP*	Index in ⎕*ERL* of error position.
T ⎕*ERS I*	Signal error number *I* with error message *T*.
T← ⎕*ERX*	Monitor code associated with I/O error.

CP-6 APL System Functions

Table E-13. CP-6 APL System Functions

Name	Description
I← ⎕*CPU*	CPU time used measured in milliseconds.
R←*W* ⎕*CVT R*	Convert data *R* into type *W*[1] using *W*[2] bits per item.
I← ⎕*DL I*	Delay execution for at least *I* seconds.
V← ⎕*EX N*	Erase objects named in *N*.
V← ⎕*EXG N*	Erase global objects named in *N*.
V← ⎕*FI T*	Convert character representation to number.
N← ⎕*GRP N*	Return namelist of group members.
⎕*IBEX T*	Erase IBEX variable named in *T*.
T← ⎕*IBLET N*	Return value of IBEX variable named in *N*.
T ⎕*IBLET N*	Assign value *T* to IBEX variable named in *N*.
R← ⎕*IBNL*	Return names of current IBEX variables.
R← ⎕*IDLOC N*	Return name correspondence of each name in *N* at each level of the state indicator.
I← ⎕*LGT*	APL invocation time in milliseconds.
I← ⎕*LOK N*	Lock functions named in *N*.
K← ⎕*NC N*	Return name class of names in *N*.
K← ⎕*NCG N*	Return global name class of names in *N*.
N← ⎕*NL K*	Return names of objects of class *K*.
N←*T* ⎕*NL K*	Like monadic ⎕*NL* but includes only names beginning with a letter in *T*.
I← ⎕*ONL*	Session mode: 0=batch,1=online.
I← ⎕*OVH*	Processor overhead time in milliseconds.
V← ⎕*RM N*	Size in bytes of objects in *N*.
V← ⎕*RMG N*	Size in bytes of global objects in *N*.
I← ⎕*SCT*	Milliseconds elapsed since APL was invoked.
T← ⎕*SI*	Text vector containing result of)*SI*.
T← ⎕*SITEID*	CP-6 site identifier.
T← ⎕*SITENAME*	Text vector containing CP-6 site name.
⎕*STEPCC I*	Sets value to use as step condition codes.
T← ⎕*SYSID*	Text vector containing current CP-6 sysid.
T← ⎕*UA*	8 item character vector of current account.
I← ⎕*UL*	Number of CP-6 system users.
T← ⎕*VERSION*	Version of the CP-6 APL processor.
V← ⎕*VI T*	Indicate legal representations of numbers.
T← ⎕*WSID*	Text vector of the current workspace name.

Shared Variable Functions

Table E-14. Shared Variable System Functions

Name	Description
I← ⎕*SC*	Wait for a shared variable event.
C← ⎕*SVC N*	Obtain controls on shared variables in *N*.
C←*C* ⎕*SVC N*	Set controls on shared variables in *N*.
I← ⎕*SVN T*	Set current process identification to *T*.
V← ⎕*SVO N*	Obtain degree of coupling for names in *N*.
V←*P* ⎕*SVO N*	Offer names in *N* to share with process *P*.
N← ⎕*SVQ P*	Obtain names of shares not accepted.
V← ⎕*SVR N*	Retract names in *N* from sharing.
R← ⎕*SVS N*	Shared variable states of names in *N*.

File I/O

File I/O functions may be used to access all CP-6 files provided access permission have been granted.

Table E-15. File I/O Example Names

Name	Description
A	CP-6 account or a fid with a ? within the filename.
F	CP-6 fid (name.account.pass).
I	Scalar integer dependent upon function.
K	Integer or character vector record identifier (key).
L	For INDEXed files: Numeric key matrix of shape (*N*,3). Column 1 is the character index of the key start, column 2 is the key length and column 3 is 0 if duplicate keys are permitted.
R	Array dependent upon function.
S	Integer scalar file I/O stream number.
T	Integer record type 1=component,2=datablock,3=raw.
X	Encryption seed (integer or 4-item character vector).
Z	Shape (*N*,17) character file access matrix. Columns 1-8 contain accounts, 9 must be blank and 10-17 contain Y or N for access control: READ, DELR, WNEW, UPDATE, DELF, NOLIST, REATTR and EXEC.

