APLSF
Language Manual

AA-H200A-TK

August 1979

This manual describes the language elements of APL-Basic and APLSF on both the TOPS-10 and TOPS-20 operating systems.

This manual supersedes the following: *DECSYSTEM-20 APLSF Programmer's Reference Manual* DEC-20-LASFA-A-D, and *DECsystem-10 APLSF Programmer's Reference Manual* DEC-10-LPLSA-A-D.

OPERATING SYSTEM:	TOPS-10 Version 6.03A TOPS-20 Version 3A
SOFTWARE:	APLSF Version 2 APL-Basic Version 2

digital equipment corporation • marlboro, massachusetts

First Printing, August 1979

DIGITAL	DECsystem-10	MASSBUS
DEC	DECtape	OMNIBUS
PDP	DIBOL	OS/8
DECUS	EDUSYSTEM	PHA
UNIBUS	FLIP CHIP	RSTS
COMPUTER LABS	FOCAL	RSX
COMTEX	INDAC	TYPESET-8
DDT	LAB-8	TYPESET-11
DECCOMM	DECSYSTEM-20	TMS-11
ASSIST-11	RTS-8	ITPS-10

CONTENTS

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

CONTENTS (CONT.)

PREFACE

This manual describes version 2 of APL on both the TOPS-10 and TOPS-20 operating systems. There are two implementations of APL on each system, APL-Basic and APLSF (System Functions). APLSF is a superset of APL-Basic. To distinguish APLSF from APL-Basic, we have shaded the text describing features available only in APLSF.

This document is not an APL tutorial manual. Therefore, if you are unfamiliar with the APL language, you should read an APL primer before reading this manual. Also, because you will be using APL in conjunction with either TOPS-10 or TOPS-20, you should have the latest editions of the following documents on hand:

1. TOPS-20 User's Guide

2. TOPS-20 Monitor Calls Reference Manual

   or

3. TOPS-10 Operating System Commands Manual

4. TOPS-10 Monitor Calls Reference Manual

Conventions Used In This Manual

⟦ ⟧	Special square brackets indicating operational information that can be omitted from a command string.
{ }	Braces indicating a choice. Choose one from the enclosed.
Lowercase letters	Lowercase characters in a command string indicating variable information you supply.
UPPERCASE LETTERS	Uppercase characters in a command string indicating fixed (literal) information that you must enter as shown.
Examples	All examples were produced on an LA37 using either TOPS-10 or TOPS-20.
Contrasting Colors	Red - Where examples contain both user input and computer output, the characters you type are in red; the characters APL prints are in black.
	Gray - The text shaded in gray or printed on gray stock indicates the features available only with APLSF.
APL	Refers to both APL-Basic and APLSF.

ACKNOWLEDGMENT

The DECsystem-10 implementation of APL was developed by Richard Fennell, Frederick Pollack, and William Price under the guidance of Dr. Alan J. Perlis of the Department of Computer Science at Carnegie-Mellon University. The APLSF enhancements were made by Frederick Pollack. The conversion to the DECSYSTEM-20 was done at Digital Equipment Corporation.

CHAPTER 1

THE APL OPERATING ENVIRONMENT

1.1 INTRODUCTION

APL (A Programming Language) is a language interpreter that runs under the control of either of two operating systems, TOPS-10 or TOPS-20. The TOPS-10 and TOPS-20 operating systems provide the APL user with standard timesharing features, such as resource allocation, job control, device handling, and usage accounting.

Because APL is a very compact programming language, it is suited for handling numeric and character array-structured data. In addition to its mathematically concise format, APL is also an efficient general data-processing language.

APL is a 2-segment system. The code segment or shared segment, is the APL interpreter consisting of code shared among all APL users. The data segment is the APL user's workspace. Each user has a data segment, but there is only one copy of the interpreter.

1.1.1 Workspaces

A workspace is a block of storage where all interaction with APL takes place. Each time you access APL, you are issued a clear workspace in which to define variables and functions as well as to execute APL statements. The size of an APL workspace is dynamic and can vary from 2K to 176K 36-bit words on TOPS-10. On TOPS-20, the figure is expressed in pages of 36-bit words, 4P to 352P. The default workspace size is 20K on TOPS-10 and 40P on TOPS-20. If you need to change the size, refer to the *)MAXCORE* command, Section 5.3.1.

There are three states your workspace can assume as you proceed through an APL session:

1. Clear workspace

2. Active workspace

3. Inactive workspace

At the beginning of an APL session, you are given a fresh workspace: the clear workspace. It has no open files, no defined variables or functions; it has a clear (empty) symbol table, and a clear (empty) state indicator. System variables are set to their default values. Once you start typing information into your workspace, it is no longer clear. The workspace you are currently using is your active workspace.

All functions and variables you define during the current APL session are stored temporarily in this workspace. You can save an active workspace as a file, in binary format, on a secondary-storage device, such as a disk or magnetic tape. An active workspace becomes an inactive workspace when you save it. You can save several workspaces in your disk area; however, only one can be active at any one time. As a group, inactive workspaces are known as a private library.

When you save a workspace, you are not only saving functions and variables, but also the APL symbol table, state indicator and some system variable settings. When you retrieve an inactive workspace from your library, it again becomes your active workspace.

A workspace can be named, copied, saved, retrieved, deleted, renamed, protected, and cleared. These workspace operations are described in Chapter 5.

1.1.2 Data Files

In addition to workspaces, APL provides another way to store and retrieve information: the data file. A data file is a collection of related elements stored in the form of records. You determine the size and content of the records, and the structure and access properties of the file.

APL can handle four types of files:

1. ASCII Sequential
2. Internal Sequential
3. Direct Access
4. Binary Access

Refer to Chapter 7 for a description of data files.

1.2 HARDWARE

The APL language consists of a special character set in which Greek letters and a variety of other special characters represent APL language functions and operators. Examples of such special characters include ι, ⎕, ∇, and ∊.

TOPS-20 and TOPS-10 support a variety of terminals for use with the APL system. Some terminals provide keyboards with the full APL character set (such as the LA37 in Figure 1-1). However, terminals without the APL keyboard can also be used to access APL. On non-APL terminals, you can use a special set of keyword mnemonics to represent APL symbols. See Table 1-2 for both character sets.

You select the APL character set or the mnemonic character set when you begin an APL session. APL prompts you with:

```
terminal..
```

You respond with one of the terminal designators listed in Table 1-1.

Table 1-1
APL Terminals and Designators

Terminal	Designator
IBM Selectric[1] -type terminal with APL typing element, or equivalent	2741
Bit-paired ASCII/APL terminal	BIT
Key-paired ASCII/APL terminal	KEY
DECwriter II model LA36 with APL option (LA37)	LA36
Tektronix[2] 4013	4013
Tektronix[2] 4015	4015
Any terminal without APL character set	TTY ⟦/terminal⟧

[1]Selectric is a registered trademark of IBM.
[2]Tektronix is a registered trademark of Tektronix, Inc.

The /terminal switch with the TTY designator can be any one of the other terminal designators in Table 1-1, for example TTY/BIT. This switch is optional. It takes effect when you use the)*OUTPUT* command (Section 7.9).

When you specify LA36, 4013, or 4015, that designator causes character font-switching sequences to be sent to the terminal when you enter and leave APL. This means that you do not have to manually switch character sets by pushing a button on the terminal. Specification of KEY does not have this effect.

1.2.1 APL-Keyboard Terminals

The keyboard illustrated in Figure 1-1 is a typical APL-keyboard terminal; you can use it in either ASCII or APL mode. When you access APL, the characters are received and interpreted by the APL system. Note that letters, numbers, and some of the special characters appear in the conventional keyboard positions. In APL mode, the letters print only in uppercase and are produced only when the keyboard is not shifted. The full APL character set is described in Table 1-2.

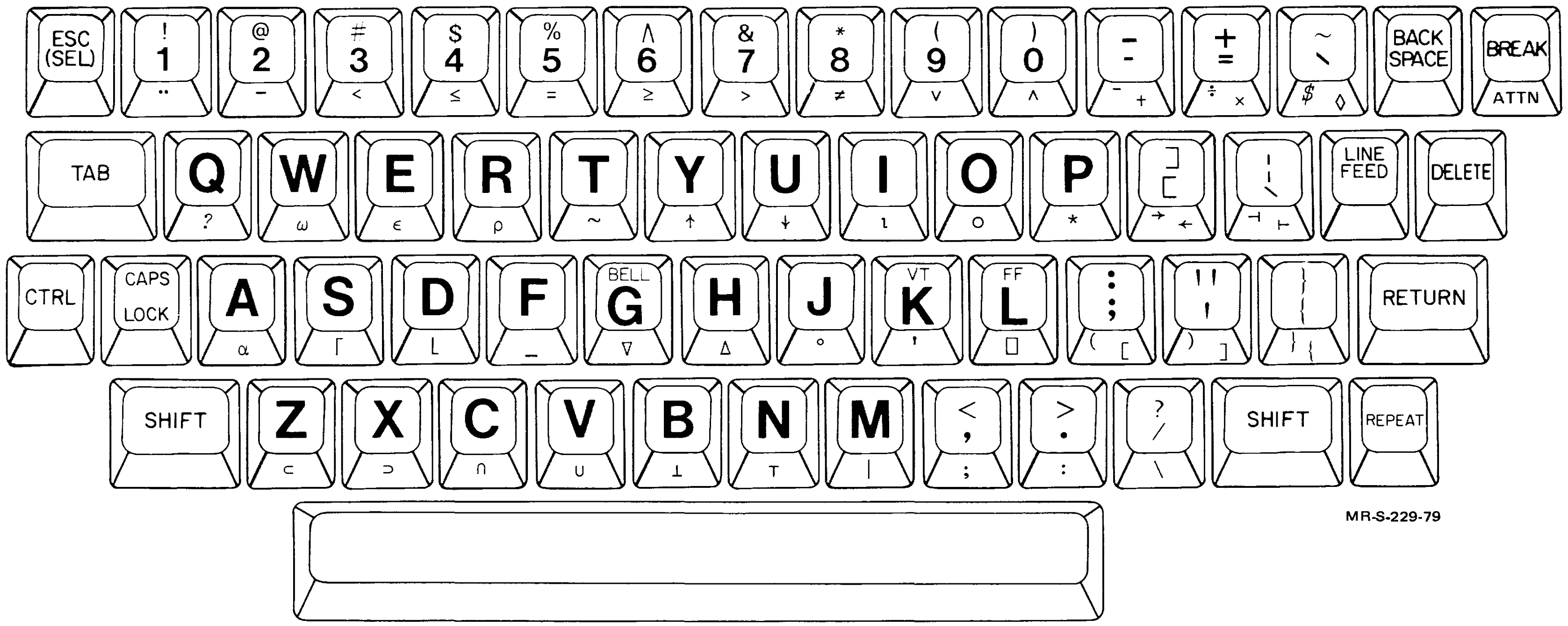


Figure 1-1 The APL Keyboard (LA37 Terminal)

1.2.2 Terminals Without the APL Keyboard

If you do not have a terminal with an APL keyboard, you can use a combination of keyword mnemonics, or escape characters, and ASCII characters to interact with the APL interpreter. First, you respond with TTY when APL prompts for a terminal designator (Section 1.2). Then you can input any of the keyword or escape-mode equivalents listed in Table 1-2.

For example, to represent the APL rho symbol (ρ), either type the mnemonic .RO or the escape character @R. To type a character in escape mode, first type the at sign (@) and then enter the desired uppercase character. No delimiting blanks are necessary, and you can mix the two input modes freely.

APL output can also be displayed in either keyword or escape modes, but you must select one or the other; they cannot be mixed. The *)MODE* command allows you to select the output mode. This is where the TTY /terminal is relevant. See Section 5.5.3.

1.3 THE APL CHARACTER SET

Table 1-2 lists all APL characters available on TOPS-10 and TOPS-20. The first column lists the APL character set. The second column, TTY set, lists the keyword mnemonic equivalents. The third column supplies names commonly associated with APL characters, and the fourth column lists the escape characters. The uppercase letters indicate the origin of the mnemonic representation.

The second section of the table lists APL overstruck characters. These are characters constructed by typing one character, one backspace, then a second character on top of the first. For example, to construct the logarithm symbol (⍟), type the circle symbol (○), then backspace, then type the exponentiation symbol (⋆). You can also type the exponentiation symbol (⋆) before the circle symbol (○); the order is not significant. On non-APL keyboard terminals, overstruck characters are represented by single-strike characters or by keyword mnemonics. Notice that dollar appears as both a single-strike and an overstruck character. On some terminals you can enter dollar as a single-strike character ($), and on other terminals you must enter dollar as an overstruck character (*S*|).

Table 1-2
APL Character Set

Single-Strike Characters			
APL Set	TTY Set	Name	Escape Mode
A-*Z*	A-Z	alphabet	
0-9	0-9	numbers	
+	+	add	
∧	&	and	
←	_	assignment (back-arrow or underline)	
,	,	concatenate	
:	:	colon	
÷	%	divide	
$	$	dollar format	
=	=	equal to	
\	\	expand (scan)	
⋆	*	exponentiate	@P
>	>	greater than	
[	[	left bracket	
(	(	left parenthesis	
<	<	less than	
×	#	multiply	
'	'	quote string	@K
?	?	question (roll and deal)	@Q
/	/	reduce	
]	]	right bracket	
)	)	right parenthesis	
;	;	semicolon	
-	-	subtract	
↑	^	take	@Y
|	.AB	residue (ABsolute value)	@M
⍺	.AL	ALpha	@A
⎕	.BX	quad (BoX)	@L
⌈	.CE	CEiling (maximum)	@S
↓	.DA	drop (Down Arrow)	@U
¨	.DD	Dieresis	
⊥	.DE	DEcode	@B
∇	.DL	DeL	@G
◇	.DM	DiaMond	
∩	.DU	Down Under	@C
⊤	.EN	ENcode	@N
∊	.EP	EPsilon	@E
⌊	.FL	FLoor	@D
≥	.GE	Greater than or Equal to	
→	.GO	GO to (branch)	
⍳	.IO	IOta	@I
{	.LB	Left curly Brace	
∆	.LD	delta (Lower Del)	@H
≤	.LE	Less than or Equal to	
⊢	.LK	Left tacK	
○	.LO	circle (Large O)	@O
⊃	.LU	Left Union	@X
≠	.NE	Not Equal to	

Table 1-2 (Cont.)
APL Character Set

Single-Strike Characters

APL Set	TTY Set	Name	Escape Mode
¯	.NG	NeGation	
~	.NT	Not	@T
⍵	.OM	OMega	@W
∨	.OR	OR	
}	.RB	Right curly Brace	
⍴	.RO	RhO	@R
⊣	.RK	Right tacK	
⊂	.RU	Right Union	@Z
∘	.SO	jot (Small O)	@J
_	.US	UnderScore	@F
∪	.UU	Up Union	@V

Overstruck Characters
(None in Escape Mode)

APL Set	Characters to Strike Over		TTY Set	Name
$	S	|	$	dollar (format)
!	'	.	!	factorial (shriek)
⍒	∇	|	.GD	Grade Down
⍋	∆	|	.GU	Grade Up
⌶	⊥	⊤	.IB	I-Beam (histogram)
⍟	○	*	.LG	LoGarithm
⍲	∧	~	.NN	NaNd
⍱	∨	~	.NR	NoR
⍀	\	-	.CB	back expansion
⊖	○	-	.CR	(Circle) Rotate
⌿	/	-	.CS	back scan
⌹	⎕	÷	.DQ	Divide Quad
⍇	⎕	←	.IQ	Input Quad
⍈	⎕	→	.OQ	Output Quad
Ø	O U	T	.OU	OUt
⍫	∇	~	.PD	Protected Del
⍍	⎕	∇	.QD	Quad Del
⍞	⎕	'	.QQ	Quote Quad
⌽	○	|	.RV	ReVersal
⍉	○	\	.TR	TRanspose
⍎	⊥	∘	.XQ	eXecute
⍕	⊤	∘	.FM	ForMat
⍝	∩	∘	"	Comment (lamp)
A̲-Z̲	A-Z	_	.ZA-.ZZ	underscored alphabetics
∆̲	∆	_	.Z@	underscored lower del

1.4 INTERACTING WITH APL

APL provides easy-to-use commands to allow you to interact with the operating system. Sections 1.4.1 through 1.4.4 describe some of the commands available. Chapter 5 discusses APL system commands.

1.4.1 Entering APL Command Level (Starting the Session)

To access APL, first log in to either TOPS-20 or TOPS-10. After a successful log in, type the following on TOPS-20:

```
    @APLSF
```

On TOPS-10, type:

```
    .R APLSF
```

In both cases, APLSF begins the session by asking for your terminal designator:

```
    terminal..
```

If you are unsure of what to respond, type H (for Help). For example:

```
    terminal..h
    give the appropriate response for your terminal
    response  your terminal
    2741[1]   ibm 2741 or similar with apl ball
    bit       ascii apl bit pairing
    key       ascii apl key pairing
    la36      la36 with apl character set option
    4013      tektronix 4013
    4015      tektronix 4015
    tty       any terminal not having apl font
    TERMINAL..
```

After receiving a valid terminal designator, APL responds with a greeting and identification message. It then supplies a clear work-space for use during the current APL session, or automatically loads the special *CONTINUE* workspace saved from the last APL session, if such a workspace exists in your disk area. (See Section 5.6.2 for a description of the *CONTINUE* workspace.) If a clear workspace is supplied, APL displays the message:

```
    CLEAR WS
```

If the CONTINUE workspace is loaded, APL outputs a standard load workspace message. For example:

```
    @aplsf
    terminal..la
    APL-20 DECSYSTEM-20 APLSF 2(407)
    TTY22) 15:22:57 TUESDAY 26-JUN-79 MASELLA
    SAVED  15:22:39 26-JUN-79 SF
```

[1]The 2741 is supported only on TOPS-10.

APL indents six spaces to signify that it is ready to accept input. APL outputs at the left margin but automatically indents six spaces before echoing your input. The first character you type will print in the seventh column from the left margin. APL thus clearly differentiates between what it prints out and what you type in.

1.4.2 Ending the Session

To log off the system while in APL mode, use one of the following commands:

`)OFF`	ends the session and logs you off the system.
`)CONTINUE`	ends the session, logs you off the system, and stores the active workspace under the name *DSK:CONTIN.APL*. This workspace, instead of a clear workspace, will be loaded the next time you run APL.

Note that APL commands begin with a right parenthesis. These commands, *)OFF* and *)CONTINUE*, are described in Sections 5.6.4 and 5.6.2, along with a description of options available for automatically returning to system command level after ending a session rather than logging off.

CAUTION

Do not end a work session by disconnecting the terminal's telephone connection or the current workspace will be lost.

1.4.3 Returning to System Command Level

To return to system command level during an APL session, type the *)MON* command. APL indicates that control has been returned to the operating system by printing:

```
MONITOR:
```

When you receive the system prompt, you can then perform a variety of system operations, including sending and receiving messages to or from other users and the operator, assigning devices to your job, inquiring about CPU usage, and performing other standard functions.

CAUTION

If you run any other program, the workspace in memory will be destroyed.

To return to APL with the workspace intact, type *CONTINUE*. APL responds with *APLSF*: to indicate that it has again received control.

1.4.4 Interrupting Execution

To interrupt APL during an operation, use the attention signal, CTRL/C. Two CTRL/Cs interrupt function or program execution and return you to APL mode. The response may be delayed for a few seconds because of system buffering.

Typing five CTRL/Cs will return you to system command level. To return to APL mode and resume APL operations, type one of the following operating system commands:

```
     @REENTER

         or

     @CONTINUE
```

1.5 KEYBOARD EDITING

The following sections describe the procedures for entering and correcting APL text on a terminal with an APL keyboard.

1.5.1 Correcting a Line Before Entering

You can type characters in an APL input line in any order. Regardless of how you enter the line, APL evaluates it exactly as it appears on the terminal; the order in which you type characters is not significant. By using the appropriate space and backspace characters, you can even type the line backwards. APL interprets the line only when you press the RETURN key. (This "random order" feature is not available on TTY terminals.)

An APL line can contain up to 390 characters. This total includes spaces and backspaces. If you type more characters than the limit and press RETURN, APL ignores the line and sends the error message:

```
     48 INPUT LINE TOO LONG
```

For a complete list of APL messages, refer to Appendix A.

NOTE

Backspacing is a method for positioning the carriage, it does not cause characters to be erased or ignored by APL.

On an APL-keyboard terminal, if you discover an error in a line before you press the RETURN key, you can backspace to the error and press the LINEFEED key. Everything from the LINEFEED to the right is ignored by APL. You can then complete the line directly below the part in error by retyping it. For example:

```
           C←'REEIVE      backspace 4 then line feed
                CEIVE'
           C
     RECEIVE
```

There are several special characters available with which to make corrections. Table 1-3 lists these characters and their meaning.

Table 1-3
Editing Characters

Character	Meaning
CTRL/C	Two CTRL/Cs interrupt APL function execution and expression evaluation. Five CTRL/Cs return you to system command level.
CTRL/U	Deletes the current input line and positions you in column one of the next line. It does not delete past the first LINEFEED it encounters. Echoes as XXX on TTYs and as ⊃⊃⊃ on LAs.
CTRL/O	Suppresses output to the terminal.
CTRL/R	Performs a LINEFEED and displays the corrected line starting at column one.
LINEFEED	In conjunction with backspace, it deletes input.
DELETE (RUBOUT)	Deletes one character at a time. On an LA, echoes one ⊢ character on TOPS-10 for entire operation. Echoes one ⊢ character for every character deleted. The ⊢ prints as a \ on a TTY.

1.5.2 Correcting a Line After Entering

An APL statement entered and processed in immediate mode can be edited according to the same line-editing rules established for user-defined functions. These rules are described in Sections 6.3.7 and 6.3.8.

CHAPTER 2

LANGUAGE SYNTAX

2.1 INTRODUCTION

This chapter describes the syntax that governs the construction of APL statements and expressions, including statement components, data types, and expression evaluation.

2.1.1 Statement Execution Modes

Two execution modes are available in APL:

1. Immediate mode, in which APL executes statements and expressions as soon as you enter them and press the RETURN key.

2. Function-execution mode, in which APL executes the statements contained in a user-defined function (Chapter 6). APL enters function-execution mode whenever it discovers a user-defined function in the statement it is currently executing, and exits from function-execution mode when the last statement in the function is executed, you suspend the function, or an error occurs.

The statement syntax is identical in both modes; however, there are a few special characters that are not generally relevant in immediate mode, but useful in function-definition mode. These characters are described in Chapter 6. Most of the examples in this chapter illustrate immediate-mode execution. Chapter 6 describes function-definition mode, in which you prepare and edit functions, and function-execution mode, in which you actually execute the function.

In immediate mode, APL clearly differentiates between what you type and what it prints. APL always indents six spaces before accepting input. After you enter text, press the RETURN key to indicate that entry is complete. APL processes your input and, if necessary, prints results beginning at the left margin. After printing output, APL then performs a carriage return/line feed and indents six spaces. For example:

```
      A←6
      A
6
      B←9
      B
9
      A+B
15
```

You can have up to 390 characters in a single line. This count includes spaces and backspaces.

2.1.2 Expression Components

An APL expression can consist of the following components:

1. Identifiers
   - Variables, Section 2.1.2.1
   - Labels, Section 6.4.2
   - User-defined Functions, Chapter 6
   - Groups, Section 5.1.3.4
2. Constants
   - Numeric, Section 2.1.2.2
   - Character, Section 2.1.2.2
3. Characters
   - I/O Functions, Section 2.5
   - Primitive Scalar Functions, Section 3.2
   - Primitive Mixed Functions, Section 3.3
   - Extended Functions, Section 3.4
   - File Functions, Chapter 7
   - Operators, Section 3.5
4. System Variables, Section 4.2
   System Functions, Section 4.3

2.1.2.1 Identifiers

2.1.2.1 Identifiers - An identifier can be a variable name, a label name, or a user-defined function name. It can consist of any number of letters and digits; however, the first character must be a letter. APL defines a letter, in this case, as any character *A* through *Z*, $\underline{A}$ through $\underline{Z}$, Δ and ⍙. Only the first 31 characters of the identifier are significant, and embedded spaces are not allowed. APL truncates all identifiers to 31 characters; therefore, you cannot create an identifier longer than 31 characters. For example:

Legal Identifiers	Illegal Identifiers	
ABC63B8	1AC75	(does not begin with a letter)
Δ74	Z94 36	(contains an embedded space)
⍙G956HΔ	F∇742E	(contains invalid character ∇)

Note that you cannot start an identifier with the characters *S*Δ or *T*Δ because of a conflict with the trace and stop vectors. Refer to Sections 6.4.5 and 6.4.6.

A variable must contain a value before you can reference it. Otherwise, you will receive the message 11 *VALUE ERROR* from APL. Section 2.1.5 describes how to assign values to variables.

Variable names and their positions have special meaning in function-definition mode. Refer to Chapter 6 for this information.

2.1.2.2 Constants - Constants can be either numeric or character data. A numeric constant is one or more decimal digits with an optional decimal point. A numeric constant can also be in exponential format; an integer or decimal quantity followed by *E* and the power of ten by which the quantity is to be multiplied. All of the following constants, for example, are valid representations of the same value.

```
712 712.0 7120E¯1 7.12E2
```

Wherever possible, APL prints numbers without decimal points and exponents.

```
      712 712.0 7120E¯1. 7.12E2
712 712 712 712
```

In APL, you represent a negative number by a numeric constant preceded by a negative sign (¯). This sign is a distinctive symbol (uppercase 2). It is not the same character as the minus sign (-) which is used to indicate subtraction. On non-APL terminals, the negative sign is .NG.

A character constant is one or more alphanumeric and/or special characters (including carriage returns and line feeds) enclosed in single quotation marks. For example:

```
'ABCDEFG'
'GEORGE'
'THIS IS A CONSTANT'
```

When APL prints a character constant, it omits the enclosing quotation marks. If you want APL to output quotation marks, type one extra single quotation mark next to the one you want to print. For example:

```
      B←'TONY''S TENNIS RACQUET'
      B
TONY'S TENNIS RACQUET
```

Numeric and character data can be structured in a variety of ways. APL supports the following types of data:

1. scalars

2. vectors

3. matrices

4. arrays of three or more dimensions

A scalar is a single numeric or character value with no dimensions. For example:

```
      32
32

      'A'
A
```

A vector is a 1-dimensional array or character string consisting of any number of values. Enter a numeric vector as a list of values separated by at least one space. For example:

```
      H←1 2 3 4 5
      H
1 2 3 4 5
```

In this example, *H* is defined as a vector whose elements are 1, 2, 3, 4, and 5. APL stores the values in the order in which you enter them.

A character vector or literal vector is entered as a string of character constants enclosed in single quotation marks. Unless you want the space character as part of the character vector, do not insert spaces between characters in the vector. Note the following example:

```
      A←'ABCDEFG HIJKLMNOP'
      A
ABCDEFG HIJKLMNOP
```

Because any characters, including carriage returns and line feeds, can be elements of a character constant, you can also enter several lines of character data as a 1-character vector. For example:

```
      A←'THIS IS A
MULTIPLE LINE
LITERAL.'
      A
THIS IS A
MULTIPLE LINE
LITERAL.
```

Although there are several lines of text, *A* is still a vector.

Note that a common error occurs when you type a character constant with an unbalanced number of quotation marks. APL thinks that you are still defining the constant when you press RETURN to enter the line. Consequently, APL includes the carriage return/line feed as part of the constant. You can spot this error by noticing that APL does not indent six spaces when you press the RETURN. Typing a single quotation mark will usually get you out of this situation.

A matrix is a 2-dimensional array consisting of rows and columns. APL supports the use of matrices as well as arrays of higher dimensions.

The rho character is used to create and reshape arrays (Sections 3.3.18 and 3.3.15). You enter values corresponding to each element of an array and also the shape or size of the array. The following examples show array output. (To input arrays, refer to Section 3.3.15.)

The following is a numeric matrix with 2 rows and 3 columns.

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

APL also supports arrays of three dimensions or more. For all practical purposes, there is no intrinsic limit on the number of dimensions in an APL array. The only restriction is that the size of the array cannot exceed your workspace size. If you have unlimited memory available, the maximum number of dimensions allowed is 2*18.

The following is an example of a 3-dimensional character array. Note that APL inserts a blank line between each plane greater than two.

```
      A←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

      2 3 4ρA
ABCD
EFGH
IJKL

MNOP
QRST
UVWX
```

2.1.3 Spaces and Comments

Spaces are usually not significant in APL. Therefore, you need not separate functions from constants or variables. This is also true on non-APL-keyboard terminals. The mnemonics for operators need not be preceded or followed by a space. The following expressions are equivalent:

```
      B     ←     35
      C←16
      A←B+1-C
      A   ←  B     +  1    -C

      .TR B
35
      .TRB
35
```

Spaces are also not required between a succession of functions or operators. For example:

```
     A←1↑⌈+⌿B
```

However, spaces must be included to separate names of adjacent user-defined functions, constants, and variables. For example, they are required when you are entering a series of numeric constants as a vector. The spaces included in the following example are necessary:

```
     2 TRIG 3        (user-defined function)
     B←3 4 5         (numeric vector)
```

You can also use comments in APL. Their use is particularly relevant in function-definition mode. Comments must appear on separate lines; they may not be included on lines containing APL statements. The first character in a comment line is the lamp character (⍝), formed by overstriking the down union character (∩) with the jot character (∘). Section 6.2.4 describes comment lines in greater detail and illustrates their use in a variety of user-defined functions. On non-APL-keyboard terminals, use a double quotation mark for the comment character.

2.1.4 File Specifications

File specifications indicate to the operating system when to locate and identify a file. Certain operations in APL require you to supply a file specification in the syntax (for example, when you assign a channel number to a file with ⎕*ASS*, Section 7.4.1).

The complete form of a file specification is:

```
     dev:filename.ext[directory]<prot>
```

where

dev:	is a device name, or a logical name you have defined. See Appendix D for defining logical names.
filename	is one to six alphanumeric characters specifying a particular file in the directory.
.ext or .typ	is one to three alphanumeric characters identifying the contents of the file.
[directory]	is the project-programmer number of the owner of the directory. You can translate a directory name on TOPS-20 to its corresponding project-programmer number by using the TRANSLATE command. See Appendix D for this information.
<prot>	is a 3-digit octal protection code specifying who can read and write the file.

You need not give the entire file specification in every case. The defaults are:

Argument	Default
dev:	DSK:
filename	No default; it must be specified
.ext or .typ	Depends on the type of file
[directory]	Your currently connected directory
<prot>	Installation-dependent

2.1.5 Statement Types

There are two general types of APL statements:

1. Branches constructed with →
2. Assignments constructed with ←

Branch statements restart a function and transfer control from one part of a function to another. These statements are most relevant in the context of user-defined functions and are described in Section 6.4.1.

Assignment statements store one or more values into an identifier. The general form of an assignment statement is illustrated by the following:

```
A←2+B
```

where *2+B* is an APL expression
← is the assignment function
A is a variable name

You can have more than one statement on a line by separating each with a semicolon. Each statement separated must contain a value. For example:

```
      C←A+B;A←64;B←92
      C
156
```

Assignment statements are, themselves, expressions and can be used in the construction of other statements. The following example illustrates a method of assigning values to more than one variable with a single statement:

```
A←3+B←4+C←7
```

Here the value 7 is assigned to *C*, 11 to *B*, and 14 to *A*. The expression is evaluated from right to left according to the rule described in Section 2.1.5.

In any mode, if you do not include an assignment or branch function in an expression, APL prints the results on the terminal when the expression is executed. For example:

```
      2+3
5
```

This type of expression has the effect of an implied print statement in APL.

2.1.6 Evaluation of APL Statements and Expressions

APL evaluates unparenthesized statements and expressions in strict right-to-left order, regardless of the particular functions in the statement. Unlike some languages, which perform multiplication and division before addition and subtraction, APL has no explicit function precedence. For example, APL evaluates the expression

```
      3×4+5
27
```

as 27, using right-to-left evaluation, rather than 17. Thus, the expression is interpreted as

```
      3×(4+5)
27
```

You can control the order in which individual functions are evaluated by enclosing part of the expression within parentheses. To cause the expression above to evaluate to 17, enter the following:

```
      (3×4)+5
17
```

APL evaluates this expression as 17 because 5 is added to the quantity 3x4, not simply to 4.

2.2 Number Precision

APL is a double-precision system with internal precision of about 18 decimal digits. Numbers are represented internally in two ways.

1. Integers less than 2 to the 35th power (2*35) are stored with full precision.
2. Non-integers and integers larger than 2 to the 35th power are carried out in floating-point format.

APL handles conversion between the two formats automatically.

Although you cannot control the internal precision of numeric representation, you do have some control over the output representation. The ⎕*PP* variable, Section 4.2.15, allows you to specify the output precision of non-integers. Also, the APL format functions ($ and ⍕) can also be used to control the printing precision of specific numbers and arrays (Sections 3.4.5 and 3.4.6).

Notice how APL outputs the numbers in these arrays:

```
      2 2⍴⌊/⍳0
  1.701411835E38    1.701411835E38
  1.701411835E38    1.701411835E38


      A←2 2⍴1
      A
1 1
1 1
      A[2;2]←⌊/⍳0
      A
  1.000000000E0     1.000000000E0
  1.000000000E0     1.701411835E38


      A←2 2⍴1
      A[1;1]←10*6
      A
  1000000                   1
        1                   1


      A←2 2⍴1
      A[2;2]←10*¯6
      A
  1                  1
  1                  0.000001


      A[1;1]←10*6
      A
  1.000000000E6     1.000000000E0
  1.000000000E0     1.000000000E¯6
```

The range of numbers you can input without receiving an error (15 *DOMAIN ERROR*) are:

For integers	¯2*35 to (2*35)-1
For non-integers	¯.14693679E-38 to 1.7014118E38

Note that Boolean is represented as 36 bits per word: (136)=136 is equal to 36ρ1.

2.3 ERROR HANDLING

When APL encounters an error, it prints three things:

1. an error message
2. the line in which the error occurred
3. a caret (^) approximately underneath the particular point at which the error was discovered

The following are examples of common error conditions:

```
         B←3
         A×B
    11 VALUE ERROR
          A×B
           ^

         1+1B+2+3
     7 SYNTAX ERROR
          1+1 B+2+3
              ^

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

In the first example, APL printed a value error because the variable named *A* had not been assigned a value. The syntax error occurred because an identifier cannot begin with a number (1*B*). The length error was a result of an unequal number of constants on either side of the plus sign.

Because APL is a highly interactive system, you can almost always respond to an error condition simply by correcting the statement in which the error occurred. This characteristic of the language also aids the trial-and-error approach to program development. In function-execution mode, APL prints an error message, the function name, and the line number of the statement at which it occurred. APL also suspends execution of the function. You then have the options of terminating the suspended function, restarting it possibly at another statement, or debugging it before resuming execution. Chapter 6 describes techniques for developing and executing functions.

If certain types of errors occur in function-execution mode, you may not want APL to halt function execution to await your corrections. APL allows you to handle error conditions under program control. The execute functions ⍎ and ⍕ (Section 3.4.3) provide some control. However, for more error-trapping facilities, refer to Section 6.5.

2.4 ARRAY INDEXING AND COMPARISONS

This section introduces the use of array indexing in APL and also the use of "fuzz" in performing comparisons. Both of these concepts are helpful in understanding the examples included in subsequent sections of this chapter.

2.4.1 Indexing Arrays

The concept of using and entering values for arrays has already been introduced in Section 2.1.2.2. To be able to access, individually, the values of the elements stored in an array, you must know the positions of the elements within the array. These positions are known as the indices. The procedure for accessing elements is called indexing. The first position in an array (index origin) can be either 0 or 1. You can set the index origin with system variable ⎕*IO*, Section 4.2.11.

To index an array, specify the array followed by the indices enclosed in square brackets, and separated with a semicolon. Each index must be an integer scalar, or an expression that evaluates to an integer. The number of indices needed to pinpoint an element depends on the array type. In general, you must specify as many indices as the number of dimensions of the array. For a vector, a single index is sufficient to identify the position of the element desired element. A matrix, a 2-dimensional array, requires two indices separated by semicolons; a 3-dimensional array requires three indices separated by semicolons; and so forth.

For example, specifying:

```
    A[1]
```

accesses the first element stored in vector *A*. If *A* consists of the vector shown below, then *A*[3] is 25.

```
          A←72 91 25 46 87
          A[3]
    25
```

If the array is a matrix, specify two indices: the first one for the row and the second one for the column:

```
          B←2 4⍴⍳8

          B
      1  2  3  4
      5  6  7  8

          B[2;3]
    7
```

Specifying the indices in the form of an array enables you to access more than one element at a time. For example:

```
      A←32 44.6 71 .80 65 97.2


      A
32 44.6 71 0.8 65 97.2

      A[3 5 6]
71 65 97.2

      M←2 4⍴⍳8

      M
  1  2  3  4
  5  6  7  8

      M[2;1]
5

      M[1 2;2 3]
  2  3
  6  7
```

The index can also be an expression which is evaluated to generate the element positions.

```
      I←2 4 5
      V←10 22 31 49 56 68 72
      V[I+1]
31 56 68
```

Here *V* and *I* are both vectors. The expression *V*[*I*] accesses the elements of *V* referenced by *I*: that is, the third, fifth, and sixth members of vector *V*.

Character arrays can also be indexed. For example:

```
      A←'ABCDEFGHIJKLMNOPQRSTUVWXYZ .'
      BABY←A[10 5 14 14 9 6 5 18 27 12 25 14 27 13 28]
      BABY
JENNIFER LYN M.
```

Note that an element can be duplicated by specifying its position more than once. The array being indexed need not be a variable. It can be a constant set of values or an expression enclosed in parentheses. For example:

```
      7 6 5 4 3 2 1[2 4]
6 4
      (2 4 8 16*2)[1 2]
4 16
```

You can omit a subscript from an index specification, but the semicolon must be included if only one array dimension is specified. If you omit the right subscript, all columns are selected from the matrix; if you omit the left subscript, all rows are selected. For example:

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

Note that a semicolon is required to indicate which subscript has been omitted. In general, the size of the result when a variable is indexed is equal to the catenation of the sizes of all the indices. For example, if *Z*←*X*[*I*1;*I*2;*I*3;...*IN*] then (⍴*Z*)=(⍴*I*1),(⍴*I*2),(⍴*I*3),...(⍴*IN*).

```
      V←'ABCDEF'
      V[3 5]
CE
      ⍝VECTOR INDEXED WITH A VECTOR
      ⍝RESULTS IN A VECTOR
      V[6 5 4 3 2 1]
FEDCBA
      ⍝VECTOR INDEXED WITH A MATRIX
      ⍝RESULTS IN A MATRIX
      M←2 3⍴2 5 4 6 5 4
      V[M]
BED
FED
      ⍝MATRIX INDEXED WITH TWO 2-DIMENSIONAL
      ⍝INDICES RESULTS IN A 4-DIMENSIONAL ARRAY
      M←2 2⍴1 2 2 1
      A←M[M;M]
      A
 1 2
 2 1

 2 1
 1 2


 2 1
 1 2

 1 2
 2 1
```

```
      ⍴A
2 2 2 2
      M←2 4⍴⍳6
      M
  1  2  3  4
  5  6  1  2
      ⍝SCALAR RESULT
      A←M[1;1]
      A
1
      ⍝NULL ARRAY
      ⍴A
                              (APL outputs a blank line)
      ⍝1-ELEMENT VECTORS
      B←,M[1;1]
      B2←M[,1;1]
      B
1
      B2
1
      ⍴B
1
      ⍴B2
1

      ⍝2-ELEMENT VECTOR
      C←M[;2]
      C
2 6
      ⍴C
2
      ⍝A 2-BY-1 MATRIX
      D←M[;,2]
      D
  2
  6
      ⍴D
2 1
```

You can also use indexing to change values of elements already stored in an array. For example:

```
      A←2 3⍴⍳6


      A
1  2  3
4  5  6
      A[1;2 3]←7 8
      A[2;1 2]←9

      A
1  7  8
9  9  6
```

```
      A[1;1]←12
      A
12  7  8
 9  9  6
      A[1;1 1]←2 3
      A
2  7  8
9  9  6
```

2.4.2 The Index Origin

In APL, the first position of a value stored in an array is called the index origin. You have the option of beginning the indices of an array at either 1 or 0. For example, if the index origin is 1, then members of a vector named *A* would be numbered *A*[1], *A*[2], *A*[3], and so forth. If the index origin is 0, elements begin at *A*[0], *A*[1], *A*[2], and so forth.

The default index origin in a clear workspace is 1, but you can change this setting to 0 or reset it to 1 with the ⎕*IO* variable (Section 4.2.11). The index origin setting is saved when you save your workspace. Refer to the)*SAVE* command, Section 5.2.4.

The value of the index origin is often used in conjunction with several monadic and dyadic functions. Chapter 3 discusses the index origin in that context.

2.4.3 Comparison Tolerance or Fuzz

APL handles the problem of performing decimal arithmetic on a binary machine with a concept known as fuzz. When two non-integer numbers are compared, for example, 7.913 and 8.019, they are considered equal if the difference between them is within a certain range. This range is referred to as the comparison tolerance or the fuzz quantity.

There are two types of fuzz:

1. Absolute fuzz - which is the tolerance used to determine whether or not a decimal number is close enough to an integer in value to be considered an integer.

2. Relative fuzz - which is the tolerance used when comparing two numbers to determine whether or not they are close enough to be considered equal.

The absolute fuzz in this version of APL is approximately 1E¯7. This setting cannot be changed. The default relative fuzz is 1E¯13 in a clear workspace. You can change the relative fuzz in your active workspace by assigning a new value to the ⎕*CT* variable, Section 4.2.7. The relative fuzz setting is saved when you save your active workspace.

The following functions use ⎕*CT* when making comparisons: <, ≤, =, >, ≥, ≠, *A*⍳*B*, *A*∊*B*, ⌊*A*, *A*⌊*B*, ⌈*A*, *A*⌈*B*.

2.5 TERMINAL I/O OPERATIONS

APL provides you with utilities to ease input and output operations on a variety of system devices. This section describes terminal input and output. Chapter 7 describes the file system that APLSF uses to handle file-oriented I/O in ASCII sequential, internal sequential, direct-access, and binary-access format.

There are several methods of input and output as illustrated by Table 2-1.

Table 2-1
Input/Output Functions

Expression	Meaning	Section
A←⎕	Quad (evaluated) input	2.5.1
A←⍞	Quote-Quad (character) input	2.5.2
A←⍔	Quad-Del (unedited) input	2.5.3
A←⍇[*N*]*C*	File Input	7.2
A	Normal output	2.5.5
A;*B*;*C*	Mixed output	2.5.6
⎕←*A*	Quad output	2.5.5
⍞←*A*	Bare output (Quote-Quad)	2.5.7
⍔←*A*	Bare output (Quad-Del)	2.5.7
A⍈[*N*]*C*	File output	7.2

All terminal I/O, except normal and mixed output, use the ⎕ symbol. All forms of I/O can be used in either immediate mode or function-execution mode. The file input and output functions in APLSF are described in Chapter 7. The following sections describe the basic forms of the quad function.

2.5.1 Evaluated Input Mode-Quad Input (⎕ or .BX)

The most basic form of quad input is called evaluated input. In this mode, the statement you enter at the terminal, in response to the ⎕ input request, is evaluated and its value is returned as the result of the ⎕ input function. By placing a ⎕ to the right of a back arrow in an expression, you signal APL to expect data input from the terminal. Normally, APL prompts you by printing ⎕: in the left margin. You can change the prompt with the *⎕SF* system variable (Section 4.2.18). Any character data input must be enclosed in single quotation marks. Otherwise, it will be considered an APL expression. For example:

```
      K←⎕
⎕:
      5
      A←2×⎕
⎕:
      5
      A
10

      MSG←⎕
⎕:
      'NO OPERATOR'
      MSG
NO OPERATOR
```

While the system is awaiting your input, you can enter and execute a system command, evaluate an expression, or define a function. The input request remains pending. If an error is encountered in the input, APL prints the appropriate message and allows you to reenter the input. If you enter a carriage return or spaces and a carriage return, APL again prints the ⎕: prompt and waits for input.

The prompt signal (⎕:) is the default. You can change it with the *⎕SF* variable, Section 4.2.18.

2.5.2 Character Input Mode-Quote-Quad (⍞ or .QQ)

APL has another version of the quad function especially for input of character data, the quote-quad function (⍞). An example of quote-quad mode is shown below:

```
      X←⍞
THAT'S AMAZING
      X
THAT'S AMAZING
```

Unlike evaluated input, quote-quad input allows you to enter character strings without enclosing them in single quotation marks. Note that APL does not print the ⎕: prompt in quote-quad mode.

When APL encounters a ⍞ symbol, it positions the carriage at the left margin and accepts the data up to the next carriage return as a character variable. If you enter a single character, APL treats it as a character scalar; it stores a string of characters as a character vector. If you enter only a carriage return, APL treats this input as a vector of length zero; this treatment is significantly different from the handling of empty input in quad-input mode, in which APL rejects the input and waits for you to reenter it correctly. If you enter a tab, APL converts it into the equivalent number of spaces accomplished by the tab.

Note that you cannot enter and execute system commands or define function during ⍞ input.

2.5.3 Unedited Input Mode-Quad-Del (⍠ or .QD)

APL has a third version of the quad function, the quad-del function (⍠). The quad-del function allows you to enter special characters, including backspace, without having APL evaluate them. The backspace is treated as a separate character, and an overstrike symbol is not created.

APL counts each character you input in quad-del mode. The length of the expression includes any spaces and backspaces. (A tab is treated as one character.) The following example illustrates the difference between quad-del and quote-quad modes in entering overstruck APL characters. The example uses the transpose function (⍉) which takes an array and transposes its values (Section 3.3.20). To input this function, type ○, press BACKSPACE, and type \. The result of the shape function (⍴), (Section 3.3.18), prints the number of characters in the array.