Table E-16. File Functions

Name	Description
R ⎕*FAPPEND S X T*	Add record or component to end of file.
K←*R* ⎕*FAPPENDR S X T*	Same as ⎕*FAPPEND* but returns key.
⎕*FCLEAR*	Close all open files.
⎕*FCLOSE S*	Close streams in *S*.
⎕*FCRPT S X*	Set default encryption seed.
⎕*FDEQ S K*	Release hold on file.
⎕*FDROP S K*	Delete record from file.
⎕*FENQ S K*	Hold a record.
⎕*FERASE S*	Close streams and delete files in *S*.
M← ⎕*FFLDS Y*	Return record field matrix.
F← ⎕*FID S*	Return CP-6 file identifier for stream *S*.
⎕*FKEYINT S I*	Set increment for ⎕*FAPPEND* to keyed file.
L← ⎕*FKEYS S*	Return matrix describing keys.
M← ⎕*FLIB A*	Return names of files in account *A*.
M←*M* ⎕*FLIB A*	Return fids of type *M* in account *A*.
T← ⎕*FMA*	Current file management account.
M← ⎕*FNAMS*	Names of open files.
R← ⎕*FNUMS*	Numbers of open files.
F ⎕*FOPEN S*	Open file I/O stream.
Z← ⎕*FRDAC S*	Return access control matrix for file *S*.
R← ⎕*FRDCI S K X*	Return component information.
R← ⎕*FREAD S K X T*	Read a record.
K← ⎕*FRKEY S I*	Return key (1=first,2=current,3=last).
I← ⎕*FSIZE S*	Return size of file *S* in bytes.
Z ⎕*FSTAC S*	Set file access matrix.
R ⎕*FWRITE S K X T*	Write a record or component.

Table E-17. File I/O Open Options

	Option	Description
*	*IN*	Open file for reading.
	CREATE	Create a new file.
	UPDATE	Update an existing file.
*	*NEWFILE*	For create, always create a new file.
	OLDFILE	For create, if file already exists use it.
	ERROR	For create, if file exists report error.
*	*NONE*	If UPDATE, open file for exclusive use, if IN, open shared with other IN,NONE users.
	ALL	Open file shared with multiple updaters.
	SHAREIN	Open file shared with multiple readers.
*	*NAMED*	Create a permanent file.
	SCRATCH	Create a scratch file.
*	*KEYED*	For create, specify file organization.
	CONSEC	For create, specify file organization.
	RANDOM	For create, specify file organization.
	UR	For create, specify file organization.
	RELATIVE	For create, specify file organization.
	INDEXED	For create, specify file organization.
	CG	For create, specify file organization.
	IREL	For create, specify file organization.
*	*DIRECT*	
	SEQUEN	
	CTG	Add file name to directory during open.
	LOAD	Build record indices on ⎕*FCLOSE*.
	COMP	Request record compression.
	REASSIGN	Use IBEX !SET for additional options.

* indicates default.

Text Editing Functions

The right argument to these functions is a vector of text vectors.

Table E-18. Text Editing Functions

Name	Description
I←⎕*SCP T T2*	Compare strings *T* and *T2*.
T←⎕*SRP T TR C1 C2*	Replaces positions *C1* through *C2* of string *T* with string *TR*.
T←⎕*SSR T TS TR*	Replaces occurrences of string *TS* with string *TR* in string *T*.
V←⎕*SSS T TS*	Returns indices of string *TS* in string *T*.
M←⎕*TIX T SDV TDV DDV*	Return matrix locating text lexemes.
V←⎕*TLEX T SDV TDV DDV*	Return text lexemes.