```
      X←⍠
⍉A
      ⍝THE BACKSPACE IS COUNTED AS A CHARACTER.
      ⍴X
4
      X←⍞
⍉A
      ⍝IN QOUTE-QUAD, THE BACKSPACE IS NOT COUNTED.
      ⍴X
2
      ⍝IN SINGLE QUOTES, IT IS NOT COUNTED EITHER.
      ⍴'⍉A'
2
```

The following example shows the particular use of quad-del mode in accepting input from non-APL-keyboard terminals. The mnemonics are not decoded.

```
      A_.QD
.ROB
      .ROA
4
      A_.QQ
.ROB
      .ROA
2
      .RO'.ROB'
2
```

As in quote-quad input mode, if you enter only a carriage return or spaces followed by a carriage return, APL treats this input as a null vector of length zero.

2.5.4 Escaping From Input Mode

To escape from an input request, you have certain escape options depending on the input mode:

In quad-input mode, type a right arrow (→)

In quote-quad (⍞) or quad-del (⍹) input mode, type *OUT* as follows:

O backspace *U* backspace *T*

or

On non-APL-keyboard terminals, type .OU

Each of these methods causes function execution to be interrupted but does not cause an exit from the function. Refer to Chapter 6 for information on function-execution mode.

2.5.5 Normal and Quad Output Modes

In normal output mode, to display output on the terminal, type an expression or an identifier without an assignment function (←) or branch function (→) as the leftmost character. The result of the expression prints on the terminal. For example:

```
      A
25
      64*3
262144
      17+A
42
```

If the quad symbol appears immediately to the left of an assignment function (←), the result of the expression to the right of the ← prints on the terminal. This is called quad output. For example:

```
      ⎕←A
25
```

Note that using quad output has the same effect as in the previous example of merely typing the variable name *A*. Quad output is especially helpful when an APL statement contains multiple assignments. For example:

```
      B←3+⎕←5×4
20
```

This statement performs the computation and displays the desired output - the result of the computation 5×4. This method is more efficient than the following:

```
      B←5×4
      B
20
      B←3+A
```

If the last operation (the leftmost expression) in an expression is an assignment or branch, then no final output is produced. The following will not cause output to be printed:

```
A←5×4
```

The following will:

```
      4+A←5
9
```

When APL outputs an array, and a row cannot fit on a single line, the remainder of the line prints on the following line, indented six spaces. For example:

```
      ⍳50
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
      20 21 22 23 24 25 26 27 28 29 30 31 32 33
      34 35 36 37 38 39 40 41 42 43 44 45 46 47
      48 49 50
```

To provide more room on a line you can alter the page width with the ⎕*PW* system variable. Refer to Section 4.2.16 for this information.

2.5.6 Mixed Output Mode

Mixed output mode allows you to print character data and numeric data on the same line. You request mixed output by entering a series of values or expressions, separated by semicolons, in the order in which they are to appear. The output displayed contains no carriage returns or spaces, except where required by the data.

Although APL evaluates from right to left, it prints items from left to right. If you display two expressions separated by a semicolon, APL will not put a space between them. You must specify a space if you want one. For example:

```
      1+1;2+2
24
      1+1;' ';2+2
2 4
```

In the first part of the next example, APL reports a value error because no value is assigned to *B*. In the second part, *B* is assigned the value first.

```
      1+B;' ';B
 11 VALUE ERROR
       1+B;' ';B
               ^
      1+B;' ';B←1
2 1
```

APL will not print a value if the leftmost expression on a line is an assignment or branch operation. You can get around this by enclosing the assignment within parentheses. Note the following examples:

```
      'THE VALUE OF A IS 25'
THE VALUE OF A IS 25

      A←5;'NOTHING WILL PRINT BECAUSE OF THE ARROW'

      (A←'THIS ');'WILL PRINT BECAUSE OF THE PARENTHESES'
THIS WILL PRINT BECAUSE OF THE PARENTHESES
```

Another way of printing the leftmost expression containing an assignment or branch, is to precede the statement with a semicolon:

```
      ;B←5
5
```

Although the semicolon is considered a statement separator, each expression in the list must return a value. For example, if *F* is a user-defined function that does not return an explicit result (regardless of the number of arguments *F* requires), the construct '*;F;*' returns an 11 *VALUE ERROR*.

```
       ∇F
[2]    'HI THERE'
[3]    ∇
       1;F;2
HI THERE
HI THERE
 11 VALUE ERROR
       1;F;2
         ∧
```

2.5.7 Bare Output Mode (Quote-Quad ⍞ or Quad-Del ⍔)

Bare output is a special mode that allows you to request input on the same line as an output string.

Like quad output mode, if you place a quote-quad or a quad-del to the left of an assignment function, the expression to the right of the function prints on the terminal. However, unlike quad output, APL does not perform a carriage return/line feed at the end of the output. Note the difference in the following example:

```
       A←⍞;⎕←'ENTER YOUR NAME '
ENTER YOUR NAME
IRENE
       A
IRENE
```

Notice that the value input is preceded by a number of spaces equal to the length of the ⍔ output.

If the last character of a character constant to be output is a comment (lamp) character ⍝, APL suppresses the printing of the ⍝ as well as the usual delimiting carriage return/line feed, thus, leaving the carriage in mid-line. This feature is useful for entering input on the same line as the previous output. For example:

```
       A←⍞;⎕←'ENTER YOUR NAME ⍝'
ENTER YOUR NAME ROSE
       A
ROSE
```

In immediate mode, bare output is the same as normal output. A bare output statement such as ⍞←*A* must be followed by an input entry at the terminal. Thus, in this instance, output is concluded by the conventional carriage return/line feed. Bare output is more appropriate in function-execution mode. Refer to Chapter 6 for more information on functions.

CHAPTER 3

APL FUNCTIONS AND OPERATORS

3.1 INTRODUCTION

APL provides several characters, known as functions and operators, that allow you to perform various operations with numeric data and character data. These functions and operators are grouped as follows within this chapter:

1. Primitive Scalar functions - arithmetic, relational, and logical, Section 3.2

2. Primitive Mixed functions - for extensive array manipulation, Section 3.3

3. Extended functions - for more advanced data formatting, Section 3.4

4. Operators - more than one function in the syntax, Section 3.5

Functions are either monadic or dyadic. A monadic function requires only one argument placed immediately to the right of it. A dyadic operator requires two arguments, with the function placed between them. Depending on the function, arguments can be variables, numbers, character strings, or expressions.

Functions are also classified as either scalar or mixed. A scalar function generally takes a single-value argument and returns a single-value result. However, scalar functions can also be used with vectors and arrays where they operate on an element-by-element basis. A mixed function can take a scalar argument and return a result in the form of a vector or an array, or take a vector or array argument and return a scalar result. Therefore, the result of a mixed function is not as apparent as a scalar function.

Both scalar and mixed functions can be either monadic or dyadic. With a scalar monadic function, the shape of the argument determines the shape of the result. For example, a scalar argument returns a scalar result; a vector argument returns a vector result, and so forth. When using scalar dyadic functions, you must specify arguments that have the same number of elements and, if arrays, the same dimensions. Table 3-1 shows the results achieved by specifying certain arguments to scalar dyadic functions.

An operator is a function that takes another function as its argument. APL operators are described in Section 3.5.

Table 3-1
Results of Scalar Dyadic Functions

Argument	Function	Argument	Result
scalar	f	scalar	scalar
scalar	f	vector	vector
vector	f	scalar	vector
vector	f	vector	vector
scalar	f	matrix	matrix
matrix	f	scalar	matrix
matrix	f	matrix	matrix

3.2 PRIMITIVE SCALAR FUNCTIONS

The primitive scalar functions are the arithmetic, relational, and logical functions. They are used primarily for basic arithmetic and logical operations, such as addition, exponentiation, maximum value, and logical OR. With a few exceptions, primitive scalar functions take numeric scalar arguments. The relational functions (≤,≥,<,>,=,≠) can take either character or numeric arguments but only the equal (=) and the not equal (≠) primitives can take both character and numeric arguments in the same expression. The logical functions (∧,∨,~,⍲,⍱) must have arguments that are equal to 0 or 1 within a tolerance of 1*E*¯7, the absolute comparison tolerance that APL uses (Section 2.4.3)

Table 3-2 summarizes the primitive scalar functions available in this version of APL. Most of the functions are straightforward and familiar arithmetic or logical operations.

Table 3-2
Primitive Scalar Functions

Monadic		Dyadic	
Function	Meaning	Function	Meaning
+Y	Y	X+Y	Add X to Y
¯Y	Negative of Y	X-Y	Subtract Y from X
×Y	Sign of Y[1]	X×Y	Multiply X and Y
÷Y	Reciprocal of Y	X÷Y	Divide X by Y
*Y	E to the Yth power	X*Y	X to the Yth power
∣Y	Magnitude of Y	X∣Y	X residue of Y (see primitive mixed operators)
⌈Y	Ceiling of Y	X⌈Y	Maximum of X and Y
⌊Y	Floor of Y	X⌊Y	Minimum of X and Y
⍟Y	Natural logarithm of Y	X⍟Y	Log of Y to the base X
!Y	Factorial of Y	X!Y	Binomial coefficient (number of combinations of Y things taken X at a time)
?Y	A random integer of ⍳Y	X?Y	X number of random integers in the range 1 through Y
○Y	Pi times Y	X○Y	Trigonometric functions (Y is in radians. See Table 3-3)

[1]Definition: ×Y is -1 if Y<0
×Y is 0 if Y=0
×Y is 1 if Y>0

Table 3-3 lists the values 0 through 7 and 0 through -7 that are needed as the left argument to the circle function (○) in order to perform trigonometric functions. The right argument, a scalar or vector, is expressed in radians.

Table 3-3
The Dyadic Circle Function

Expression	Result	Expression	Result
0○*X*	(1-*X**2)*.5		
1○*X*	sine *X*	-1○*X*	arcsin *X*
2○*X*	cosine *X*	-2○*X*	arccos *X*
3○*X*	tangent *X*	-3○*X*	arctan *X*
4○*X*	(1+*X**2)*.5	-4○*X*	(-1+*X**2)*.5
5○*X*	sinh *X*	-5○*X*	arcsinh *X*
6○*X*	cosh *X*	-6○*X*	arccosh *X*
7○*X*	tanh *X*	-7○*X*	arctanh *X*

The following examples illustrate ways in which primitive scalar functions can be extended to arrays:

```
   A←3 3⍴5 6 8 3 2 1 6 4 2
   A
 5 6 8
 3 2 1
 6 4 2

   ⍝ELEMENT-BY-ELEMENT MULTIPLICATION
   A×A
 25 36 64
  9  4  1
 36 16  4

   2×A
 10 12 16
  6  4  2
 12  8  4

   2*0 1 2 3 4 5 6 7 8
1 2 4 8 16 32 64 128 256

   4 9 16 25 36*0.5
2 3 4 5 6
```

3.2.1 Relational Functions

In APL, the relational functions (≤,≥,<,>,=,≠) return results; they are not simply comparison functions. An expression of the form *A*≤*B* yields the result of 1 if true (*A* is less than or equal to *B*), and 0 if false. For example:

```
      9>6
1
      4>6
0
      'C'>'A'
1
```

These functions can take either numeric or character arguments, but only the equal and not equal functions can have mismatched arguments, that is, one numeric and one character argument simultaneously. For example:

```
      'A'=5
0
```

When you use relational functions with Boolean arguments (0 and 1), the relational functions can perform logical operations. For example, the not equal (≠) function performs an exclusive OR operation if its arguments are 0s and 1s:

```
      0≠0;0≠1;1≠0;1≠1
0110
      0011≠0101
1
```

3.2.2 Logical Functions

The following table is a truth table that describes the results of logical operations:

Table 3-4
Truth Table

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

3.2.3 | or .AB – The Residue Function

Format

dyadic

Argument Types

Both arguments can be either scalars or vectors and either integer or noninteger.

Definition

Obtains the remainder or residue of a number. The residue is a unique number whose value is in the range between the value of the left argument and zero. It is obtained by adding or subtracting multiples of the left argument from the right argument. For example, for positive arguments, the remainder is obtained by dividing right arguments from left arguments. The result of a residue operation has the same sign as the sign of the left argument.

The formal definition of the residue function is as follows:

```
A|B IS B-A×⌊B÷ A+A=0
```

If the left and right arguments are equal, the residue is 0. If the left argument is 0, then the residue is equal to the value of the right argument.

If the left argument is not 0, then the residue is in the range of the left argument through 0; it may equal 0 but not equal the value of the left argument.

Examples

```
       5|8
   3
       ¯5|7
   ¯3
       7|7
   0
       7|0
   0
       0|7
   7
       2|5.8
   1.8
       5 5|8 8
   3 3
       5 5 5|2
   2 2 2
       5|2 2 2
   2 2 2
```

Related Functions

None

3.2.4 *?* – The Roll Function

Format

monadic

Argument Types

The argument is an array of positive integers.

Definition

Generates an array of independent random integers. Each element is in the range 0 or 1 (depending on ⎕*IO*) to the value of the corresponding element in the argument. There may be duplicate values. Roll also changes ⎕*RL*. See Section 4.2.17.

The term "roll" relates to the analogy between the operation performed by this function and the rolling of several dice.

Examples

```
      ?5 10 15 20 25
2 6 2 14 8
      ?5 10 15 20 25
5 6 13 2 21

      A←2 3⍴⍳6
      A
  1  2  3
  4  5  6
      ?A
  1  2  3
  2  5  4
      ?A
  1  2  2
  4  2  6
      ?A
  1  1  2
  3  3  4
```

Related Functions

Deal, Section 3.3.3

Deal differs from roll in that deal generates a set of random numbers in which no number is selected twice.

3.3 PRIMITIVE MIXED FUNCTIONS

Unlike the primitive scalar functions discussed in the previous sections, the functions presented in the following sections are considered primitive mixed functions. Scalar functions usually take scalar arguments, return scalar results, and are extended to arrays on an element-by-element basis. Primitive mixed functions, however, can take vector arguments and return scalar or vector results, or can take scalar arguments and return vector results. In expressing primitive mixed functions for arrays of higher rank, you may need to specify the particular coordinate of the array to which the function applies.

The primitive mixed functions provide the capability of extensive array manipulation.

Table 3-5 lists the primitive mixed functions available in this version of APL. The operators are also listed.

Table 3-5
Primitive Mixed Functions

Function	Section	Meaning
X,*Y*	3.3.1	Catenate *X* to *Y* along the last dimension of *X*
X/*Y*	3.3.2	*X* (logical) compression along the last dimension of *Y*
X/[*N*]*Y*	3.3.2	*X* (logical) compression along the *N*th dimension of *Y*
X⌿*Y*	3.3.2	*X* (logical) compression along the first dimension of *Y*
X?*Y*	3.3.3	Deal *X* integers selected randomly in range 1 through *Y* without duplication
X⊥*Y*	3.3.4	Decode the representation of *Y* in number system *X*
X↓*Y*	3.3.5	For *X*>0, drop first *X* elements of *Y* - for *X*<0, drop last ∣*X* elements of *Y*
X⊤*Y*	3.3.6	Encode *Y* in number system *X*
X\Y	3.3.7	*X* (logical) expansion along the last dimension *Y*
X\\[*N*]*Y*	3.3.7	*X* (logical) expansion along the *N*th dimension of *Y*
X⍀*Y*	3.3.7	*X* (logical) expansion along the first dimension of *Y*
X⍒*Y*	3.3.8	Generate an index vector such that *X*[⍒*Y*] is in descending order

Table 3-5 (Cont.)
Primitive Mixed Functions

Function	Section	Meaning
X⍋Y	3.3.9	Generate an index vector such that X[⍋Y] is in ascending order
⍳Y	3.3.10	Generate the first Y consecutive integers from current origin
X⍳Y	3.3.11	Find the first occurrence of Y in vector X
X,[N]Y	3.3.12	Laminate X to Y along the Nth dimension of X
X∊Y	3.3.13	Determine the membership of X in array Y
,Y	3.3.14	Return the ravel of Y (make Y a vector)
⌽Y	3.3.16	Reverse along the last dimension of Y
⌽[N]Y	3.3.16	Reverse along the Nth dimension of Y
⊖Y	3.3.16	Reverse along the first dimension of Y
⍴X	3.3.18	Return the shape of X
X⍴Y	3.3.15	Reshape Y to make dimension X
X⌽Y	3.3.16	Rotate by X along the last dimension of X
X⌽[N]Y	3.3.16	Rotate by X along the Nth dimension of Y
X⊖Y	3.3.16	Rotate by X along the first dimension of Y
X↑Y	3.3.19	For X>0, take first X elements of Y - for X<0, take last |X elements of Y
⍉Y	3.3.20	Transpose the dimensions of Y (for a matrix, exchange the rows and columns)
X⍉Y	3.3.21	Transpose array Y according to X

3.3.1 , – The Catenate Function

Format

dyadic

Argument Types

scalars, vectors, or arrays
Both arguments can be either numeric or character data.

Definition

Chains two scalar or vectors to form a new vector. Catenation joins constants or variables along an existing dimension. Any number of items can be catenated. The order in which values are catenated is the order in which you specify them in the APL statement. Actually, the value(s) of the argument to the right of the catenate function is appended to the value(s) of the argument to the left of the function.

The result of a catenation can be formally expressed as follows: if ρ*A* is 5 and ρ*B* is 3, then ρ*R*←*A*,*B* is 8, *R*[ι5] is *A* and *R*[5+ι3] is *B*.

You can also catenate literals. APL does not allow you to catenate numbers to characters and vice versa. If you attempt this, you will receive a 15 *DOMAIN ERROR*.

The catenate function also allows you to join multidimensional arrays along an existing coordinate as long as they have the same length over the other dimensions. You include the coordinate within square brackets along with the right argument of the catenate specification. For example, *A*,[1]*B*. The coordinate is 1 for first dimension (row), 2 for second dimension (column), and so forth.

You can also catenate constants to an array or matrix. If you do not specify the coordinate, APL assumes the highest rank of the array being catenated, that is, the last dimension. See examples.

When catenating arrays, you must follow two general rules:

1. Using the expression *A*,[*K*]*B*, if the arrays have equal rank ((ρρ*A*)=ρρ*B*), then *K* must be in ιρρ*A* and ρ*A* must equal ρ*B* except in the *K*th dimension. This is illustrated in the following:

```
      ρA
3 4 5
      ρB
3 6 5
      R←A,[2]B
      ρR
3 10 5
```

Here *A* is equal to *R*[;ι4;] and *B* to *R*[;4+ι6;].

2. If the arrays have different rank ((ρρ*A*)≠ρρ*B*) then one of the arguments must be a scalar (1=|(ρρ*A*)-ρρ*B*), and ρ*B* must equal ρ*A* without its *K*th coordinate. This is shown below:

```
      ρA
3 4 5
      ρB
4 5
      R←A,[1]B
      ρR
4 4 5
```

Here, *A* is equal to *R*[⍳3;;] and *B* to *R*[4;;].

Examples

The following example catenates two vectors to each other and to several scalar values:

```
      A←5 8 9
      B←7 8
      A,B
5 8 9 7 8

      10,A,B,12
10 5 8 9 7 8 12

      'NAME','XY'
NAMEXY

      A←2 3ρ1 2 3 4 5 6
      B←2 3ρ7 8 9 10 11 12
      A
 1  2  3
 4  5  6
      B
  7   8   9
 10  11  12

      A,[1]B
  1   2   3
  4   5   6
  7   8   9
 10  11  12

      A,[2]B
  1   2   3   7   8   9
  4   5   6  10  11  12

      A,[1]7
 1  2  3
 4  5  6
 7  7  7
      A,7
 1  2  3  7
 4  5  6  7
```

```
      X←2 3⍴8 7 3 9 4
      Y←2 3⍴0 1 2 3 4

      X,[1]Y
8  7  3
9  4  8
0  1  2
3  4  0

      X,Y
8  7  3  0  1  2
9  4  8  3  4  0
```

Related Functions

Laminate, Section 3.3.12
Ravel, Section 3.3.14

3.3.2 / and ⌿ - The Compression Function

Format

dyadic

Argument Types

The right argument can be a scalar or any array. The left argument must be the scalar argument 0 or 1, or a Boolean vector (a vector containing only 0s and 1s).

Definition

Builds a new vector or array from an old one by specifying the elements to be deleted and the elements to be preserved. For example:

```
      A←5 7 9 11 13
      B←1 1 0 1 0
      ⎕←A←B/A
5 7 11
```

Elements in *A* whose positions correspond to the positions of nonzero elements of *B* are preserved; elements corresponding to zeros in *B* are dropped. If *B* contains only 1s, all elements of *A* are preserved; if *B* contains only 0s, the result is an empty vector.

The lengths of both arguments, for example *A* and *B*, must generally be the same. However, if *A* is of length 1, it will automatically be extended to the length of *B*; if *B* is of length 1, it will be extended to the length of *A*. Thus:

```
      A←5 7 9 11 13
      B←1 1 0 1 0
      B/5
5 5 5

      1/A
5 7 9 11 13

      0/A
                                     (APL outputs a blank line)
```

The expression 0/*A* produces an empty vector because all elements of *A* are dropped.

You can also compress arrays by specifying, within square brackets, the coordinate to be compressed. (The coordinate is dependent upon the index origin, Section 4.2.11.) For a matrix, compression along the first coordinate can cause certain rows to be omitted; compression along the second coordinate can cause columns to be dropped. The result in all cases is a matrix.

If you omit the coordinate in square brackets, APL compresses the highest-ranking coordinate of the array. By specifying the special compression symbol ⌿ (.CS) without including a coordinate, APL compresses the first coordinate.

Examples

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

      1 0 1/[1]A
  1   2   3   4
  9  10  11  12
        1 0 1 0/[2]A
  1   3
  5   7
  9  11

      ⍴(0/A)
3 0

      X←2 3⍴⍳6
      X
 1  2  3
 4  5  6

      0 1 1/X
 2  3
 5  6

      1 0⌿X
 1  2  3
```

Related Functions

Expansion, Section 3.3.7

3.3.3 *?* - The Deal Function

Format

dyadic

Argument Types

Both arguments must be positive scalars or single-element vectors.

Definition

Generates a vector of integers randomly selected from the right argument vector without selecting any number more than once.

The length of the vector produced by the operation is specified by the left argument. You can set the seed of the pseudo-random-number generator with the ⎕*RL* system variable, Section 4.2.17. Everytime you use the deal function, *?*, you change ⎕*RL*.

Examples

```
      5?5
1 3 5 2 4

      5?1.0E7
6595053 6514970 5824656 4389382 7540976

      5?1.0E7
2010075 9444312 5995397 3627744 3545552

      5?1.0E7
5923563 4257710 6323814 5360300 2709926
```

Related Functions

Roll, Section 3.2.3

Unlike the roll function, deal is like dealing a number of cards from a deck with no two cards alike. Roll is like rolling several dice independently. Roll may generate duplicates but deal will not.

3.3.4 ⊥ or .DE - The Decode Function

Format

dyadic

Argument Types

scalars, vectors, or arrays

Definition

Reduces a representation in a number system to a value. It is the converse of the encode function (⊤); equivalent examples of the two functions as they operate on a quantity expressed in yards, feet, and inches, are shown below:

```
      1760 3 12⊤63
1 2 3
      1760 3 12⊥1 2 3
63
```

The expressions *A*⊤*B* and *A*⊥*B* differ only in the value included in *B*; *A* expresses the number base in both cases.

The number of elements in both arguments, for example *A* and *B*, must generally be the same; the first element in *A* expresses the base in which the first element in *B* is decoded, and so on. However, if *A* is a scalar or a single-element array, it is extended so that its length is the same as that of *B*. For example, the following expression has the effect of producing the base 10 value of the base 8 number 3777 (octal to decimal conversion).

```
      8⊥3 7 7 7
2047
```

You can also specify the decode function with multidimensional arrays. The expression *A*⊥*B* is equal to *W*+.X*B* where *W* is the weighting vector given by the expression *W*[ρ*A*] 1 and *W*[(-*N*)+ρ*A*] is equal to *A*[(-*N*)+1ρ*A*]×*W*[(-*N*)+1+ρ*A*]. The value of *A*[1] is irrelevant.

The arrays you specify as arguments must conform to the following rules using *A* and *B* as arguments:

1. *A* or *B* is a scalar.

2. The results of -1↑ρ*A* and 1↑ρ*B* are equal.

3. Either 1^ρ*A* or 1^ρ*B* equals 1.

Examples

```
        ⍝CONVERTS 3 YDS, 2 FT, 4 INCHES TO INCHES
        1 3 12 ⊥ 3 2 4
136
        ⍝IS 2.5 A ZERO OF THE POLYNOMIAL 6X*2-7X-20
        2.5⊥6 ¯7 ¯20
0
        ⍝BASE-10 EQUIVALENT OF BASE-5 NUMBER
        5⊥4 3 4
119
```

Related Functions

Encode, Section 3.3.6
Inner Product, Section 3.4.1

The decode function can be viewed as a form of the inner product operator. The following example illustrates two equivalent operations:

```
        A←1760 3 12
        B←1 2 3
        A⊥B
63
        36 12 1+.×B
63
```

3.3.5 ↓ or .DA - The Drop Function

Format

dyadic

Argument Types

The right argument must be an array. In most cases, the left argument must be a scalar; it can be a vector if the right argument is a multidimensional array.

Definition

Builds a new vector or array by dropping a specified number of elements from an existing array. For example:

```
      ⎕←V←⍳5
1 2 3 4 5
      ⎕←X←2↓V
3 4 5
```

The expression drops the first two elements of *V* and forms a new vector with the remaining elements. If the value of the scalar is greater than the number of elements in *V*, then the result is the null vector.

The drop function handles negative scalar values by dropping the elements from the end of the array instead of from the beginning. For example:

```
      ¯2↓⍳5
1 2 3
```

You can also specify multidimensional arrays with the drop function. In this case, the left argument must be a vector containing one element for each dimension of the array. In the expression, *S*↓*V*, the value of *S*[1] indicates the number of elements to be dropped along the first coordinate of *V*, and so on.

Examples

```
      ⎕←A←3 5⍴⍳15
 1   2   3   4   5
 6   7   8   9  10
11  12  13  14  15

      ¯1 3↓A
 4   5
 9  10

      1 3↓A
 9  10
14  15
```

Related Functions

Take, Section 3.3.19

3.3.6 ⊤ or .EN - The Encode Function

Format

dyadic

Argument Types

The right argument identifies the scalar or array to be translated. The left argument is a vector that represents the number base in which the value is to be expressed. The vector contains one element for each column representation.

Definition

Represents a scalar or an array in any number system. For example, to encode the decimal value 7 in four columns of binary representation, the following expression can be specified:

```
      2 2 2 2⊤7
0 1 1 1
```

You can also specify mixed bases for the number to be represented. The encode function can express some number of inches in miles, yards, feet, and inches; or some number of milliseconds in days, hours, minutes, seconds, and milliseconds. The following examples illustrate these two situations:

```
      ⍝MILES, YARDS, FEET, INCHES
      0 1760 3 12⊤273125
4 546 2 5
      ⍝DAYS, HOURS, MINUTES, SECONDS, MILLISECONDS
      0 24 60 60 1000⊤719732523
8 7 55 32 523
```

In the expression *A*⊤*B*, *A* can be considered as the representation rule to be applied by *B*. Each element of the vector *A* is defined in terms of the element immediately to its left. Thus, in encoding a number as miles, yards, feet, and inches, the following elements are specified from right to left:

1. 12 inches in 1 foot

2. 3 feet in 1 yard

3. 1760 yards in 1 mile

In the previous example, a miles specification is not defined in terms of another quantity, so 0 is printed in the miles column.

The following examples of base 3 conversions demonstrate the specification of different numbers of columns in the rule vector and illustrate the way in which negative numbers are encoded:

```
      3 3 3⊤17
1 2 2

      3 3⊤17
2 2

      3 3 3⊤¯17
1 0 1
```

Another useful application of encode is to return the integer and fractional portions of a number:

```
      X←823.7513
      0 1⊤X
823 0.7513
```

You can also specify the encode function with multidimensional arrays. The shape of the result of the expression *R*←*A*⊤*B* is always (⍴*A*),⍴*B*.

Examples

```
      ⎕←A←⍉3 2⍴2 3
  2  2  2
  3  3  3
      B←5 2
      ⎕←R←A⊤B
  1  0
  1  0
  1  0

  2  2
  2  2
  2  2
      ⍴B
2
      ⍴A
2 3
      ⍴R
2 3 2

      C
  865  429
  103  692
```

```
      10 10 10⊤C
  8  4
  1  6

  6  2
  0  9

  5  9
  3  2

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

Related Functions

Decode, Section 3.3.4

3.3.7 \ and ⍀ - The Expansion Function

Format

dyadic

Argument Types

The right argument can be any array. The left argument must be a scalar value 0 or 1 or a Boolean vector, a vector containing only 0s and 1s. If the right argument is a character vector, spaces are used instead of 0s. The number of 1s in the Boolean vector must generally be the same as the number of values in the array included as the right argument.

Definition

Builds a new vector or array by expanding the elements of another vector into a new format specified by the function. For example:

```
      A←1 2 3
      V←1 0 1 0 1
      V\A
1 0 2 0 3
      V\'APL'
A P L
```

The function expands the elements of *A* into the format specified by *V*. The values of *A* are inserted in positions corresponding to the occurrences of 1s in *V*. For numeric values, zeros are inserted in positions corresponding to 0s in the Boolean vector. If the right argument is a character string, as in the second example above, spaces are used instead of zeros.

A scalar Boolean value as the right argument is extended as in the following example:

```
      1 0 1\5
5 0 5
```

You can also expand multidimensional arrays along a particular coordinate. (The coordinate is dependent upon the index origin, Section 4.2.11.) You include the coordinate within square brackets. The syntax is the same as the compression function, Section 3.3.2. If you omit the coordinate, APL expands along the last coordinate of the array. To specify expansion along the first coordinate, use the special symbol ⍀, or type .CB.

Examples

```
      ⎕←A←2 3⍴⍳6
 1  2  3
 4  5  6
      1 0 1\[1]A
 1  2  3
 0  0  0
 4  5  6
      1 0 1 1\[2]A
 1  0  2  3
 4  0  5  6
      0 0 0\⍳0
0 0 0
      ⎕←A←0 0 0\''
                                   (APL outputs a blank line)
      X
⋆THISISAN
EXPANSION
EXAMPLE⋆⋆
      ⍴X
3 9
      V←1 1 1 1 1 0 1 1 0 1 1
      V\X
⋆THIS IS AN
EXPAN SI ON
EXAMP LE ⋆⋆
      1 0 1 1⍀X
⋆THISISAN

EXPANSION
EXAMPLE⋆⋆
```

Related Functions

Compression, Section 3.3.2

3.3.8 ⍒ or .GD - The Grade Down Function

Format

monadic

Argument Types

The argument can be a vector or a matrix.

Definition

Creates an index to sort a vector or matrix in descending order. The ⍒ function creates a permutation vector that APL can use to sort the original vector. Duplicate values are ordered by their relative positions in the original vector. You can also reorder character arrays with ⍒. The grade down function does not use fuzz in performing comparisons.

The symbol ⍒ is formed by overstriking the del (∇) with the residue (|).

Examples

```
      A←2 9 7 4 3 10 3
      ⎕←B←⍒A
6 2 3 4 5 7 1
      A[B]
10 9 7 4 3 3 2
      A[⍒A←'MANUAL']
UNMLAA
```

Related Functions

Grade Up, Section 3.3.9

3.3.9 ⍋ or .GU - The Grade Up Function

Format

monadic

Argument Types

The argument can be a vector or a matrix.

Definition

Creates an index to sort a vector or matrix in ascending order. If two or more elements of a vector or matrix have the same value, the order of the elements is determined by their relative positions in the original array. (Fuzz is not used in comparing the elements.)

The ⍋ symbol is formed by overstriking the delta (∆) with the residue (|).

Grade up does not actually sort the vector. It creates a permutation vector of the index numbers of the elements. This vector is then used to sort the original vector.

If the array to be sorted is a matrix, the simplest operations cause each row of the matrix to be treated as a string. The result of the grade up operation is a vector whose length is equal to the number of rows in the matrix.

You can cause a matrix to be sorted by rows; and by subscripting the function, you can also sort on the basis of columns. For matrices, the expression ⍋*M* is equal to ⍋[2]*M*.

Examples

```
      A←2 9 7 4 3 10 3
      ⎕←B←⍋A
1 5 7 4 3 2 6
      A[B]
2 3 3 4 7 9 10

      A
STEVE
SAM
STAN
      ⍴A
3 5
      ⍋A
2 3 1
      A[⍋A;]
SAM
STAN
STEVE
```

```
      B
  3  2  1  5  0
  3  1  9  7  0
  3  2  0  8  0

      ⍴B
3 5
      ⍋B
2 3 1
      B[⍋B;]
  3  1  9  7  0
  3  2  0  8  0
  3  2  1  5  0
```

Related Functions

Grade Down, Section 3.3.8

3.3.10 ⍳ or .IO - The Index Generator Function

Format

monadic

Argument Types

The argument can be a nonnegative integer scalar or a 1-element array.

Definition

Generates a number of consecutive integers equal to the value specified as the argument, starting from the value of the index origin, Section 2.4.2.

The expression ⍳*N* generates a vector containing *N* components. If the index origin is set to 1, these components have values 1 through *N*. If the index origin is set to 0, then the resulting vector has values 0 through *N*-1.

The index origin default is 1 in a clear workspace, but this setting can be changed with the)*ORIGIN* command (Section 5.5.4) or the ⎕*IO* variable (Section 4.2.11).

Examples

```
      ⎕←A←⍳4
1 2 3 4
      ⍴A
4
      ⍝POWERS OF 2
      2*⍳12
2 4 8 16 32 64 128 256 512 1024 2048 4096
      ⍝OFTEN USED WITH RHO
      ⍴⍳50
50
      X←7 1 3 4
      ⍳⍴X
1 2 3 4
      ⍝GENERATES A NULL VECTOR
      ⍳0
                              (APL outputs a blank line)
      ⍴⍳0
0
```

Related Functions

Index of Section 3.3.11.
Reshape and Shape, Sections 3.3.15 and 3.3.18

3.3.11 ⍳ or .IO - The Index Of Function

Format

dyadic

Argument Types

The left argument must be a vector. The right argument can be any scalar or array.

Definition

Returns the index in the left argument of the first occurrence of the value in the right argument. The result of a dyadic iota operation always has the same shape as the right argument. That is, the result returns an index for each of the values in the right argument.

If the value is not located in the vector specified as the left argument, APL reports a value equal to the number of values in the vector plus 1, ((⍴*A*)+1).

The right argument need not be a single-element array; it may have many elements and many dimensions. The right argument can also contain literal characters.

The result of a dyadic iota expression, for example *X*←*B*⍳*A*, always has the same shape as the right argument, formally ⍴*X* is the same as ⍴*A*. If *A* is a matrix, the correspondence between *A* and *X* can be expressed as *X*[*I*;*J*] is the smallest *K* such that *A*[*I*;*J*] is equal to *B*[*K*].

Examples

```
      B
4 9 6 8
      A
6
      B⍳A
3
      X←B⍳⌈/B
      ⎕←A←X⍳⌈/X
1
      B[X]
9
      B[B⍳⌈/B]
9
      B←0 1 2 3 4 5 6 7 8 9
      A←3 2⍴6 5 3 2 0 9
      X←B⍳A
      B
0 1 2 3 4 5 6 7 8 9
```

```
      A
 6  5
 3  2
 0  9
      X
  7   6
  4   3
  1  10

      'ABCDEFGH'⍳'HEADED'
8 5 1 4 5 4
      B←5 4 2 3 7 8
      A←2 2⍴⍳4
      B⍳A
  7  3
  4  2


       ⍝ 'A'=2 IS LEGAL,  SO IS ⍳ WITH CHAR AND NUMERIC ARGS

       'AAA' ⍳ 2
 4
```

Related Functions
     Index Generator, Section 3.3.10

3.3.12 , - The Laminate Function

Format

dyadic

Argument Types

Both arguments can be scalars, vectors, or arrays.

Definition

Joins scalars, vectors, or arrays along a new dimension. The syntax is the same as the catenate function, Section 3.3.1. However, the coordinate specification ([]) is usually a fraction to indicate a position between existing coordinates in which the new coordinate is to be placed. (The coordinate is dependent upon the index origin, Section 4.2.11.)

If two arguments in a laminate operation do not have the same dimensions, then at least one of them must be a scalar value.

Examples

```
      ⎕←X←'ABC',[0.5]'DEF'
ABC
DEF
      ρX
2 3
      ⍝CREATES A NEW DIMENSION BEFORE
      ⍝THE FIRST ONE;ADDS A ROW

      ⎕←X←'ABC',[1.3]'DEF'
AD
BE
CF
      ρX
3 2
      ⍝CREATES A NEW DIMENSION AFTER
      ⍝THE FIRST ONE;ADDS A COLUMN

      ⎕←D←3 2ρ'UVWXYZ'
UV
WX
YZ
      A←3 2ρ'ABCDEF'
      R←A,[.2]D
      ρR
2 3 2
      R←A,[1.9]D
      ρR
3 2 2
      R←A,[2.3]D
      ρR
3 2 2
      R←A,[.5]'Z'
      ρR
2 3 2
      ⎕←R←'X',[1.5]A
XX
AB

XX
CD

XX
EF
      ρR
3 2 2
      'Y',[2.5]A
YA
YB

YC
YD

YE
YF
```

Related Functions

Catenate Function, Section 3.3.1
Ravel Function, Section 3.3.14

3.3.13 ∊ or .EP - The Membership Function

Format

dyadic

Argument Types

Both arguments can be arrays of any dimension; the left argument contains the elements by which membership in the right argument is determined. The arrays need not have the same rank.

Definition

Determines whether or not particular elements of one array occur as elements of another array. The result is a Boolean array whose shape is the same as that of the left argument.

The result consists of only 0s and 1s; a 1 indicates that the corresponding element in the left array is present in the right array, 0 indicates that it is not present.

Examples

```
      ⎕←A←'ABCDEFG'∊'HEADED'
1 0 0 1 1 0 0
      A/'ABCDEFG'
ADE
```

The compression function /, is helpful here in identifying the particular characters that are members of the vector.

```
      A←2 3⍴7 8 2 4 6 9
      A∊⍳6
 0 0 1
 1 1 0
      3 4∊'34'
0 0
      3 4∊⍳0
0 0


       ⍝ 'A'=2 IS LEGAL,  SO IS ∊ WITH CHAR AND NUMERIC ARGS
       'AAA' ∊ 2
 0 0 0
```

Related Functions

Index of, Section 3.3.11

3.3.14 , - The Ravel Function

Format

monadic

Argument Types

The argument can be any scalar or array.

Definition

Produces a vector from any scalar or array. The vector produced has the same length as the original array. The elements of the array are preserved in the resulting vector in row order. If the argument is a scalar, then the ravel function produces a vector containing one element.

Examples

```
      A
  1  2  3
  4  5  6
      ρA
2 3
      ⎕←B←,A
1 2 3 4 5 6
      ρB
6
      ⎕←A←2 3 4ρ⍳90
   1   2   3   4
   5   6   7   8
   9  10  11  12

  13  14  15  16
  17  18  19  20
  21  22  23  24
      ,A
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
      ρ,A
24
      ρA
2 3 4
```

Note the difference in the shape of a scalar and the shape of a scalar to which the ravel function has been applied

```
      ρ4
                             (APL outputs a blank line)
      ρ,4
1
```

Related Functions

Catenate, Section 3.3.1
Laminate, Section 3.3.12

3.3.15 ρ or .RO - The Reshape Function

Format

dyadic

Argument Types

The left argument can be a scalar or a vector. The right argument can be a numeric constant, or a literal character, or the name of an existing array. A literal array can be constructed by including a character string as the right argument and enclosing the string within single quotation marks.

Definition

Constructs an array or reshapes an existing one. The left argument specifies the shape of the array; the right argument specifies the values to be assigned to each element of the array. The shape of the array describes both the number of dimensions of the array and the number of elements in each dimension. The values are placed in the array in row order; that is, the first value is placed in row 1 column 1; the second value is placed in row 1 column 2; the third value is placed in row 1 column 3, and so on.

If the left argument is a single value, a 1-dimensional array is created.

The array being reshaped need not have the same number of values as the array from which the values are taken. If the right argument has too many values, the excess values are ignored from the right. If there are too few arguments, the values are duplicated from the right.

Any number of array elements can be specified in a reshape operation as long as the number is not negative or fractional and does not generate a vector or array too large for your workspace.

Examples

```
      3ρ5
5 5 5
      9ρ'ABC'
ABCABCABC

      X←8 9 7 4
      Y←2 2ρX
      Y
  8  9
  7  4

      A←1 2 3 4
      2 5ρA
  1  2  3  4  1
  2  3  4  1  2

      3ρA
1 2 3
```

```
      ⎕←A←2 2⍴⍳4
1  2
3  4
```

The following expressions each generate a null vector:

```
          ⍝SHAPE WITH A SCALAR
          A←1
          ⍴A                              (APL outputs a blank line)

          0⍴''                            (APL outputs a blank line)

          0⍴0                             (APL outputs a blank line)

          ⍳0                              (APL outputs a blank line)

          ⎕←A←3 0 5 4 ⍴A

          ⍴A
3 0 5 4
```

Related Functions

Shape, Section 3.3.18

3.3.16 ⌽ or .RV and ⊖ or .CR - The Reverse Function

Format

monadic

Argument Types

The argument can be a vector or array.

Definition

Reverses a vector or the elements of one coordinate (last dimension) of an array. It changes the order of the elements, not their dimension.

To specify the coordinate to be reversed, include it in square brackets. (The coordinate is dependent upon the index origin, Section 4.2.11.) The default is the highest coordinate (last dimension) of the array. The special character ⊖ (.CR) reverses the first coordinate of the array.

The reverse is formed by overstriking the circle ○ with the residue function |. The ⊖ character is formed by overstriking the circle (○) with the minus (-).

Examples

```
      ⎕←⍝←2 4⍴⍳8
1  2  3  4
5  6  7  8
      ⌽[1]⍝
5  6  7  8
1  2  3  4
      ⌽[2]⍝
4  3  2  1
8  7  6  5
      ⊖⍝
5  6  7  8
1  2  3  4
```

The following example reverses a matrix in both dimensions simultaneously:

```
      ⎕←X←2 3⍴⍳6
1  2  3
4  5  6
      ⌽⌽[1]X
6  5  4
3  2  1
```

Reverse is not the same as transpose:

```
      ⍉X
1  4
2  5
3  6
```

Related Functions

Rotate, Section 3.3.17

3.3.17 ⌽ or .RV - The Rotate Function

Format

dyadic

Argument Types

The left argument can be a scalar or a vector. The right argument can be any array.

If a vector is being rotated, the left argument must be a scalar or a 1-element vector. If a multidimensional array is being rotated, the left argument must be a scalar, a single-element vector, or a vector whose elements correspond to dimensions of the array being rotated, with the dimension being rotated omitted from the vector.

The rotate function is formed by overstriking the circle ○ with the residue function |.

Definition

Rotates an array by a specified number of places. A positive rotation causes a left shift; a negative rotation causes a right shift.

You rotate a multidimensional array by specifying the coordinate along which rotation is to take place. (The coordinate is dependent upon the index origin, Section 4.2.11.) The default is the highest coordinate of the array.

To specify the first coordinate, use the special symbol ⊖ (.CR) which is formed by overstriking the circle ○ with a minus -.

Examples

```
      3⌽⍳5
4 5 1 2 3
      ¯3⌽⍳5
3 4 5 1 2
      X←3 4⍴'ABCDEFGHIJKL'

ABCD
EFGH
IJKL
      0 1 2⌽X
ABCD
FGHE
KLIJ
      1 1 2 3⌽[1]X
EFKD
IJCH
ABGL
      ⎕←A←3 5⍴'ABCDEFGHIJKLMNO'
ABCDE
FGHIJ
KLMNO
```

```
      1 ¯1 2 2 2⊖A
FLMNO
KBCDE
AGHIJ
      5 2 ¯1⌽A
ABCDE
HIJFG
OKLMN
```

Related Functions

Reverse, Section 3.3.16

3.3.18 ρ or .RO - The Shape Function

Format

monadic

Argument Types

The argument can be scalar, vector, or array.

Definition

Returns the shape of the argument, that is, it returns the length of a vector or the dimensions of an array. For example, if the argument *B* is a character vector consisting of '*ABCDEF*', then the following expression returns the number of characters in the array:

```
      B←'ABCDEF'
      B
ABCDEF
      ρB
6
```

If the argument is a matrix, rho returns the number of rows and columns it has. For example:

```
      A←5 6ρι10
      ρA
5 6
```

If the argument is a scalar and not a vector or array, then the rho of that scalar is a null vector, a vector of length zero. APL outputs a blank line in response to the shape operation with a scalar. Two shape functions (ρρ) return the number of dimensions (rank) of the arguments as follows:

Argument	ρρ*K*
scalar	0
1-dimensional array	1
2-dimensional array	2
3-dimensional array	3
and so on.	

This effect is the result of the fact that ρ*K* is a vector containing one element for each dimension of *K*, so its ρ (ρρ*K*) is a 1-element vector consisting of the number of dimensions of *K*.

Examples

```
           A
      1    2    3    4    5    6
      7    8    9   10    1    2
      3    4    5    6    7    8
      9   10    1    2    3    4
      5    6    7    8    9   10
           ⍴A
   5  6
           ⍴⍴A
   2
           K←3
           ⍴K
                                        (APL outputs a blank line)
```

Related Functions

Reshape, Section 3.3.15

3.3.19 ↑ or ^ - The Take Function

Format

dyadic

Argument Types

The left argument can be a scalar. However, if the right argument is a multidimensional array, the left argument must be a vector containing one element for each dimension of the array. The right argument can be any array.

Definition

Builds a new vector or array by taking a specified number of elements from an existing array. If the value of the scalar is greater than the number of elements in the vector, the resulting vector is extended so that its length is the value of the scalar. Zeros extend numeric vectors, and spaces extend character vectors.

In the expression *R*←*S*↑*V*, if *S* is positive, then *R* consists of the first *S* elements of *V*. If *S* is negative, then *R* contains the last |*S* elements of *V*. If |*S* is greater than the number of elements in *V*, then zeros or blanks are inserted in *R* before the values of *V*.

Examples

```
      2↑⍳3
1 2
      4↑⍳3
1 2 3 0
      15↑'APLSF';8↑5
APLSF          5 0 0 0 0 0 0 0
      ¯6↑12 24 35 48
0 0 12 24 35 48
      ¯5↑⍳3
0 0 1 2 3
      5↑⍳3
1 2 3 0 0
      ¯20↑'TEST'
                TEST
```

Related Functions

Drop, Section 3.3.5

3.3.20 ⍉ or .TR - The Monadic Transpose Function

Format

monadic

Argument Types

The argument can be a matrix or higher-dimensional array.

Definition

Transposes the dimensions of an array. For a matrix, it exchanges rows and columns. If you use a vector as the argument, it will have no effect. For example:

```
      A←1 2 3 4 5
      ⍉A
1 2 3 4 5
```

To form the ⍉ symbol, overstrike the circle ○ with the slash \.

Examples

```
      ⎕←A←2 3⍴⍳6
  1  2  3
  4  5  6
      ⎕←C←⍉A
  1  4
  2  5
  3  6
      ⍴⍉A
3 2
      ⎕←B←2 3 4⍴⍳8
  1  2  3  4
  5  6  7  8
  1  2  3  4

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

```
          ⍉E
    1  5
    5  1
    1  5

    2  6
    6  2
    2  6

    3  7
    7  3
    3  7

    4  8
    8  4
    4  8

           ⍴⍉E
4  3  2
```

Related Functions

Dyadic Transpose, Section 3.3.21

3.3.21 ⍉ or .TR - The Dyadic Transpose Function

Format

dyadic

Argument Types

The left argument must be a vector containing one element for each of the dimensions of the array to be transposed. The right argument can be an array. The shape of the vector expresses the rank of the right argument. For example, in the expression *V*⍉*A*, the rank of the right argument can be expressed as: ρ*V* which must be equal to ρρ*A*. Thus, *V* must have two elements if *A* is a matrix, three if *A* is a 3-dimensional array, and so on.

If the rank of the array is 3, then valid values for the left-argument vector can be 1 1 1, 1 2 1, 1 3 2, 3 1 2, but not 1 3 1 (2 is missing).

Definition

Permutes the coordinates of an array. The following table lists transpositions for a variety of arrays:

Table 3-6
Transpose Definitions

Expression	Shape of *R*	Definition
R←1⍉*V*	ρ*V*	*R*←*V*
R←1 2⍉ *M*	ρ*M*	ρ←*M*
R←2 1⍉*M*	(ρ*M*)[2 1]	*R*[*I*;*J*]←*M*[*J*;*I*]
R←1 1⍉*M*	⌊/ρ*M*	*R*[*I*]←*M*[*I*;*I*]
R←1 2 3⍉*A*	ρ*A*	*R*←*A*
R←1 3 2⍉*A*	(ρ*A*)[1 3 2]	*R*[*I*;*J*;*K*]←*A*[*I*;*K*;*J*]
R←2 3 1⍉*A*	(ρ*A*)[3 1 2]	*R*[*I*;*J*;*K*]←*A*[*J*;*K*;*I*]
R←3 1 2⍉*A*	(ρ*A*)[2 3 1]	*R*[*I*;*J*;*K*]←*A*[*K*;*I*;*J*]
R←1 1 2⍉*A*	(⌊/(ρ*A*)[1 2]),(ρ*A*)[3]	*R*[*I*;*J*]←*A*[*I*;*I*;*J*]
R←1 2 1⍉*A*	(⌊/(ρ*A*)[1 3]),(ρ*A*)[2]	*R*[*I*;*J*]←*A*[*I*;*J*;*I*]
R←2 1 1⍉*A*	(⌊/(ρ*A*)[2 3]),(ρ*A*)[1]	*R*[*I*;*J*]←*A*[*J*;*I*;*I*]
R←1 1 1⍉*A*	⌊/ρ*A*	*R*[*I*]←*A*[*I*;*I*;*I*]

Examples

```
       ⎕←A←2 3⍴⍳6
 1  2  3
 4  5  6
       ⍝DYADIC SOMETIMES SAME AS MONADIC
       ⍉A
 1  4
 2  5
 3  6
       2 1⍉A
 1  4
 2  5
 3  6
       A←2 3⍴⍳6
       A
 1  2  3
 4  5  6

       A←2 3 4⍴⍳6
       A
 1  2  3  4
 5  6  1  2
 3  4  5  6

 1  2  3  4
 5  6  1  2
 3  4  5  6

       2 1 1⍉A
 1  1
 6  6
 5  5
```

Related Functions

Monadic Transpose, Section 3.3.20

3.4 EXTENDED FUNCTIONS

APLSF provides extended functions to perform a variety of matrix operations and to aid in data formatting. These functions are:

1. The Matrix Inverse and Matrix Divide functions ⌹, Sections 3.4.1 and 3.4.2
2. The Execute or Unquote functions ⍎ and ⊥, Section 3.4.3
3. The Extended Execute function ⍎, Section 3.4.4
4. The Dollar Format function $, Section 3.4.5
5. The Monadic and Dyadic Format functions ⍕, Sections 3.4.6 and 3.4.7
6. The Quote function ⊤, Section 3.4.8

3.4.1 ⌹ or .DQ - The Matrix Inverse Function (Quad-Divide)

Format

monadic

Argument Types

The argument can be a scalar, a vector, or a matrix.

Definition

Inverts a matrix to facilitate matrix division and a variety of other matrix operations.

The domino symbol is formed by overstriking the quad character (⎕) with the division character (÷).

Examples

```
      ⎕←A←÷(⍳3)∘.+¯1+⍳3
1              0.5            0.3333333333
0.5            0.3333333333   0.25
0.3333333333   0.25           0.2
      ⎕←X←⌹A
  9             ¯36              30
¯36             192            ¯180
 30            ¯180             180
      X×A
  9            ¯18            10
¯18             64           ¯45
 10            ¯45            36
```

Related Functions

Matrix Divide Section 3.4.2. The monadic expression ⌹*X* is equal to the dyadic *I*⌹*X*, where *I* is an identity matrix whose order can be described as 1↑⍴*X*. If the argument of the matrix inverse is a scalar, the expression ⌹*X* is equal to ÷*X*.

3.4.2 ⌹ or .DQ - The Matrix Divide Function (Quad-Divide)

Format

dyadic

Argument Types

Both arguments can be scalars, vectors, or matrices.

Definition

Performs more complicated matrix operations than the inversions described in Section 3.4.1. In the expression *X*⌹*Y*, *X* and *Y* must conform to the following:

1. *Y* must have a rank of 2 or less.

2. If the dimensions of *Y* are *M* by *N*, then *M*≥*N*.

3. *X* must have a rank of 2 or less and (1↑⍴*Y*)=1↑⍴*X*

This implies that matrices *X* and *Y* have the same number of rows, and the columns of *Y* are linearly independent. If *Z*←*X*⌹*Y*, then ⍴⍴*Z* is the same as ⍴⍴*X* and +\((*Y*+.×*Z*)-*X*)*2 is minimized (least squares solution).

The matrix divide treats scalar arguments as matrices containing one row and one column. The expression *X*⌹*Y* is equal to scalar division *X*÷*Y*, except that the operation 0⌹0 produces an error condition. If the arguments are vectors, they are treated as matrices with a single column. If *I* is an identity matrix of the same dimension as *X*, then ⌹*X* is equal to *I*⌹*X*.

Examples

The following example illustrates the use of the matrix division function in solving these linear equations:

3A+B=9
2A-B=1

In the expression *X*⌹*Y*, *Y* is a matrix whose values are the coefficients of the equations, and *X* is a vector containing the values 9 and 1.

```
      X←9 1
      Y←2 2⍴3 1 2 ¯1
      X⌹Y
2 3
```

The result is a vector in which the first element is the value of *A* in the linear equations, and the second is the value of *B*.

The following examples illustrate the use of the matrix divide, including a least squares solution:

```
      ⎕←A←(2 1⍴2 5),1
  2  1
  5  1
      B←10 19
      ⍴X←B⌹A
2
      A+.×X
10 19
      ⎕←A←(5 1⍴⍳5),1
  1  1
  2  1
  3  1
  4  1
  5  1
      B←2.001 2.998 4.002 4.997 6.01
      ⎕←X←B⌹A
1.0017 0.9965
      B←A+.×X
      ⎕←X←⌹A
 ¯2.000000000E¯1  ¯1.000000000E¯1   2.923875822E¯20
        1.000000000E¯1   2.000000000E¯1
  8.000000000E¯1   5.000000000E¯1   2.000000000E¯1
       ¯1.000000000E¯1  ¯4.000000000E¯1
      X+.×A
  1.000000000E0    5.421010862E¯20
 ¯1.301042607E¯18  1.000000000E0
```

Related Functions

Matrix Inverse, Section 3.4.1

3.4.3 ϵ or ⊥ .EP or .DE - The Execute Function or Unquote Function

Format

monadic

Argument Types

The argument can be a scalar or a vector. If the scalar is numeric, the value of ϵ*A* is equal to *A*. If the scalar or vector is a literal, APL evaluates it exactly as quad input from the terminal would be evaluated.

Definition

Executes a character string as an APL statement. The scalar or vector included as the right argument of the function is evaluated as the character string to be executed by APL. The ϵ and ⊥ can be used interchangeably to indicate the execute function.

APL treats carriage return/line feeds in the argument as statement separators, just as they would be if they were input from the terminal, so multiple lines are allowed. The result of the expression *R*←ϵ*A* is the value of the last statement evaluated in *A*. If the last statement has no value, *R* is a null vector.

Errors encountered in the character string processed by the execute function are handled exactly as if they occurred in statements entered from the terminal. If an error is encountered while evaluating the execute string, an error message is output, and the segment of the execute string currently being evaluated is displayed. No further evaluation of the string is performed. The ϵ*A* returns a null array whose shape is 0 *E*, where *E* is a number indicating the error that was encountered. Appendix A contains a complete description of all APL error conditions.

The execute function is also known as the unquote function, because it strips quotes from the value entered as its argument. Other uses of this function include:

1. Function definition (character-editing commands are not permitted)

2. Conversion of vectors of characters representing numeric constants into numeric values

3. Passing an unevaluated APL name to a function. (The argument can be evaluated with ϵ inside the function.)

Examples

The following examples illustrate the use of ϵ in function definition, system command execution, and APL statement evaluation:

```
        A←'∇Z←F
Z←B+3
Z←Z×Z
∇'
        B←4
        ⍝F IS NOT DEFINED
        F
  11 VALUE ERROR
         F
          ∧
        C←εA
        ⍴C
0
        ⍝F IS NOW DEFINED
        F
49
        C←ε')FNS'
        C
F
        D←ε'5+4
3+2
6'
9
5
        D
6
        E
 11 VALUE ERROR
         E
          ∧
        E←ε'5+5
3+2,
4'
10
  7 ε SYNTAX ERROR
         3+2,
             ∧
        ⍴E
0 7


      ε')ERASE',ε')VARS'

      ⍝ALL VARIABLES ARE NOW ERASED
      E
11 VALUE ERROR
       E
        ∧
      )WSID THISWS
WAS CLEAR WS
```

```
      ∇H
[1]   'THIS IS HARD TO BELIEVE'
[2]   Z←⍎')SAVE THISWS'
[3]   'WHEN LOADED, EXECUTION RESUMES AFTER EXECUTE
AUTOMATICALLY'
[4]   ∇
      H
THIS IS HARD TO BELIEVE
WHEN LOADED, EXECUTION RESUMES AFTER EXECUTE
AUTOMATICALLY
      )LOAD THISWS
WHEN LOADED, EXECUTION RESUMES AFTER EXECUTE
AUTOMATICALLY

      ⍝THE NEXT EXPRESSION DOES NOT PRINT A VALUE
      ⍎'A←5'

      ⍝THE NEXT ONE DOES
      ⎕←⍎'A←5'
5
      R←⍎''
      ⍴R
0
```

The last example illustrates that the execute function always returns a value. Because, in this case, there is no value expressed in the character string, the value of the operation is simply a null vector. Similarly, if the character string contains a branch (→), the execute function does not transfer control but returns the null vector.

Note that you can use *)ECHO (OFF)*,Section 5.5.2, to suppress the error message from ⍎.

Related Functions

Extended Execute, Section 3.4.4

3.4.4 ⍎ or .XQ - The Extended Execute Function

Format

monadic

Argument Types

The argument can be a scalar or a vector.

Definition

Processes system commands and supports the entry of multiple lines. The execute symbol ⍎ is formed by overstriking the decode or unquote character ⊥ with the jot character ∘.

The ⍎ function is very similar to the ∊ or ⊥ functions. However, there are two major differences. If an error is encountered in the character string being executed, ⍎ does not return a null vector indicating the type of error. Instead, ⍎ generates an error message for the line on which the actual execute occurred.

The second major difference is that if the ⍎ character string contains a branch (→), control passes to the specified function line.

Examples

```
      ∇F A
[1]   B←⍎A
[2]   ∇
      F '3,'
  7 ⍎ SYNTAX ERROR
       3,
         ^
 25 EXECUTE ERROR
F[1]    B←⍎A
         ^
      )SI
F[1]    *
      B
 11 VALUE ERROR
        B

        ^
      F ' '

 11 VALUE ERROR
F[1]    B←⍎A
         ^
```

In the last example, a value is required in the execute string, but none is included. APL generates an error message and suspends function execution.

Related Functions

Execute or Unquote, Section 3.4.3

3.4.5 *$* - The Dollar Format Function

Format

dyadic

Argument Types

The right argument can be one or more scalars, vectors, or multidimensional arrays containing numeric or character fields to be formatted. The left argument is a character vector containing one or more format fields describing the type of formatting to be performed on the specified fields. The left argument is enclosed in single quotation marks.

Table 3-7 summarizes the syntax of the format fields.

Table 3-7
Format Fields

Format	Meaning
'*M*A*W*'	Character data - cannot be used for numeric values
'*M*E*W*.d'	Floating-point numeric data with exponent
'*MQ*F*W*.d'	Fixed-point numeric data
'*MQ*I*W*'	Integer numeric data with automatic rounding
'*M*X*W*'	Blanks inserted in edited line
'*M*⍞text⍞'	Literal text inserted in edited line

where

M	is an optional repetition factor (number of values to which the format is to be applied).
W	is the width of the field.
d	is the number of decimal positions.
Q	is any number of qualifiers (see Table 3-8).

The lamp character (⍝), formed by overstriking the down union (∩) with the jot (∘), can be used instead of the quote-quad (⍞). On non-APL-keyboard terminals, .QQ replaces ⍞ and " replaces ⍝.

Table 3-8
Qualifiers

Qualifier	Meaning
B	Blank field if value is 0
C	Insert commas
L	Left justify
Z	Fill with zeros
M⍞text⍞	Insert text left of negative result
N⍞text⍞	Insert text right of negative result
P⍞text⍞	Insert text left of nonnegative result
Q⍞text⍞	Insert text right of nonnegative result
R⍞text⍞	Insert text in background

If more than one format field is included in the left argument, the fields must be separated by commas. Successive fields apply to successive vectors or arrays represented by the right argument of the function. If you include a repetition factor in one of the format fields, this factor indicates the number of vectors to which that format is to be applied.

Definition

The result of a format operation is one or more lines of edited text. Each resulting line consists of one edited row of each array in the right argument, where each vector (*V*) is treated as an array of dimensions (⍴*V*) by 1. The total number of lines produced by a format is equal to the longest column in the array contained in the right argument. The columns of values with shorter columns are extended with blanks.

As many as 18 significant digits can be specified in a format statement. A format field that requests more than 18 significant digits will cause digit positions to the right of the decimal point to be filled with blanks, and digit positions to the left of the decimal point to be filled with underscores. A minus sign is output on non-APL-keyboard terminals.

A format field that does not specify sufficient room for all significant digits plus any inserted characters causes the entire field to be filled with stars on APL terminals and asterisks on TTYs.

If a format expression produces a very large matrix that is not assigned explicitly to a variable name, APL saves storage in the workspace by displaying each line of the matrix as it is formatted and not saving the results.

As in other languages that support format specification similar to this APL function, parentheses can be used to repeat groups of fields. They can be nested to three levels.

Examples

```
      'I4'$(⍳5;10+⍳6)
   1  11
   2  12
   3  13
   4  14
   5  15
      16
      'ZF4.1'$1 2 3
01.0
02.0
03.0
      A←'F4.2,3(I2,E8.1),A1'
      ⍝IS EQUAL TO
      A←'F4.2,I2,E8.1,I2,E8.1,I2,E8.1,A1'
      'I5'$3 ¯7 4.7 4.2
    3
   ¯7
    5
    4
      'I5'$(3;¯7;4.7;4.2)
    3   ¯7    5    4
      ⎕←A←'F6.2'$(3;6;¯7;4.3)
  3.00  6.00 ¯7.00  4.30
      ⎕←A←'F6.2'$(3;6 ¯7;4.3)
  3.00  6.00  4.30
       ¯7.00

      ⍴A
2 18
      S←2E25
      'F30.2'$S
 20000000000000000________.
      'F3.2'$500
***
      A←2 3⍴(6?9999)÷100
      'I4,⍞|⍞,F6.2,X3,E10.2'$A
  35| 51.19      8.9E0
  70| 29.36      9.0E1
      'A1'$'AB'
A
B
      'A3'$'ABC'
  A
  B
  C


      T←4 7⍴'PENS   PENCILSPAPER  TOTAL  '
      COST←.19 .05 .01
      AMNT←20 50 2589
      TCOST←AMNT×COST

      '7A1,⍞|⍞,I6,F8.2,F12.2'$(T;AMNT;COST;TCOST,+/TCOST)
PENS   |    20    0.19        3.80
PENCILS|    50    0.05        2.50
PAPER  |  2589    0.01       25.89
TOTAL  |                     32.19
```

```
      X←¯23456.78 ¯25 ¯0.4 .8 0 100
      'BF10.1'$X
  ¯23456.8
     ¯25.0
      ¯0.4
       0.8

     100.0

      'CBI10'$X
   ¯23,457
       ¯25

         1

       100

      'ZF9.1'$X
¯023456.8
¯000025.0
¯000000.4
0000000.8
0000000.0
0000100.0

      'L19'$X
  7 SYNTAX ERROR
       'L19'$X
       ^


      'LI9'$X
¯23457
¯25
0
1
0
100
      'M⎕¯¯⎕F11.2'$X
 ¯¯23456.78
    ¯¯25.00
     ¯¯0.40
       0.80
       0.00
     100.00
```

```
      'R⎕+⎕F⎕+⎕I11'$X
+++++¯23457
++++++++¯25
++++++++++0
++++++++++1
++++++++++0
++++++++100

      'M⎕¯$⎕F⎕$⎕BF11.2'$X
 ¯$23456.78
    ¯$25.00
     ¯$0.40
      $0.80

    $100.00

      'N⎕ ⎕CI20'$X
            ¯23,457
                ¯25
                   0
                   1
                   0
                 100

      'R⎕*⎕F⎕$⎕CF11.2'$|X
*$23,456.78
*****$25.00
******$0.40
******$0.80
******$0.00
****$100.00

      'R⎕      NONE⎕BF9.2'$X
¯23456.78
   ¯25.00
    ¯0.40
     0.80
     NONE

   100.00

      'M⎕(⎕ N⎕)⎕ Q⎕ ⎕ F10.2'$X
(23456.78)
   (25.00)
    (0.40)
     0.80
     0.00
   100.00

      'N⎕ DB⎕ Q⎕ CR⎕ M⎕⎕ BF14.2'$X
   23456.78 DB
      25.00 DB
       0.40 DB
       0.80 CR

     100.00 CR
```

Related Functions

None

3.4.6 ⍕ or .FM - The Monadic Format Function

Format

monadic

Argument Types

The argument can be a scalar or an array of any shape and either numeric or character data.

Definition

Converts numeric arrays to character arrays. When applied to a character scalar or array, the result of the format *R*←⍕*A* is an array identical to *A*. If *A* is numeric, then the character array represented by *R* will be identical to *A* as it appears when displayed by APL. However, the blank characters displayed along with the values of *A* will actually be a part of the new array *R*. The format of a scalar number is always a vector.

Examples

The following example illustrates the difference between the shapes of a displayed numeric array and a formatted character array:

```
      A←2 4⍴⍳8
      R←⍕A
      A
  1  2  3  4
  5  6  7  8
      ⍴A
2 4
      R
  1  2  3  4
  5  6  7  8
      ⍴R
2 12
      R[;3×⍳4]
1234
5678
```

Related Functions

Dyadic Format, Section 3.4.7

System variables ⎕*PP* Section 4.2.15
⎕*PW* Section 4.2.16

3.4.7 ⍕ or .FM - The Dyadic Format Function

Format

dyadic

Argument Types

The right argument must be a numeric array. The left argument can be a scalar, a pair of numbers, or a vector whose length is no more than twice the number of columns in the numerical array. The left argument controls the format of the result. Two numbers are usually supplied as the left argument. The first number specifies the width of the numeric field, and the second sets the precision of that field.

Definition

Provides output control exceeding that available with the monadic format. It offers a number of formatting options but does not provide the comprehensive formatting capability available with the dollar format function ($).

Dyadic format provides a powerful tool for formatting tables, headings, and labels. Precision is expressed differently for decimal values and in scaled exponential forms of output. The form is determined by the sign of the precision argument. For decimal output, precision is a positive number, expressed as the number of digits to the right of the decimal point. For scaled output, precision is negative and is the number of digits in the multiplier.

If the width of the specification is zero or omitted from the expression, APL provides a default width such that at least one space is inserted between pairs of numbers. If only one number is provided as the left argument, the number is assumed to represent the precision of the result, not its width.

In general, the width must be large enough to accommodate the number field. However, APL does not require that space be inserted between columns.

You can specify width and precision arguments for each column of the array to be formatted or even for each element of the array.

A format operation can also be specified for a multidimensional array and applied to the last two coordinates.

Examples

```
      X
   31.16           0             ¯1.07
  ¯15.578          8            ¯235.61
      ⍴X
2 3
      ⎕←Y←12 3⍕X
      31.160       0.000      ¯1.070
     ¯15.578       8.000    ¯235.610
      ⍴Y
2 36
      A←9 2⍕X
      A

    31.16     0.00    ¯1.07
   ¯15.58     8.00  ¯235.61
      ⎕←R←6 0⍕X
    31      0    ¯1
   ¯16      8  ¯236
      ⍴R
2 18
      ⎕←B←9 ¯2⍕X
  3.1E1     0       ¯1.1E0
 ¯1.6E1     8.0E0   ¯2.4E2

      ⎕←C←7 ¯1⍕X
  3E1     0     ¯1E0
 ¯2E1     8E0   ¯2E2

      ⍝COLUMN FORMATTING
      ⍴⎕←8 0 0 ¯2 8 0 ⍕X
      31  0            ¯1
     ¯16  8.0E0      ¯236
2 25

      ⍝FORMATTING A MULTIDIMENSIONAL ARRAY
      ⎕←A←2 2 2⍴⍳8
  1  2
  3  4

  5  6
  7  8
      5 2⍕A
 1.00 2.00
 3.00 4.00

 5.00 6.00
 7.00 8.00

      B
  3.1E1     0       ¯1.1E0
 ¯1.6E1     8.0E0   ¯2.4E2


      R←3 3⍴1 0 0 1 0 1 1 1 1
      ⍝TABLE FORMATTING

      ROWS←5 7⍴'APL    FORTRANCOBOL  BASIC  PL1
      COLS←'  USERS  PROGS  SYSTS'
      FORM←5 3⍴A
      ('       ',[1]ROWS),COLS,[1]7 0⍕FORM
         USERS  PROGS  SYSTS
APL          1      2      3
FORTRAN      4      5      6
COBOL        7      8      1
BASIC        2      3      4
PL1          5      6      7
```

Related Functions

Monadic Format, Section 3.4.6

3.4.8 ⍕ or .EN - The Quote Function

Format

monadic

Argument Types

The argument can be a scalar or an array with either numeric or character data. If numeric, APL converts it to a character string.

Definition

Converts numeric values to character strings and also provides aid in preparing text to be processed by the execute function.

If the argument is already a character string, APL determines whether or not the string represents an identifier (for example, a variable name or function name). If the character string is not an identifier, then APL returns a null vector. If the argument if a variable, APL returns the value of the variable. If the argument is a function, APL returns the lines of the function definition, separated by pairs of carriage return/line feed characters.

Examples

In the following example, array *A* is converted to a 20-character vector (spaces output by APL are included in the size) in which the character representations of 1 through 6 are members, but the corresponding numeric values are not.

```
      A←2 3⍴⍳6
      B←⍕A
      B
 1  2  3
 4  5  6
      ⍴B
20
      '123456'∊B
1 1 1 1 1 1
      (⍳6)∊B
0 0 0 0 0 0
```

In the following example, the user defined function *G* is effectively restored by the use of the character string that represented this user-defined function in an execute operation.

```
        ∇Z←A G B
[1]     Z←(3×A)+4×B
[2]     Z←Z*2
[3]     ∇
        2 G 1
100
        ⎕←C←⍕'G'
∇ Z←A G B
 Z←(3×A)+4×B
 Z←Z*2
∇
        )ERASE G
        2 G 1
  7 SYNTAX ERROR
          2 G 1
            ∧
        ⍎C
        2 G 1
100
```

Related Functions

Unquote, Section 3.4.3

3.5 OPERATORS

An operator differs from a function in that an operator takes a function as its argument. The following operators are available in APL:

1.	Inner Product	Section 3.5.1
2.	Outer Product	Section 3.5.2
3.	Reduction	Section 3.5.3
4.	Scan	Section 3.5.4

3.5.1 f . g - The Inner Product Operator

Format

dyadic

Argument Types

Both arguments can be vectors, matrices, arrays, or higher-dimensional arrays. If either argument is a scalar or a 1-element vector, it is extended so that its dimensions match the dimensions of the other argument.

Both f and g can be any dyadic scalar function as long as both are of the same type, that is, both arithmetic, both logical, and so on.

Definition

Obtains the common algebraic matrix product, and also extends this capability to other arithmetic operations and other array dimensions.

You can also specify an inner product in which an operation other than multiplication is performed. It is possible to locate values containing specific characters by this method or to search for a row of one array in which all the elements are equal to those in a column of another array.

The two arguments, say *A* and *B*, must conform to certain rules to be used in an inner product operation. The two arguments conform if any of the following is true:

1. *A* or *B* is a scalar.

2. Results of -1↑ρ*A* and -1↑ρ*B* are equal.

3. Either -1↑ρ*A* or -1↑ρ*B* equals 1.

If the third characteristic is the case, then the corresponding argument is extended so that the arguments have equal lengths along the specified coordinate. The basic test for conformability is whether or not the length of the last dimension of the left argument matches the length of the first dimension of the right argument. The dimensions of the result can then be considered all dimensions of *A* except the last, catenated to all dimensions of *B* except the first.

In Table 3-9, the letters have the following meaning:

f	is a primitive function.
g	is a primitive function.
A	is an argument.
B	is an argument.
Z	is the result.
C,*D*,*E*,*F*	are the respective shapes of the arguments.
I,*J*	are indices.

Note that when one or both of the arguments are scalar the shape of the arguments need not conform to any rules.

Table 3-9
Inner Product Description

Definition of	Shape		Result
Z←*A*f.g*B*	ρ*A*	ρ*B*	ρ*Z*
Z←f/*A*g*B*	scalar	scalar	scalar
Z←f/*A*g*B*	scalar	*E*	scalar
Z←f/*A*g*B*	*D*	scalar	scalar
Z[*I*]←f/*A*g*B*[;*I*]	scalar	*E* *F*	*F*
Z[*I*]←f/*A*[*I*;]g*B*	*C* *D*	scalar	*C*
Z[*I*]←f/*A*g*B*[;*I*]	*D*	*D* *F*	*F*
Z[*I*]←f/*A*[*I*;]g*B*	*C* *D*	*D*	*C*
Z[*I*;*J*]←f/*A*[*I*;]g*B*[;*J*]	*C* *D*	*D* *F*	*C* *F*

Examples

```
          ⎕←A←2 3⍴⍳6
    1  2  3
    4  5  6
          ⎕←B←⍳3
  1 2 3
          A+.×B
  14 32
          (⍳3)+.×⍳3
  14
          2 6+.≤A
  0 1 2
          A+.×⍉A
    14  32
    32  77
          ⎕←X←4 3⍴'ONETWOSIXTEN'
  ONE
  TWO
  SIX
  TEN
          ⍴X
  4 3
          V←'SIX'
          ⍴V
  3
          X∧.=V
  0 0 1 0
```

Related Operators

Outer Product, Section 3.5.2

3.5.2 ∘ . f or .SO . f - The Outer Product Operator

Format

dyadic

Argument Types

Both arguments can be any array. The ∘ symbol is the jot character, not the circle.

The f is any dyadic scalar function. The period (.) is the connector between the jot and the function.

Definition

Specifies an operation to be performed by every element of one array on every element of another array. For example, in the expression *R*←*A*.f*B*, *R* is any array that results from applying f to every pair of elements of *A* and *B*. The shape of *R* is the dimensions of *A* catenated to the dimensions of *B*, or (ρ*A*),ρ*B*. Unlike inner product, outer product performs only one operation.

Table 3-10 describes the results of using a variety of arrays.

The letters have the following meaning:

f	is a primitive function.
g	is a primitive function.
A	is an argument.
B	is an argument.
C,*D*,*E*,*F*	are the respective shapes of the arguments.
I,*J*,*K*,*L*	are indices.
Z	is the result.

Table 3-10
Outer Product Description

Definition of	Shape		Result
Z←A∘.gB	ρA	ρB	ρZ
Z←AgB	scalar	scalar	scalar
Z[I]←AgB[I]	scalar	E	E
Z[I]←A[I]gB	D	scalar	D
Z[I;J]←A[I]gB[J]	D	E	D E
Z[I;J]←AgB[I;J]	scalar	E F	E F
Z[I;J]←A[I;J]gB	C D	scalar	C D
Z[I;J;K]←A[I]gB[J;K]	D	E F	D E F
Z[I;J;K]←A[I;J]gB[K]	C D	E	C D E
Z[I;J;K;L]←A[I;J]gB[K;L]	C D	E F	C D E F

Examples

```
        1 2 3∘.×2 3 4 5
    2   3   4   5
    4   6   8  10
    6   9  12  15

        A←1 2 3 2 2 1
        (⍳3)∘.=A
  1 0 0 0 0 1
  0 1 0 1 1 0
  0 0 1 0 0 0

        +/(⍳3)∘.=A
2 3 1
        ⍝THERE ARE 2 ONES, 3 TWOS, AND 1 THREE IN A
```

Related Operators

Inner Product, Section 3.5.1

3.5.3 f/ - The Reduction Operator

Format

monadic

Argument Types

The argument is a scalar, a vector or one coordinate of an array.

The f can be any scalar dyadic function.

Definition

Specifies that an operation is to be used to combine the elements of a vector or elements along a specified dimension of an array. The result of reducing any vector is a scalar value.

The result of reducing an array has a rank that is one less than the rank of the original array. Thus, the reduction of a matrix yields a vector. To specify a coordinate, include it within square brackets. (The coordinate is dependent upon the index origin, Section 4.2.11.) The default is the highest coordinate. The special symbol to specify the first coordinate is ⌿ (.CS).

If the argument is an empty vector, then the result of a reduction is the identity element of the operator, if one exists. Table 3-11 lists the identity elements for scalar dyadic functions.

Table 3-11
Identity Elements

Dyadic Operator	Symbol	Identity Element
Plus	+	0
Minus	-	0
Times	×	1
Divide	÷	1
Power	*	1
Residue	|	0
Maximum	⌈	¯1.701411835E38
Minimum	⌊	1.701411835E38
Logarithm	⍟	none
Combination	!	1
Circle	○	none
And	&	1
Or	∨	0
Nand	⍲	none
Nor	⍱	none
Less	<	0
Not Greater	≤	1
Equal to	=	1
Not Less	≥	1
Greater	>	0
Not Equal	≠	0

Examples

```
            ⎕←X←⍳6
1 2 3 4 5 6
            +/X
21
            ×/X
720
            ⌈/X
6
            ⌊/X
1
            +/5 7 8
20
            +/1 6 7
14
            -/1 6 7
2
            ⍝SAME AS 1-6-7
            ⌈/7
7
            ×/⍳0
1

            ⎕←A←2 4⍴⍳6
  1  2  3  4
  5  6  1  2
            +/[2]A
10 14
            +/[1]A
6 8 4 6
```

Related Operators

Scan, Section 3.5.4

3.5.4 f\ - The Scan Operator

Format

monadic

Argument Types

The argument can be a scalar, a vector, or one coordinate of an array.

The f is any scalar dyadic function.

Definition

Returns partial results in calculating the reduction of an array. The shape of the result of a scan is the same as the shape of the original vector. The first element of the result is always identical to the first element of the original vector. The last element is equal to a reduction of the entire original vector. For example:

```
      +\3 4 5
3 7 12
```

If the argument is a null vector, then the result of the scan is a null vector.

You can also specify a scan for one particular coordinate of a multidimensional array. You specify the coordinate to be scanned by including a bracketed number with the function. The syntax is the same as that of the reduction function. If you omit the coordinate within brackets, APL scans the last coordinate of the array. You can specify a scan on the first coordinate by using the symbol ⍀ (.CB), which is formed by overstriking the scan (backslash) with the minus sign.

If the dyadic function specified with scan is associative (for example, + or ×) APL performs the scan in a way that is different from the conventional scan, in order to increase efficiency by reducing the number of operations. The definition of *R*←f*A* is equal to *R*[*I*]=f/*I*↑*A* as follows:

```
    R[1]=A[1]
    R[I]=R[I-1]fA[I]FOR I∊1↓⍴A
```

This definition requires fewer operations than the traditional scan. The result of an associative operation of this kind may differ slightly from the nonassociative approach and should be used carefully if the results require a high degree of precision.

Examples

```
      A←1E6 ¯1E6 1E¯16
      A
1000000 ¯1000000 1.000000000E¯16
      +\A
1000000 0 1.000000000E¯16
      +/A
0


      ×\2 2 2
2 4 8

      ∨\0 1 0 0
0 1 1 1


      ×\⍳7
1 2 6 24 120 720 5040


      ÷\8
8

      ⎕←A←2 3⍴⍳6
  1  2  3
  4  5  6

      +\A
   1   3   6
   4   9  15

      +\[1]A
  1  2  3
  5  7  9

      +\[2]A
   1   3   6
   4   9  15

      +⍀A
  1  2  3
  5  7  9
```

Related Operators

Reduction, Section 3.5.3

CHAPTER 4

APL SYSTEM COMMUNICATION

4.1 INTRODUCTION

There are a variety of ways in which you can communicate with the APL system to change parameters, determine hardware or operational characteristics, and modify processing methods. The APL system commands in Chapter 5 facilitate many of these system operations. The elements in Chapter 4 that aid system communication are:

> system variables - They are similar to ordinary variables but are distinguished by special names that begin with a quad character, for example, *⎕LX*.
>
> system functions - They allow you to interact with APL by specifying distinguished names beginning with a quad character, for example, *⎕CR*.

Section 4.2 describes system variables and Section 4.3 describes system functions.

4.2 SYSTEM VARIABLES

APL system variables allow you to perform such operations as the following:

1. Set the index origin and relative fuzz
2. Change the output precision and line width
3. Specify an operation to be performed when the workspace is activated
4. Save the active workspace automatically after editing

The syntax of APL system variables is similar to ordinary variables in that you can use both types of variables in any language expression or function. APL system variables have distinguished names; they begin with a quad character (⎕). They differ from ordinary variables because of their special significance to the system.

APL system variables cannot be used as names for user-defined functions. Also, you cannot copy, erase, or collect them in a group.

The 25 system variables described in this chapter serve as an interface between APL and the operating system you are using. The workspace and the APL processor can each use values specified by the other as appropriate to the particular operation being performed. The value of a system variable being used in a workspace can sometimes be different from the value last specified by the user of the workspace.