I-D-S/II Functions

Table E-19. I-D-S/II System Functions

Name	Description
R←⎕DBACCEPT D	Return db-key or area name.
I←⎕DBBUFFERS I	Set number of buffers to use.
I←⎕DBCHECK I	Indicate a checkpoint.
R←⎕DBCONNECT D	Connect a record to a set.
⎕DBCRPT I	Indicate an area encryption seed.
R←⎕DBDISCONNECT D	Remove record from a set.
R←⎕DBERASE D	Delete a record.
R←⎕DBFIND D	Locate a record.
R←⎕DBFINISH D	Close area(s).
R←⎕DBFROM D	Obtain data from workarea.
R←⎕DBGET D	Read a record from database.
R←⎕DBIF D	Test for set membership.
V←⎕DBMEMBER D	Return member indices of set or record.
R←⎕DBMODIFY D	Re-write a record.
T←⎕DBMSG	Return latest I-D-S/II error message.
M←⎕DBNAMES	Names of all objects in current subschema.
V←⎕DBOWNER D	Return owner indices of set or record.
R←⎕DBPRIVACY D	Specify privacy locks.
R←⎕DBREADY D	Open an area.
R←⎕DBRETAIN D	Indicate currencies to be retained.
⎕DBROLL	Roll back updates to last checkpoint.
R←⎕DBSTATS	Report statistics.
R←⎕DBSTATSOFF	Turn statistics gathering off.
R←⎕DBSTATSON	Turn statistics gathering on.
R←⎕DBSTORE D	Write a new record.
T←⎕DBSUB T	Indicate subschema name to use.
R←⎕DBTO D	Move data into work area.
R←⎕DBTRACEON I	Initiate I-D-S/II trace.
R←⎕DBTRACEOFF I	Terminate I-D-S/II trace.
R←⎕DBTYPES	Indicate types of objects in subschema.
N←⎕DBUSE V	Namelist of use procedures for errors in *V*.
V←N ⎕DBUSE V	Set use procedures associated with errors.

Terminal Control System Functions

Table E-20. Terminal System Functions

Name	Description
V← ⎕TATTR	Return terminal attributes.
⎕TECHO I	Control terminal echo: 0=off,1=on.
⎕TIN T	Set input re-read line.
R←I ⎕TSQZ T	Translate to or from mnemonics.
⎕TTIME I	Set timeout on terminal reads.
R← ⎕TWINDOW	Return terminal window locations and sizes.

Table E-21. ⎕*FMT* Format Controls

Form	Description
r*A*w	Blank insertion
r*E*w.s	Exponential notation (scientific format)
r q*F*w.d	Fixed decimal notation
r q*G*<text>	Picture format
r q*I*w	Integer
*T*w	Column tabbing
r*X*w	Blank insertion
r⍞text⍞	Text insertion
r⎕text⎕	Text insertion
r¨text¨	Text insertion
r<text>	Text insertion
Where:	
r	optional replication count
w	total field width
s	digits in *E* format
d	digits to include after the decimal point
c	column at which the next field will start
q	optional qualifier
Optional qualifiers	
B	field is blank if it is exactly zero
C	insert comma between triads of digits
L	formatted field is left justified
M<text>	prefixes negative fields with text
N<text>	postfixes negative fields with text
P<text>	prefixes positive fields with text
Q<text>	postfixes positive fields with text
R<text>	sets background for formatted value
S<text>	substitutions for '9Z*0_,.'

Table E-22. Report Formatting Functions

Name	Description
M←*Q* ⎕*FMT L*	Format *L* according to specification *Q*.
⎕*HDR T*	Set heading for output pages to *T*.
I← ⎕*NLS*	Return lines remaining on output page.
⎕*PGE*	Begin future output on a new page.
⎕*VFC B*	Set VFC character for next output line.
T←*X* ⎕*XL T*	Translate characters in *T* using vector *X*.

Blind I/O Functions

Blind I/O is provided to access CP-6 devices using the special variables ⎕0 through ⎕9. The APL system command)*SET* is used to direct blind I/O to a particular device.