System variables fall into two categories:

1. System variables to which you can assign a value. These variables retain the value until you override it with another value or clear the workspace. You can save the value with the workspace, and you can also localize them in a function definition. These system variables do have default values. The following system variables are in this category:

⎕*AUS*	Section 4.2.5
⎕*CT*	Section 4.2.7
⎕*ERROR*	Section 4.2.9
⎕*GAG*	Section 4.2.10
⎕*IO*	Section 4.2.11
⎕*LX*	Section 4.2.13
⎕*PP*	Section 4.2.15
⎕*PW*	Section 4.2.16
⎕*RL*	Section 4.2.17
⎕*SF*	Section 4.2.18
⎕*TIMELIMIT*	Section 4.2.19
⎕*TRAP*	Section 4.2.21

2. System variables that have values you cannot change. If you assign a value to this type of variable, you will not receive an error; however, the assignment will have no effect. The following system variables are in this category:

⎕*AI*	Section 4.2.1
⎕*ALPHA*	Section 4.2.2
⎕*ALPHAU*	Section 4.2.3
⎕*ASCII*	Section 4.2.4
⎕*AV*	Section 4.2.6
⎕*CTRL*	Section 4.2.8
⎕*LC*	Section 4.2.12
⎕*NUM*	Section 4.2.14
⎕*TIMEOUT*	Section 4.2.20
⎕*TS*	Section 4.2.22
⎕*TT*	Section 4.2.23
⎕*UL*	Section 4.2.24
⎕*WA*	Section 4.2.25

4.2.1 ⎕*AI* - Storing Account Information

The ⎕*AI* (account information) system variable stores the following account information during a work session:

1. User identification (for the project-programmer number [PROJ,PROG] this is PROG+PROJ×2*18)

2. Computer time (CPU time) used during the current APL session

3. Connect time used during the current APL session

4. Keying time (time during which the keyboard is unlocked) used during the current APL session

All times are expressed in milliseconds.

For example:

```
        ⎕AI
    1048708 1235 440967 382458
```

4.2.2 ⎕*ALPHA* - Alphabetic Characters

The ⎕*ALPHA* system variable is a subset of the ⎕*AV* system variable, Section 4.2.6. The value contained in ⎕*ALPHA* is a vector of the 27 alphabetic characters *∆* and *A* through *Z*.

For example:

```
        ⎕ALPHA
    ∆ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

4.2.3 ⎕*ALPHAU* - Underlined Alphabetics

The ⎕*ALPHAU* system variable is a subset of the ⎕*AV* system variable, Section 4.2.6. The value contained in ⎕*ALPHAU* is a vector of the 27 underlined characters *⍙* and *A̲* through *Z̲*.

For example:

```
        ⎕ALPHAU
    ⍙A̲B̲C̲D̲E̲F̲G̲H̲I̲J̲K̲L̲M̲N̲O̲P̲Q̲R̲S̲T̲U̲V̲W̲X̲Y̲Z̲
        ⎕AV⍳⎕ALPHAU
    481 482 483 484 485 486 487 488 489 490 491 492 493 494 495
          496 497 498 499 500 501 502 503 504 505 506 507
```

4.2.4 ⎕*ASCII* - ASCII Character Set

The ⎕*ASCII* system variable contains 128 ASCII characters. This distinguished variable is designed primarily for use with ASCII files, as an easy way to output arbitrary ASCII codes to those files while using the APL character set. The first 32 characters of ⎕*ASCII* are the control characters in ⎕*CTRL*, Section 4.2.8. The rest are pure ASCII, not translatable by APL.

⎕*ASCII* is a subset of ⎕*AV*. The indices into ⎕*AV* that contain ⎕*ASCII* are ⎕*AV*[257] through ⎕*AV*[288] and ⎕*AV*[381] through ⎕*AV*[476].

⎕*ASCII* is intended for use with output functions, especially output to the terminal. It accepts a list of numbers that it translates into ASCII codes for transmission. It does not go through the usual translation from APL 9-bit to ASCII.

When you write to an ASCII sequential file (Section 7.5.1) using ⎕*ASCII*, you can output the full range of ASCII characters. The following example writes seven ASCII characters to an ASCII sequential file while in APL mode. The APL equivalents of these characters print during output. Once back at operating system command level, you can access the ASCII characters.

For example:

```
          ⎕ASS 'THIS/AS'
    12
          ⎕ASCII[36,38,39,43,65,95,96]⍞12
    <=>≠¯≥_

          ⎕CLS 12

          )MON
    MONITOR:
    @TY THIS.AAS
    #%&*@^_
    @
    @
```

Table 4-1 contains the decimal indices and octal values of ⎕*ASCII*.

Table 4-1
The ASCII Character Set
⎕*ASCII* (⎕*IO*←1)

Index	ASCII Char	Octal Value
1	NUL	000
2	SOH	001
3	STX	002
4	ETX	003
5	EOT	004
6	ENQ	005
7	ACK	006
8	BEL	007
9	BS	010
10	HT	011
11	LF	012
12	VT	013
13	FF	014
14	CR	015
15	SO	016
16	SI	017
17	DLE	020
18	DC1	021
19	DC2	022
20	DC3	023
21	DC4	024
22	NAK	025
23	SYN	026
24	ETB	027
25	CAN	030
26	EM	031
27	SUB	032

Table 4-1 (Cont.)
The ASCII Character Set
⎕*ASCII* (⎕*IO*←1)

Index	ASCII Char	Octal Value
28	ESC	033
29	FS	034
30	GS	035
31	RS	036
32	US	037
33	space	040
34	!	041
35	"	042
36	#	043
37	$	044
38	%	045
39	&	046
40	'(apostrophe)	047
41	(	050
42	)	051
43	*	052
44	+	053
45	,	054
46	-	055
47	.	056
48	/	057
49	0	060
50	1	061
51	2	062
52	3	063
53	4	064
54	5	065
55	6	066
56	7	067
57	8	070
58	9	071
59	:	072
60	;	073
61	<	074
62	=	075
63	>	076
64	?	077
65	@	100
66	A	101
67	B	102
68	C	103
69	D	104
70	E	105
71	F	106
72	G	107
73	H	110
74	I	111
75	J	112
76	K	113
77	L	114
78	M	115
79	N	116
80	O	117
81	P	120

Table 4-1 (Cont.)
The ASCII Character Set
⎕*ASCII* (⎕*IO*←1)

Index	ASCII Char	Octal Value
82	Q	121
83	R	122
84	S	123
85	T	124
86	U	125
87	V	126
88	W	127
89	X	130
90	Y	131
91	Z	132
92	[	133
93	\	134
94	]	135
95	^(uparrow)	136
96	_(underscore)	137
97	`(grave)	140
98	a	141
99	b	142
100	c	143
101	d	144
102	e	145
103	f	146
104	g	147
105	h	150
106	i	151
107	j	152
108	k	153
109	l	154
110	m	155
111	n	156
112	o	157
113	p	160
114	q	161
115	r	162
116	s	163
117	t	164
118	u	165
119	v	166
120	w	167
121	x	170
122	y	171
123	z	172
124	{	173
125	¦	174
126	}	175
127	~	176
128	DEL	177

⎕*ASCII*[124] through [127] print differently on other terminals such as on a VT05 or VT52.

4.2.5 ⎕*AUS* - Saving a Workspace Automatically

The ⎕*AUS* (automatic save) system variable activates a special feature that allows you to save the currently active workspace automatically at periodic intervals. Workspace backup is often critical when you are performing an extensive amount of function editing and debugging or typing a large table of values. Normally, you would stop editing and issue the appropriate APL commands (Chapter 5) to save the workspace on disk to ensure that a system crash does not destroy the current workspace. If ⎕*AUS* is set to 1, APL automatically saves the workspace on disk every time a function is closed or a quad-input request is sent to the terminal. This ensures that you will probably have to reenter only a small amount of input in the event of a system crash.

The default value for ⎕*AUS* in a clear workspace is installation dependent and can be changed by the System Manager. APL saves the value of ⎕*AUS* with the workspace, and you can localize it in the same manner as ⎕*IO* and several other variables.

When saving a workspace, ⎕*AUS* creates a disk file and assigns a name to it in the format:

    XNNWWW.TMP:

where

X	is *A*, if the job number divided by 100 equals 0.
	is *B*, if the job number divided by 100 equals 1, and so on.
NN	is the job number modulo 100.
WWW	is the first three characters of the current workspace name.
TMP	is extension or filetype.

For example, if the job number is 79 and the workspace name is *TESTS*, then the temporary filename will be *A*79*TES.TMP*. If the job number is 179, the temporary file will be *B*79*TES.TMP*.

If the system crashes and is reloaded, you can verify that a temporary disk file exists for the workspace by issuing a)*LIB* command. For example:

```
      )LIB *.TMP
DSK:
A79TES.TMP
```

After APL displays the sign-on message, you can load the backup file as the active workspace by issuing a)*LOAD* command (Section 5.2.4).

```
      )LOAD A79TES.TMP
SAVED    9:49:59  2-JUL-79 5P
```

APL prints the *)LOAD* message. The name of the active workspace is now the name that the workspace had before the backup was performed, not the name of the temporary file.

```
          )WSID
      TESTS [4,204]
```

After completing the editing of a function or entering data you should explicitly save the active workspace and delete the temporary backup on disk.

APL deletes a *.TMP* file that it has written when any one of the following conditions occurs:

1. A *)SAVE* has been completed successfully.
2. An *)OFF*, *)CONTINUE*, *)CALL*, or *)RUN* command is executed.

APL intends to write a new *.TMP* file with a different name.

For example:

```
          ⎕AUS←1

          )WSID ABC
   WAS CLEAR WS
          ∇F
   [1]    A←25×713
   [2]    B←A÷68
   [3]    ∇
          ⍝HERE APL WRITES THE .TMP FILE
          )LIB *.TMP
   DSK:
   A18ABC.TMP
          )WSID XYZ
   WAS ABC [4,204]
          ∇F
   [3]    B
   [4]    ∇
          ⍝HERE APL WRITES THE NEW .TMP FILE
          )LIB *.TMP
   DSK:
   A18XYZ.TMP
```

4.2.6 ⎕*AV* - Atomic Vector

The ⎕*AV* (atomic vector) system variable contains a vector of every character in APL. Table 4-2 lists the characters with their positions in the vector. Note that the positions are based on an index origin of 1. In Table 4-2 characters that are normally non-printable are output as ⌷, a squish quad.

Table 4-2
The Atomic Vector ⎕*AV* (⎕*IO*←1)

⎕*AV*[]	Symbol	TTY Set	Name
1 through 16	⌷	.SQ	squish quad
17	!	!	factorial
18	⌽	.RV	reversal
19	⍉	.TR	transpose
20	⌶	.IB	I-Beam
21	⍞	.QQ	quote quad
22	⍟	.LG	logarithm
23	⍲	.NN	Nand
24	⍱	.NR	Nor
25	⍝	"	Comment (lamp)
26	⍋	.GU	grade up
27	⍒	.GD	grade down
28	⊖	.CR	circle (rotate)
29	⌿	.CS	back scan
30	⍀	.CB	back expansion
31	⌷	.SQ	squish quad
32	⍫	.PD	protected del
33	⍔	.QD	quad del
34	⍇	.IQ	input quad
35	⍈	.OQ	output quad
36	⌹	.DQ	divide quad (domino)
37	$	$	format (dollar)
38	⌽̲	.FI	fix
39	⍎	.XQ	execute
40	⍕	.FM	format
41 through 46	⌷	.SQ	squish quad
47	¨	.DD	dieresis
48	<	<	less than
49	≤	.LE	less than or equal
50	≥	.GE	greater than or equal
51	>	>	greater than
52	?	?	question (roll, deal)
53	⍵	.OM	omega
54	∊	.EP	epsilon
55	⍴	.RO	rho
56	↑	^	take
57	↓	.DA	drop (down arrow)
58	⍳	.IO	iota
59	⍺	.AL	alpha
60	⌈	.CE	ceiling
61	⌊	.FL	floor
62	(	(	left parenthesis
63	)	)	right parenthesis
64	[	[	left square bracket
65	]	]	right square bracket
66	⊂	.RU	right union
67	⊃	.LU	left union
68	∪	.UU	up union
69	_	.US	underscore
70 through 72	⌷	.SQ	squish quad
73	←	_	left arrow (assignment)
74	→	.GO	right arrow (branch)
75	⎕	.BX	quad (box)
76	≠	.NE	not equal
77	=	=	equal
78	∩	.DU	down union
79	⊥	.DE	decode

Table 4-2 (Cont.)
The Atomic Vector ⎕*AV* (⎕*IO*←1)

⎕*AV*[]	Symbol	TTY Set	Name
80	∧	&	and
81	∨	.OR	or
82	~	.NT	not
83	/	/	reduce
84	\	\	expand
85	*	*	exponentiate (star)
86	×	#	multiply
87	÷	%	divide
88	+	+	add
89	-	-	subtract
90	○	.LO	circle (large o)
91	|	.AB	residue (absolute)
92	;	;	semicolon
93	,	,	comma
94	∘	.SO	jot (small o)
95	⊤	.EN	encode
96	∇	.DL	del
97	:	:	colon
98	'	'	quote
99	(none)	(backspace)	for internal use
100	(none)	(line feed)	for internal use
101	(none)	(double line feed)	for internal use
102	(none)	(carriage return)	for internal use
103	(none)	(null)	for internal use
104	(none)	(space)	for internal use
105	(none)	(escape)	for internal use
106	(none)	(formfeed)	for internal use
107	(none)	(tab)	for internal use
108	⌷	.SQ	squish quad
109	.	.	period
110	¯	.NG	negation
111	⌷	.SQ	squish quad
112	(none)	(at @)	for internal use
113	(none)	(left bracket)	for internal use
114	⊢	.LK	left tack
115	(none)	(right bracket)	for internal use
116	(none)	(uparrow)	for internal use
117	(none)	(left arrow)	for internal use
118	◊	.DM	diamond
119	{	.LB	left curly brace
120	⊣	.RK	right tack
121	}	.RB	right curly brace
122	(none)	(tilde)	for internal use
123	(none)	(delete)	for internal use
124 through 256	⌷	.SQ	squish quad
257	⎕*CTRL*[1]	.BXCTRL[1]	null (NUL)
258	⎕*CTRL*[2]	.BXCTRL[2]	start of heading (SOH)
259	⎕*CTRL*[3]	.BXCTRL[3]	start of text (STX)
260	⎕*CTRL*[4]	.BXCTRL[4]	end of text (ETX)
261	⎕*CTRL*[5]	.BXCTRL[5]	end of transmission (EOT)
262	⎕*CTRL*[6]	.BXCTRL[6]	enquiry (ENQ)
263	⎕*CTRL*[7]	.BXCTRL[7]	acknowledge (ACK)

Table 4-2 (Cont.)
The Atomic Vector ⎕*AV* (⎕*IO*←1)

⎕*AV*[]	Symbol	TTY Set	Name
264	⎕*CTRL*[8]	.BXCTRL[8]	bell (BEL)
265	⎕*CTRL*[9]	.BXCTRL[9]	backspace (BS)
266	⎕*CTRL*[10]	.BXCTRL[10]	horizonal tabulation (HT)
267	⎕*CTRL*[11]	.BXCTRL[11]	line feed (LF)
268	⎕*CTRL*[12]	.BXCTRL[12]	vertical tab (VT)
269	⎕*CTRL*[13]	.BXCTRL[13]	form feed (FF)
270	⎕*CTRL*[14]	.BXCTRL[14]	carriage return (CR)
271	⎕*CTRL*[15]	.BXCTRL[15]	shift out (SO)
272	⎕*CTRL*[16]	.BXCTRL[16]	shift in (SI)
273	⎕*CTRL*[17]	.BXCTRL[17]	data link escape (DLE)
274	⎕*CTRL*[18]	.BXCTRL[18]	device control 1 (DC1)
275	⎕*CTRL*[19]	.BXCTRL[19]	device control 2 (DC2)
276	⎕*CTRL*[20]	.BXCTRL[20]	device control 3 (DC3)
277	⎕*CTRL*[21]	.BXCTRL[21]	device control 4 (DC4)
278	⎕*CTRL*[22]	.BXCTRL[22]	negative acknowledge (NAK)
279	⎕*CTRL*[23]	.BXCTRL[23]	synchronous idle (SYN)
280	⎕*CTRL*[24]	.BXCTRL[24]	end of transmission block (ETB)
281	⎕*CTRL*[25]	.BXCTRL[25]	cancel (CAN)
282	⎕*CTRL*[26]	.BXCTRL[26]	end of medium (EM)
283	⎕*CTRL*[27]	.BXCTRL[27]	substitute (SUB)
284	⎕*CTRL*[28]	.BXCTRL[28]	escape (ESC)
285	⎕*CTRL*[29]	.BXCTRL[29]	file separator (FS)
286	⎕*CTRL*[30]	.BXCTRL[30]	group separator (GS)
287	⎕*CTRL*[31]	.BXCTRL[31]	record separator (RS)
288	⎕*CTRL*[32]	.BXCTRL[32]	unit separator (US)
289 through 304	⌷	.SQ	squish quad
305	0	0	zero
306	1	1	one
307	2	2	two
308	3	3	three
309	4	4	four
310	5	5	five
311	6	6	six
312	7	7	seven
313	8	8	eight
314	9	9	nine
315 through 352	⌷	.SQ	squish quad
353	∆	.LD	delta
354	*A*	A	A
355	*B*	B	B
356	*C*	C	C
357	*D*	D	D
358	*E*	E	E
359	*F*	F	F
360	*G*	G	G
361	*H*	H	H
362	*I*	I	I
363	*J*	J	J
364	*K*	K	K
365	*L*	L	L
366	*M*	M	M
367	*N*	N	N
368	*O*	O	O
369	*P*	P	P

Table 4-2 (Cont.)
The Atomic Vector ⎕*AV* (⎕*IO*←1)

⎕*AV*[]	Symbol	TTY Set	Name
370	*Q*	Q	Q
371	*R*	R	R
372	*S*	S	S
373	*T*	T	T
374	*U*	U	U
375	*V*	V	V
376	*W*	W	W
377	*X*	X	X
378	*Y*	Y	Y
379	*Z*	Z	Z
380	▯	.SQ	squish quad
381	⎕*ASCII*[33]	.BXASCII[33]	space
382	⎕*ASCII*[34]	.BXASCII[34]	exclamation point (!)
383	⎕*ASCII*[35]	.BXASCII[35]	double quote (")
384	⎕*ASCII*[36]	.BXASCII[36]	number sign (#)
385	⎕*ASCII*[37]	.BXASCII[37]	dollar sign ($)
386	⎕*ASCII*[38]	.BXASCII[38]	percent (%)
387	⎕*ASCII*[39]	.BXASCII[39]	ampersand (&)
388	⎕*ASCII*[40]	.BXASCII[40]	apostrophe (')
389	⎕*ASCII*[41]	.BXASCII[41]	left parenthesis (()
390	⎕*ASCII*[42]	.BXASCII[42]	right parenthesis())
391	⎕*ASCII*[43]	.BXASCII[43]	asterisk (*)
392	⎕*ASCII*[44]	.BXASCII[44]	plus (+)
393	⎕*ASCII*[45]	.BXASCII[45]	comma (,)
394	⎕*ASCII*[46]	.BXASCII[46]	hyphen (-)
395	⎕*ASCII*[47]	.BXASCII[47]	period (.)
396	⎕*ASCII*[48]	.BXASCII[48]	slash (/)
397	⎕*ASCII*[49]	.BXASCII[49]	zero (0)
398	⎕*ASCII*[50]	.BXASCII[50]	one (1)
399	⎕*ASCII*[51]	.BXASCII[51]	two (2)
400	⎕*ASCII*[52]	.BXASCII[52]	three (3)
401	⎕*ASCII*[53]	.BXASCII[53]	four (4)
402	⎕*ASCII*[54]	.BXASCII[54]	five (5)
403	⎕*ASCII*[55]	.BXASCII[55]	six (6)
404	⎕*ASCII*[56]	.BXASCII[56]	seven (7)
405	⎕*ASCII*[57]	.BXASCII[57]	eight (8)
406	⎕*ASCII*[58]	.BXASCII[58]	nine (9)
407	⎕*ASCII*[59]	.BXASCII[59]	colon (:)
408	⎕*ASCII*[60]	.BXASCII[60]	semicolon (;)
409	⎕*ASCII*[61]	.BXASCII[61]	less than (<)
410	⎕*ASCII*[62]	.BXASCII[62]	equal (=)
411	⎕*ASCII*[63]	.BXASCII[63]	greater than (>)
412	⎕*ASCII*[64]	.BXASCII[64]	question mark (?)
413	⎕*ASCII*[65]	.BXASCII[65]	at sign (@)
414	⎕*ASCII*[66]	.BXASCII[66]	A
415	⎕*ASCII*[67]	.BXASCII[67]	B
416	⎕*ASCII*[68]	.BXASCII[68]	C
417	⎕*ASCII*[69]	.BXASCII[69]	D
418	⎕*ASCII*[70]	.BXASCII[70]	E
419	⎕*ASCII*[71]	.BXASCII[71]	F
420	⎕*ASCII*[72]	.BXASCII[72]	G
421	⎕*ASCII*[73]	.BXASCII[73]	H
422	⎕*ASCII*[74]	.BXASCII[74]	I
423	⎕*ASCII*[75]	.BXASCII[75]	J
424	⎕*ASCII*[76]	.BXASCII[76]	K
425	⎕*ASCII*[77]	.BXASCII[77]	L
426	⎕*ASCII*[78]	.BXASCII[78]	M

Table 4-2 (Cont.)
The Atomic Vector ⎕*AV* (⎕*IO*←1)

⎕*AV*[]	Symbol	TTY Set	Name
427	⎕*ASCII*[79]	.BXASCII[79]	N
428	⎕*ASCII*[80]	.BXASCII[80]	O
429	⎕*ASCII*[81]	.BXASCII[81]	P
430	⎕*ASCII*[82]	.BXASCII[82]	Q
431	⎕*ASCII*[83]	.BXASCII[83]	R
432	⎕*ASCII*[84]	.BXASCII[84]	S
433	⎕*ASCII*[85]	.BXASCII[85]	T
434	⎕*ASCII*[86]	.BXASCII[86]	U
435	⎕*ASCII*[87]	.BXASCII[87]	V
436	⎕*ASCII*[88]	.BXASCII[88]	W
437	⎕*ASCII*[89]	.BXASCII[89]	X
438	⎕*ASCII*[90]	.BXASCII[90]	Y
439	⎕*ASCII*[91]	.BXASCII[91]	Z
440	⎕*ASCII*[92]	.BXASCII[92]	left square bracket ([)
441	⎕*ASCII*[93]	.BXASCII[93]	backslash (\)
442	⎕*ASCII*[94]	.BXASCII[94]	right square bracket (])
443	⎕*ASCII*[95]	.BXASCII[95]	up arrow (^)
444	⎕*ASCII*[96]	.BXASCII[96]	underscore (_)
445	⎕*ASCII*[97]	.BXASCII[97]	grave (`)
446	⎕*ASCII*[98]	.BXASCII[98]	a
447	⎕*ASCII*[99]	.BXASCII[99]	b
448	⎕*ASCII*[100]	.BXASCII[100]	c
449	⎕*ASCII*[101]	.BXASCII[101]	d
450	⎕*ASCII*[102]	.BXASCII[102]	e
451	⎕*ASCII*[103]	.BXASCII[103]	f
452	⎕*ASCII*[104]	.BXASCII[104]	g
453	⎕*ASCII*[105]	.BXASCII[105]	h
454	⎕*ASCII*[106]	.BXASCII[106]	i
455	⎕*ASCII*[107]	.BXASCII[107]	j
456	⎕*ASCII*[108]	.BXASCII[108]	k
457	⎕*ASCII*[109]	.BXASCII[109]	l
458	⎕*ASCII*[110]	.BXASCII[110]	m
459	⎕*ASCII*[111]	.BXASCII[111]	n
460	⎕*ASCII*[112]	.BXASCII[112]	o
461	⎕*ASCII*[113]	.BXASCII[113]	p
462	⎕*ASCII*[114]	.BXASCII[114]	q
463	⎕*ASCII*[115]	.BXASCII[115]	r
464	⎕*ASCII*[116]	.BXASCII[116]	s
465	⎕*ASCII*[117]	.BXASCII[117]	t
466	⎕*ASCII*[118]	.BXASCII[118]	u
467	⎕*ASCII*[119]	.BXASCII[119]	v
468	⎕*ASCII*[120]	.BXASCII[120]	w
469	⎕*ASCII*[121]	.BXASCII[121]	x
470	⎕*ASCII*[122]	.BXASCII[122]	y
471	⎕*ASCII*[123]	.BXASCII[123]	z
472	⎕*ASCII*[124]	.BXASCII[124]	left brace ({)
473	⎕*ASCII*[125]	.BXASCII[125]	vertical bar (¦)
474	⎕*ASCII*[126]	.BXASCII[126]	right brace (})
475	⎕*ASCII*[127]	.BXASCII[127]	tilde (~)
476	⎕*ASCII*[128]	.BXASCII[128]	delete (DEL)
477 through 480	⌷	.SQ	squish quad
481	⍙	.Z@	underscored delta
482	*A̲*	.ZA	underscored A
483	*B̲*	.ZB	underscored B
484	*C̲*	.ZC	underscored C
485	*D̲*	.ZD	underscored D

Table 4-2 (Cont.)
The Atomic Vector ⎕*AV* (⎕*IO*←1)

⎕*AV*[ ]	Symbol	TTY SET	Name
486	*E̲*	.ZE	underscored E
487	*F̲*	.ZF	underscored F
488	*G̲*	.ZG	underscored G
489	*H̲*	.ZH	underscored H
490	*I̲*	.ZI	underscored I
491	*J̲*	.ZJ	underscored J
492	*K̲*	.ZK	underscored K
493	*L̲*	.ZL	underscored L
494	*M̲*	.ZM	underscored M
495	*N̲*	.ZN	underscored N
496	*O̲*	.ZO	underscored O
497	*P̲*	.ZP	underscored P
498	*Q̲*	.ZQ	underscored Q
499	*R̲*	.ZR	underscored R
500	*S̲*	.ZS	underscored S
501	*T̲*	.ZT	underscored T
502	*U̲*	.ZU	underscored U
503	*V̲*	.ZV	underscored V
504	*W̲*	.ZW	underscored W
505	*X̲*	.ZX	underscored X
506	*Y̲*	.ZY	underscored Y
507	*Z̲*	.ZZ	underscored Z
508 through 512	▯	.SQ	squish quad

Subsets of ⎕*AV* are:

⎕*ALPHA*,	Section 4.2.2
⎕*ALPHAU*,	Section 4.2.3
⎕*ASCII*,	Section 4.2.4
⎕*CTRL*,	Section 4.2.8
⎕*NUM*,	Section 4.2.14

4.2.7 ⎕*CT* - Comparison Tolerance

The ⎕*CT* (comparison tolerance) system variable sets the degree of tolerance or relative fuzz (not absolute fuzz) to be applied in performing comparisons. The meaningful range of ⎕*CT* values is 0 through 1*E*¯8. The default value is 1*E*¯13.

You can specify ⎕*CT* in conjunction with the following functions:

⌈	Ceiling
⌊	Floor
>	Greater than
≤	Less than or equal to
=	Equal to
≥	Greater than or equal to
<	Less than
≠	Not equal to

The ⎕*CT* value is saved when you save the active workspace. See the description of fuzz in Section 2.4.3.

For example:

```
      ⎕CT
1.136864040E¯13
```

4.2.8 ⎕*CTRL* - Control Characters

The ⎕*CTRL* system variable is a subset of ⎕*AV* (Section 4.2.6). It contains a vector of the 32 characters listed in Table 4-3.

Table 4-3
⎕*CTRL* (⎕*IO*←1)

Index	Character Name	Octal Value
1	NUL(Null)	000
2	SOH(Start of Heading)	001
3	STX(Start of Text)	002
4	ETX(End of Text)	003
5	EOT(End of Transmission)	004
6	ENQ(Enquiry)	005
7	ACK(Acknowledge)	006
8	BEL(Bell)	007
9	BS(Backspace)	010
10	HT(Horiz. Tabulation)	011
11	LF(Line Feed)	012
12	VT(Vert. Tabulation)	013
13	FF(Form Feed)	014
14	CR(Carriage Return)	015
15	SO(Shift Out)	016
16	SI(Shift In)	017
17	DLE(Data Line Escape)	020
18	DC1(Device Control 1)	021
19	DC2(Device Control 2)	022
20	DC3(Device Control 3)	023
21	DC4(Device Control 4)	024
22	NAK(Negative Acknowledge)	025
23	SYN(Synchronous Idle)	026
24	ETB(End of Transmission Block)	027
25	CAN(Cancel)	030
26	EM(End of Medium)	031
27	SUB(Substitute)	032
28	ESC(Escape)	033
29	FS(File Separator)	034
30	GS(Group Separator)	035
31	RS (Record Separator)	036
32	US(Unit Separator)	037

Note that for any formatting-control character the internal code that appears in ⎕*CTRL* is not the same as the internal code used by APLSF for that formatting-control character.

For example:

```
          ⍝INDEX ORIGIN IS 1
          ⎕AV⍳⎕CTRL[14]
  270
          ⍝WHILE TAKING THE FIRST HALF OF
          ⍝A CRLF IS
          ⎕AV⍳1↑'
  '
  102
```

4.2.9 *⎕ERROR* - Storing Error Messages

The *⎕ERROR* system variable contains text that identifies what error occurred and where it occurred. APL sets *⎕ERROR* during immediate mode and function-definition mode as well as function-execution mode. *⎕ERROR* contains one error at a time. When a new error occurs, the new message overwrites the old one. You can, however, localize *⎕ERROR* within a function to save error information within the environment of a particular function. You can also set *⎕ERROR* to contain your own message. In this way, you can clear *⎕ERROR* by assigning a null string to it, *⎕ERROR*''.

The text that APL sets contains a character vector of variable-length lines, each delimited by a carriage return/line feed. The text has the following format:

```
     nn error message
     funcname [n] line containing the error caret pointing to symbol
         in error
```

where

nn error message is the number and text of the message.

funcname is the name of the function in which the error occurred.

[n] is the line number where the error occurred.

For example:

```
          ∇ABC;⎕TRAP;⎕ERROR
  [1]     ⎕TRAP←'→ LAB'
  [2]     1A←5
  [3]     LAB:⎕BREAK 'CHECK ERROR MESSAGE'
  [4]     'RESUME AT LINE 4'
  [5]     ∇
          ABC
  CHECK ERROR MESSAGE
          ⎕ERROR
    7 SYNTAX ERROR
  ABC[2] 1 A←5
            ^
```

```
      )SI
ABC[3] *
      ⍝AT THIS POINT, CAN CONTINUE EXECUTION
      →⎕LC+1
RESUME AT LINE 4
      ⍝FUNCTION HAS FINISHED EXECUTION
      ⍝NEXT IS AN IMMEDIATE MODE ERROR
      C↓A
 11 VALUE ERROR
       C↓A
        ^
      ⍝CHECK GLOBAL VALUE OF ⎕ERROR
      ⎕ERROR
 11 VALUE ERROR
       C↓A
        ^

      )SI
```

The three lines of text in ⎕*ERROR* are exactly the same three lines APL displays on the terminal.

⎕*ERROR* can contain up to 384 characters. If the line containing the error is too long to fit in ⎕*ERROR*, APL truncates the line and prints the significant portion containing the error. The last character will contain -1↑⎕*AV*.

Note that if an error occurs during an ⍎ execute, then ⎕*ERROR* contains six lines of text. For example:

```
      ⍎'WW'
11 ⍎ VALUE ERROR
      WW
       ^
25 EXECUTE ERROR
      ⍎'WW'
      ^
      ⎕ERROR
11 ⍎ VALUE ERROR
      WW
       ^
25 EXECUTE ERROR
      ⍎'WW'
      ^
```

For more information on ⎕*ERROR*, refer to Section 6.5.

4.2.10 ⎕*GAG* - Preventing Interruptions

The ⎕*GAG* system variable allows you to prevent certain messages from appearing on your terminal. On both the TOPS-10 and TOPS-20 operating systems, users have the ability to send each other messages. On TOPS-10, a user can send a message with the SEND command; on TOPS-20, a user can send a message by LINKing to you with the TALK command. You set ⎕*GAG* to either 1 or 0. The default is installation-dependent.

```
On TOPS-10, ⎕GAG←1 means "TTY NO GAG" - Accept Messages
            ⎕GAG←0 means "TTY GAG" - Refuse Messages

On TOPS-20, ⎕GAG←1 means "RECEIVE LINKS" - Accept Messages
            ⎕GAG←0 means "REFUSE LINKS" - Refuse Messages
```

On TOPS-10, if you return to monitor level, the ⎕*GAG* setting will not inhibit messages. On TOPS-20, the ⎕*GAG* setting remains in effect at monitor level.

For example:

```
      ⍝TOPS-20

      ⎕GAG
1
      )MON
MONITOR:
@i term
 TERMINAL LA36
 TERMINAL SPEED 300
 RECEIVE LINKS
 REFUSE ADVICE
 RECEIVE SYSTEM-MESSAGES
 TERMINAL NO ^C
@cont
APLSF:
      ⎕GAG
1
      ⎕GAG←0
      )MON
MONITOR:
@i term
 TERMINAL LA36
 TERMINAL SPEED 300
 REFUSE LINKS
 REFUSE ADVICE
 RECEIVE SYSTEM-MESSAGE^C
@


      ⍝TOPS-10

      ⎕GAG
0
      )CONT HOLD
 15:46:24 11-JUL-79 2 BLKS
TTY44) 15:46:24 11-JUL-79
CONNECTED    0:01:07 CPU TIME    0:00:00
2 STATEMENTS 0 OPERATIONS
KILO-CORE-SECS 10

EXIT
```

```
.I TTY

    RZ363A KL #1026/1042 15:46:35 TTY44 system 1026/1042
Connected to Node KL1026(26) Line # 44
Job 61    User MASELLA,S.    [27,26173]
DSKC:    KL1026 System disk DSKC
ERIC:    TOPS-10 Benchmark STR with 7.00 monitor
DSKB:    KL1026 System disk DSKB
NOLC NOTABS NOFORM ECHO CRLF WIDTH:72 GAG NODISPLA
FILL:1 NOTA

.RU APLSF[611,51723]

terminal..LA
APL-10 DECSYSTEM-10 APLSF 2(412)
TTY44) 15:47:08 WEDNESDAY 11-JUL-79 MASELLA,S.    [27,26173]
SAVED  15:46:24 11-JUL-79 2K
      ⎕GAG
0
      ⎕GAG←1
      )CONT HOLD
 15:47:33 11-JUL-79 2 BLKS
TTY44) 15:47:34 11-JUL-79
CONNECTED    0:00:25 CPU TIME    0:00:00
2 STATEMENTS 1 OPERATIONS
KILO-CORE-SECS 9

EXIT

.I TTY

    RZ363A KL #1026/1042 15:47:42 TTY44 system 1026/1042
Connected to Node KL1026(26) Line # 44
Job 61    User MASELLA,S.    [27,26173]
DSKC:    KL1026 System disk DSKC
ERIC:    TOPS-10 Benchmark STR with 7.00 monitor
DSKB:    KL1026 System disk DSKB
NOLC NOTABS NOFORM ECHO CRLF WIDTH:72 NOGAG NODISPLA
FILL:1 NOTAPE TYPE^C

.
```

You can localize ⎕*GAG* in a user-defined function. ⎕*GAG* cannot be saved with your workspace, however, its setting remains unchanged during a)*LOAD* or a)*CLEAR* operation.

4.2.11 ⎕*IO* - Index Origin

The ⎕*IO* (index origin) system variable changes the setting of the index origin. This setting determines whether the values of a vector or array are indexed beginning with position 0 or 1. The default is 1.

The index origin is important in array operations and in conjunction with roll and deal (Sections 3.2.4 and 3.3.3). The value is saved when the active workspace is saved and is only meaningful if it is 0 or 1. This system variable is equivalent to the)*ORIGIN* command, Section 5.5.4. ⎕*IO* is used with the following operations: ⍳*A*, ?*A*, *A*?*B*, *A*[], *A*⍳*B*.

For example:

```
          ⎕IO←1

          ⍳3
    1 2 3
          ⎕←A←2 4⍴⍳6
      1  2  3  4
      5  6  1  2

          +/[2]A
    10 14
          +/[1]A
    6 8 4 6
          +/[0]A
      8 INDEX ERROR
            +/[0]A
             ∧

          ⎕IO←0

          ⍳3
    0 1 2

          +/[2]A
      8 INDEX ERROR
            +/[2]A
             ∧
          +/[1]A
    10 14
          +/[0]A
    6 8 4 6
```

4.2.12 ⎕*LC* - Reporting on Executing Functions

The ⎕*LC* (line counter) system variable is used to obtain a partial report on functions that are currently being executed. It is stored as a vector of the line numbers contained in the state indicator, arranged in order of the most recently suspended function first. The default value for ⎕*LC* is 0.

The ⎕*LC* system variable is particularly useful in branch statements (Section 6.4.1). You can specify that execution is to resume immediately following the line number at which function execution was most recently suspended with ⎕*LC*.

For example:

```
          ∇NEW
    [1]   →1
    [2]   ∇
          NEW
     18 ATTENTION SIGNALED
    NEW[1] →1
             ∧

          ⎕LC
    1
```

4.2.13 ⎕*LX* - Latent Expression

The ⎕*LX* (latent expression) system variable takes the expression you assigned to it before you saved the workspace and executes that expression automatically when the workspace is loaded. The value you assign to ⎕*LX* must be a character string. APL processes the expression as if you had specified ⍎⎕*LX*, the execute function. Any error messages you receive from the use of ⎕*LX* are produced by the execute function (Section 3.4.3). The default value for ⎕*LX* is ''.

The ⎕*LX* system variable is often used to display a message when the workspace in which it is defined is loaded.

For example:

```
      ⎕LX←' ''NOTE NEW LINEPRINTER IN OPERATION'' '
```

This system variable is also useful in restarting a suspended function. For example:

```
      ⎕LX←'→⎕LC'
```

It is also useful in invoking a particular user-defined function (Chapter 6) when you load the workspace. For example:

```
      ⎕LX←' ''STARTUP'' '
```

Upon loading a workspace, APL executes ⎕*LX* when the)*SI* stack is either empty or has a suspended function on top. If a function executes a)*SAVE* command, the function will continue when the workspace is reloaded, but the ⎕*LX* system variable will not be executed because the function on top of the)*SI* stack is not suspended.

4.2.14 ⎕*NUM* - Digits

The ⎕*NUM* system variable is a subset of the ⎕*AV* system variable (Section 4.2.6). ⎕*NUM* contains a vector of the ten digits, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

For example:

```
      ⎕NUM
0123456789

      ⎕AV⍳⎕NUM
305 306 307 308 309 310 311 312 313 314
```

4.2.15 ⎕*PP* - Output Precision

The ⎕*PP* (print precision) system variable determines the precision of noninteger output by allowing you to set the number of significant digits to be displayed. Legal values for ⎕*PP* are integers 1 through 18. The default is 10. The ⎕*PP* system variable does not affect the precision of internal calculations or the display of numeric constants.

For example:

```
      ⎕PP
10

      123456789.123456789
123456789.1

      ⎕PP←5
      123456789.123456789
1.2346E8


      ⎕PP←15

      123456789.123456789
123456789.123457
```

APL rounds off a number if it contains more digits that the setting. The precision you specify is saved when the active workspace is saved. ⎕*PP* is equivalent to the)*DIGITS* command (Section 5.5.1).

The ⎕*PP* system variable is also relevant to the conversion of numbers to characters with the monadic format function (⍕), Section 3.4.6.

4.2.16 ⎕*PW* - Determining the Width of the Output Line

The ⎕*PW* (page width) system variable sets the maximum number of characters that can appear on a terminal output line, before a carriage return/line feed is performed. Legal values for ⎕*PW* are integers 30 through 390. The default is 120. ⎕*PW* has no effect on the display of messages or the length of input lines.

For example:

```
      ⎕PW←30
      ⎕←A←'THIS IS A TEST OF THE PAGE WIDTH VARIABLE.'
THIS IS A TEST OF THE PAGE WID
      TH VARIABLE.
```

The width is saved along with the active workspace. ⎕*PW* is equivalent to the)*WIDTH* command (Section 5.5.6).

4.2.17 ⎕*RL* - Setting a Random Link

The ⎕*RL* (random link) system variable sets the seed of the pseudo-random-number generator in APL. This generator is used with the roll and deal functions (Sections 3.2.4 and 3.3.3). The range of meaningful values you can specify is -1+2*1 through -1+2*35. The default value is 0.

Every time you specify either a roll or deal operation, you change the value of the random link.

For example:

```
      ⎕RL
0
      5?5
1 3 5 2 4
      ⎕RL
30388006192
      5?5
1 4 5 3 2
      ⎕RL
9311234783
```

The value of ⎕*RL* is saved when you save the active workspace.

4.2.18 ⎕*SF* - Setting the Evaluated Input Prompt

The ⎕*SF* (signal for evaluated input) system variable changes the standard signal message used as the prompt in accepting evaluated input. You can use any printing character(s) as the prompt for evaluated output. The default is:

⎕: carriage return/line feed 6 spaces

For example:

```
      A←3+⎕+5
⎕:
      5
      A
13
      B←⎕
⎕:
      'INPUT'
      B
INPUT
      ⎕SF←'WHAT IS YOUR NAME? '
      C←⎕
WHAT IS YOUR NAME? 'SARAH'
      C
SARAH
```

Note that you must enclose the character input within single quotation marks in evaluated input.

4.2.19 ⎕*TIMELIMIT* - Setting a Time Limit

The ⎕*TIMELIMIT* system variable sets a limit to the amount of time you have to respond to a quote-quad input request (⍞) or a quad-del input request (⍔). The range of meaningful values you can specify is -1 to 262143 milliseconds.

Example:

```
      ⎕TIMELIMIT←5000
      A←⍞
YOU HAVE FIVE SECONDS
      A
YOU HAVE FIVE SECONDS
      A←⍞
YOU HAVE FIVE SE                CONDS
 11 VALUE ERROR
       CONDS
        ^
      ⍝RAN OUT OF TIME
      A
YOU HAVE FIVE SE
```

If you exceed the time limit, APL accepts only the data you typed before you ran out of time. Any input accepted ends with a carriage return/line feed.

Note that by specifying a negative argument (-1) to ⎕*TIMELIMIT*, you can set APL to accept type-ahead input. This feature can be useful if you are accepting input from a pseudo-terminal (PTY).

To find out whether you or another user ran out of time, use the ⎕*TIMEOUT* system variable (Section 4.2.20).

4.2.20 ⎕*TIMEOUT* - Reporting on Time Limit

The ⎕*TIMEOUT* system variable reports whether a user ran out of time during a quote-quad input request (⍞) or a quad-del input request (⍔) with a ⎕*TIMELIMIT* set. ⎕*TIMEOUT* is set to either 1 or 0: a 1 means that the user ran out of time, a 0 means the user did not run out of time.

For example:

```
        ⎕TIMELIMIT←5000
        A←⍞
YOU HAVE FIVE SECONDS
        ⎕TIMEOUT
0

        ⍝DID NOT RUN OUT OF TIME

        A←⍞
YOU HAVE FIVE          SECONDS
 11 VALUE ERROR
         SECONDS
          ∧
        ⎕TIMEOUT
1
        ⍝RAN OUT OF TIME
```

The value of ⎕*TIMEOUT* remains constant until you type one of the following:

1. Quote-quad input from the terminal (⍞).
2. Quad-del input from the terminal (⍹).
3. Input from a pseudo-terminal (PTY).

4.2.21 ⎕*TRAP* - Trapping Errors

The ⎕*TRAP* system variable takes an expression you assign to it and executes that expression when an error occurs only during function execution. The value you assign ⎕*TRAP* can be any valid APL expression in the form of a character string. The default value is null, or ⍳0.

You can set ⎕*TRAP* as a global variable or localize it in a function. If an error occurs, APL searches for the most local ⎕*TRAP*. If ⎕*TRAP* is set to anything other than null, APL executes ⎕*TRAP* in the environment of the function where the error occurred.

For example:

```
        )SI

        ∇R;⎕TRAP
  [1]   ⎕TRAP←'→L'
  [2]   A←5
  [3]   B←0
  [4]   C←A÷B
  [5]   ⍝DIVISION BY 0 IS DOMAIN ERROR
  [6]   L:'TRAPPED ERROR, THEN CONTINUED'
  [7]   'EXECUTED LAST LINE'
  [8]   ∇

        R
TRAPPED ERROR, THEN CONTINUED
EXECUTED LAST LINE

        ⎕ERROR
 15 DOMAIN ERROR
R[4]    C←A÷B
           ^

        )SI
```

The following example illustrates what happens without *⎕TRAP* set:

```
        )SI

        ∇G
  [1]   A←5
  [2]   B←0
  [3]   C←A÷B
  [4]   ∇

        G
 15 DOMAIN ERROR
G[3]    C←A÷B
           ^
        )SI
G[3]    *
```

All the errors that are trappable with the ⍎ execute function you can also trap with *⎕TRAP*. *⎕TRAP* also traps the attention signal (two CTRL/C's). In fact, only the following errors cannot be trapped with *⎕TRAP*:

1. Function halts due to setting the trace and stop vectors (*S∆F* and *T∆F*)

2. Input halts from typing ⍟ (.OU) to ⍞ or ⍠ input or typing → to ⎕ input

3. Errors in ∊ or ⊤ execute

Because you can execute a function call from a ⎕*TRAP*, you may want to localize ⎕*TRAP* to avoid unwanted loops.

For information on error handling, refer to Section 6.5.

4.2.22 ⎕*TS* - Reporting Current Time and Date

The ⎕*TS* (time stamp) system variable obtains the current time and date and stores it as a 7-element vector in base 10 format. This vector is known as a timestamp and contains the following elements:

current year, month, day, hour, minute, second, millisecond

For example:

```
      ⎕TS
1979 7 2 13 44 47 70
```

4.2.23 ⎕*TT* - Reporting Terminal Type

The ⎕*TT* (terminal type) system variable contains a value that relates to the type of terminal being used for the current APL session. When you run APL, you specify the terminal type in response to

```
TERMINAL..
```

APL stores this information according to the following table:

Table 4-4
⎕*TT* Terminal Types

Value	Meaning
0	TTY-type terminal
1	TTYCOM terminal
2	LA36 or Tektronix 4013, 4015
3	APL keyboard- or typewriter-paired ASCII/APL terminal
4	APL bit-paired ASCII/APL
5	APL ONTEL
6-9	Reserved
10	2741 Selectric-Type

For example:

```
      ⎕TT
2
```

4.2.24 ⎕*UL* - Reporting the Job Number

The ⎕*UL* (user load) system variable contains the system job number associated with the current APL session. The value is stored in base 10 format.

For example:

```
      ⎕UL
18
```

4.2.25 ⎕*WA* - Reporting the Available Work Area

The ⎕*WA* (work area) system variable contains the amount of available storage space in the active workspace. This value allows you to determine the maximum amount to which your workspace can increase.

The size is given in bytes, not words. APL obtains the value by subtracting the current data-segment size from the maximum data-segment size.

For example:

```
      ⎕WA
73232
```

4.3 SYSTEM FUNCTIONS

The system functions described in the following sections allow you to perform such operations:

1. Expressing the canonical representation of a function and storing the function definition as data
2. Expunging a named object
3. Constructing a name list of labels, variables, or functions and returning the classification of a named object
4. Delaying execution of a function for a specified period of time

There are certain system functions that are relevant only in conjunction with the file system, for example, ⎕*APPEND* and ⎕*ASS*. These systems functions are described in Chapter 7.

System functions are an integral part of the APL language and can be used freely in all APL function definitions. The names of the 13 system functions described in this section all begin with a quad (⎕) character and are reserved words. Like system variables, system functions cannot be copied, erased, or collected in a group.

You access a system function by simply stating its name with arguments, as you would access a primitive or user-defined function.

The system functions described in this chapter are:

⎕*BREAK*	Section 4.3.1
⎕*CR*	Section 4.3.2
⎕*DL*	Section 4.3.3
⎕*EX*	Section 4.3.4
⎕*FI*	Section 4.3.5
⎕*FX*	Section 4.3.6
⎕*NC*	Section 4.3.7
⎕*NL*	Section 4.3.8
⎕*QCO*	Section 4.3.9
⎕*QLD*	Section 4.3.9
⎕*QPC*	Section 4.3.9
⎕*SIGNAL*	Section 4.3.10
⎕*VI*	Section 4.3.11

4.3.1 ⎕*BREAK* - Suspending Execution

Format

⎕*BREAK* arg

where

arg is any APL object.

The ⎕*BREAK* system function suspends execution of the function in which it is contained and returns you to immediate mode.

⎕*BREAK* is a monadic system function. It takes any APL object as an argument and prints that argument before breaking to the terminal.

For example:

```
      ∇FUNC
[1]   'FIRST LINE'
[2]   ⎕BREAK 'BREAK AT LINE 2'
[3]   'RESUME AT LINE 3'
[4]   ∇
      FUNC
FIRST LINE
BREAK AT LINE 2

      )SI
FUNC[2] *

      →⎕LC+1
RESUME AT LINE 3
```

To return to function execution after a break, you can either go to a specific line number (→3) or use the system variable ⎕*LC*. Specifying ⎕*LC* would return you to the line where the ⎕*BREAK* executes. To resume at the line after the breakpoint, specify ⎕*LC*+1.

Note that ⎕*BREAK* is illegal from immediate mode and ⍎ execute.

For more information on the use of ⎕*BREAK*, refer to Section 6.5.

4.3.2 ⎕*CR* - Obtaining a Canonical Representation

Format

⎕*CR* arg

where

arg can be a function name enclosed in single quotation marks or a variable name whose value is the name of the function. The rank of the arg (for example *A*) cannot be greater than 1, 1≥⍴⍴*A*.

The function you specify must be a defined and unlocked function.

The ⎕*CR* (canonical representation) system function provides a canonical representation of a defined function. A canonical representation is a character matrix with rows consisting of the original lines of the function definition reformatted to be of equal length. The ∇ symbols, line numbers, and brackets are removed from the definition. Lines that contain labels are shifted to the right so all text begins at the same character position. Lines are then right-padded with blanks to make all lines equal to the longest line of the function. This formatting allows you to treat the function as data.

The following example illustrates the original function to be referenced by ⎕*CR* and the matrix or canonical representation that results from the operation of the system function.

For example:

```
          ∇MEANX←NSUBJ MEAN X
    [1]   ⍝SUM VECTOR X
    [2]   SUMX←+/X
    [3]   MEANX←SUMX÷NSUBJ
    [4]   ∇

          ⎕←A←⎕CR 'MEAN'
    MEANX←NSUBJ MEAN X
    ⍝SUM VECTOR X
    SUMX←+/X
    MEANX←SUMX÷NSUBJ

          ⍴A
    4 18

          X←8 6 3 9 5 4 2 1 7 4
          10 MEAN X
    4.9
```

If the argument to ⎕*CR* does not represent the name of a defined and unlocked function, the resulting matrix is a null matrix, 0 by 0. APL returns a 9 *RANK ERROR* if the function name is not a vector or a scalar, and a 15 *DOMAIN ERROR* if the argument is not a character array or if the name is not enclosed in quotation marks.

The ⎕*FX* system function (Section 4.3.6) reverses the effects of ⎕*CR*.

4.3.3 ⎕*DL* - Delaying the Execution of a Function

Format

    ⎕DL arg

where

> arg is the number of seconds you want to delay execution. The argument must be a scalar or vector with a single numerical value (1≥⍴⍴*ARG*) or APL returns a 9 *RANK ERROR* or a 15 *DOMAIN ERROR*. There is no limit to the value of the argument.

Although ⎕*DL* specifies the desired duration of the delay of the function, the actual delay can be somewhat different. Other demands on the APL system at the time that the ⎕*DL* is issued can affect the accuracy of the delay. In addition, you can use a single attention signal, CTRL/C, at any time to abort the delay and cause an interrupt in the function in which the ⎕*DL* appears.

For example:

```
          DEL←⎕DL 2
          DEL
    1
```

DEL is a scalar value equal to the actual delay incurred as a result of the 2-second ⎕*DL* specification.

The ⎕*DL* function uses a negligible amount of computer time; you can issue it freely in situations where tests are required at periodic intervals to determine whether or not an event has occurred as expected.

This is helpful in simplifying interuser and interprogram communication of various kinds. Another way to wait specifically for input is to use ⎕*TIMELIMIT*, Section 4.2.19.

4.3.4 ⎕*EX* - Erasing a Named Object

Format

    ⎕EX arg

where

> arg is a function name enclosed in single quotation marks or a variable name whose value is a matrix of function names. Therefore, the argument can have a rank of 2 or less (2≥⍴⍴*ARG*).

The ⎕*EX* (expunge) system function erases a variable or function name so that you can reuse it without confusion. ⎕*EX* operates on global or dominant local variables. You cannot erase a named object that refers to a label, a group, a suspended or pendent function, or a system variable.

When you erase a name (or names) ⎕*EX* returns a 1 or a 0 depending upon whether the name was successfully erased. A 1 signifies that the name was erased; a 0 signifies that the name cannot be erased. You also receive a 0 if the name is not a legal APL variable name.

APL returns a 9 *RANK ERROR* if the argument has a rank higher than 2, and returns a 15 *DOMAIN ERROR* is the argument is not a character string.

For example:

```
          )FNS
ABCD       GROW       TEST

          A←3 4⍴'ABCDTESTGROW'

          A
ABCD
TEST
GROW
          ⎕EX A
1 1 1
          )FNS
```

(APL outputs at blank lines)

4.3.5 ⎕*FI* - Converting Characters to Numerics

Format

⎕*FI* arg

where

arg is a character scalar, vector, or one-element array.

The ⎕*FI* system function takes a character argument and converts it into a numeric, placing zeros in each position that does not correspond to a valid number. Note that a minus sign preceding a number is not part of the number but is rather an operation to be performed on the number. However, in the expression ¯5, the negative sign is a valid part of the number in APL.

For example:

```
          ∇Z←AVERAGE
[1]       Z←, ⍞◊⎕←'ENTER A LIST OF NUMBERS'
[2]       Z←(⎕VI Z)/⎕FI Z
[3]       Z←(+/Z)÷⍴Z
[4]       ∇

          AVERAGE
          A LIST OF NUMBERS
1 3.5 A 0 +2 ¯.5 6. .
2
```

In the previous example,

⎕*VI* is 1 1 0 1 1 1 1 0
⎕*FI* is 1 3.5 0 0 2 ¯.5 6 0

4.3.6 ⎕*FX* - Establishing a Function

Format

    ⎕*FX* arg

where

    arg is the name of the character matrix that contains a canonical representation of a function. The rank of the argument must equal 2, 2=⍴⍴*ARG*.

The ⎕*FX* system function reverses the operation of the ⎕*CR* system function (Section 4.3.2). If a function already exists in your workspace with the same name, ⎕*FX* replaces it.

For example:

```
          A[3;6]←'X'

          ⎕FX A
    MEAN
          ∇MEAN[⎕]∇
        ∇  MEANX←NSUBJ MEAN X
    [1]   ⍝SUM VECTOR X
    [2]    SUMX←×/X
    [3]    MEANX←SUMX÷NSUBJ
        ∇
          X
    8 6 3 9 5 4 2 1 7 4
          10 MEAN X
    145152
```

The rules for local names apply to the names of any functions established by the ⎕*FX* function.

⎕*FX* will not establish a function if the name of the function to be established is the same as that of an existing label, variable, or group, or an existing function that is currently pendent or suspended. A pendent function is usually one that is awaiting return from another function. The ⎕*FX* executes properly if the matrix referenced by ⎕*FX* is identical to a canonical representation, except for the addition of blank characters in rows other than those consisting only of blanks.

If ⎕*FX* cannot establish a function, APL returns a scalar index representing the row in the matrix where the error occurred. No change is made to any function or matrix in your workspace.

APL returns a 9 *RANK ERROR* if the argument is not a matrix, and a 15 *DOMAIN ERROR* if the argument is not a character array. If ⎕*FX* is successful, its value is the name of the function defined.

4.3.7 ⎕*NC* - Returning a Name Classification

Format

```
    ⎕NC arg
```

where

arg is a character matrix of names or a vector or scalar consisting of one name. The rank of the argument is 2≥ρρ*ARG*.

The ⎕*NC* (name classification) system function returns the classification of a name or group of names. If the argument is a matrix, ⎕*NC* returns the class of the name represented by each row in the matrix. If the argument is a vector or scalar, ⎕*NC* returns the class of a single name. ⎕*NC* returns a numeric value representing each name class. Table 4-5 lists these values.

Table 4-5
⎕*NC* Classes

Value	Meaning
0	Name available for any use
1	Label name
2	Variable name
3	Function name
4	Not available for use as a name

A value of 4 implies that the argument is not a valid name or that it is currently in use as a group name (Section 5.4.4).

For example:

```
      ⎕NC 'AVER'
0
```

4.3.8 ⎕*NL* - Constructing a List of Labels, Variables, or Functions

Format

⟦a⟧⎕*NL* n

where

a is a scalar or vector of alphabetic characters. The letters must be supplied in alphabetic order. This parameter is optional. (Do not type the square brackets.)

n is one or more integer scalars or a vector from the following list:

Values	Meaning
1	Labels
2	Variables
3	Functions

The ⎕*NL* (name list) system function can be either monadic or dyadic depending on whether you supply the left argument. In both forms, the function constructs a list of named objects residing in the active workspace. The "n" parameter identifies the type of named objects to be included in the list.

For example:

```
      X←⎕NL 1 2
```

causes the names of all labels and variables in the workspace to be included in the name list *X* in alphabetic order. Each row of the matrix will contain the name of one label or variable. The number of columns is determined by the length of the longest name. ⎕*NL* fills the shorter names with blanks to the length of the longest name.

The dyadic form of ⎕*NL* allows you to restrict the name list to names beginning with specified characters by including an "a", left argument, in the expression. For example:

```
      NLIST←'ABCDEF' ⎕NL 3
      NLIST
AVERAGE
CAR
FUNC
```

causes a name list to be constructed of function names whose initial letters are *A* through *F*; the list is arranged in alphabetic order.

The ⎕*NL* system function is useful for a variety of purposes. Some of these are described below:

1. ⎕*NL* can interact with ⎕*CR* (Section 4.3.2) in creating functions that can automatically display the definitions of all or a subset of functions in the workspace. You can also use it to analyze interactions between variables and functions. (Remember to remove the blanks on the right, if any exist.)

2. In conjunction with *⎕EX* (Section 4.3.4), the *⎕NL* function can cause all of the named objects in a certain category to be erased dynamically. It also helps the design of a function that can be used to clear a workspace of all but a preselected collection of named objects.

3. In its dyadic form, *⎕NL* can guide you in choosing names while developing or interacting with a workspace.

The following example illustrates the construction of a matrix containing the names of variables in the active workspace that begin with the letter *V*.

```
      NLIST←'V' ⎕NL 2
      NLIST
VAR1
VAR2
VAR203
VAR99
VBMAX
```

4.3.9 *⎕QLD*, *⎕QCO*, *⎕QPC* - Loading and Copying a Workspace

Format

⎕QLD arg
⎕QCO arg
⎕QPC arg

where

arg in each case is a character vector (enclosed in single quotation marks) representing the workspace name, an optional password and an optional list.

The password is the password associated with the owner of the workspace. The password is necessary only if you are not privileged to access the particular workspace. (Do not type the square brackets.)

The list is an optional parameter used to identify specific objects to be copied. If you omit this parameter, all functions, variables, and groups in the workspace are copied. (Do not type the square brackets.)

The *⎕QLD*, *⎕QCO*, and *⎕QPC* system functions perform the same operations as the *)LOAD*, *)COPY*, and *)PCOPY* commands described in Chapter 5. *⎕QLD* loads a workspace, *⎕QCO* copies a workspace, and *⎕QPC* copies a workspace with certain protection considerations.

Unlike the system commands, the system functions do not return messages to verify a successful load or copy. The system functions, however, return the messages *OBJECTS NOT FOUND* and *NOT COPIED*: when applicable. These are not inhibited.

Also, APL does not output a blank line to indicate that no value was returned. Therefore, you can use one of these system functions alone or as a function line without blank lines appearing on your terminal. If the ⎕*LX* has a value in the workspace, it also executes with the ⎕*QLD*.

The ⎕*QPC* system function does not cause APL to return the names of objects that were not copied.

If an error occurs during the execution of any of these three system functions, APL prints an error message. Therefore, you can trap errors as usual with the use of the execute function (Section 3.4.3).

For example:

```
        )LOAD T
SAVED  13:53:13 11-JUL-79 5P
        A
1
        )CLEAR
CLEAR WS
        ⎕QLD 'T'
        )WSID
T <007> [4,204]
        A
1
        )CLEAR
CLEAR WS
        )COPY T A B
SAVED  13:53:13 11-JUL-79 6P
        A
1
        B
2
        C
  11 VALUE ERROR
         C
          ^
        )CLEAR
CLEAR WS
        ⎕QCO 'T'
        A
1
        B
2
        C
3
        )CLEAR
CLEAR WS
        ⎕QCO 'T A B'
        A
1
        B
2
        C
  11 VALUE ERROR
         C
          ^
```

```
      )CLEAR
CLEAR WS
      )COPY T A D
SAVED  13:53:13 11-JUL-79 6P
OBJECTS NOT FOUND:        D
      A
1
      B
 11 VALUE ERROR
       B
        ∧
      )CLEAR
CLEAR WS
      ⎕QCO 'T A D'
OBJECTS NOT FOUND:        D
      )CLEAR
CLEAR WS

      A←20
      )PCOPY T
SAVED  13:53:13 11-JUL-79 6P
NOT COPIED:       A
      A
20
      B
2
      C
3
      )CLEAR
CLEAR WS
      A←20
      ⎕QPC 'T'
NOT COPIED:       A
      A
20
      B
2
      C
3
      )CLEAR
CLEAR WS
      Z←⎕QCO 'T A D'
OBJECTS NOT FOUND:        D
      ⍴Z
0
      )CLEAR
CLEAR WS
      Z←⎕QPC 'T A D'
OBJECTS NOT FOUND:        D
      ⍴Z
0
```

4.3.10 ⎕*SIGNAL* - Signalling Errors

The ⎕*SIGNAL* system function allows you to pass an error up the stack ()*SI*) one level to the caller of the function in error. The syntax of ⎕*SIGNAL* also allows you to make up your own messages. The syntax has the following format:

error message ⎕*SIGNAL* error number

where

error message is an optional character string.

error number is a scalar or a single-element array of any rank.

The value you supply as the error number can be any APL error number as listed in Appendix A or a number from 500 to 999 inclusive. The message sent by ⎕*SIGNAL* is stored in ⎕*ERROR*. The error number you supply ⎕*SIGNAL* becomes the number in the first line of ⎕*ERROR*. If error number is the number of an APL error, then the message stored in ⎕*ERROR* is the error message that coincides with that number, regardless of the left argument. If you supply your own number, you can either make up a message or leave the left argument blank, in which case you would receive the default message:

ERROR SIGNALED BY FUNCTION

For example:

```
        ⍝FUNCTION F HAS ⎕SIGNAL
        ∇F A
[1]     →(A>0)/3
[2]     'WILL NOT ACCEPT NEGATIVE NUMBERS'⎕SIGNAL 501
[3]     'FUNCTION CONTINUES NORMALLY'
[4]     ∇

        ⍝FUNCTION H CALLS F

        ∇H A
[1]     F A
[2]     ∇
        H 5
FUNCTION CONTINUES NORMALLY
        ⎕ERROR

        )SI

        H ¯7
501 WILL NOT ACCEPT NEGATIVE NUMBERS
H[1]     F A
         ^

        ⎕ERROR
501 WILL NOT ACCEPT NEGATIVE NUMBERS
H[1]     F A
         ^

        )SI
H[1]     *
```

Notice that the error is signaled at the level of the caller, *H*, not *F*.

It is illegal for you to execute ⎕*SIGNAL* in immediate mode or with ⍎ execute or ∊ execute.

For more information on the use of ⎕*SIGNAL* refer to Section 6.5.

4.3.11 ⎕*VI* - Validating Input

Format

⎕*VI* arg

where

arg is a character vector, scalar, or 1-element array.

The ⎕*VI* (validating input) system function is used in conjunction with the ⎕*FI* system function (Section 4.3.5). While ⎕*FI* converts a character vector into a numeric vector, ⎕*VI* returns a Boolean vector that contains a 1 in each position corresponding to a valid number that can be converted with ⎕*FI*. It returns a 0 for nonvalid numbers.

For example:

```
      A←'1.5 3 A ¯5 3.. 1.0E15 +1 ¯3'

      ⎕VI A
1 1 0 1 0 1 0 1

      ⎕FI A
1.5 3 0 ¯5 0 1.000000000E15  0 ¯3

      (⎕VI A)/⎕FI A
1.5 3 ¯5 1.000000000E15  ¯3
```

CHAPTER 5

SYSTEM COMMANDS

5.1 INTRODUCTION

APL provides a wide variety of system commands to communicate with the APL system and to control the operating environment in which the APL session is conducted. System commands allow you to examine or change the state of the system. For example, they allow you to:

1. Clear, save, or name the active workspace
2. Load or copy a workspace from a secondary storage device
3. List workspace, variable, function, and group names
4. Determine memory and workspace size, time and system resources used, and version and device information
5. Display the status of functions and local variables in the workspace
6. Set the index origin, the maximum number of significant digits, and the output line width

System commands are not considered a part of the APL language itself, but can be viewed as an interface between you and the language interpreter. System commands implemented for use with the APL file system are described in Chapter 7. Appendix B provides a summary of the format of all system commands in alphabetic order.

This chapter is structured in the following way. Section 5.1 provides an overview of the format for system commands, the two ways of using system commands (action and inquiry), characteristics of workspaces, and APL libraries that allow you to share programs with other APL users. Sections 5.2 through 5.7 describe the system commands themselves by category:

Section	Commands
5.2	Basic Workspace-Control
5.3	Extended Workspace-Control
5.4	Workspace-Content
5.5	Workspace-Environment
5.6	APL Termination
5.7	Miscellaneous

Section 5.8 discusses the special function of the execute function (⍎) in relation to system commands.

5.1.1 System Command Format

Unlike other APL statements, system commands begin with a right parenthesis, as shown in the following format:

)command-name ⟦parameter-list⟧

You can abbreviate the command-name to its shortest unique form, which usually requires no more than four characters. Some system commands require one or more parameters or arguments in the command string. If you include required or optional parameters, you must separate the individual elements of the command string with at least one space. (Do not type the square brackets.)

The following examples illustrate the format of several system commands:

```
      )DIGITS 5
WAS 10
      )DIGI
5
      )DIGIT
5
      )SAVE MYWORK
 14:38:11 11-JUL-79 1 PGS
      )COPY WS40-SESAME A B C VAR6 N
SAVED  14:36:45 11-JUL-79 6P
```

The first three examples invoke the same system command, *)DIGITS*, since the first four letters of each of the command names are the same; note that extending a command name beyond its unique form (four letters) has no effect. In the fourth example, *MYWORK* serves as an additional argument to the *)SAVE* system command. The fifth example illustrates the inclusion of a series of parameters in the *)COPY* command. Only *WS*40 is a required argument; the password and name list are optional (see Section 5.4.1).

5.1.2 Action and Inquiry Commands

You can use APL system commands for two distinct purposes:

To obtain information - inquiry commands

To change the state of a workspace or operating environment - action commands

Action commands cause some change in the state of the APL system. Inquiry commands report on the state of the system but do not change this state in any way.

The *)MON* command is an example of an action command. To return to operating system command level, specify the following:

```
        )MON
MONITOR:
@
```

The *)SI* command, on the other hand, operates as an inquiry command. It reports the status of APL program execution. For example:

```
        )SI
PRIMES[2] *
```

You can use many system commands as both action and inquiry commands. The distinction between action and inquiry is made by the inclusion of optional parameters. The *)ORIGIN* command is an example of a command you can use in both ways. The *)ORIGIN* command can either (1) change the index origin setting associated with an array specification (action) or (2) return the current setting of the index origin (inquiry).

The first example below shows the use of *)ORIGIN* as an action command; this command sets the index origin to 0 and also reports that the previous setting was 1. The second example shows the use of *)ORIGIN* as an inquiry command; this command reports that the current setting of the index origin is 0.

```
        )ORIGIN 0
WAS 1
        )ORIGIN
0
```

5.1.3 Workspace Characteristics

The APL system uses a buffer in your memory area to store functions, variables, and values, information on the status of functions, group descriptions, and any temporary results obtained while executing APL statements. When available in memory, this buffer area is known as the active workspace.

You can enter system commands that cause this active workspace to be saved on a secondary-storage device; subsequently, you can load the saved workspace into the buffer area to function as the active workspace once again. The term "workspace" is used to refer to either the active workspace or a version of an active workspace now saved in secondary storage.

Many of the system commands described in this chapter aid in changing the status of a workspace. For instance, you can clear, save, load, name, lock, and delete a workspace. You can also copy functions, variables, and other elements from a saved workspace into an active workspace and display their names. Workspace size, owner, and password information can be reported. As an APL user, you have extensive control over the activity and characteristics of the workspace in your system.

5.1.3.1 Workspace Names - Each APL workspace defined in your disk area has a unique name associated with it. In the command formats presented in this chapter, this name is represented by the parameter "wsname". The workspace name has five distinct parts:

1. Device name
2. Filename
3. Extension or type
4. Protection code
5. Directory

Legal formats for these name components correspond closely to standard TOPS-10 naming conventions and are summarized on the following page. For TOPS-20 users not familiar with TOPS-10 conventions, refer to Appendix D.

Part	Format
Device name	Maximum of six characters followed by a colon (for example, *MTA*:).
Filename	Maximum of six characters (for example, *TEST*01). The rest will be ignored.
Extension or file type	Period or comma followed by a maximum of three characters (for example, .*APL*). If you are using TTY mnemonics, you must use a comma (for example, ,APL).
Protection code	Octal number in the range 0 through 777, enclosed in angle brackets (for example, (<377>).
Directory	Directory is a project-programmer number, enclosed in square brackets (for example, [145,7231]). For TOPS-20 users with directory names, use the TRANSL command to find out the project-programmer number associated with a directory name. (Refer to Appendix D for more information on TRANSL.)

Characters you use in device names, filenames, extensions, and types can be *A*-*Z*, <u>*A*</u>-<u>*Z*</u>,∆,<u>∆</u>, and 0-9. An example of a complete workspace name is:

```
DSK:MYWORK.APL<157>[147,3216]
```

The system commands do not always require all five parts of the workspace name in the command format. When you omit parts of the name, the default values take over. These defaults are summarized in Table 5-1.

Table 5-1
Workspace Name Defaults

Component	Default
Device name	*DSK:*
Filename	Name of active workspace
Extension or type	*.APL*
Protection code	Installation-dependent
Directory	Your directory

5.1.3.2 The *CONTINUE* Workspace - When you terminate an APL session with the)*CONTINUE* command (see Section 5.6.2) APL saves your workspace on disk and names it *CONTIN.APL*. The *CONTINUE* workspace is an image of the active workspace as it existed at the time of termination. It has the workspace name:

```
DSK:CONTIN.APL<std prot>[directory]
```

where

<std prot> is an installation-dependent standard protection code

[directory] is your directory

APL recognizes *CONTIN* as a special workspace. If a *CONTINUE* workspace exists in your disk area, APL loads it as your active workspace when you begin the APL session. This means that when you run APL instead of having a clear workspace, you will automatically have the workspace you saved with)*CONTINUE*. The *CONTINUE* workspace replaces any existing one on disk.

You can also backup your workspace periodically by setting the ⎕*AUS* system variable to 1. Refer to Section 4.2.5 for more information on ⎕*AUS*.

When you *)CALL* something, your active workspace is in *CONTIN.APL* and will start up at the line 1 plus the *)CALL* line.

5.1.3.3 Workspace Passwords - In addition to the unique name, each APL workspace can also have a password associated with it. A password begins with a hyphen and can have up to eight more characters. The default is a hyphen (-). This is considered a null password. For example:

```
    -SESAME

    -
```

Workspace passwords provide additional workspace protection. If you want to use a password-protected workspace, you must specify the password associated with that workspace before APL allows you to retrieve it from secondary storage.

5.1.3.4 Groups - Various functions and variables in a workspace can sometimes be easier to work with when they are treated as elements in a single logical collection. This approach is aided by the "group" concept in APL, which allows you to treat functions and variables as logical entities. Several system commands are available that allow you to define a new group, list the members of a group, add members to or delete members from an existing group, erase or "undefine" a group. See Sections 5.4.4 through 5.4.6.

5.1.3.5 The State Indicator - Every APL workspace contains a status vector known as the state indicator. This indicator stores information on the execution of functions within the workspace. You can obtain a report on the status of APL functions by issuing an *)SI* or *)SIV* system command (Sections 5.4.8 and 5.4.9). These commands list the contents of the state indicator, which identifies suspended and pendent functions. A suspended function is one that has stopped executing for some reason. A pendent function is one that contains a call to a function that has not completed. The pendent function is waiting for the called function to return.

If the state indicator is "empty", no functions are currently suspended or pendent. The use of the state indicator in debugging and executing functions is described in Section 5.4.8.

5.1.4 APL Libraries

A special library facility is available through the APL system that allows you to make your programs available to all APL users. You can save programs in workspaces on a library device and subsequently allow other users to retrieve them. All workspaces you store on a library device must be assigned a library-device name as the device name portion of the workspace name. A library-device name consists of the characters *LIB*, followed (without intervening blanks) by an integer in the range 1 through 999, followed by a colon. Some examples of possible library-device names follow:

```
LIB1:

LIB445:

LIB001:
```

The third example, *LIB*001:, is equal to both *LIB*1: and *LIB*01:.

Workspaces specified with a library device do not have file extensions or file types associated with them.

CAUTION

LIB: activates a special password system, so if the library workspaces are removed to another area, you might need to rename them.

5.2 BASIC WORKSPACE-CONTROL COMMANDS

This section describes the basic workspace-control commands. These commands allow you to manipulate APL workspaces in a variety of ways. You can:

1. Clear or name the active workspace
2. Delete workspaces when no longer needed
3. List the names of workspaces in your area or in a library
4. Specify, change, or return a password for a workspace
5. Save the active workspace on a secondary storage device and retrieve it when required

5.2.1)*CLEAR* - Clearing the Active Workspace

Command

```
      )CLEAR
```

Example

```
          )CLEAR
      CLEAR WS
```

The)*CLEAR* system command closes all open files and clears the active workspace by replacing it with a special workspace known as the clear workspace.)*CLEAR* resets all the workspace constants to their defaults. APL also gives you a clear workspace when you begin a work session unless you have the *CONTINUE* workspace (*CONTIN.APL*) in your disk area. The clear workspace contains the default values for all workspace related system functions:

1. Contains no functions, groups, or variables.

2. Has an index origin of 1 (⎕*IO*).

3. Has an output line length determined by the operating system width specification. You can change the output width with ⎕*PW*. See Section 4.2.16.

4. Displays numbers with 10 significant digits. This output precision can be changed with ⎕*PP*. See Section 4.2.15.

5. Has a clear symbol table and state indicator.

6. Has the name *CLEAR WS*
   Cannot be saved without being given a name with either)*WSID*, or)*SAVE*.

7. Has the null password hyphen (-).

8. Requests quad input with the message ⎕: followed by a carriage return/line feed and six blanks. (⎕*SF*)

APL uses a symbol table to record function, variable, and group names and constants. The size of this symbol table expands as new names are specified and is limited only by the size of the workspace.

5.2.2 *)DROP* - Deleting Stored Workspaces or Files

Command

)*DROP* wsname ⟦switch-list⟧

Examples

```
          )DROP INT.B20
    9:15:30   6-JUL-79
             )LIB TEST.*
    DSK:
      TEST.ABI
      TEST
      TEST.B20
      TEST.MAC
             )DROP TEST.*
    DROPPED:
      TEST.ABI
      TEST
      TEST.B20
      TEST.MAC
      9:16:14   6-JUL-79
```

The)*DROP* system command is an action command that deletes stored workspaces or files.)*DROP* can delete any system file for which you have the necessary protection privileges. You can erase one, several, or all of the files on a directory device. To delete a single workspace or file, specify the name as the parameter. As described in the)*LIB* command, Section 5.2.3, you can use an asterisk as a wildcard designator. If you use the asterisk, APL lists the deleted files. In both cases, APL displays the time and date of the)*DROP* request.

You can also include a switch-list in the)*DROP* command string. These switches are the same as the ones listed in Table 5-2. If you include a switch,)*DROP* displays the information applicable to the file just prior to its deletion. For example:

```
             )DROP WS9 /B
    DROPPED:
      FILE       BLKS
       WS9         1
                   1
      9:17:23   6-JUL-79
```

The example above displays the number of blocks freed by deleting the workspace named *WS*9.

On TOPS-20, the file is deleted but not expunged.

5.2.3 *)LIB* - Listing Workspace Names

Command

```
      )LIB ⟦wsname⟧⟦switch-list⟧
```

Examples

```
        )LIB
DSK:
 ALPHA
  CHAR
GEORGE
 PRIME
        )SAVE WS40
 14:56:23 11-JUL-79 1 PGS
        )LIB
DSK:
 ALPHA
  CHAR
GEORGE
 PRIME
  WS40

        )SAVE WS40.VAR
 14:57:32 11-JUL-79 1 PGS
        )LIB WS40.*
DSK:
  WS40
  WS40.VAR



        )LIB *.*
DSK:
 ALPHA
  CHAR
GEORGE
 LOGIN.CMD
 LOGIN.EXE
 PRIME
  WS40
  WS40.VAR

        )LIB LOGIN.CMD /B
   FILE     BLKS
DSK:
 LOGIN.CMD     1
               1
```

The *)LIB* system command is an inquiry command that displays a list of workspaces in your disk area. *)LIB* assumes that any file in your disk area with the extension or file type *.APL*, contains a workspace.

You can use the *)LIB* command to list the names of all or selected files on any directory device. These files need not be APL workspaces.

If you specify the wsname, you can specify the filename or file type to be displayed. You can identify a particular file or use the "wildcard" character, the asterisk, to substitute for the filename and/or extension or type. The asterisk matches any name. For example, this command lists the names of all files that have *WS40* as their filename:

```
)LIB WS40.*
```

This command lists the names of all files on device *DSKH*:.

```
)LIB DSKH:*.*
```

The optional switch-list parameter is used to obtain information about files on directory devices. A description of all switches supported with the *)LIB* command are listed in Table 5-2. You can specify more than one switch, but each switch must consist of a slash(/) followed by one of the letters shown in the table. The information returned is displayed on the same line as the filename.

Table 5-2
)LIB Switches

Switch	Meaning
/A	Access: the date the file was last read (disk only)
/B	Blocks: the number of blocks required for the file (Divide the number of blocks by four to determine pages.)
/C	Creation: The creation date of the file (disk only)
/L	Long: same as typing */B/P/C*
/M	Mode: the mode in which the file was written (disk only); TOPS-10 only
/N	No header: suppresses printing of the display header line
/P	Protection: the protection code associated with the file (disk only)
/T	Time: the creation time of the file (disk only)

5.2.4 *)LOAD* - Retrieving a Workspace

Command

)LOAD ⟦magtape-position⟧ wsname ⟦password⟧

Examples

```
          )LOAD WS35
SAVED     15:49:56 11-JUL-79 2K
          )LOAD LIB1:APLSF
SAVED      9:52:57 13-SEP-77 4K

          )LOAD PRIME
SAVED     14:52:57 11-JUL-79 5P
          )LOAD AVER
SAVED     15:45:03 24-OCT-78 5P
```

The *)LOAD* system command is an action command that retrieves a workspace from a secondary storage device. When you load a workspace, it becomes your active workspace, replacing the currently active workspace and destroying its contents. You must specify the name of the file in order to retrieve it. However, APL assumes the rest of the wsname; that is, it assumes *.APL* as the file type or extension, disk as the storage device, current user directory and a null password (-). If a password was submitted when the workspace was saved, you must specify it, or APL will not retrieve the workspace.

If the workspace is stored on magnetic tape, you can specify the number of tape marks to skip before APL tries to load the workspace. The magtape position is an integer corresponding to the number of end-of-file marks to skip on the tape.

When you load a workspace the *)LOAD* system command responds by displaying the word *SAVED*, followed by the time and date when the workspace was saved, followed by the size of the newly active workspace. If the newly active workspace contains a suspended function, APL also prints an asterisk (*).

The ⎕*QLD* system function, Section 4.3.9, performs the same operation as the *)LOAD* command, except without verifying messages.

5.2.5 *)PASSWORD* - Determining the Workspace Password

Command

)*PASSWORD* ⟦password⟧

Examples

```
          )PASSWORD -SESAME
WAS -
          )PASSWORD
-SESAME
          )PASSWORD -
WAS -SESAME
          )PASSWORD
-
```

The)*PASSWORD* system command allows you to either display the current password associated with a workspace or change the password. The password parameter you supply must begin with a hyphen and can contain up to eight more characters from *A*-*Z*, $\underline{A}$-$\underline{Z}$; Δ, $\underline{\Delta}$, 0-9; the first character after the hyphen must be alphabetic (*A*-*Z*, $\underline{A}$-$\underline{Z}$, Δ, $\underline{\Delta}$). The default or null password is a hyphen (-).

5.2.6 *)SAVE* - Saving a Copy of the Active Workspace

Command

)*SAVE* ⟦magtape-position⟧⟦wsname⟧⟦password⟧

Examples

```
          )WSID
CLEAR WS
          )SAVE
 60 WS NOT SAVED, THIS WORKSPACE IS CLEAR WS
          )WSID WS30
WAS CLEAR WS
          )SAVE
 16:08:40 11-JUL-79 1 PGS   WS30 [4,204]
          )WSID WS10
WAS WS30 [4,204]
          )SAVE
 16:08:51 11-JUL-79 1 PGS   WS10 [4,204]
          )WSID WS30
WAS WS10 [4,204]
          )SAVE WS10
 60 WS NOT SAVED, THIS WORKSPACE IS WS30 [4,204]
          )WSID WS35
WAS WS30 [4,204]
```

The)*SAVE* system command is an action command that saves a copy of the active workspace on a secondary storage device. The saved workspace can be stored as a file on disk, DECtape[1], or magnetic tape.

[1]Some systems do not support DECtapes. Check with your System Administrator.

The *)SAVE* system command assumes that you want to save your active workspace on disk. All three parameters in the command string are optional. If you specify a wsname, *)SAVE* stores the active workspace under that name. The default wsname is the name of the currently active workspace. In both cases, the file type or extension is *.APL*. The protection code is the standard one for your installation. As shown in the first example, APL will not save a clear workspace. If your workspace is clear, you must use *)WSID* to give it a name before you can use the *)SAVE* command.

If you are saving a workspace on magnetic tape, you can also specify the position at which the save is to start. The magtape-position parameter in the command string is an integer representing the number of end-of-file marks you want to skip before the save begins. If you omit this parameter, APL makes no attempt to position the magnetic tape. For example:

```
      )SAVE 3 MTA1:WS35
16:26:49 11-JUL-79 3 BLKS
```

In the example above, APL skips three tape marks before it starts to save the workspace.

When you save a workspace, you have the option of saving it under its current name (check the *)WSID*) or renaming it. However, you cannot save a workspace under a name that already exists in your storage area unless the *)WSID* is that name. APL refuses to save the workspace. If you specify a new name with the *)SAVE* command, you not only store your active workspace under that name but also change the name of the currently active workspace to the new name specified.

NOTE

If your current *)WSID* is the same as a workspace you have already saved and you save it under this name, APL overwrites the old file with the new one.

The *)SAVE* system command also provides the option of specifying a password for your workspace. The default is a null password (-). Subsequently, you must know the password of the workspace to retrieve it from storage.

If you interrupt a function execution by typing two CTRL/Cs and then save your workspace, the function is suspended in your storage area. Therefore, when you load this workspace, the function does not continue automatically. You can cause an automatic start-up by using the *⎕LX* system variable (Section 4.2.13).

If you execute a *)SAVE* command within a function, for example, ⍎ '*)SAVE*', APL saves the workspace and continues executing the function. The next time you load that workspace APL will begin the session by executing that particular function at the line after the ⍎ '*)SAVE*'. APL will not execute *⎕LX* in this instance. *⎕LX* executes only if there is a suspended function on top of the *)SI* stack.

The *)SAVE* command responds by displaying the time, date, and amount of space required to store the workspace. If you have not included the wsname, APL also displays the current name.

5.2.7 *)WSID* - Identifying the Active Workspace

Command

)*WSID* ⟦wsname⟧⟦password⟧

Examples

```
        )WSID DSK;MYWORK,APL
WAS CLEAR WS
        )WSID
MYWORK [4,204]
        )WSID MTA1;
WAS MYWORK [4,204]
        )WSID
MTA1;MYWORK [4,204]
        )WSID[4,311]
WAS MTA1;MYWORK [4,204]
        )WSID
MYWORK [4,311]
```

The)*WSID* system command can be used as either an action command or inquiry command. As an action command,)*WSID* allows you to change the name and password of the active workspace. As an inquiry command,)*WSID* returns the name of the active workspace. When you use)*WSID* as an action command, you must specify the wsname parameter. However, you need not specify the entire name. APL uses the defaults listed in Table 5-1. With)*WSID* you can also specify a password parameter. This causes the password associated with the active workspace to be changed to the specified password. The)*PASSWORD* system command (Section 5.2.5) allows you to change only the password.

As shown in the examples above, the)*WSID* system command returns a workspace name when used either as an action command or as an inquiry command. When)*WSID* returns a workspace name it always returns the workspace filename. Those additional parts that are the same as the defaults are not displayed. The password is never displayed with)*WSID*.

5.3 EXTENDED WORKSPACE-CONTROL COMMANDS

This section describes a variety of system commands that extend the basic workspace-manipulation functions detailed in Section 5.2. These commands can be used to:

1. Determine the maximum and minimum size of the active workspace

2. Report workspace owner and version information

3. Turn the workspace lock on and off to control access by other users

4. Report how long the active workspace has been in use

5. Determine how large the active workspace would be on a secondary storage device

5.3.1 *)MAXCORE* - Determining the Maximum Workspace Size

Command