Table E-23. Blind I/O Example Variable Names

Name	Description
I	Integer scalar.
J	Optional integer scalar whose presence indicates a set mode operation and whose absence causes the current setting to be returned.
K	Scalar integer or character vector key.
L	Matrix of shape (*N*,*M*) where 4≤*M*≤7 and the following column meanings are assigned: 1-field number, 2-line, 3-column, 4-length, 5-attributes, 6-input attributes, 7-initial value.
S	Blind I/O stream number (0 through 9).
T	Character vector.
V	Vector of form field numbers.

Table E-24. Blind I/O Functions

Name	Description
I← ⎕*BBIN S J*	Set or query binary I/O mode.
⎕*BCLOSE S I*	Close blind I/O stream *S*.
L ⎕*BFLD S*	Define forms mode fields.
K← ⎕*BKEY S*	Key of current file position.
K← ⎕*BKR S*	Key returned by last operation.
V← ⎕*BLINES S*	Return lines per page and lines remaining.
L ⎕*BMFLD S*	Modify forms mode fields.
⎕*BPFILE S I*	Position to beginning or end of file.
⎕*BPRECORD S I*	Forward or backward space in file.
⎕*BREW S*	Rewind file or tape.
⎕*BREWRITE S J*	Set rewrite option.
V ⎕*BRFLD S*	Release forms mode fields.
⎕*BRR S J*	Set re-read option.
I← ⎕*BRS S*	Size of record last read (F$DCB.ARS#).
⎕*BSEED S I*	Set encryption seed.
V ⎕*BSFLD S*	Select forms mode fields.
I← ⎕*BSIZE S J*	Set read size.
I← ⎕*BTRANS S J*	Set or query transparency.
I← ⎕*BVFC S J*	Set or query VFC.
V ⎕*BXFLD S*	Expunge forms mode fields.
T← ⎕*UNSET S*	Return character vector containing CP-6 fid and)*SET* options.

Table E-25. CP-6 APL Character Set

Index	APL	ASCII	Index	APL	ASCII	Index	APL	ASCII	Index	APL	ASCII
0		NUL	64	⍺	@	128			192	⌊	$MIN
1		SOH	65	A	A	129			193	∊	$E
2		STX	66	B	B	130			194		
3		ETX	67	C	C	131			195	∆	$DLT
4		EOT	68	D	D	132			196	⍳	$I
5		ENQ	69	E	E	133			197	×	#
6		ACK	70	F	F	134			198	÷	%
7		BEL	71	G	G	135			199	⌈	$MAX
8		BS	72	H	H	136			200	↓	$DRP
9		HT	73	I	I	137			201		
10		LF	74	J	J	138			202	⍵	$W
11		VT	75	K	K	139			203	⊃	$DSC
12		FF	76	L	L	140			204	∧	&
13		CR	77	M	M	141			205	∇	"
14		SO	78	N	N	142			206	¯	—
15		SI	79	O	O	143			207	≤	$LE
16		DLE	80	P	P	144			208	≥	$GE
17		DC1	81	Q	Q	145			209	∨	$OR
18		DC2	82	R	R	146			210	⋄	$DMD
19		DC3	83	S	S	147			211	⊢	$LTK
20		DC4	84	T	T	148			212	⊣	$RTK
21		NAK	85	U	U	149			213	⎕	$Q
22		SYN	86	V	V	150			214	○	$O
23		ETB	87	W	W	151			215	→	$GO
24		CAN	88	X	X	152			216	←	_
25		EM	89	Y	Y	153			217	⊥	$DCD
26		SUB	90	Z	Z	154			218	⊤	$ECD
27		ESC	91	[	[	155			219	⍝	$COM
28		FS	92	\	\	156			220	≡	$EQV
29		GS	93	]	]	157			221	≢	$NQV
30		RS	94	↑	$TAK	158			222	⍸	$FDI
31		US	95	_	$_	159			223	⍷	$FND
32		BL	96	⊂	$ENC	160			224		
33	!	!	97	A̲	a	161	∘	$SC	225		
34	¨	$"	98	B̲	b	162	⌿	$RD1	226		
35	≠	$NE	99	C̲	c	163	∪	$CUP	227		
36	$	$	100	D̲	d	164	⍀	$XP1	228		
37	⍴	$R	101	E̲	e	165	⍒	$GD	229		
38	∩	$CAP	102	F̲	f	166	⍫	$LOK	230		
39	'	'	103	G̲	g	167	⍎	$XEC	231		
40	(	(	104	H̲	h	168	⍕	$FMT	232		
41	)	)	105	I̲	i	169	⍞	$QQ	233		
42	⋆	*	106	J̲	j	170	⍟	$LOG	234		
43	+	+	107	K̲	k	171	⊖	$RV1	235		
44	,	,	108	L̲	l	172	⌹	$MDV	236		
45	−	$−	109	M̲	m	173	⍑	$TBR	237		
46	.	.	110	N̲	n	174	⌶	$IB	238		
47	/	/	111	O̲	o	175	⍙	$UDL	239		
48	0	0	112	P̲	p	176	⍉	$TPS	240		
49	1	1	113	Q̲	q	177	⍋	$GU	241		
50	2	2	114	R̲	r	178	⍲	$NND	242		
51	3	3	115	S̲	s	179	⍱	$NOR	243		
52	4	4	116	T̲	t	180	⌽	$REV	244		
53	5	5	117	U̲	u	181	⎕0	$Q0	245		
54	6	6	118	V̲	v	182	⎕1	$Q1	246		
55	7	7	119	W̲	w	183	⎕2	$Q2	247		
56	8	8	120	X̲	x	184	⎕3	$Q3	248		
57	9	9	121	Y̲	y	185	⎕4	$Q4	249		
58	:	:	122	Z̲	z	186	⎕5	$Q5	250		
59	;	;	123	{	{	187	⎕6	$Q6	251		
60	<	<	124	|	|	188	⎕7	$Q7	252		
61	=	=	125	}	}	189	⎕8	$Q8	253		
62	>	>	126	~	~	190	⎕9	$Q9	254		
63	?	?	127		DEL	191			255		