```
)MAXCORE [[{K-of-memory}]]
           {P-of-memory}
```

Examples

```
          ⍝TOPS-10
          )MAXCORE
  20K/ 176K
          )MAXCORE 50
WAS  20K

          ⍝TOPS-20
          )MAXCORE
  40P/ 352P
          )MAXCORE 70
WAS  40P
```

The *)MAXCORE* system command can be used as either an action command or an inquiry command. As an action command, *)MAXCORE* changes the current setting for the maximum workspace size to the amount you specify and displays the previous setting. As an inquiry command, *)MAXCORE* should be typed without a parameter; it returns the current maximum workspace size and the system memory limit for the data segment.

The standard APL default is 20K words for the data segment on TOPS-10 and 40P words on TOPS-20. The maximum value for K-of-memory is either 176K words or the system memory limit, whichever is smaller. The maximum value for P-of-memory is 352P words. Note that you do not type P or K in the command string.

5.3.2 *)MINCORE* - Determining the Minimum Workspace Size

Command

$$\textit{)MINCORE}\ \left[\!\left[\begin{Bmatrix}\text{K-of-memory}\\\text{P-of-memory}\end{Bmatrix}\right]\!\right]$$

Examples

```
       ⍝TOPS-10
       )MINCORE
 0K
       )MINCORE 10
WAS    0K
       ⍝TOPS-20
       )MINCORE
 0P
       )MINCORE 35
WAS    0P
```

The *)MINCORE* system command can be used as either an action command or an inquiry command. As an action command, *)MINCORE* changes the current setting for the minimum workspace size to the amount you specify and displays the previous setting. As an inquiry command, *)MINCORE* should be typed without a parameter; it returns the current minimum workspace size. The standard APL default on both operating systems is 0. Note that you do not type P or K in the command string.

NOTE

APL does not allow you to specify a minimum workspace size that is larger than the setting of *)MAXCORE*.

The *)MINCORE* system command is very useful in dealing with very large arrays or with operations requiring large amounts of intermediate storage that cause the workspace to expand and contract. *)MINCORE* causes APL to retain at least the amount of memory specified and thus to speed up operations by precluding the frequent acquisition and release of large blocks of memory.

5.3.3 *)OWNER* - Identifying the Owner of a Workspace

Command

```
    )OWNER
```

Example

```
        )OWNER
    CREATED ON 12-JUL-79 BY [4,2043] AT TTY22
```

The *)OWNER* system command is an inquiry command that displays information about the creation of the currently active workspace. *)OWNER* returns the date on which the workspace was created, the directory of the creator of the workspace, and the terminal number of the device at which it was created.

5.3.4 *)SEAL* - Turning the Workspace Seal On or Off

Command

$$)SEAL\left[\left\{\begin{matrix} ON \\ OFF \end{matrix}\right\}\right]$$

Examples

```
        )SEAL
    OFF
        )SEAL ON
    WAS OFF
```

The *)SEAL* system command is both an action command and an inquiry command. When you use it as an action command, you can turn the workspace seal (lock) on or off. The default setting is *OFF*. When the workspace seal is on, only the user who turned the seal on can copy objects from the workspace or turn the seal off. The *)SEAL* command has no effect on the *)LOAD* command.

As an inquiry command, *)SEAL* without a parameter returns the current setting of the seal.

5.3.5)*SIZE* - Reporting the Workspace Size

Command

```
)SIZE
```

Examples

```
      ATOPS-20
      )SIZE
35P   1 PGS

      ATOPS-10
      )SIZE
3K    2 BLKS
```

The)*SIZE* system command is an inquiry command that displays information on the size of the active workspace.)*SIZE* returns two numbers: the first is the current size of the data segment, the second is the amount of disk space that would be required to store this workspace if it were saved in its present state.

5.3.6)*TIME* - Reporting the Time Used

Command

```
)TIME
```

Examples

```
      )CLEAR
CLEAR WS
      )TIME
CONNECTED  0:00:04  CPU TIME  0:00:00
      )LOAD PRIME
SAVED  14:52:57 11-JUL-79 35P
      )TIME
CONNECTED  0:18:10  CPU TIME  0:00:03
```

The)*TIME* system command is an inquiry command that reports the amount of connect time and CPU time accumulated while the currently active workspace has been active. This command is useful in determining the amount of time expended by programming projects. The time begins to accumulate when the workspace is created as a clear workspace, and runs until the session is terminated or the current workspace is saved. The)*COPY* command does not affect the time accumulation.

5.3.7)*VERSION* - Displaying the APL Version Number

Command

```
    )VERSION
```

Examples

```
          )VERSION
    2(412)
```

The)*VERSION* system command is an inquiry command that displays the APL version number with which the currently active workspace was last saved. If your workspace is a clear workspace, APL prints the current version of the interpreter, in the format ver(edit#) where edit is in octal.

5.4 WORKSPACE-CONTENT COMMANDS

This section describes the system commands that examine and control workspace elements such as functions, variables, and groups. The following operations can be performed:

1. Copy variables, functions, and groups from a stored workspace, and erase these elements from the active workspace when desired
2. Display a list of functions defined in the active workspace
3. Collect named objects into a group, disperse the group, display the members of the group, and display a list of groups defined in the active workspace
4. Display the APL state indicator to report on the execution of functions in the workspace
5. Display a list of variables defined in the active workspace

5.4.1 *)COPY* - Copying Objects from a Workspace

Command

```
      )COPY  wsname ⟦password⟧⟦named-object-list⟧
```

Examples

```
          )COPY AVER
SAVED  15:45:03 24-OCT-78 35P
          )COPY AVER B
SAVED  15:45:03 24-OCT-78 35P
          )COPY AVER C
SAVED  15:45:03 24-OCT-78 35P
OBJECTS NOT FOUND:       C
```

The *)COPY* system command is an action command that retrieves functions, variables, and groups from a stored workspace and places them into your active workspace. If there is a password associated with the workspace, you must include it in the command string. You have the option of copying all the named objects in a workspace or a subset of them. The named-object-list identifies the specific objects to be copied. If you omit this parameter, all user-defined functions, variables, and groups are copied.

)COPY does not transfer local values for variables and functions, nor does it copy the state indicator, the width, origin, and significant-digits setting. Only global values of user-defined objects are copied, since a *)COPY* writes a fresh user symbol table. That is, if *A* is a local variable with a value of 3 and a global value of 15, APL copies the global value 15. Also, if your active workspace contains objects with the same name as those in the copied workspace, APL replaces the global values in your active workspace with the copied ones. For example, if *B* is a variable in the active workspace with a global value of 10 and a local value of 5, and the workspace being copied has a variable *B* with a global value of 20 after the *)COPY*, the active workspace will have a variable *B* with a global value of 20 and a local value of 5. Objects that have the same name in both workspaces but are pendent functions or functions still being defined are not replaced.

When you copy a group, all members of the group are copied along with their values. However, if a member of a group is itself a group, APL copies only the group names and not the values from this level.

If you specify an object that is not located in that workspace, APL returns a message *OBJECTS NOT FOUND*.

The *)COPY* command is the same as the ⎕*QCO* system function except that ⎕*QCO* does not display messages to confirm that the copy was successful. See Section 4.3.9 for ⎕*QCO* information.

5.2.4)*ERASE* - Erasing Global Names

Command

)*ERASE* name-list

Examples

```
        A←2 3 4
        A
2 3 4
        )ERASE A
        A
 11 VALUE ERROR
          A
          ^

        ∇R←F
[1]     )ERASE F
NOT ERASED:        F
[1]
```

The)*ERASE* system command is an action command that erases global names from the active workspace by undefining the functions, variables, and groups specified in the name-list parameter. You can undefine a suspended function but not a function in the process of being defined. If you specify a group name, then the group name is erased from the active workspace along with the members of the group.

If a member of the named group is itself a group, the group name is erased but the members of the subgroup remain intact. For example:

```
        )FNS
ARC     COS     DIAM    RADIUS  SIN     TAN
        )GRP TRIG
CIRCLE  COS     SIN     TAN
        )GRP CIRCLE
ARC     DIAM    RADIUS
        )ERASE TRIG
        )GRP TRIG
 22 INCORRECT PARAMETER
        )GRP TRIG
             ^
        )GRP CIRCLE
 22 INCORRECT PARAMETER
        )GRP CIRCLE
             ^
        )FNS
ARC     DIAM    RADIUS
```

The)*ERASE* command does not distinguish between pendent functions and other functions. You should avoid erasing pendent functions because of problems you could create. APL attempts to alleviate such problems by displaying the following message after performing the)*ERASE* operation:

```
 13 POSSIBLE SI DAMAGE
```

This warns you that remedial action might be required before execution of the function continues. (*SI* refers to the state indicator.)

Note that)*ERASE* leaves a slot in the symbol table for the erased name (symbol). Although you erase a symbol, the slot in the symbol table still exists. If you reuse a name that was in the symbol table, APL places it in the same slot where it was before. If you do a)*COPY* of the workspace, APL rebuilds the workspace thus erasing the slot as well as the symbol.

5.4.3)*FNS* - Displaying a List of Functions

Command

)*FNS* ⟦letter⟧

Examples

```
      )FNS
ALPH    HILB    I       INV     LSQ
      )FNS I
I       INV     LSQ
```

The)*FNS* system command is an inquiry command that displays an alphabetic list of global names used as user-defined function names (Chapter 6) in the active workspace. The optional letter parameter identifies the letter at which the alphabetic listing is to begin. If you omit the letter the entire set of global function names is displayed.

5.4.4)*GROUP* - Defining or Dispersing a Group

Command

)*GROUP* group-name ⟦group-member-list⟧

Examples

```
)GROUP FINANCIAL INT FUTUAL PRESVAL
)GROUP FINANCIAL TAX FINANCIAL
)GROUP FINANCIAL
)GROUP FINANCIAL X
```

The)*GROUP* system command is an action command that places a collection of named objects under one group name and can disperse an existing group. The objects can be variables, functions, and other group names. The)*GROUP* command is used primarily with the)*COPY* and)*PCOPY* commands. After collecting a set of functions and variables under one group name, you can specify this name in a)*COPY* or)*PCOPY* command to copy the entire collection at one time. In the first example above, the functions and variables named *INT*, *FUTUAL*, and *PRESVAL* are collected into a group named *FINANCIAL*.

To add a new member to an existing group, you must list the groupname as an element in the member list. This is illustrated in the second example where the variable *TAX* is added to the group named *FINANCIAL*.

To disperse a group, specify the group name without a group-member-list. The group name will no longer be defined but the individual members of the group still exist under their original names. The third example above disperses the group *FINANCIAL*.

A group name is always global; you cannot localize it. For example:

```
          A←1
          B←2
          ∇F;C
    [1]   ⍎')GROUP C A B'
    [2]   ∇
          F
     24 ⍎ NOT GROUPED, NAME IN USE
    )GROUP C A B
            ^
```

5.4.5)*GRP* - Displaying the Members of a Group

Command

)*GRP* group-name

Examples

```
          )GROUP FINANCIAL INT FUTUAL FRESVAL
          )GRP FINANCIAL
    INT      FUTUAL   FRESVAL
```

The)*GRP* system command is an inquiry command that displays the members of the group named in the command string. The members are listed in the order in which they entered the group.

5.4.6)*GRPS* - Displaying a List of Groups

Command

)*GRPS* ⟦letter⟧

Examples

```
          )GRPS
    FINANCIAL           INVENTORY
          )GRPS H
    INVENTORY
```

The)*GRPS* system command is an inquiry command that displays an alphabetic list of global names you specified as group names in the active workspace. The optional letter parameter identifies the letter at which the list is to begin. If you omit the letter, the entire set of group names is displayed.

5.4.7 *)PCOPY* - Copying from a Workspace with Protection

Command

```
    )PCOPY  wsname ⟦password⟧⟦named-object list⟧
```

Examples

```
        A←25
        PLUSROW←40
        )PCOPY MYWORK F PLUSROW PRIMES A
SAVED  10:24:30 12-JUL-79 35P
NOT COPIED:      A        PLUSROW

        )PCOPY MYWORK G B F
SAVED  10:24:30 12-JUL-79 35P
OBJECTS NOT FOUND:       G
NOT COPIED:      F
```

The *)PCOPY* (protected copy) system command is an action command that performs in much the same way as the *)COPY* system command. However, *)PCOPY* protects you from accidentally replacing objects in your active workspace, that is, *)PCOPY* does not replace objects in the active workspace with objects of the same name in the copy workspace. Instead, APL returns the message *NOT COPIED*: along with the names of the objects involved.

When copying groups, the *)PCOPY* command does not copy any members of the group that have the same name in the active workspace. If the group name itself is the same as a group name in the active workspace, APL does not copy the group name but does copy any member of the group that does not have the same name in the active workspace.

Named objects that cannot be found in the copy workspace or cannot be copied are displayed as shown in the examples.

The *)PCOPY* system command operates the same as the *⎕QPC* system function except that *⎕QPC* does not return messages to verify the success of the copy. See Section 4.3.9 for information on *⎕QPC*.

5.4.8 *)SI* - Displaying the State Indicator

Command

```
    )SI
```

Examples

```
        A←⎕
⎕:
        ⍎')SI'
        A
⍎
⎕
F[2]    *
G[3]
```

The *)SI* system command is an inquiry command that displays the state indicator of the active workspace. The state indicator contains the status of the execution of user-defined functions in the workspace. By analyzing the *)SI* listing, you can determine such function-status conditions as the following:

1. suspended functions (*)

2. pendent functions

3. pending quad input requests

4. operations involving the execute function

A bracketed line number following the function name indicates the line at which the function stopped executing. An asterisk following the bracketed line number indicates that the function is currently suspended. If there is no asterisk, the function is a pendent function (one awaiting the return of a called function).

Instead of a line number, the *)SI* display can contain only an asterisk (*), a quad character (⎕), or an execute function (∊ or ⍎). In this case, an asterisk indicates that a suspended function has been erased. A quad character indicates pending quad input. An execute function indicates an execute operation.

The order in which functions are displayed in the *)SI* list is significant; the function that was most recently active is listed first.

5.4.9 *)SIV* - Displaying the State Indicator and Local Variables

Command

```
      )SIV
```

Examples

```
          )SIV
    F[2]   *        R        A        B
    G[3]   *        T        C        A        D
```

The *)SIV* system command is an inquiry command that acts much the same as *)SI*. However, *)SIV* also displays a list of variable names local to the halted function including localized system variables. The *)SIV* command displays the status of pendent and suspended functions, pending quad input requests, and operations involving the execute operator. Unlike *)SI*, *)SIV* also displays the current line of any pending execute string.

5.4.10)*VARS* - Displaying a List of Variables

Command

)*VARS* ⟦letter⟧

Examples

```
          )VARS
A         I         J         K         N
          )VARS K
K         N
```

The)*VARS* system command is an inquiry command that displays an alphabetic list of global names used as variable names in the active workspace. The optional letter parameter identifies the letter at which the listing is to begin. If you omit the letter, the entire list of global variable names is displayed. Local variables are not listed.

5.5 WORKSPACE-ENVIRONMENT COMMANDS

This section describes a variety of system commands that allow you to control the characteristics of the workspace environment. These commands can be used to:

1. Specify the maximum number of significant digits to be displayed
2. Determine the index origin setting
3. Determine the terminal output mode, displaying error lines, setting terminal tab stops
4. Set or return the width of the output line

5.5.1)*DIGITS* - Determining the Output Precision

Command

)*DIGITS* ⟦n⟧

Examples

```
          )DIGITS
10
          1.23456789123456789
1.234567891
          )DIG 5
WAS 10
          1.23456789123456789
1.2346
```

```
      )DIG 2
WAS 5
      1.23456789123456789
1.2
```

The)*DIGITS* system command is both an action command and an inquiry command. As an action command,)*DIGITS* allows you to specify the maximum number of significant digits you want APL to display for noninteger output only. As an inquiry command,)*DIGITS* displays the current setting. The parameter n can be from 1 to 18; the default setting is 10.

APL rounds off a number if it has more digits than the current setting.

The)*DIGITS* system command does not affect the precision of internal calculations or the display of numeric constants. The setting is preserved when you save the active workspace.

The)*DIGITS* command performs the same operation as the ⎕*PP* system variable (Section 4.2.15).

5.5.2)*ECHO* - Determining Error Line Echoing

Command

$)ECHO \left[\left\{\begin{matrix} ON \\ OFF \end{matrix}\right\}\right]$

Examples

```
      )ECHO
ON
      ε'A'
11 ε VALUE ERROR
       A
         ^

      ⊥'A'
11 ε VALUE ERROR
       A
         ^

      ⍎'A'
11 ⍎ VALUE ERROR
       A
         ^
25 EXECUTE ERROR
       ⍎'A'
       ^
      )ECHO OFF
WAS ON
      ∊'A'

      ⊥'A'

      ⍎'A'
25 EXECUTE ERROR
```

The *)ECHO* system command is both an action command and an inquiry command. As an action command, *)ECHO* allows you to select whether or not to have APL echo statements that contain errors. As an inquiry command, *)ECHO* returns the current state of echoing. The parameter is either the word *ON* or the word *OFF*. The default is *ON*.

If the echoing status is *OFF* and you perform an execute (∊ or ⍳) operation (Sections 3.4.3 and 5.8), APL prints neither the error message nor the error line. The value of the execute expression is a null array of rank two. The first dimension of the null array is zero and the second is a number that identifies the type of error encountered (see Appendix A). This feature aids the handling of error conditions under program control and is illustrated in the final example at the beginning of this section.

The echoing status is preserved when the active workspace is saved.

5.5.3 *)MODE* - Determining the Terminal Output Mode

Command

$$)MODE \left[\left[\begin{Bmatrix} ESCAPE \\ KEYWORD \end{Bmatrix}\right]\right]$$

Examples

```
      )MODE
KEYWORD
      )MODE ESCAPE
WAS KEYWORD
      A_'@A@K@K@Z'
      A
@A'@Z
      )MODE KEYWORD
WAS ESCAPE
      A
.AL'.RU
```

The *)MODE* system command is both an action command and an inquiry command. As an action command, *)MODE* allows you to select the mode of output on terminals that do not have an APL character set (see Section 1.3). As an inquiry command, *)MODE* displays the current setting.

The parameter can be either the word *KEYWORD* or the word *ESCAPE*. Either word can be abbreviated to one letter. In *ESCAPE* mode, on output, @p, @K, @Q, @Y, print as *, ', ?, ^. The default is *KEYWORD*. This setting has effect only if you responded to the *TERMINAL*.. prompt, at the beginning of the session, with TTY.

The mode setting is preserved when the active workspace is saved.

5.5.4)*ORIGIN* - Determining the Index Origin

Command

)*ORIGIN* ⟦n⟧

Examples

```
        )ORIGIN
1
        ⍳5
1 2 3 4 5
        )ORIGIN 0
WAS 1
        ⍳5
0 1 2 3 4
```

The)*ORIGIN* system command is both an action command and an inquiry command. As an action command)*ORIGIN* allows you to change the setting of the index origin for array operations and returns the previous setting. The parameter "n" can be either 0 or 1. The default setting is 1. As an inquiry command,)*ORIGIN* displays the current setting.

The effect of the)*ORIGIN* command is to renumber the elements of arrays to begin at 0 or 1, depending on the setting. This command is particularly relevant when used with the ⍳ function. (Sections 3.3.10 and 3.3.11.) The index origin setting is saved when the active workspace is saved.

The)*ORIGIN* system command performs the same operation as the ⎕*IO* system variable (Section 4.2.11).

5.5.5)*TABS* - Determining Tab Stops on the Terminal

Command

)*TABS* ⟦n⟧

Examples

```
        )TABS
0
        )TABS 8
WAS 0
```

The)*TABS* system command is both an action command and an inquiry command. As an action command,)*TABS* sets the increment between tab settings for APL output. As an inquiry command,)*TABS* returns the current tab setting. The integer parameter "n" specifying the tab increment can be from 0 to the page width. The default setting is 0.

If you reset the page width with either the ⎕*PW* system variable (Section 4.2.16) or the)*WIDTH* command (Section 5.5.6), the)*TABS* setting is reset to the new page width.

For example:

```
      ⎕PW
72
      )TABS 70
WAS 8
      ⎕PW←50
      )TABS
50
      )WIDTH 45
WAS 50*
      )TABS
45
```

The asterisk in the above example indicates that)*TABS* has been reset.

NOTE

The)*TABS* system command is designed for use on terminals with physical tab stops. The tab setting is not saved with the active workspace.

APL will output a TAB instead of a string of blanks if the next non-blank character to output comes after a tab stop.

5.5.6)*WIDTH* - Determining the Width of the Output Line

Command

```
)WIDTH ⟦n⟧
```

Examples

```
      )WIDTH
72
      ⍳15
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
      )WIDTH
WAS 72*
      ⍳15
1 2 3 4 5 6 7 8 9 10 11 12 13
      14 15
      )WIDTH
30
```

The)*WIDTH* system command is both an action command and an inquiry command. As an action command, you can set the maximum number of characters that can appear in an output line and display the previous setting. As an inquiry command, the)*WIDTH* command returns the current width of the output line. The parameter "n" must be an integer within the range 30 through 390. The default is determined by the system width setting. You can change the system width for your current job at operating system command level by using:

```
@TERMINAL WIDTH       !TOPS-20

.SET TTY WIDTH        !TOPS-10
```

The *)WIDTH* system command does not affect the display of messages on the terminal or the allowable length of input lines. The width setting is preserved when you save the active workspace.

The *)WIDTH* system command performs the same operation as the *⎕PW* system variable (Section 4.2.16).

5.6 APL TERMINATION COMMANDS

This section describes the various system commands that can terminate an APL session. You can exit from APL in a variety of ways. You can:

1. Terminate the session, save the active workspace, and run a program
2. Terminate the session and save the active workspace
3. Return to system command level
4. Terminate the APL session, optionally returning to system command level
5. Terminate a session and run a program

5.6.1 *)C* and *)CALL* - Running a Program and Returning to APL

Command

)C ⟦n⟧ file specification

)CALL ⟦n⟧ file specification

Examples

```
      )CALL SYS:FORTRA
*,TTY: = TTY:
       I=1
       END
^Z

  MAIN.                        FORTRAN V.5A(621) /KI   12-JUL-79        13:00
PAGE 1


00001              I=1
00002              END


SUBPROGRAMS CALLED
```

```
SCALARS AND ARRAYS [ "*" NO EXPLICIT DEFINITION - "%" NOT REFERENCED ]

*I        1

MAIN.    [ NO ERRORS DETECTED ]
* RUN:APLSF/RUNOFFSET:1
TERMINAL..LA
APL-20 DECSYSTEM-20 APLSF 2(412)
TTY22) 13:01:33 THURSDAY 12-JUL-79 MASELLA        [4,204]
CLEAR WS
```

The *)C* and *)CALL* system commands perform the same operation as *)R* and *)RUN* except that *)C* and *)CALL* also save your active workspace as *CONTIN.APL*. They also write an APL *.TMP* file (nnn*APL.TMP*) or *.TMPCOR* file so that, if APL is subsequently run with a CCL linkage, the processor is able to determine the terminal type from the file written and does not prompt you with *TERMINAL...*

The difference between *)C* and *)CALL* is the default device searched. The *)C* command defaults to *SYS*:. The *)CALL* command defaults to *DSK*:.

5.6.2 *)CONTINUE* - Saving the Workspace and Ending the Session

Command

)CONTINUE ⟦*HOLD*⟧

Examples

```
          )CONTINUE HOLD
  9:35:53 12-JUL-79 2 BLKS
TTY44)   9:35:54 12-JUL-79
CONNECTED     0:01:49 CPU TIME      0:00:00
0 STATEMENTS 0 OPERATIONS
KILO-CORE-SECS 13

EXIT

.
```

The *)CONTINUE* system command is an action command that ends an APL session after saving the currently active workspace. *)CONTINUE* operates the same was as *)OFF* (displaying the same statistics) except that before ending the session *)CONTINUE* saves the active workspace in your disk area under the name *CONTIN.APL*. The workspace is saved only if it is indeed active, that is, contains at least one symbol in the symbol table. The saved workspace replaces any other disk file that you previously saved with the name *CONTIN.APL*.

The next time you begin an APL session, instead of a clear workspace, you will receive the *CONTIN.APL* workspace as your active workspace.

The *HOLD* parameter returns you to operating system command level after ending the APL session. *)CONTINUE* not only prints the same summary information as the *)OFF* command, but also displays an initial line that specifies the time, date, and size of the saved workspace.

5.6.3 *)MON* - Returning to Operating System Command Level

Command

```
      )MON
```

Examples

```
      ⍝TOPS-20
      )MON
MONITOR:
@CONTINUE
APLSF:

      ⍝TOPS-10
      )MON
MONITOR:

.REENTER

APLSF:
```

The *)MON* system command is an action command that returns control to operating system command level. The *)MON* command does not save the active workspace as the *)CONTINUE HOLD* does. Therefore, if you intend to return to APL and save the workspace, be careful not to destroy your memory image while at command level. This could occur if you issue a command that runs a program.

You can return to APL by typing the operating system command CONTINUE or REENTER. For more information on returning to APL, refer to Section 1.4.4.

5.6.4 *)OFF* - Terminating the APL Session

Command

```
      )OFF [[HOLD]]
```

Examples

```
      )OFF
TTY22) 15:40:22 12-JUL-79
CONNECTED    0:00:34 CPU TIME     0:00:01
5 STATEMENTS 6 OPERATIONS
KILLED JOB 16, USER MASELLA, ACCOUNT APL, TTY 22,
  AT 12-JUL-79 15:40:22,  USED 0:0:1 IN 0:0:42

      )OFF HOLD
TTY22) 15:41:18 12-JUL-79
CONNECTED    0:00:19 CPU TIME     0:00:00
0 STATEMENTS 0 OPERATIONS

EXIT
@
```

The *)OFF* system command is an action command that terminates your APL session. If you specify the *HOLD* parameter, APL terminates your session and returns you to operating system command level. Without the *HOLD*, APL not only terminates your session but also logs you off the system.

The)*OFF* commands outputs several lines of information before terminating the session. The lines contain the following:

1. Your terminal identification
2. Current time
3. Current date
4. Length of time connected to APL
5. Amount of computer CPU time used
6. Number of statements
7. Number of operations executed
8. Kilo-core-seconds-used (TOPS-10 only)

The)*OFF* command destroys the currently active workspace.

5.6.5)*R* and)*RUN* - Ending the Session and Running a Program

Command

)*R* ⟦n⟧ file specification

)*RUN* ⟦n⟧ file specification

Examples

```
          )RUN DSK:TEST

EXIT
@
          )R FORTRA
*
```

The)*R* and)*RUN* system commands perform essentially the same function as)*OFF HOLD*. They terminate the current APL session and return you to the operating system, but unlike)*OFF HOLD*, they also run the program you specify as the filespec in the command string. The file you specify must contain a program ready to run, that is, a program with a file extension or file type of *.EXE*. The optional parameter "n" is an integer value that is added to the starting address of the file to be run; this facility is useful when starting from alternate entry points (for example, CCL entry points are equal to 1).

The)*R* and)*RUN* commands do not save the currently active workspace nor the reentry point to APL before ending the APL session. If the program you identify cannot run for some reason, you will be at operating system command level.

The difference between the *)R* command and the *)RUN* command is the default device that is searched. The *)R* command default device is *SYS*:. The *)RUN* command default device is *DSK*:.

5.7 MISCELLANEOUS COMMANDS

This section describes the additional system commands that perform such tasks as the following:

1. Generating a mask to protect confidential data

2. Displaying a record of activity during the current APL session

5.7.1 *)BLOT* - Generating a Mask

Command

)BLOT ⟦n⟧

Examples

```
          )BLOT 30
GHOUEWWRGUMEEKMRDCGEEFZRWNRRMK
```

The *)BLOT* system command is an action command that generates a mask in a random pattern to conceal confidential input such as passwords. The optional parameter "n" specifies the length of the mask. If you omit the parameter, the default length is 25 characters.

5.7.2 *)CHARGE* - Displaying APL Session Information

Command

)CHARGE

Examples

```
      )CHARGE
TTY22) 15:43:39 12-JUL-79
CONNECTED    0:00:35  CPU TIME      0:00:00
0 STATEMENTS  0 OPERATIONS
```

The *)CHARGE* system command is an inquiry command that displays a record of activity during the current APL session. Information to be displayed includes the statistics as the *)OFF* and *)CONT* commands display:

1. connect time

2. CPU time

3. number of APL statements

4. number of operations executed

5. kilo-core seconds (TOPS-10 only)

5.8 SYSTEM COMMANDS AND THE EXECUTE FUNCTION

The execute function (ϵ,⊥,⍎) allows you to use any system command as an argument, as long as you enclose the command in single quotation marks. The value returned by the execute is always a character vector or matrix. When displayed, the response of the execute function resembles the response of issuing the command by itself. This allows you to execute a system command from within a user-defined function. For example:

```
          A←⍎')VARS'
          A
    A          B          C
```

The shape of the result returned by the execute function after executing a system command indicates the number of characters in the command response. (Carriage returns and line feeds are not included in the count.) Table 5-3 lists the shape of values resulting from executing a system command with ϵ. See Sections 3.4.3 and 3.4.4 for a detailed description of APL execute functions.

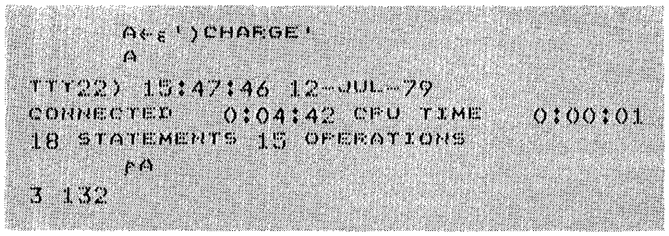
Table 5-3
System Commands and Execute

Response	Rank	Rho (ρ)
no characters	1	0
one character	0	null vector
more than one character but only one line	1	number of characters
multiple lines	2	number of lines, current width

The following example illustrates several uses of the execute function, with comments describing the characters included in the count:

```
          A←⍎')ORIGIN 1'
          A
    WAS 0
          ⍴A
    5
          ⍝SHAPE IS 5 CHARACTERS
          ⍝SPACE IS A CHARACTER

          A←⍎')WIDTH 132'
          A
    WAS 72
          ⍴A
    6
```



```
      A←⍎')CHARGE'
      A
TTY22) 15:47:46 12-JUL-79
CONNECTED    0:04:42 CPU TIME     0:00:01
18 STATEMENTS 15 OPERATIONS
      ⍴A
3 132
```

CHAPTER 6

DEFINING AND EXECUTING FUNCTIONS

6.1 INTRODUCTION

APL language statements operate in any of three modes:

Immediate mode (or execution mode): in this desk-calculator mode, APL statements and expressions are executed immediately after you terminate the line.

Function-definition mode: in this mode, you name, develop, edit, and save functions for use at a future time.

Function-execution mode: in this mode, you execute the function you created in function-definition mode.

The language syntax is the same in all modes. However, in function-definition mode there are a variety of special APL characters available and a number of practical considerations you need to take into account when you construct a function.

This chapter discusses:

1. Function definitions, headers, and variables

2. Editing procedures

3. Branching and the use of labels, trace vectors, stop vectors, and the state indicator

4. Use of locked and suspended functions

5. Error trapping

6.2 DEFINING THE FUNCTION

APL provides a comprehensive facility for defining, changing, and calling user-defined functions that supplement the large set of primitive functions discussed in Chapter 3. Once you write or define a function, you can save it and recall it with the ease of a primitive function.

You construct a user-defined function in two parts, a function header and a function body. For the function header, you define the name of the function and the syntax of the function call. The function body consists of a series of statements and expressions that define the actions to be performed by the function when it is executed.

To enter function-definition mode, type a del character (∇) followed by the function header and a carriage return/line feed. This signals the APL processor not to execute subsequent lines as you enter them, as it would in immediate mode. However, system commands are executed immediately. In function-definition mode, APL prompts you for successive lines of the function body by displaying successive bracketed line numbers for every line. All the lines you enter are treated as function lines until you type another del (∇). The second del returns you to immediate mode. Functions can have up to 1000 lines.

The format of a function is shown in the following:

```
          ∇ function header
    [1]   .
    [2]   .
    [3]   .
    [4]   function body
    [5]   .
    [6]   .
    [7]   .
    [8]   .
          ∇
```

There are no restrictions on the type of statements you can include in a function definition. You can include system commands in a function but you must precede them with the execute function. If you do not, APL executes them immediately. See Section 5.8.

You can define and execute functions in quad input mode. The input request remains pending until you leave function-definition mode.

You delete a function from your workspace with the system function ⎕*EX* (Section 4.3.4).

6.2.1 The Function Header

The function header contains the name of the function and the syntax of the function call. You type the function header on the same line as the del (∇) that signals function-definition mode. There are six types of functions that you can define depending on the number of arguments the function header takes and whether or not the function returns an explicit result. Table 6-1 displays the formats of the six function types.

Table 6-1
Function Headers

Type	Explicit Result	Type	No Explicit Result
niladic	∇ variable ←name	niladic	∇ name
monadic	∇ variable ←name arg	monadic	∇ name arg
dyadic	∇ variable ←arg2 name arg1	dyadic	∇ arg2 name arg1

The arg, arg1, and arg2 in the function header are dummy arguments. They look like variable names but they have no values assigned to them. A dummy argument is a placeholder for an actual argument (value) you supply when you call the function. The number of dummy arguments in the function header determines the calling syntax of the function (niladic, monadic, dyadic). You must include the same dummy arguments in the function definition as in the function header.

The variable in the function-header syntax (Table 6-1) is also another dummy argument. However, the presence of this variable in the function header determines whether or not the function returns an explicit result. This variable temporarily stores the results of the function execution. You assign the function name to the variable in the function header. If a function returns an explicit result, you can use this function in subsequent computations by including its name in an expression just as you would an APL function. A function that does not return an explicit result (no variable assigned in the function header), may also return a result when you execute it. However, you cannot use this function result for further work as a value within another function.

Functions that return explicit results can be included as part of any expression. Functions that do not return explicit results must be either the only statement on the line or the last statement in a multi-statement line (last statement on the left).

The following function header returns an explicit result:

```
    ∇A←B PROG C
```

In this function header, *A* is the variable (dummy argument) that designates this function to be one that returns an explicit result. The result of the execution is stored temporarily in *A*. The variable names *B* and *C* are dummy arguments, and *PROG* is the function name.

When you call a function containing dummy arguments, you must supply the values for APL to use during execution. You include the values in the calling syntax of the function name. For example, if the function header has two dummy arguments:

```
    ∇A NAME B
```

you must supply values for *A* and *B*.

```
    25 NAME 42
```

When APL executes the function, the values 25 and 42 are used in the calculations wherever you placed *A* and *B* in the function definition.

You can also include local symbols in a function header. Local names must be unique from dummy names in the same function definition. See Section 6.2.2.1.

6.2.2 Symbol Classification

An APL workspace contains an area that is known as the symbol table. This area is empty in a clear workspace. Every time you create a variable name, function name, or group name (Section 5.4.4), this name is written, and referred to as a symbol, in the symbol table. (Dummy arguments are not recorded in the symbol table). Any values you assign to these symbols are also stored in the symbol table. When you save an active workspace, APL also saves the symbol table.

Symbols are classified as being either local or global symbols depending on how their values are treated before and after function execution. The following subsections describe the characteristics of local and global symbols.

6.2.2.1 Local Symbols - A local symbol has significance only during the execution of a particular function. To specify a symbol as being a local symbol, include that symbol in the function header and then assign it a value within the function definition. Separate local symbols from the function name with semicolons. All local symbols are placed to the right of the function name. For example:

```
    ∇NAME;I;LOC;G
```

In this function header, *I*, *LOC* and *G* are local variables. Local variables have no significance in determining whether a function is niladic, monadic, or dyadic. Dummy variables do. See Section 6.2.1.

During execution of a function, the local value is always dominant. For example, if a local variable has the same name in two separate functions, the execution of one function does not affect the value of the local variable in the other function. You initialize local variables when you call the function, and any local values are lost upon exiting from the function. Using a local variable before you have assigned it a value results in an 11 *VALUE ERROR*. Note that there is no need to include local variables in the function call.

Function-line labels (Section 6.4.2) are treated as local variables and are also initialized when you call the function; however, they cannot be assigned a value.

System variables (Section 4.2) can also be localized within a function definition.

6.2.2.2 Global Symbols - A global symbol has the same significance (value) throughout the scope of its workspace, whether or not it is inside or outside a function. You can change a global symbol, erase it, or expunge it throughout that scope. However, you can have only one global definition at a time for a symbolic name.

Because naming conventions for functions and variables are the same, you cannot have a global function and a global variable with the same name. You can, however, have a local and a global with the same name. In this case, certain rules apply for determining which value takes precedence, global or local. See Section 6.2.2.3 for this explanation.

6.2.2.3 Dynamic Localization - The phrase "dynamic localization" refers to the precedence of local symbols over global symbols with the same name. During function execution, the value of a local variable supersedes the value of a global variable with the same name. Also, depending on the particular function being executed, a local variable can supersede another local variable of the same name.

For example, if you have a global variable *A* and you execute a function containing a local variable, *A*, APL uses the value of the local variable *A* during function execution. Once APL exits from the function, *A* retains its global value.

If two functions have a local variable with the same name, APL uses the value from the function in which it is currently executing. For example, the local variable *B* has a value of 10 in *FUNC*1 and a value of 25 in *FUNC*2. While executing *FUNC*1, APL uses 10; while executing *FUNC*2, APL uses 25. Upon APL's return to *FUNC*1, *B* resumes the value of 10. Finally, upon APL's exit from *FUNC*1, *B* has no value.

6.2.3 Function Input and Output

You can input and output data and the results of function execution by using the standard APL input/output functions described in Chapter 2. All the quad symbols (⎕) can be used in both immediate and function-definition mode. These are:

1. Quad (⎕) or evaluated input, Section 2.5.1
2. Quote-quad (⍞) or character input, Section 2.5.2
3. Quad-del (⍔) or unedited input, Section 2.5.3
4. File input (⍇), Section 7.2
5. Normal output, expressed by simply typing a variable name, Section 2.5.4
6. Mixed output, expressed by typing values separated by semicolons, Section 2.5.6
7. Quad output (⎕), Section 2.5.5
8. Bare or character output (⍞ or ⍔), Section 2.5.7
9. File output (⍈), Section 7.2

File input and output are discussed in Chapter 7. The other varieties of input and output are described in detail in Section 2.5.

One aspect of APL I/O, escaping from an input request within an infinite loop, is particularly relevant to a discussion of function execution. In this case, you may not be able to escape by typing the attention signal, two CTRL/Cs. You can escape by typing several right-arrows (→). Also, you can escape from either quote-quad or quad-del input mode by typing the following:

O backspace *U* backspace *T* (or .OU from a non-APL terminal)

Either form of escape has the same effect as function suspension (Section 6.4.3); it causes function execution to be interrupted but does not result in an exit from the function.

6.2.4 Comment Lines

You can include comment lines anywhere in the APL function. They are particularly useful in annotating statements in the definition. Comments must appear on separate lines and cannot be included on lines containing APL statements. The first character in a comment line must be a lamp character (⍝), formed by overstriking the down union (∩) and the jot (∘). APL treats the text following this symbol as a comment; and the text can consist of any combination of valid APL characters. Note that a comment cannot extend across line boundaries.

When you display a function definition, APL begins a comment line one character position to the left of the rest of the text. Actually, the lamp character (⍝), itself, prints one position to the left. An example of this is shown in Section 6.2.5.3.

6.2.5 Examples of Defined Functions

This section contains three different examples of defined functions.

6.2.5.1 Niladic Function

6.2.5.1 Niladic Function - The following niladic function does not return an explicit result. Note the value of *VECTOR*, as a global variable outside the definition of *AVG* and as a local variable inside *AVG*.

```
        ∇AVG;VECTOR
[1]     ⍞←'ENTER THE VECTOR TO BE AVERAGED: '
[2]     'THE RESULT IS ';(+/VECTOR)÷ρ,VECTOR←⍎⍞
[3]     ∇

        VECTOR←'ABCD'
        AVG
ENTER THE VECTOR TO BE AVERAGED: 3 4 5 6 7
THE RESULT IS 5
        VECTOR
ABCD
```

6.2.5.2 Monadic Function - The following monadic function returns an explicit result:

```
        ∇ANS←AVERAGE VEC
[1]     ANS←(+/VEC)÷ρ,VEC
[2]     ∇
        AVERAGE 3 5 4 6 7
5
        100×AVERAGE 3 5 4 6 7
500
```

6.2.5.3 Dyadic Function - The following dyadic function included below does not return an explicit result:

```
          ∇LETTER IN STRING;T
    [1]   ⍝RETURNS NUMERIC POSITION WHERE CHARACTER
    [2]   ⍝APPEARS IN STRING
    [3]   T←(LETTER=STRING)/⍳⍴,STRING
    [4]   →0,⎕←T;→5×⍳0=⍴T
    [5]   'NO OCCURENCES'
    [6]   ∇
          LETTER←'C'
          T←'GLOBAL'
          LETTER IN 'ABCACBC'
    3 5 7
          LETTER IN 'LMNOP'
    NO OCCURENCES
          T
    GLOBAL
```

6.3 EDITING THE FUNCTION

You can edit a function definition in a variety of ways. There is no need to go to a text editor outside of APL. APL has a line editor that allows you to add, delete, and change definition lines and also alter the function header. You can even edit individual characters in a line. See Section 6.3.7 for character-editing procedures.

You must be in function-definition mode to edit a function. To open a function for editing, type:

```
    ∇ function name
```

After an addition, replacement, insertion, deletion, or display operation, APL displays a line number to allow you to add or enter additional text. If you do not want to do this, type a del (∇) to close the function and thus shift from function definition to immediate mode. You can also type a del on an edit line. For example:

```
          ∇STAT
    [7]    [∆5]∇
```

In this example, APL deletes line [5] and then exits from function-definition mode. The del can be included on any line except a comment line.

After you return to immediate mode, the lines in the function are automatically renumbered sequentially, beginning at line [1]. Therefore, lines you insert with fractional numbers affect the function only while it is open for editing.

6.3.1 Adding Function Lines

You can add lines to the end of a function in a very convenient manner. When you open an existing function, APL assumes that you want to add new lines and it displays the next available line number. For example, the function named *STAT* can exist in the following form before you edit it to remove errors.

```
         ∇STANDX←NSUBJ STAT X
   [1]   SUMX←+X
   [2]   SUMX2←+/(X*2)
   [3]   ⍝COMPUTE MEAN, VARIANCE, STANDARD DEVIATION
   [4]   MEANX←SUMX÷NSUBJS
   [5]   MEANX←SUMX÷NSUBJ
   [6]   ∇
```

You can add lines in response to the bracketed line numbers displayed as shown below.

```
         ∇STAT
   [6]   ⍝FUNCTION RETURNS VALUE OF STANDARD DEVIATION OF X
   [7]   STANDX←VARX*0.5
   [8]   ∇
```

To terminate the specification of additional lines, enter a del to transfer from function-definition mode to immediate mode.

6.3.2 Replacing Function Lines

You can replace existing lines in a function definition by entering function-definition mode, then specifying the affected line number and the new text of the line. In the next example, APL displays the line number [8] indicating that the existing function has seven lines. The first line of the function is replaced with the new text.

```
         ∇STAT
   [8]   [1] SUMX←+/X
   [2]   ∇
```

APL then displays the next line number after the replaced line, in this case, [2]. At this point, you can either enter new text for line [2], or escape from function-definition mode by typing ∇.

The line number you specify must be a positive number less than 1000. If the line does not currently exist, it is inserted (see Section 6.3.3). You can include a decimal point but you cannot specify more than three decimal places.

6.3.3 Inserting Function Lines

You can insert new lines between existing lines of a function definition by first entering function-definition mode. Then, specify the new line number followed by the text. For example, to insert a line between [5] and [6] you can specify any number from [5.001] to [5.999]. To insert a line before line [1], you can use any number from [0.001] through [0.999]. Note the following example:

```
        ∇STAT
[8]     [0.5] ⍝SUM ELEMENTS OF ARRAY X
[0.6]   [5.5] VARX←(SUMX2÷NSUBJ)−MEANX*2
[5.6]   ∇
```

The new lines are inserted between existing lines [0] and [1] and [5] and [6] respectively. In each case, APL prompts with the next line number after the inserted line. To derive the line that is next in an inserted sequence, APL increments the present line number by *1E¯D* where *D* is the number of decimal places in that line number. The next line after [0.5] is thus [0.6], the next line after [5.5] is [5.6], and the next line after [8.29] would be [8.3]. Line number [6] is after [5.9], and line number [7], not [6.1], is after [6]. You can enter new text for the line number displayed, override the line number by specifying another one, or return to immediate mode by typing a ∇.

After you close the function, APL renumbers the lines to consecutive integers beginning with [1]. Line numbers you insert must be positive numbers up to but not including 1000, with or without a decimal point, and with no more than three decimal places. The renumbered function definition now exists in the following form.

```
      ∇   STANDX←NSUBJ STAT X
[1]     ⍝SUM ELEMENTS OF ARRAY X
[2]      SUMX←+/X
[3]      SUMX2←+/(X*2)
[4]     ⍝COMPUTE MEAN, VARIANCE, STANDARD DEVIATION
[5]      MEANX←SUMX÷NSUBJS
[6]      MEANX←SUMX÷NSUBJ
[7]      VARX←(SUMX2÷NSUBJ)−MEANX*2
[8]     ⍝FUNCTION RETURNS VALUE OF STANDARD DEVIATION OF X
[9]      STANDX←VARX*0.5
      ∇
```

6.3.4 Deleting Function Lines

To delete existing lines in a function-definition mode, you type a delta (∆) and the line number both within square brackets. For example, to delete line [5] of *STAT*, type the following:

```
        ∇STAT
[10]    [∆5]
[5]     ∇
```

APL displays the number of the line just deleted to give you the opportunity to type a new version of the deleted line. You can enter new text or escape from function-definition mode by typing ∇. After you close the function, APL renumbers the lines.

6.3.5 Displaying Function Lines

APL gives you the ability to display an individual line, a part of the function definition from the specified line to the end, or the entire function definition.

To display an individual line, type the line number and a quad character (⎕) within square brackets. For example, to display line [3] of function *STAT*, type:

```
          ∇STAT[3⎕]
   [3]     SUMX2←+/(X*2)
   [3]    ∇
```

APL prints the number of the line just displayed to give you the opportunity to specify a new version of the line. You can now perform any editing operation or escape from function-definition mode by typing ∇.

To display the function definition from a particular line number to the end, you type the quad character first and then the line number from which lines are to be displayed. For example:

```
          ∇STAT[⎕7]
   [7]    ⍝FUNCTION RETURNS VALUE OF STANDARD DEVIATION OF X
   [8]     STANDX←VARX*0.5
        ∇
```

APL displays the number of the next line after the final line of the function definition, in this case [9], to give you the opportunity to add more text.

To display the entire function definition, type the quad character without a line number. For example:

```
          ∇STAT[⎕]∇
        ∇  STANDX←NSUBJ STAT X
   [1]    ⍝SUM ELEMENTS OF ARRAY X
   [2]     SUMX←+/X
   [3]     SUMX2←+/(X*2)
   [4]    ⍝COMPUTE MEAN, VARIANCE, STANDARD DEVIATION
   [5]     MEANX←SUMX÷NSUBJ
   [6]     VARX←(SUMX2÷NSUBJ)-MEANX*2
   [7]    ⍝FUNCTION RETURNS VALUE OF STANDARD DEVIATION OF X
   [8]     STANDX←VARX*0.5
        ∇
```

The ∇ characters preceding line [1] and following line [8] indicate the delimiters of the function and identify its name. They do not change the mode as the function prints. APL displays the number of the next line after the final line of the function to give you the opportunity to add new text.

If you want to display a line or an entire function and return to immediate mode after the display, type the closing ∇ on the same line as the display request. For example:

```
     ∇STAT[⎕]∇
```

6.3.6 Editing the Function Header

You can edit the name or arguments of a function header by accessing line number [0]. You can display and replace the function header just like any other line in the function. However, you cannot delete the header using the delta character Δ. Also, you must include a valid function header before leaving function-definition mode.

The following example displays the function header:

```
           ∇STAT
     [9]   [0⎕]
         ∇  STANDX←NSUBJ STAT X
     [0]   ∇
```

Notice that the header is displayed without a line number. When you specify a character position in the header (see Section 6.3.7), APL types the header with line number [0] and without the ∇. For example:

```
           ∇STAT
     [9]   [0⎕7]
     [0]    STANDX←NSUBJ STAT X
            ∧
```

The caret in the above example indicates the position of the terminal head. It does not print on your terminal. Line-editing positions are discussed in Section 6.3.7.

6.3.7 Character-Editing Procedures

Besides providing a way to edit a function definition line by line, APL allows you to edit a function definition character by character. Character editing is a multiple-step process. The first step involves deleting characters no longer needed and inserting blanks in the line to allow additional desired text to be typed. The second step involves typing in the new text. Repetition of these steps is often necessary. The final appearance of the line should be identical to a function line just entered from the keyboard.

To modify a line, specify (1) the line number followed by (2) a quad character (⎕) followed by (3) the estimated character position at which editing is to begin.

For example:

```
          ∇DIESEL
    [7]   [1⎕10]
    [1]    A←R⍟GAMMA−1
              ^
```

APL displays the line, performs a carriage return/line feed, and then positions the cursor or terminal head at the position you specified. In the example above, the caret (^) indicates the position specified; it does not print on the terminal. If you specify position 0, APL places you at the end of the line. Once you are in the desired position, you can do any of the following:

1. Delete a character by typing a slash(/) beneath each character you want to delete.

2. Insert a space by typing a digit or letter beneath each character before which you want to insert a space. Typing the digit 1 beneath a character inserts one space before that character. A 2 inserts two spaces, and so forth. You can insert multiples of five spaces by using letters. Typing an *A* inserts five spaces; typing a *B* inserts ten spaces and so forth. If the number of spaces you specify plus the current line length exceeds the length of the terminal line (the value of ⎕*PW*) you will receive a 5 *DEFN ERROR* error message.

All other characters you type are ignored.

When you press the RETURN key after inserting spaces and deleting characters, APL displays the line with the inserted spaces and without the deleted characters. It then performs a carriage return and positions you at the first inserted space on the line to be edited. If you did not insert spaces, APL positions you at the end of the line. You can enter new text in the spaces or make further modifications to the existing text. On APL terminals, you can backspace to insert new characters and can create illegal overstrikes to aid in retyping the line.

If you change the line number while you are editing the line, any edits you make correspond to the new line number. The original line remains unchanged.

The following example illustrates the use of character-editing in correcting the line:

```
[1]    T←(LETTR=STRING/⍳8P,STRING
```

There are several errors in this line:

1. *LETTER* is misspelled *LETTR*.
2. The right parenthesis is missing after *STRING*.
3. The "8" should not appear at all.
4. The "P" should be ρ.

Because the first error occurs in *LETTR*, you could edit the line this way:

```
        ∇FUNC
[5]     [1⎕14]
[1]      T←(LETTR=STRING/⍳8 P,STRING
                1       1 / /
[1]      T←(LETTER=STRING)/⍳ρ,STRING
[2]     ∇
```

The cursor or terminal head is now positioned at the space between *T* and *R*. You can now enter the new characters, spacing over the text you want to preserve. To do so, type the following:

1. *E* in the space between *LETT* and *R*
2.) in the space between *STRING* and /
3. ρ in the space between ⍳ and ,

The new line looks like this:

```
        ∇FUNC[1⎕]∇
[1]      T←(LETTER=STRING)/⍳ρ,STRING
```

When you press the RETURN key, this line replaces the existing function line [1] in your function definition.

You can type a deliberate character error, for example, an illegal overstrike, after a character-editing display to cancel the revision of the function line. When APL encounters a character error, it displays both an error message and the line up to the point at which the error occurred. However, you cannot escape from character-editing mode except by completing the line.

6.3.8 Performing Immediate-Mode Editing

You can edit lines during immediate mode as well as function-definition mode. In immediate mode, line edits affect the last immediate line entered from the keyboard. Because immediate-mode lines do not have line numbers, type (1) any arbitrary legal line number (that is a number less than 1000) followed by (2) a quad character (⎕) followed by (3) the character position at which editing is to begin. For example:

```
    ACRON←INIT1,INIT2[INIT3
7 SYNTAX ERROR
     ACRON←INIT1,INIT2[INIT3
                        ^
    [1⎕23]
    ACRON←INIT1,INIT2[INIT3
                      /1
    ACRON←INIT1,INIT2,INIT3
```

Immediate-mode editing proceeds exactly as in function-definition mode. However, after you press RETURN to conclude the final edits, APL executes the line. Note that the DELETE key (RUBOUT), CTRL/U, and CTRL/R also work in immediate mode.

6.4 EXECUTING THE FUNCTION

The process of defining a function associates the function header provided by you with the statements you enter as the function body. When you decide to execute the function, you use the function name as you would use a primitive APL function. The information provided in the function header specifies the number of arguments to be supplied in the function call and determines whether or not a value will be returned. Section 6.2.5 provides examples of defined functions and their corresponding function calls.

It is also possible to issue function calls from within other functions. You can nest functions to any depth.

The following sections provide information on function execution. They focus on branching, suspending, tracing, stopping, and locking functions, using the state indicator, and trapping errors.

6.4.1 Branching Within a Function

APL statements in a function definition normally execute in the order determined by their line numbers. Execution begins at the first statement following the function header, terminates after the last statement in the definition, and executes only once. You can modify this standard order of execution by including branch statements in the function definition. A branch statement changes the sequence of execution by transferring control to another line in the function definition. Branching allows you to execute loops within the body of the function.

There are two types of branch statements: unconditional and conditional. An unconditional branch statement consists of a branch symbol (→), followed by the number of the function line or label to which you want to transfer control. For example:

```
    [5]   →1
```

This statement causes an unconditional branch from line [5] to line [1]. Line [1] is thus the next statement to execute.

The line number you specify after the → can be in the form of a constant, a variable, or an expression. It must be equivalent to an integer line number within the current function definition to allow execution to continue. If the integer does not reference a line number in the current function, the branch statement closes the function and returns you to immediate mode or to the caller. (APL users often deliberately specify an out-of-range number to stop execution.) Line number [0], the function header, is not a legitimate reference for a branch statement. Therefore, specifying →0 also closes the function and returns you to immediate mode or to the caller.

If the object of the branch is a nonempty vector, control passes to the line number referenced by the first element of the vector. If the vector is empty, the branch statement is not meaningful and the normal order of execution continues.

You can include a branch statement in the middle of a multi-statement line. However, if the branch executes, the rest of the expression to the left of it is ignored. If the branch does not execute, the result of the statement scanned so far is considered the empty vector. Therefore, the expression to the left of the branch is executed.

APL also allows you to include conditional branches in a function definition. A conditional branch executes as the result of evaluating a logical expression, not in response to any specific IF logic. There are two popular ways of doing a conditional branch. The first format is:

→ line number × logical expression

For example:

```
    →9×I>B
```

APL evaluates the logical expression to the right of the line number specification (→9). Logical expressions return either a 1 (true) or a 0 (false). Therefore, if *I* is greater than *B*, the value of the expression is 9×1 and control passes to line number [9]. In the expression →(*A*<*B*)/13, if the logical expression *A*<*B* evaluates to 1, (1/13), then control passes to line number [13]. (If the logical expression evaluates to 0, (0/13) returns a null so control passes to the next line.)

You can include more than one line number in a conditional branch but each line number must have a corresponding logical expression. Only one expression can evaluate to 1. Both the line numbers and the expressions are separated by commas. For example:

```
    →((A>B),(A<B),A=B)/7,8,9
```

APL transfers control to the line number whose expression evaluates to 1.

Note that you should use labels instead of line numbers in branch statements because APL renumbers when lines are added or deleted. See Section 6.4.2 for a description of labels.

6.4.2 Statement Labels

Because APL automatically renumbers function lines as consecutive integers when exiting from function-definition mode, problems can occur when you refer to explicit line numbers in branch statements. Instead, you can associate a line number with a label and reference the label, not the line number, as the object of the branch. For example:

```
    [15] INCR: I←I+1
           .
           .
           .
    [27] → INCR×I<IMAX
```

As shown above, a statement label consists of a distinct identifier, followed by a colon(:). When you specify the label in the branch statement, you do not include the colon. The internal value of the label identifier is the line number with which it is associated.

A label acts like a local variable in that its value is local to the function in which it appears. Label values are internally respecified upon each exit from function-definition mode. You cannot explicitly define a value for a statement label, and you cannot place a statement label in the function header.

Following are two examples of defined functions that use branching and statement-labeling techniques. Note that APL prints lines containing labels or comments one character to the left of the rest of the text.

```
      ∇FACTORIAL[⎕]∇
    ∇  R←FACTORIAL N
[1]    R←1
[2]    →0×⍳0=N
[3]    R←R×N
[4]    N←N-1
[5]    →2
    ∇
      FACTORIAL 5
120
      ∇FAC[⎕]∇
    ∇  Z←FAC N
[1]    →NZERO×⍳N=0
[2]    Z←N×FAC N-1
[3]   ⍝NOTICE THAT RECURSIVE DEFINITIONS
[4]   ⍝ARE PERMITED
[5]    →0
[6]   NZERO:Z←1
    ∇
      FAC 5
120
```

6.4.3 Suspending Function Execution

Function execution can be suspended before normal completion by means of:

1. An error report
2. An attention signal, two CTRL/Cs
3. The stop control vector
4. The ⎕*BREAK* system function

When an error occurs, APL suspends function execution and displays an error message and the line number where the error occurred. Appendix A lists possible errors you may encounter. The attention signal, two CTRL/Cs, is described in Section 1.4.4. The stop vector is described in Section 6.4.6. The ⎕*BREAK* system function is discussed in Section 4.3.1.

When function execution is suspended, APL displays the name of the suspended function and the line number of the statement that would have been executed next. APL then begins a new line, indents six spaces, and awaits input in immediate mode. You can perform any operation at this time, including erasing the suspended function (see the)*ERASE* system command, Section 5.4.2).

The suspended function remains active until you terminate it, erase it, or clear the currently active workspace. You can resume execution at any time by typing:

```
→ line
```

where line identifies the line number at which execution is to be continued. You can terminate a suspended function by typing:

```
→0 or just →
```

When a function is suspended, its local variables remain active. You can examine these variables and can specify their values by using an immediate-mode assignment.

⎕*LC* contains the line number of the line where execution was suspended. Therefore, →⎕*LC* restarts the suspended function at the beginning of the line that was interrupted.

6.4.4 Examining the State Indicator

The state indicator is a status vector that resides in your active workspace. You can examine the state indicator to determine the status of all active functions by specifying an)*SI* system command (Section 5.4.8). The)*SI* system command lists active functions as in the following example:

```
        )SI
T[3]    *
S[7]
R[6]
F[2]    *
```

The listing displays functions in the order in which they were most recently active. The example included above indicates that execution was suspended during execution of statement [3] of function *T*, which was called during line [7] of function *S*, which was called during line [6] of function *R*. (Before this sequence of calls, execution was suspended during execution of line [2] of function *F*.)

In the)*SI* display, an asterisk (*) following the name and line number indicates a suspended function. The other functions in the list are pendent. A pendent function is one which is awaiting return from another function - possibly a suspended one - which it called. You can edit a suspended function but not a pendent one. Although you can erase both suspended and pendent functions, you can cause considerable confusion by erasing a pendent function and then resuming execution of a suspended function that was called by that pendent function. Following is an example of an operation of this kind:

```
        )SI
A[3]    *
B[4]
C[5]    *

        )ERASE A
        )SI
 *
B[4]
C[5]
```

You can resume execution of *C* at this point but not *B*. In the *)SI* display, an asterisk without a line number indicates an erased suspended function. Whenever you erase a pendent function, APL displays:

```
    13 POSSIBLE SI DAMAGE
```

to warn you to consider the status of existing functions before you resume execution.

From the *)SI* listing you can also determine when quad-input requests are pending or an execute operation (∊, ⊥, or ⌽) has been invoked. Local functions (suspended or pendent) also appear in the *)SI* listing. An example of both of these special conditions is shown below:

```
          )SI
F[2]       *
A[7]
          T←⎕
⎕:
          ∊')SI'
          T
∊
⎕
F[2]       *
A[7]
```

You can clear the state indicator by terminating the execution of each suspended function in the list. There are several ways to accomplish this:

1. You can type a right arrow (→) for each function marked by an asterisk.

2. You can issue the I-beam function ⌶30 to clear the state indicator completely (Appendix C).

3. You can clear the state indicator by saving the active workspace, then clearing and copying it again (see the *)COPY* system command, Section 5.4.1).

If the state indicator is clear, APL outputs a blank line in response to *)SI*.

You can use the *)SIV* system command (Section 5.4.9) to obtain a more extensive display of the state indicator. In addition to the information accessible to *)SI*, *)SIV* returns a list of local and dummy variables for each function displayed. The current line being executed by the execute function is also displayed. The following is an example of an *)SIV* display:

```
        )SIV
S[6]    *         U         V
F[2]    *         T
A[7]              T         W         Z
T[3]

        ∇Z←A M B;⎕IO;Q
[1]     ⎕BREAK 'LINE 1 OF M'
[2]     ∇

        1 M 2
LINE 1 OF M

        )SIV
M[1]    *         Z         Q         ⎕IO       B         A
S[1]    *         V         U
```

This indicates that the variable *T*, local to function *F* is currently dominant, and that the variable *T* local to function *A*, as well as the function named *T*, are currently inaccessible.

6.4.5 The Trace Vector

You may find it helpful for debugging purposes to obtain an automatic printout of intermediate results of function execution. As a program tracing aid, you can output the values computed by one or more function statements each time those statements execute.

To set the trace vector, use the following format:

T∆function name←line number(s)

where function name is the name of the function you want to trace and the line number(s) are the lines you want information on.

You can set the trace vector in either immediate mode or within a function definition. For each execution of the line numbers you specify, the trace vector causes the following information to be displayed in the order shown:

function name
bracketed statement line number
final value returned by each statement on the line

An example of a trace operation is shown below:

```
              T∆F←4 6 7
              F
      F[4]    32.5 37.9
      F[6]    9
      F[7]    16 1.7
```

If the statement being traced is a branch statement, then the value printed is the line number to which control is passed by the branch. In the example above, line [6] was →9.

To trace all the statements of a function, for example *F*, you can supply the following specification:

```
      T∆F←⍳N
```

where *N* is a number at least as large as the number of statements in *F*.

To disable the trace vector, type the following:

```
      T∆FUNCTION NAME←⍳0
```

For example:

```
      T∆F←⍳0
```

The trace control vector can be set within a function to aid in selective tracing or setting breakpoints. For example, you may want to initiate tracing if certain conditions are in effect and disable it as soon as a specified value has exceeded a defined maximum.

If you edit a function for which you have defined a trace control vector, you clear the trace vector. Also, when you lock a function, you automatically clear the trace vector. The trace vector setting is saved with your workspace.

Note that APL identifiers cannot start with *T∆* because this is reserved for the trace vector syntax.

6.4.6 The Stop Vector

APL allows you to suspend execution of a function at predetermined points. A stop control vector is available with a syntax similar to that of the trace vector. To cause statement execution to stop before executing a particular line, you can type the following:

```
    SΔfunction name←line number(s)
```

where function name is the name of the function you want to suspend and line number(s) specify where you want to suspend execution. You can set the stop vector either in immediate mode or within a function definition. When you execute the function, the stop vector suspends execution at the first line number you specify. It displays the function name and the line number. You can resume execution by typing a branch to the desired line number (→5) or continue by typing →⎕*LC*. The stop vector will then suspend execution at the next line you specified.

When you edit a function for which you have defined a stop control vector, you automatically clear the stop vector. Also, when you lock a function, you automatically clear the stop vector. The stop vector setting is saved with your workspace.

Note that APL identifiers cannot start with *S*Δ because this is reserved for the stop vector syntax.

6.4.7 Locking a Function

APL allows you to lock a function definition to protect it from unauthorized use, to maintain security, or to treat a function as a proprietary program. To create a locked function, or to lock an existing function, you open or close the function with a del-tilde (⍫) character (protected del) rather than a simple del (∇). The del-tilde (⍫) is created by overstriking (∇) and (~).

The following example illustrates the locking of a previously unlocked function definition:

```
          ∇TRIG
    [19]    ⍫
```

A locked function cannot be edited in any way; if you try to edit a locked function, you will receive the error message, 5 *DEFN ERROR*. You cannot add, change, delete, or display a function line. Trace and stop vectors cannot be defined or changed for the function. Any trace or stop settings in effect at the time you lock the function are automatically cleared.

If an error occurs during execution of a locked function, the function name and line number at which the error occurred are displayed, but the contents of the line are not displayed. APL then causes an exit from all pendent functions that are locked until the function on the top of the *SI* stack is not locked. If all functions on the *SI* stack are locked, APL clears the *SI* stack and enters immediate mode. Note that you cannot unlock a function once it is locked. However, you can delete a locked function by using *)ERASE*.

CAUTION

If a locked function calls an unlocked function and the unlocked function becomes suspended, the environment of the locked function is available for examination.

6.5 ERROR TRAPPING

APL provides a form of error trapping that allows you to handle errors from within a user-defined function. With error-trapping aids you can handle a failure that occurs within a user-defined function in the same way as an APL primitive function handles a failure, that is, by informing its caller that it failed and why.

Normally, if APL detects errors while executing a function, it suspends execution and prints the error information on the terminal. However, if you plan ahead you can prepare alternatives that can save you time in the event of an error. What you need is the ability to gain control once an error occurs. You need to know what the error was and where it occurred, that is, the function name and line number of the statement that failed along with an indication of where the error occurred in the line. It would also be useful if you could halt execution of a function, check the flow of logic, then reenter the function and resume execution.

The following system variables and system function, whether used together or separately, give you the tools to create your own error handling routines:

1. ⎕*BREAK* system function, Section 4.3.1
2. ⎕*ERROR* system variable, Section 4.2.9
3. ⎕*SIGNAL* system function, Section 4.3.10
4. ⎕*TRAP* system variable, Section 4.2.21

6.5.1 Considerations for Error Handling

There are certain questions you should answer before you begin to define error routines. These questions are:

1. What error do you want to trap?
2. Where (in what function (stack level)) do you want to handle the error?
3. What errors do you want someone else to handle? (⎕*SIGNAL* to caller)

4. When do you hit the panic button and which panic button do you hit? (⎕*BREAK* to immediate mode or let APL have it.)

5. Where do you go after you fix your mistake?

Taking these questions into consideration before you define a function can help you optimize your error trapping.

Section 6.5.2 contains an example of how to handle one error trapping situation.

6.5.2 Error Trapping Examples

The first example sets a trap to handle the occurrence of an index origin set to 0. If an error occurs during function execution, APL executes ⎕*TRAP* which, in this case, is set to go to label *IO*. At label *IO*, APL checks to see if ⎕*IO* is equal to 0. If ⎕*IO* is 0, APL proceeds to label *SET*, where ⎕*IO* is reset to 1. Execution then proceeds to label *DIV*, where APL executes the expression again, this time with ⎕*IO* set to 1.

```
      ∇  Z←A DIVIDE B;⎕TRAP
[1]     ⍝THIS FUNCTION TAKES A NUMBER, A, AND
[2]     ⍝DIVIDES IT BY ⍳B.
[3]     ⍝EXAMPLE: 2 DIVIDE 3 WILL RETURN
[4]     ⍝A 3-ELEMENT VECTOR
[5]     ⍝OF 2÷1, 2÷2, 2÷3.
[6]      ⎕TRAP←'→IO'
[7]     DIV:Z←A÷⍳B
[8]      →0
[9]     IO:→(0=⎕IO)/SET
[10]     ⎕BREAK 'DIVIDE ERROR'
[11]    SET:⎕IO←1
[12]     →DIV
      ∇
        ⎕IO←0
        25 DIVIDE 5
25 12.5 8.333333333 6.25 5

        ⎕ERROR
 15 DOMAIN ERROR
DIVIDE[7]DIV:Z←A÷⍳B
                  ^

        )SI

        ⎕IO
1
```

Notice that APL executed the function even though ⎕*IO* is set to 0. ⎕*ERROR* contains the error that occurred, but the trap handled the situation. ⎕*IO* is now set to 1.

The state of *⎕IO* is the only check point in this function. If another error occurs, the function suspends execution and breaks to the terminal because of *⎕BREAK*.

```
          'A' DIVIDE 5
      DIVIDE ERROR
          ⎕ERROR
       15 DOMAIN ERROR
      DIVIDE[7]DIV;Z←A÷⍳B
                        ^

          )SI
      DIVIDE[10] *
```

⎕BREAK prints the argument you supplied and suspends execution. *⎕BREAK* does not set *⎕ERROR* but *⎕SIGNAL* does.

The next example works with three functions: *MASTER*, *ERROR*, and *SNIGGLE*. *MASTER* takes a right argument of either a scalar or a vector. It adds 50 to each element and then passes the vector to *SNIGGLE*. *SNIGGLE* tries to turn the vector into a square matrix. If the matrix cannot be squared (does not have an integer square root) *SNIGGLE* signals a user-defined error '550 *NOT SQUARE*' and pops back to *MASTER*. *MASTER* has a trap set to send errors to the function *ERROR*. Here are the functions:

```
          ∇MASTER[⎕]∇
        ∇  Z←MASTER VECTOR;⎕TRAP
      [1]   ⎕TRAP←'ERROR'
      [2]   ⍝FEED MASTER A VECTOR, ADD 50 TO EACH ELEMENT
      [3]   ⍝AND PASS IT TO SNIGGLE TO MAKE IT SQUARE
      [4]    SNIGGLE VECTOR+50
      [5]   ⍝IF IT REFUSES TO GET SQUARE, SEND IT TO ERROR
      [6]    Z←MATRIX
        ∇
          ∇SNIGGLE[⎕]∇
        ∇  SNIGGLE VEC;⎕TRAP
      [1]   ⍝THIS PROGRAM TURNS THE VECTOR INTO A SQUARE MATRIX
      [2]   ⍝IF THE VECTOR WON'T GO, SIGNAL UP TO THE CALLER.
      [3]    ⎕TRAP←'→ SNIG'
      [4]    MATRIX←(((⍴VEC)*0.5),(⍴VEC)*0.5)⍴VEC
      [5]    →0
      [6]   SNIG:'NOT SQUARE' ⎕SIGNAL 550
        ∇
          ∇ERROR[⎕]∇
        ∇  Z←ERROR;A
      [1]    →(550≠⎕FI 3↑⎕ERROR)/EEK
      [2]    A←⌈(⍴VECTOR)*0.5
      [3]    →0,Z←MASTER VECTOR←VECTOR,((A*2)-⍴VECTOR)⍴0
      [4]   EEK:⎕BREAK 'UH OH'
        ∇
```

The following function executes *MASTER*.

The sequence of events is as follows:

1. *MASTER* is called with a vector that is not square, ⍳8.

2. *MASTER* initializes the localized ⎕*TRAP*.

3. *MASTER* calls *SNIGGLE* with the argument ⍳8+50.

4. *SNIGGLE* initialized its localized ⎕*TRAP*.

5. At *SNIGGLE*[4], the expression causes a 15 *DOMAIN ERROR*, thereby invoking the error trap.

6. ⎕*TRAP* is equal to '→*SNIG*' so ⍎ ⎕*TRAP* redirects execution to line [6].

7. At this point, ⎕*ERROR* contains the 15 *DOMAIN ERROR*.

8. At line [6],*SNIGGLE* uses ⎕*SIGNAL* to set ⎕*ERROR* to the user-defined error message.

9. ⎕*SIGNAL* terminates execution of *SNIGGLE* and forces an error at *MASTER*[4], where *SNIGGLE* was called.

10. At *MASTER*[4], error trapping is invoked. (Note that *MASTER* is on top of the)*SI* stack now.)

11. ⎕*TRAP* is equal to '*ERROR*', so ⍎ ⎕*TRAP* executes the function *ERROR*. (Note that ⎕*LC* at this moment is line [4], where the error occurred.)

12. The function *ERROR* checks to see if ⎕*ERROR* contains error number 550. If it does not, *ERROR* breaks to the terminal for assistance.

13. The function *ERROR* corrects error 550 by extending the argument *VECTOR* with enough zeroes to make it square. (Note that *VECTOR* and *Z* in *ERROR*[3] are local to the first call to *MASTER*.)

14. *MASTER* is called with the vector 1 2 3 4 5 6 7 8 0, which sets the global matrix to:

```
51  52  53
54  55  56
57  58  59
```

    then returns to the function *ERROR*.

15. The function *ERROR* returns to ⍎ ⎕*TRAP* at *MASTER*[4].

16. The trap expression does not cause control transfer and is complete. Therefore, APL breaks to the terminal with the)*SI* equal to *MASTER*[4]*.

```
        MASTER ⍳8
  51  52  53
  54  55  56
  57  58  50
        ⎕ERROR
550 NOT SQUARE
MASTER[4] SNIGGLE VECTOR+50
            ^

        )SI
MASTER[4] *

        →
```

Note that the error is signalled in *MASTER* not *SNIGGLE*.

The next execution of *MASTER* causes a break to the terminal:

```
        MASTER 'ABC'
UH OH
        ⎕ERROR
 15 DOMAIN ERROR
MASTER[4] SNIGGLE VECTOR+50
            ^

        )SI
ERROR[4] *
⍎
MASTER[4] *
```

The following functions cause surprising results when executed:

```
        ∇F[⎕]∇
    ∇  Z←F A
[1]     ⎕TRAP←'→4'
[2]     Z←G A
[3]     →0
[4]     'MATRIX INVERSION ERROR, MATRIX WAS:'
[5]     A
[6]     ⎕ERROR
    ∇

        ∇G[⎕]∇
    ∇  Z←G A
[1]     Z←⌹A
    ∇
```

Note the following sequence of events:

```
        A←2 2⍴ 1E20 0 0 1E¯20
        F A
MATRIX INVERSION ERROR, MATRIX WAS:
  1.000000000E20    0
  0                 1.000000000E¯20
 11 VALUE ERROR
F[2]    Z←G A
          ^

 11 VALUE ERROR
        F A
        ^

       ⎕ERROR
 11 VALUE ERROR
        F A
        ^

       )SI
```

This is what happened:

1. *F*[1] sets ⎕*TRAP* to go to line [4], where the inversion error is reported.
2. *F*[2] calls *G* with the singular matrix *A*.
3. *G*[1] gets a domain error which invokes error trapping.
4. ⎕*TRAP* is now set to the global value '→4', so at line [4], APL does ⍎ ⎕*TRAP*.
5. There is no line [4] in *G* so *G* exits back to *F*[2], without setting a return value for *Z*.
6. APL reports the error

   ```
     11 VALUE ERROR
   F[2] Z←G A
          ^
   ```

   and invokes error trapping which again does ⍎ ⎕*TRAP* and goes to line [4].
7. The message, the value of *A*, and an unexpected ⎕*ERROR* are printed.
8. *F* exits but never sets its return value *Z*. Therefore, APL complains with

   ```
     11 VALUE ERROR
        F A
        ^
   ```

The specific error expression '→4' in *F* had surprising consequences when executed in *G*.

CHAPTER 7

THE FILE SYSTEM

7.1 INTRODUCTION

The file system is an integral part of the APLSF language itself. It allows you to store data files on a number of system devices.

APL can create and handle a variety of file types. You determine the size and content of the records and the structure and access properties of the file. You can write records into a file in either immediate mode or function-execution mode and subsequently retrieve them. One of the most significant extensions of this implementation of APL is the inclusion of a powerful data-file capability.

APL allows you to create and store four types of files:

1. ASCII sequential
2. Internal sequential
3. Direct access
4. Binary access

ASCII sequential files allow you to create and/or read any standard ASCII file while at APL level. You can read and work with an ASCII file created by another language, or you can create an ASCII sequential file to be passed to a program in another language such as FORTRAN or COBOL. Internal sequential files and direct-access files can be created and read only by APL functions (or by you in immediate mode). Binary-access files can be accessed in any format as random-access memory, and can be read and written in almost any language.

The file system has three distinct components:

1. File functions that allow you to read from a file (⍇) and write to a file (⍈)
2. System functions that allow you to assign, deassign, close, rename, and append files
3. System commands that create a direct-access file and divert terminal input and output to other devices

This chapter focuses on the following:

1. Access methods
2. APL file input and output functions
3. Basic file system functions
4. Sequential files
5. Random-Access files
6. Utility system functions
7. Synchronizing shared-file access
8. Handling I/O from non-terminal devices

7.2 ACCESS METHODS

The methods that you use to store or retrieve data in a file are determined by the file's organization. The organization of a file is fixed at the time you create it, but, depending on the access allowed, an access method can change each time the file is opened. In some cases, you can vary the access during function execution or during immediate mode. You can use two types of record access: sequential or random. Sequential indicates that the records are accessed in a serial order; random indicates that records can be accessed directly at any point in the file.

Table 7-1 shows the relationships between file organization and record access.

Table 7-1
Access Methods

File Organization	Access Method Allowed
ASCII sequential	Sequential only
Internal sequential	Sequential only
Direct access	Sequential and random
Binary access	Sequential and random

The following sections discuss each type of record access.

7.2.1 Sequential Access

All file organizations allow you to access records sequentially. Sequential record access is employed when you issue a series of requests for the next record. The record operations are performed in terms of a predecessor-successor record relationship. For each successfully accessed record (except the last) there is a succeeding record somewhere in the file.

Sequentially organized files (ASCII sequential, internal sequential, and binary-access sequential) allow only sequential access. In these files, each record except the last is physically adjacent to the next record. Sequential access to a sequential file means that records are accessed in the order of their insertion into the file. A particular record can be read only after each preceding record has been successfully read. Similarly, once a record has been read or written, you must reposition the file to the beginning before preceding records can be accessed.

When you assign a sequential file with either the */AS* switch (ASCII sequential) or the */IS* switch (internal sequential), APL positions the file pointer at the beginning of the file. You must then do sequential reads to get to the particular record you want. If you do a write operation at the beginning of the file, you overwrite the existing file, not just the existing record.

When you assign a sequential file with either the */AS** switch or the */IS** switch, APL positions the file pointer at the end of the file. You can then append records to the end-of-file with a write operation; a read gets an end-of-file.

Direct-access and binary-access files can be accessed sequentially. Sequential access to a direct-access or binary-access file means that records are accessed in ascending order according to record number (for direct-access) or word number (for binary-access). A sequential read from one of these files finds the next record by adding 1 to the value of the record/word number used in the previous I/O operation.

Binary-access files allow the writing of data based on word position. Empty words are assigned a value of integer 0. Direct-access files allow empty record positions that can be caused by a record deletion or by your purposely leaving positions empty. APL maintains the predecessor-successor relationship through its ability to recognize a record position as either empty or occupied.

7.2.2 Random Access

Random access allows you to control the order of record access. The predecessor-successor relationship has no effect on random access. You identify each record of interest in each operation. This procedure allows you to access records in any order at any point in the file. Random access is not permitted on ASCII and internal sequential files because of the strict physical relationship maintained among records. Direct-access and binary-access files do allow random access.

By specifying a record number in a direct-access file or by specifying a word number in a binary-access file, you can access any record in the respective file. You can also alternate the type of access to these files, sequential or random.

Records in APL random-access files are called components. There is no restriction that all records in these files be the same length.

7.3 FILE INPUT/OUTPUT FUNCTIONS ⍇ AND ⍈ OR .IQ AND .OQ

You can initiate input and output to and from a file either in immediate mode or during function execution. When you perform a read operation, you are requesting input data from the file. When you perform a write operation, you are outputting data to the file.

APL provides two quad functions for files, one to perform file input and one to perform file output: ⍇ (.IQ) and ⍈ (.OQ). These functions work in much the same way as the basic quad input and output functions described in Section 2.5. The file input and output characters are formed by overstriking the quad (⎕) with either the left arrow (←) or the right arrow (→).

The syntax of the I/O functions is explained with each file organization.

7.4 BASIC FILE SYSTEM FUNCTIONS

The following sections describe four basic system functions that perform the following file operations:

1. Assigning a file ⎕*ASS*
2. Deassigning a file ⎕*DAS*
3. Closing a file ⎕*CLS*
4. Renaming a file ⎕*RENAME*

The names of these system functions, like those described in Chapter 4, begin with a quad (⎕) character and are considered to be distinguished names. That is, you cannot use them for user-defined function names, and you cannot copy, erase, or collect them in a group.

7.4.1 ⎕*ASS* - Assigning a File

Format

⎕*ASS*' ⟦channel⟧ filename ⟦password⟧⟦/file org⟧⟦/share⟧⟦/dump⟧ '

where

channel is an integer scalar in the range 1 through 12 inclusive. If you do not specify a channel number, APL assigns you one.

filename identifies the name of the file to be read or written on the specified channel. The filename has the same format as a workspace filename. See Section 5.1.3.1.

password is optional. The default is (-).

/file org is one of the file organizations listed in Table 7-2. If you do not specify this switch, the default is */DA*.

/share is a switch that allows you to extend multiple-user characteristics. It is relevant to direct-access and binary-access files only (Sections 7.6.2 and 7.6.5).

/dump is a switch used for magnetic tape I/O. It is relevant only to binary-access files (Section 7.6.8.1). The single quotation marks are required.

The ⎕*ASS* system function assigns a file to a specified channel number. In this way, you can refer to the channel number rather than the filename specification when performing I/O. You can also use ⎕*ASS* to return information concerning a file. By specifying the channel as the argument, you receive the name of the file currently assigned to that number plus any other characteristics you may have specified previously.

For example:

```
      ⎕ASS 2
DIRACC.EXM [4,204]/DA
```

If the channel you specify is currently unassigned, APL returns a null vector.

The ⎕*ASS* system function operates in the same manner as any other (APL function. ⎕*ASS* returns the value that is the channel number you assigned. Therefore, you can specify a variable to receive the value of the channel number.

For example:

```
      CHAN←⎕ASS'TEST/AS'
      CHAN
12
```

Because the range of channels is 1 through 12 inclusive, you cannot access more than 12 files simultaneously. If you do not specify a channel, APL assigns an available channel in the system and returns this number as the function result. If you assign a channel number (for example 12) that has already been assigned to a file, APL closes the first file and deassigns it from the channel, then assigns the new file to that channel number. If a syntax error is encountered in the ⎕*ASS* function or if there are no available channels, APL returns a function result of 0, which means your assign failed.

The ⎕*ASS* system function does not cause input or output to be performed. It establishes a connection between a filename and a specified channel.

Table 7-2
File Organization Switches

File org Switch	Default File Extension	Type of File
/AS	*.AAS*	ASCII sequential
*/AS**	*.AAS*	ASCII sequential; file is positioned at end of file to allow appending
/IS	*.AIS*	Internal sequential
*/IS**	*.AIS*	Internal sequential; file is positioned at end of file to allow appending
/DA	*.ADA*	Direct-access which supports reading and writing
/DI	*.ADA*	Direct-access which supports reading only
/BS	*.ABI*	Binary-access which supports reading or writing, but not both; file can be read by multiple users, but written by only one user at a time
*/BS**	*.ABI*	Binary access; file is positioned at end of file to allow appending and same user capability as */BS*
/BU	*.ABI*	Binary-access which supports reading and writing; file can be used by only one user at a time

7.4.2 ⎕*DAS* - Deassigning a File

Format

⎕*DAS* channel(s)

where

channel(s) is either a numeric scalar or vector, or a null vector.

The ⎕*DAS* system function deassigns the files on one or more channels in the system. In general, ⎕*DAS* reverses the operations performed by the ⎕*ASS* system function. It disassociates the channel number(s) with the file(s). If the files associated with the channel numbers being deassigned have not been closed (⎕*CLS*), ⎕*DAS* closes these files automatically.

Like the ⎕*ASS* system function, ⎕*DAS* returns a function result. In all cases, this result is a null vector.

For example:

```
    ⎕DAS 1
```

This example outputs a blank line and deassigns the file associated with channel 1. The following example deassigns files from three channels:

```
    ⎕DAS 2 3 5
```

You can deassign the files on all channels by specifying a null vector as the argument (or ⍳12).

For example:

```
    ⎕DAS ⍳0

    ⍝SAME AS
    ⎕DAS ⍳12
```

7.4.3 ⎕*CLS* - Closing a File

Format

⎕*CLS* channel(s)

where

channel(s) is either a numeric scalar or vector, or a null vector.

The ⎕*CLS* system function closes the files on one or more channels in the system. However, ⎕*CLS* does not deassign the channel from the file. This capability is useful when you want to return to the beginning of a sequential file after performing an operation. After you close the file, the next read operation reads the first record in the file; the first write truncates the file. There is no need to reassign the file to the channel.

The following example closes the file associated with channel 2:

```
    ⎕CLS 2
```

The ⎕*CLS* system function always returns a null vector as a result. You can specify more than one file or all files.

For example:

```
    ⎕CLS 2 3

    ⎕CLS ⍳0

    ⍝SAME AS
    ⎕CLS ⍳12
```

7.4.4 ⎕*RENAME* - Changing the File Specification

Format

'filespec' ⎕*RENAME* channel

where

filespec is the new file specification. You can specify all or part of a new file spec. Every element except the name is optional. The single quotation marks are required.

channel is the number associated with the file.

Specifying elements of the filespec-a new device name, extension or filetype, protection code, or directory- is optional. The old value is the default.

The ⎕*RENAME* system function renames a currently assigned file. Therefore, before you can use ⎕*RENAME*, you must assign the file with ⎕*ASS*. When you execute ⎕*RENAME*, you also close the file if it is open.

If other users have the file open when you issue a ⎕*RENAME*, the ⎕*RENAME* will fail, but the close will be done in any case. Also, if other users have the file assigned but not opened (have not performed a read or write) they cannot use the file until they reassign it under the new name. However, you, the user performing the ⎕*RENAME*, will automatically have the file reassigned on the same channel under its new name.

For example:

```
      ⎕ASS'TEST/AS'
10
      'NEW' ⎕RENAME
WAS TEST,AAS [4,204]/AS

      ⎕ASS 10
NEW,AAS [4,204]/AS
```

7.5 SEQUENTIAL FILES

APL supports three types of sequential files: ASCII sequential internal sequential, and binary-access sequential files (*/BS* and */BS**). The first type, ASCII sequential, is a standard operating system ASCII sequential file that can be read or written by APL or by other languages. This file format is line-oriented; a record is delimited by a carriage return. Therefore, each line is considered a record in the file, and records in the file can be of different lengths. You can display ASCII sequential files on terminals and high-speed printers. To display an ASCII sequential file on the terminal, return to operating system command level and type a standard system command:

```
TYPE filename
```

To display such a file on a line printer, type the operating system command:

```
PRINT filename
```

The second type of sequential file, internal sequential, is a file that can be read or written only by APL. In this file format, information is read and written in internal binary format. In internal-sequential files, a record is all the data written to the file in a single output operation, rather than a single line in an ASCII sequential file.

The third type, binary-access sequential, is a file that can be read or written by a file in another language such as FORTRAN, as well as APL.

APL does not open a sequential file for input or output until the moment when the first read or write request is made. APL does not normally close the file until you direct it to do so with a ⎕*DAS* or ⎕*CLS* system function, or a *)LOAD*, *)CLEAR*, *)OFF*, or *)CONTINUE* system command. This implies that if user *A* is writing a sequential file and user *B* subsequently starts to read the same file, user *B* will read the copy of the file that does not have the updates just made by user *A*.

If your first reference to a sequential file is an input request (read) all subsequent I/O requests to that file must be input requests, until you close the file. Similarly, if the first file reference is an output request, all subsequent I/O requests must be output requests. Note that you can store sequential files on many types of system devices.

The following subsections describe the way in which I/O functions for ASCII sequential and internal sequential file organizations are formatted. The description of binary-access files begins at Section 7.6.4.

7.5.1 ASCII Sequential I/O

The following format requests input from an ASCII sequential file:

```
    ⍇ ⟦[mode]⟧ channel
```

where

> ⍇ is the input quad function.
>
> [mode] is one of the integer scalars listed in Table 7-3. It specifies both the type of data and the character set of the data being read. mode is optional but, if present, you must enclose it in square brackets.
>
> channel is the channel number associated with the file. The value of the ⍇ function is the data read.

The following format requests that output be written to a file:

```
    data ⍈ ⟦[mode]⟧ channel
```

where

> data is the information you want to write to the file.
>
> ⍈ is the output quad function.
>
> [mode] is one of the integer scalars listed in Table 7-3. It specifies both the type of data and the character set of the data being written. mode is optional, but if present, you must enclose it in square brackets.
>
> channel is the channel number associated with the file. The value of ⍈ is data.

Because you can write different types of data to a file, you must tell APL how to read the data by specifying an input mode. When reading, you also specify whether the record was written with mnemonics or with the APL character set. Table 7-3 lists input modes and their meaning.

Table 7-3
Input Modes

Input Type	Character Set	Mode
⎕	TTY	1
⍞	TTY	2
⍔	TTY	3
⎕	APL	4
⍞	APL	5
⍔	APL	6

The default input mode for TTY is 1; the default input mode for APL is 4. For more information on input quad types, refer to Section 2.5. For ⍈, modes 1, 2, and 3 are equivalent and modes 4, 5, and 6 are equivalent. When accessing APL, you have the option of specifying a particular APL character set on output when you respond to *TERMINAL..* with TTY. The modes 4, 5, and 6 use the setting you specified at access time. See Section 1.2 for terminal designators and Section 7.9 for *)INPUT* and *)OUTPUT* to non-terminal devices.

When you are reading or writing to an ASCII sequential file, the first thing to do is to assign the file to a channel. If you want to append to the file, specify */AS***. For example:

```
        ⎕ASS '2 FILE/AS*'
  2
```

When you are writing to a file, using the output quad ⍈ is the same as using ⎕←*A*, except that the output is written to a file and not the terminal. A record is a string of ASCII characters terminated by a carriage return. APL inserts the carriage return/line feed into the data. The current width value associated with the active workspace is used to determine the maximum length of the line to be output. This means that the current page width determines the length of the record if the data is longer than the maximum. APL inserts a carriage return/line feed when you reach the maximum setting. You can change the setting with the ⎕*PW* system variable (Section 4.2.15). You can also divert output to devices other than your terminal with the *)INPUT* and *)OUTPUT* commands, Section 7.9. If you output to a line printer, you should change the width value to 130.

The following example illustrates the writing and reading of an ASCII sequential file. If data is an expression, you must enclose it within parentheses.