Index

Note: Index references indicate the page on which the paragraph containing the index term actually ends. Should the paragraph straddle two pages, the actual indexed term might be on the first page, while the index reference is to the second page.

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE: CP-6
APL REFERENCE MANUAL

ORDER NO. CE38-04

DATED MAY 1986

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

⇨ Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. □

FROM: NAME ______________________________ DATE ______________

TITLE ______________________________

COMPANY ______________________________

ADDRESS ______________________________

____

PLEASE FOLD AND TAPE–
NOTE: U.S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 39531 WALTHAM, MA 02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

HONEYWELL INFORMATION SYSTEMS
Technical Publications Remarks Form

TITLE	CP-6 APL REFERENCE MANUAL	ORDER NO.	CE38-04
		DATED	MAY 1986

ERRORS IN PUBLICATION

SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Your comments will be investigated by appropriate technical personnel and action will be taken as required. Receipt of all forms will be acknowledged; however, if you require a detailed reply, check here. □

FROM: NAME ______________________ DATE ____________

TITLE ______________________

COMPANY ______________________

ADDRESS ______________________

____

,-LEASE FOLD AND TAPE–
NOTE: U.S. Postal Service will not deliver stapled forms

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 39531 WALTHAM, MA 02154

POSTAGE WILL BE PAID BY ADDRESSEE

HONEYWELL INFORMATION SYSTEMS
200 SMITH STREET
WALTHAM, MA 02154

ATTN: PUBLICATIONS, MS486

Honeywell

Together, we can find the answers.

Honeywell

Honeywell Information Systems
U.S.A.: 200 Smith St., MS 486, Waltham, MA 02154
Canada: 155 Gordon Baker Rd., Willowdale, ON M2H 3N7
Mexico: Av. Constituyentes 900, 11950 Mexico, D.F. Mexico
U.K.: Great West Rd., Brentford, Middlesex TW8 9DH **Italy:** 32 Via Pirelli, 20124 Milano
Australia: 124 Walker St., North Sydney, N.S.W. 2060 **S.E. Asia:** Mandarin Plaza, Tsimshatsui East, H.K.

45575, 5C686, Printed in U.S.A.