```
        ⎕ASS '2 OUTPUT/AS'
  2


        ⍝⌹ USES THE APL CHARACTER SET
        ⍝TERMINAL IS AN LA37

        'FIRST RECORD'⌹2
  FIRST RECORD
        'SECOND RECORD'⌹2
  SECOND RECORD
        (2 4⍴⍳8)⌹2
    1   2   3   4
    5   6   7   8
        ⎕CLS 2

        ⌹[5]2
  FIRST RECORD
        ⌹[5]2
  SECOND RECORD
        ⌹[4]2
  1 2 3 4
        ⌹[4]2
  5 6 7 8
        A←⌹[4]2
        ⍴A
  0 75
```

In the previous example, the first two records are strings and the third record is a numeric expression. To read the output, you first close the file because you cannot do both input and output at the same time. You do not need to assign the channel; *⎕CLS* does not deassign the channel.

Notice that, because the numeric expression is a matrix, APL had to insert a carriage return/line feed to format it properly. Therefore, even though you write an array as one record, it resides in the file as more than one record. For example, the above matrix had two records when read.

The expression *A*←⌹[4]2 requests input and assigns the value to *A*. Since the file pointer was at the end-of-file, the value was a null array. The ⍴*A* returns the null array of shape 0 75: the error number is 75. (See Appendix A for error messages). This error message means end-of-file. A blank line in a file is ignored in mode [1] and is a null vector in modes [2] and [3]. A blank line is ignored in mode [4] and is a null vector in modes [5] and [6].

To write to the end of an ASCII sequential file, assign the file with the */AS** switch. This positions the file pointer at the end of the file so you can append to it with write operations without overwriting current records. A read to a */AS** file gives an EOF.

7.5.2 Internal Sequential Files

The following format performs a read from an internal sequential file:

⍇ channel

where

⍇ is the input quad function.

channel is the channel number assigned to the file. The value of ⍇ is the data read.

The following format writes a record to an internal sequential file:

data ⍈ channel

where

data is the information you want to write into the file.

⍈ is the output quad function.

channel is the channel number assigned to the file. The value of ⍈ is data.

When performing I/O on internal sequential files, you need not specify an input mode as you do with ASCII sequential files. Information in an internal sequential file is stored in the internal format of APL, which is very different from ASCII format. No conversion effort is incurred by using internal sequential files; compared with ASCII sequential files, there is practically no overhead involved in reading and writing internal sequential files.

When you are reading or writing an internal sequential file, the first thing to do is to assign the file to a channel. If you want to append to the file, use the */IS** switch. For example:

```
      ⎕ASS 'INT/IS*'
5
```

In an internal sequential file, a record is all the information previously written in a single output operation. A single read will retrieve all information output at a single write rather than a single line that would be retrieved by an ASCII sequential operation. Also, when you write an array to an internal sequential file, APL includes the shape of the array along with the array itself. Therefore, you need only a single read to retrieve the array. The array is stored in one record.

Another comparison between ASCII sequential and internal sequential is that 0 75 indicates end-of-file in internal sequential as well as ASCII sequential. A blank record in */IS* returns blanks. For example:

```
          ⎕ASS 'INT/IS'
    4
          'TOPS-10\APL'⌷4
    TOPS-10\APL
          'TOPS-20\APL'⌷4
    TOPS-20\APL
          (2 4⍴⍳8)⌷4
      1  2  3  4
      5  6  7  8
          ⎕CLS 4

          ⌷4
    TOPS-10\APL
          ⌷4
    TOPS-20\APL
          ⌷4
      1  2  3  4
      5  6  7  8
          ⌷4

          A←⌷4
          ⍴A
    0 75
```

Note that internal sequential files can reside on many system devices.

7.6 RANDOM ACCESS FILES

The following sections describe the characteristics of files that you can access randomly as well as sequentially: direct-access files and binary-access files.

7.6.1 Direct-Access Files

A direct-access file is structured as a collection of variable-length records with a directory containing pointers to each record. A record can be any size. A direct-access file can reside only on disk but the only limit imposed on the size of the file is the amount of disk space available. You must specify the number of records you plan to write to the file when you first create the file. Unlike the procedure for creating ASCII sequential, internal sequential, and binary-access files, the procedure for creating a new direct-access file requires that you use a system command, *)CREATE*.

The *)CREATE* command has the following format:

```
     )CREATE filesize filespec ⟦password⟧⟦blocking factor⟧
```

where

> filesize is an integer that specifies the number of records you plan to write. You are only limited by the amount of disk space you have available.
>
> filespec is the device name, filename, extension or type, directory, and protection code. See Section 2.1.4. Everything but the filename is optional.
>
> password is optional. The default is a hyphen (-). (Direct access is the only file type with a password.)
>
> blocking factor is one of the following values: 8, 16, 32, 64, and 128 words. The default is 16 words.

When you specify the *)CREATE* command, APL sets up the parameters of the new file according to your specifications. It creates a directory that acts as an index for the file. The filesize determines how many records the directory, and subsequently the file, will contain. For every record you write, there is an entry one word long (36 bits) in the directory. This entry contains information about the record such as its file position and its size. This directory allows you to access records randomly by the number position in the file. The directory keeps track of where the records are.

Because you can delete records in direct-access, there is another area in the file called the free-space area. The free-space area keeps track of where the holes are in the file. When you delete a record, an entry is made in the free-space area.

The blocking factor you specify with the *)CREATE* command allows you to allocate space more efficiently in your file. By specifying a blocking factor you actually set up a fixed length for each record. If the data you enter is less than the number of words you need to block it, APL fills the rest of the record with nulls. For example, if you specify a blocking factor of 16 words and the data you enter is 35 words, the record will occupy 48 words of space. By specifying a smaller blocking factor, this type of waste can be minimized but writing records with smaller blocking factor takes longer if the record lengths are bigger than the blocking factor.

Note that you do not need to use the *)CREATE* command for direct-access files that already exist. Once you create a direct-access file, its characteristics remain until you delete the entire file.

7.6.2 Sharing Direct-Access Files

Before you do any I/O on a direct-access file, you must assign it to a channel with ⎕*ASS*. Because you can share direct-access files, you have the option of specifying one of three switches with the ⎕*ASS* operation. They are:

/DI	multiple-user access; file can only be read.
/DA	single-user access; file can be read and written only by one user at a time.
/DA/SHARE	multiple-user access; file can be read and written.

If one user opens a file in */DA* mode and another user attempts to access the same file in */DA* or */DA/SHARE* mode, the operation will fail and the error message 32 *FILE BEING MODIFIED* will be displayed.

Sharing of direct-access files is synchronized by the ⎕*ENQ* and ⎕*DEQ* system functions described in Section 7.8.

7.6.3 Direct-Access I/O

The following format performs a read from a direct-access file:

⍇ ⟦[record]⟧ channel

where

⍇ is the input quad function. The value of ⍇ is the data read.

[record] is the component number of the record you want to access. If [record] is not specified, the default is the next record. If [record] is specified, you must enclose it in square brackets.

channel is the channel number of the file.

The following format writes a record to a direct-access file:

data ⍈ ⟦[record]⟧ channel

where

data is the information you want to write to the file.

⍈ is the output quad operator.

[record] is the component number you want associated with the record. The default is the next record position after the last operation.

channel is the channel number associated with the file. The value of the ⍈ is data.

The following format writes the record at the end:

data ⎕*APPEND* channel

where

data is the information you want to write to the end of the file.

⎕*APPEND* is the system function that allows you to write to the end-of-file.

channel is the channel number associated with the file.

The value of ⎕*APPEND* is data.

⎕*APPEND* acts like ⍈ except for where the new record is logged into the directory. ⎕*APPEND* finds the highest record number used and adds one to that number. This becomes the record number of the new record. The new record is logged into the directory and the record is written for random access. If you proceed to do a read, you will find you are at the end-of-file. If you then do another write ⍈, you perform the same operation. ⎕*APPEND* will not enlarge the maximum number specified in)*CREATE*.

When you write a record to a direct-access file, you or APL, associate a record number with the data written. For input, you specify the number of the record you want to read. Because the directory keeps track of the record numbers, you can read and write records in any order. For example, you can write record 10 before you write record 9. If you try to read a record that does not exist, the value of the read is a null array. The shape of this array is 0 75, where 75 indicates an end-of-file.

To delete a record, write a null array of shape 0 75 in its place. For example:

```
      (0 75⍴0)⍈[5]C
```

You can access records both randomly and sequentially. If you do not specify a record number in your read or write operation, APL uses as the record number, 1 plus the value of the record position used in the previous I/O operation. If the current ⍇ is the first I/O operation on the channel, record one, the first record in the file is used as the default.

The following is an example of creating and accessing a direct-access file:

```
          )CREATE 100 DIRACC,EXM
          ⎕ASS '2 DIRACC,EXM/DA'
      2
          R←(2 4⍴⍳8)⍈[98]2
          'RECORD 8'⍈[8]2
      RECORD 8
          ⍴D←⍇[98]2
      2 4
      ⍝NEXT OPERATION WRITES TO RECORD 99
      (⍳4)⍞2
1 2 3 4
      ⍞[8]2
RECORD 8
      ⍞[99]2
1 2 3 4

      ⎕CLS 2
```

For the sake of efficiency, APL will not write a record across a block boundary (128 words) unless the record is larger than 128 words. Because of this limitation, it is very inefficient to have files where all records are between 65 and 100 words. It is more efficient to write fixed-length records. For example, if you realize that all records will be between 10 and 32 words, you can specify a blocking factor of 32. This will reduce CPU time and the real time involved in I/O operations. The file size will, however, be larger than if you had specified a blocking factor of 8 or 16. If most of your records are larger than 128 words, you should set the blocking factor to 128.

You can also reduce processing time by updating the file in order of record numbers - for example, by writing records 5, 9, and 200 in order instead of 5, 200, and 9. When updating a file, you should perform deletions first and then replacements and additions.

The formula for determining the number of words that data (for example, the letter *A*) will take up, is:

```
3+(⍴⍴A)+⌈(×/⍴A)÷L
```

where

L is	0.5	to indicate floating-point
	1	for integer
	4	for character
	36	for Boolean

Note that direct-access files can reside only on disk or drum devices.

7.6.4 Binary-Access Files

APL treats a binary-access file as random-access memory. By using a binary-access mode, you can access a file of any format or characteristics. For example, you can read and write FORTRAN or COBOL random-access files with an APL binary-access function.

To read or write this type of file, you specify the following:

1. The word of the file at which reading or writing is to begin

2. The type of values to be found beginning at the specified word (for example, integer, ASCII)

3. The number of values to be read or written

Data is stored in variable-length records that you can access randomly or sequentially by specifying a word position, rather than record position as in direct-access files.

Because you specify data type when performing I/O (see Sections 7.6.6 and 7.6.7), you can read or write any type of file organization including ASCII sequential, internal-sequential, and direct-access files.

For magnetic tapes in binary-access mode, you can perform a variety of magnetic tape operations with the ⎕*MTP* system function. Refer to Section 7.6.8.2 for more information on ⎕*MTP*.

Binary-access files can be shared with other users. See Section 7.6.5 for file sharing information.

7.6.5 Sharing Binary-Access Files

Before you can perform I/O on a binary-access file, you must assign it to a channel number with ⎕*ASS*. You have the option of sharing binary-access files with other users. You control the extent of the sharing by specifying certain switches with ⎕*ASS*. They are:

/BS	File can be read or written but not both; file can be read by multiple users but written only by one user at a time. Sequential access only.
*/BS**	Same access privileges as */BS*, except sequential output begins at end-of-file. Sequential access only.
/BU	File can be read and written; only one user can have a */BU* file open at any time. If another user attempts to read or write a */BU* file that is already in use, an error message 32 *FILE BEING MODIFIED* will be returned.
/BU/SHARE	File can be read and written; multiple users can simultaneously have the file open. A user cannot access a file in */BU/SHARE* mode if another user has already opened the file in */BU* mode.

Note that in the case of */BS* and */BS**, on output, if you specify a word position that is less than the last word output, you erase the entire file and create a new file with your output written at that word position.

The ⎕*ENQ* and ⎕*DEQ* system functions, described in detail in Section 7.8, synchronize the sharing of binary-access files.

7.6.6 Binary-Access I/O

The following format performs a read from a binary-access file opened as */BU*, */BU/SHARE*, */BS*, or */BS**:

⍞ ⟦[word]⟧ channel ⟦,header ⟦,type ⟦,length⟧ ⟧ ⟧

where

⍞ is the input-quad function. The value of ⍞ is the data read.

[word] is an integer specifying the word position at which reading is to begin. The default is 1 (first word). The file pointer is at either the beginning of the file, or at the next word after the previous I/O operation.

channel is the channel number assigned to the file.

header is either the value 0 for no header, or 1 if a header is available. The default is 1. If you specify header, you must specify type and length. If header is 0, APL takes the correct type and length from the data.

type is an integer from 1 to 6, specifying the type of data being read. See Table 7-4 for data type values. If a header exists, you need not specify type.

length is an integer indicating the number of values to be read; not the number of words. If a header exists, you need not specify the length.

Table 7-4
Data Types

Type	Data	Value Size
1	integer	1 per word
2	Boolean	36 per word
3	single-precision floating-point	1 per word
4	double-precision floating-point	1 per 2 words
5	APL 9-bit	4 per word
6	ASCII 7-bit	5 per word

The following format writes a record to a binary-access file.

data ⍈ ⟦[word]⟧ channel ⟦,header ⟦,type ⟦,length⟧ ⟧ ⟧

where

data is the information you want to write to the file.

⍈ is the output-quad function. The value of ⍈ is data.

[word] is an integer specifying where you want the write to begin. The default is 1 if the file pointer is at the beginning of the file or the next word after the previous operation.

channel is the channel number associated with the file.

header is either 0 if the header does not exist, or 1 if the header is available. The default is 1.

type is an integer from 1 to 6 specifying data type. See Table 7-4. If you do not specify type, APL attempts to write the data in the proper type.

length is an integer indicating the number of values to be written. When you specify a header, the length specification is ignored.

The following format writes a record to the end of a binary-access file opened as */BU* or */BU/SHARE*:

data ⎕*APPEND* channel ⟦,header ⟦,type ⟦,length⟧ ⟧ ⟧

where

data is the information you want to append to the end-of-file.

channel is the channel number associated with the file.

header is either 0 or 1.

type is one of the data types listed in Table 7-4.

length is an integer indicating the number of values to be written. The value of ⎕*APPEND* is data.

⎕*APPEND* writes to the physical end-of-file. The default word number for the next read or write to the file is now the end-of-file. In other words, the sequence ⎕*APPEND*, then ⍈ with no word specification always gives an end-of-file, while the sequence ⎕*APPEND* then ⍈ with no word specification appends a second record to the end-of-file.

To write to the end of a binary sequential file, use the */BS** switch when assigning a channel number. Then, the first write operation opens the file at the end to prevent overwriting.

When you write a record with a header, APL sets up the record with the header information first. The header size is two words plus one word for every dimension in your data. The data type and length make up the first word of the header (one half-word each). The rank of the data is in the second word, the number of rows in the third word, the number of columns in the fourth word, and so on.

Figure 7-1 illustrates the format of the header.

Bit	0 — 17	18 — 35
Word 1	type (1-6)	length (in words) of value plus header
Word 2	Rank (Number of dimensions in data)	
Word 3	rho 1	
Word 4	rho 2	
	⋮	

Figure 7-1 Format of Header in a Binary-Access File

If you do not specify a header when performing I/O, you must specify the type and length of the data for input and output.

When reading a record with a header, you have the option of actually accessing the header information itself or going right to the data. If you specify no header (0) on a read of a record with a header, you will access the header information when requesting word 1, 2 and so on. APL views the header as data in this case. If you specify header, APL goes right to the values you input instead. Because APL creates a header as the default, if you read the end-of-file without specifying 0 in the header position, you will receive a 15 *DOMAIN ERROR*.

The following example illustrates the writing then reading of a character matrix whose shape is 3 by 2. A header is specified along with ASCII data type (6) and length (number of values (6)). Figure 7-2 illustrates the format of the header and the data.

```
      ⎕ASS '1 FILE/BU'
1
      (3 2⍴'ABCDEF')⌹1,1,5,6
AB
CD
EF
      ⌹[1]1,0,1,1
1310726
      ⌹[2]1,0,1,1
2
      ⌹[3]1,0,1,1
3
      ⌹[4]1,0,1,1
2
      ⌹[5]1,0,5,6
ABCDEF

      ⎕CLS 1
```

The data is stored as follows:

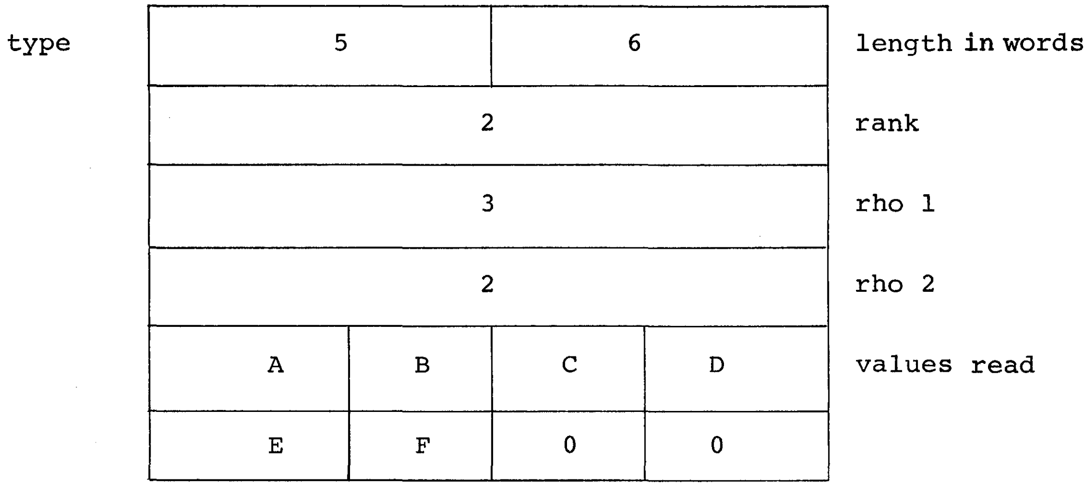


Figure 7-2 Record Format of a Binary-Access File

Note that the final value word is right-filled with zeroes.

In the previous example, the first read produced an integer value for the entire word 1310726. To break this down into type and length, use the encode function as in the following example:

```
        (0,2*18)⊤1310726
  5 6
```

The following examples illustrate the use of the file input and output functions for binary-access files:

```
        ⎕ASS '2 BIN/BU'
  2
        ⍝WRITES REQUEST;H=0,NO HEADER,T=1,INTEGER
        (⍳5)█2,0,1
  1 2 3 4 5
        ⍝WRITES REQUEST;SEQUENTIAL
        (5+⍳8)█2,0,1
  6 7 8 9 10 11 12 13
        ⍝READ BEGINNING AT 8TH WORD;4 VALUES READ
        █[8]2,0,1,4
  8 9 10 11
        ⍝READ;SEQUENTIALLY-NEXT WORD;2 VALUES
        █2,0,1,2
  12 13
        ⍝WRITE BEGINNING AT 13TH WORD;ASCII DATA
        'TOPS-10 APLSF'█[13]2,0,6
  TOPS-10 APLSF
        ⍝READ 1 WORD BEGINNING AT 13TH WORD
        █[13]2,0,6,5
  TOPS-
        ⍝READ 2 WORDS(10 VALUES)(AT NEXT WORD)
        █2,0,6,10
  10 APLSF
        ⍝WRITE BEGINNING AT 50TH WORD
```

```
         ⍝HEADER EXISTS
         (3 2⍴20+⍳6)▮[50]2
   21   22
   23   24
   25   26
         ⍝READ 10 VALUES BEGINNING AT 50(NO HEADER)
         ▮[50]2,0,1,10
262154 2 3 2 21 22 23 24 25 26
         ⍝TRANSLATE 1ST WORD INTO DECIMAL
         (0,2*18)⊤262154
1 10
         ⍝READ BEGINNING AT 50
         ▮[50]2
   21   22
   23   24
   25   26
         ⍝NULL VECTOR INDICATES END-OF-FILE
         A←▮2,0,1,1
         ⍴A
0 75
```

If you open a binary-access file with */BS*, you can access the file only sequentially. Because you can also specify a word number when writing to the file, you could destroy the file by specifying a word number that is less than the last word number written. For example, if the last output was written in word 40, and you specify word 15, the data will be written in word 15 but all data before and after that word is erased.

You can, however, write to a word number greater than the last word number in the file (41 in this case). APL considers this preserving the sequence. Note the following example:

```
         CHAN←⎕ASS'BIN/BS'
         (⍳10)▮CHAN
1 2 3 4 5 6 7 8 9 10
         (10+⍳10)▮CHAN
11 12 13 14 15 16 17 18 19 20
         (40+⍳10)▮[50]CHAN
41 42 43 44 45 46 47 48 49 50
         (100+⍳10)▮[3]CHAN
101 102 103 104 105 106 107 108 109 110
         ⎕CLS CHAN

         CHAN←⎕ASS'BIN/BS'
         ▮[50]CHAN
  15 DOMAIN ERROR
          ▮[50]CHAN
          ^
         ▮[3]CHAN
101 102 103 104 105 106 107 108 109 110
         ▮[2]CHAN
  15 DOMAIN ERROR
          ▮[2]CHAN
          ^
```

As in direct-access files, to delete a record in binary-access, specify a null vector in its place (0 75)⍴0. Also, an end-of-file is indicated by the null vector (0 75),⍴0 and the error message 75.

With binary-access files, you can also input and output mixed data in one logical record. See Section 7.6.7 for the ⎕*CIQ* and ⎕*COQ* system functions.

7.6.7 ⎕*CIQ* and ⎕*COQ* - Accumulating Data

The ⎕*CIQ* and ⎕*COQ* system functions allow you to accumulate data of different types into a variable for storage as one logical record.

For output, use the following format:

data ⎕*COQ* ⟦header ⟦,type⟧ ⟧

where

data is the value of ⎕*COQ*.

⎕*COQ* is the system function that packs data for output.

header is either 0 or 1, with the same meaning as in Section 7.6.6.

type is a data type listed in Table 7-4. The value of ⎕*COQ* is the packed data.

The following format unpacks data:

variable ⎕*CIQ* ⟦header ⟦,type⟧ ⟧

where

variable is the name where the value being read is stored.

⎕*CIQ* is the system function that unpacks the data.

header is either 0 or 1.

type is a data type listed in Table 7-4. The value of ⎕*CIQ* is the packed data.

The ⎕*COQ* system function takes any data type and turns it into an integer vector. In this way you can assign ⎕*COQ* packed ASCII characters to a variable and assign floating-point or APL 9-bit to another variable and catenate them. Then you can write the record to a file with ⍈ using the integer data type.

For example:

```
      A←⍳5
      F←A ⎕COQ 1
      F
262152 1 5 1 2 3 4 5
      (0 2*18)⊤F[1]
1 8
```

To retrieve it and translate the data back, first read the data with ⍈, then use ⎕*CIQ* to unpack the data types.

For example:

```
          P ⎕CIQ 1
    1 2 3 4 5


          ∇ΔPACK[⎕]∇
        ∇  P←ΔPACK LIST;I
    [1]   ⍝ ΔPACK USES ⎕COQ TO PACK A SET OF VALUES INTO A SINGLE
    [2]   ⍝     VARIABLE.  THE VALUES CAN BE OF DIFFERENT TYPES.  USE
    [3]   ⍝     ΔPACK WHEN CATENATE WON'T WORK
    [4]   ⍝ LIST IS A CHARACTER MATRIX, EACH ROW OF WHICH CONTAINS THE
    [5]   ⍝      NAME OF A VARIABLE WHOSE VALUE IS TO BE PACKED
    [6]   ⍝ P IS THE RESULTANT PACKED VALUE — IT IS AN INTEGER ARRAY
    [7]    P←⍳0
    [8]    I←1
    [9]   TEST:→(I>1↑⍴LIST)/0
    [10]   P←P,(⍎LIST[I;])⎕COQ 1
    [11]   I←I+1
    [12]   →TEST
        ∇
          ∇ΔUNPACK[⎕]∇
        ∇  P ΔUNPACK LIST;DATA;I;J;LEN;ENTRY
    [1]   ⍝ ΔUNPACK USES ⎕CIQ TO UNPACK A VARIABLE CREATED BY ΔPACK
    [2]   ⍝      INTO A SET OF VARIABLE NAMES, USING ASSIGNMENT
    [3]   ⍝ P IS THE PACKED VALUE, CREATED BY ΔPACK
    [4]   ⍝ LIST IS A CHARACTER MATRIX, EACH ROW OF WHICH CONTAINS THE
    [5]   ⍝      NAME OF A VARIABLE TO RECEIVED ONE OF THE PACKED VALUES.
    [6]   ⍝      THE VALUES UNPACKED OUT OF P ARE STORED INTO SUCCESSIVE
    [7]   ⍝      VARIABLES NAMED IN LIST
    [8]    DATA←P
    [9]    I←⍴DATA
    [10]   J←1
    [11]  TEST:→(I≤0)/0
    [12]   →(J>1↑⍴LIST)/0
    [13]   LEN←1↓(0 2 *18)⊤DATA[1]
    [14]   ENTRY←DATA[⍳LEN]
    [15]   ⍎LIST[J;],'←ENTRY ⎕CIQ 1'
    [16]   DATA←LEN↓DATA
    [17]   J←J+1
    [18]   I←I−LEN
    [19]   →TEST
        ∇



          ⍝DEFINE SOME NUMERIC VARIABLES
          A←1
          AA←1 1
          AAA←1 1 1

          ⍝DEFINE SOME CHARACTER VARIABLES
          B←'B'
          BB←'BB'
```

```
      ⍝MAKE A LIST OF INPUT VARIABLE NAMES
      L1←5 3⍴'A  AA AAAB  BB '
      L1
A
AA
AAA
B
BB
      ⍝PACK THESE VARIABLES INTO ONE ITEM
      ⍝CATENATE WILL NOT WORK

      P←ΔPACK L1
      P
524291 0 ¯34359738368 524292 1 2 ¯17179869184 524292 1
      3 ¯8589934592 1310723 0 ¯21206401024 1310724 1 2
      ¯21113602048

      ⍝MAKE A LIST OF OUTPUT VARIABLE NAMES
      L2←5 3⍴'X  XX XXXY  YY '
      L2
X
XX
XXX
Y
YY
      ⍝UNPACK THE PREVIOUS DATA INTO THESE NEW VARIABLES

      P ΔUNPACK L2

      ⍝THE RESTORED DATA IS THE SAME AS THE DATA
      ⍝THAT WAS PACKED

      X = A
1
      XX = AA
1 1
      XXX = AAA
1 1 1
      Y = B
1
      YY = BB
1 1
```

7.6.8 Binary-Access Magnetic Tape Files

Binary-access mode is often used to read magnetic tape files that have been created on other systems. The following sections, Sections 7.6.8.1, and 7.6.8.2, describe the use of the */BS/DUMP* switches and the ⎕*MTP* system function for magnetic tape.

7.6.8.1 *<u>/BS/DUMP</u>* <u>Switches</u> - If you plan to read or write a binary-access file on magnetic tape, you should specify the */BS* switch when assigning the file to a channel (⎕*ASS*). Output is then written to a binary-access magnetic tape file in fixed-length blocks. The size of these blocks is determined by the operating system; the default size is 128 words. You can override the setting with the SET BLOCKSIZE or SET TAPE RECORD-LENGTH monitor commands.[1]

If your file consists of variable-length blocks, you should also include the */DUMP* switch. When you specify */BS/DUMP*, each read or write request reads or writes one magnetic tape block.

When you perform a read, the size of each input request must be at least as large as the block to be read or you will receive the message 74 *BLOCK TOO BIG*. If the length specification in the input request is larger than the input block, the value read will reflect the actual block size. You should specify length with ⊟ because APL cannot know the length of the data before reading the tape.

NOTE

> The primary use for */BS/DUMP* is to read magnetic tapes generated on other systems. It is not recommended for general use.

7.6.8.2 <u>⎕*MTP* - Operating on Magnetic Tape</u> - The ⎕*MTP* system function performs a variety of magnetic tape operations including rewinding, setting density, and returning data modes.

Format

channel ⎕*MTP* operation(s)

where

channel is the channel number associated with the file.

⎕*MTP* is the system function for magnetic tape operations.

operation(s) is a vector containing one or more codes indicating the particular magnetic tape operations to be performed. See Table 7-5.

Some of the operations read characteristics of the tape. For each read operation the result of that operation is placed into a result vector. In other words, if there are 3 read operations in the operations array, the result array will be 3 elements long. If there are no read operations, the result array will be null.

There are four basic types of magnetic tape operations.

1. Tape positioning operations - These operations do such things as rewind the tape and move it forward and backward. They have codes in the range 0-511.

2. Read operations - These operations read characteristics of the tape such as density and track status. The results of

[1] See the <u>TOPS-20 Monitor Calls Manual</u> (version 3A or later) or the <u>TOPS-10 Monitor Calls Manual</u> (version 6.03A or later).

these read operations are placed in the result array, as explained above. They have codes in the range 512-1023.

3. Write operations - These operations set characteristics of the tape. The value which is written is taken from the next element of the operations array. For example:

```
    12 ⎕MTF 1025
```

   This example sets the density of the tape to 1600 bpi, the 4 is the value written as the tape's density. These operations have codes in the range 1024-1535.

4. Reserved for user definition - These must be defined by the user. Since APL does not know whether these are read or write operations it assumes they are both. In other words, it passes the next element of the operations array to the operation and puts the result of the operation in the result array. These operations have codes in the range 1536-2047.

On TOPS-10 the operation codes and results returned are identical to those for the TAPOP UUO. (See the TOPS-10 Monitor Calls Manual.) On TOPS-20 the operations listed in Table 7-5 are available.

Table 7-5
Operation Codes

1 -	Wait for I/O to stop	
2 -	Rewind the tape to the load point	
3 -	Rewind and unload the tape	
4 -	Skip forward one block	
5 -	Skip forward one file	
6 -	Skip to the logical end of the tape	
7 -	Skip backward one block	
8 -	Skip backward one file	
9 -	Write a tape mark	
513 -	Read density, possible values are	
	0 -	unit default bits/inch
	1 -	200 bits/inch
	2 -	556 bits/inch
	3 -	800 bits/inch
	4 -	1600 bits/inch
	5 -	6250 bits/inch
	6-17 -	reserved for DIGITAL
519 -	Read data mode, possible values are	
	0 -	DEC-compatible core dump format (7-track and 9-track)
	1 -	DEC-compatible dump format (9-track)
	2 -	Industry-compatible, 8-bit mode (4 bytes/word)
	3 -	6-bit mode (9-track, TU70 only)
	4 -	7-bit mode (TU70 only)
	5 -	DEC-compatible 7-track core dump (SIXBIT)
520 -	Read track status, 1=7-track, 0=9-track	
521 -	Read write-lock bit; returns 1 if set, 0 if not set	
1025 -	Set density (same values as read)	
1031 -	Set data mode, same values as read	

On success, ⎕*MTP* returns a 1-dimensional array with one element for each read or user-defined operation. If any of the operations returns an error, ⎕*MTP* returns an integer indicating the cause of the error. For TOPS-10 the errors are the same as for the TAPOP UUO. For TOPS-20 the following errors can be returned:

-2	Unknown error
-1	Address check while storing answer
0	Illegal function code specified
1	Function code requires privileges
3	Value is not in legal range
4	Address check while reading arguments, or too few arguments
6	Tape has not been initialized
8	Termination code error
9	Job number associated with unit is incorrect

In addition, for both TOPS-10 and TOPS-20, the following errors can be generated:

12 *RANGE ERROR* -	CHANNEL < 1 or CHANNEL > 12
64 *CHANNEL NOT ASSIGNED* -	CHANNEL not assigned
9 *RANK ERROR* -	Rank of argument array NEQ 1 and length of argument array NEQ 1
15 *DOMAIN ERROR* -	Operation code LSS 0
10 *LENGTH ERROR* -	Write operation specified but there is no argument to write.
62 *NOT A PROPER DEVICE* -	Device is not a magtape

NOTE

When you specify ⎕*MTP*, APL first writes out any in-core buffers before performing the ⎕*MTP* operations. You should therefore issue this function with great caution, and you should use it only between magnetic tape files.

7.7 UTILITY SYSTEM FUNCTIONS

The following sections describe four utility system functions that return file organization, device characteristics, record size, and other information about files that are in the system.

7.7.1 ⎕*CHS* - Returning File Organization and Status

Format

⎕*CHS* channel(s)

where

⎕*CHS* is the system function.

channel(s) is one or more channel numbers associated with the files.

The ⎕*CHS* system function returns the file organization and the open status of the files on one or more channels in the system. The channel argument can be a numeric scalar, vector, or a null vector.

For example:

```
      ⎕CHS 8
4 1
```

This expression returns information about the file associated with channel number 8. If the channel number is a vector, APL returns information on all channels specified. One row containing two values is returned for each channel specified. The first value identifies the file organization, and the second value identifies the open status of the file.

Tables 7-6 and 7-7 list the meanings of the values.

Table 7-6
File Organization

Code	Organization
0	channel free
1	*/AS*
2	*/IS*
3	*/DI*
4	*/DA*
5	*/BS*
6	*/BU*

Table 7-7
Open Status

Code	Status
0	channel free
1	file assigned but not open
2	file open for output
3	file open for input
4	file open for input and output

If you specify a single channel number, the result of the function is a 2-element vector. If the function contains *N* arguments, the result is an array of shape *N* by 2. For example, the following function results in a 3-by-2 array:

```
          ⎕←FILS←⎕CHS ⍳3
    4  1
    5  1
    0  0

          ⍴FILS
 3 2
```

If the argument to ⎕*CHS* is a null vector:

```
          ⎕CHS ⍳0
    4  1
    5  1
    0  0
    0  0
    6  1
    0  0
    4  1
    4  1
    0  0
    0  0
    0  0
    1  1
```

APL returns information on all channels in the APL system. This specification is the same as ⎕*CHS* ⍳12.

7.7.2 ⎕*DVC* - Returning Device Characteristics

Format

⎕*DVC* channel(s)

where

⎕*DVC* is the system function.

channel(s) is one or more channel numbers associated with files. This argument can be a numeric scalar, vector, or a null vector.

The ⎕*DVC* system function returns a device-characteristics word and block size for the files on one or more channels in the system. The device-characteristics word returned by ⎕*DVC* has the same format and meaning as that interpreted by the DEVCHR UUO[1].

The syntax is identical to that of ⎕*CHS*: one row containing two values is returned for each channel specified. The first value is the device-characteristics word, and the second value represents the block size for the device, in words. ⎕*DVC* returns a 2-element vector if a single channel is specified. If the function contains *N* arguments, the result is an array of shape *N* by 2. If the argument is a null vector, APL returns information on all channels, and the result is an array of shape 12 by 2.

It is usually helpful to convert the device-characteristics word to binary format before examining it. The following example illustrates the conversion of the word returned in the example included at the beginning of this section with the APL encode function (⊤).

For example:

```
      ⎕CHS 1
4 1
      A←⎕DVC 1
      A[1]
17324376063
      '01'[1+(36⍴2)⊤A[1]]
010000001000100111001111111111111111
 │ │        │ │      │ │
 │ └ LPT:   │ └ MTA: │ └ Device can do output
 └── DSK:   └── TTY: └── Device can do input
```

[1]See the TOPS-20 Monitor Calls Manual (version 3A or later) or the TOPS-10 Monitor Calls Manual (version 6.03A or later).

7.7.3 ⎕*FLS* - Returning File Sharing Information

Format

⎕*FLS* channel(s)

where

⎕*FLS* is the system function.

channel(s) is one or more channel numbers associated with files. The argument can be a numeric scalar, vector, or null vector.

The ⎕*FLS* system function returns the sharing status and other information about files on one or more channels in the system. The syntax of ⎕*FLS* is similar to that of ⎕*CHS* and ⎕*DVC* except that one row containing four values is returned for each channel specified. The meaning of the values differs according to the file organization. ⎕*FLS* is meaningful only for direct-access and binary-access files. If the channel number you specify is associated with an ASCII sequential or internal sequential file, the values returned are all zeros.

The values returned for direct-access files (/*DA* and /*DI*) have certain meanings depending on their position in the vector. Starting from left to right:

First value	Share bit: 0 means no sharing, 1 means sharing
Second value	Value of the next record number to be used for reading or writing (if subscript record in the file I/O request is defaulted)
Third value	Maximum record number permitted
Fourth value	Blocking factor

Starting from left to right, the values for binary-access files have the following meaning:

First value	Share bit: 0 means no sharing; 1 means sharing
Second value	File-word pointer to the next word to be read or written (if subscript word in the file I/O request is defaulted)
Third value	Length of file in words (cannot be determined for magnetic tape device)
Fourth value	Size of last read or write request in words (Not for magnetic tapes)

For example:

```
          ⎕FLS 1
    0 20 2052 50
```

There is another important difference between ⎕*FLS* and ⎕*CHS*. The file on the channel must be open to return values. In the case of a */BS* file in which only reading or writing can be in effect at any one time, there must be a way of specifying which type of access you intend to perform on the next operation if the file has not already been accessed. APL allows you to include a special channel number specification for */BS* files that have not already been accessed. A channel number in the range 1 through 12 indicates that you will be reading the file. A channel number in the range 101 through 112 indicates that you will be writing the file.

⎕*FLS* returns a 4-element vector if a single channel number is specified. If the function contains *N* channels, the result is an array of shape *N* by 4. If you specify a null vector, APL returns information on all channels, and the result is an array of shape 12 by 4.

7.7.4 ⎕*FCM* - Returning File Information

Format

⎕*FCM* channel

where

⎕*FCM* is the system function.

channel is the channel number associated with the file.

The ⎕*FCM* system function returns information about the records in a direct-access file. Unlike ⎕*CHS* and the other utility function, the ⎕*FCM* function requires that the channel number be an integer scalar, not a vector. ⎕*FCM* is meaningful only for direct-access files (*/DA* and */DI* files). If the channel number is associated with another file organization, the result will be a null array of shape 0 by 2.

For direct-access files, ⎕*FCM* returns one row containing two values, one row for every record in the file. The first value is the record number; the second value is the number of blocks in the record. If the file contains *N* records, the result is an array of shape *N* by 2.

For example:

```
          ⎕FCM 3
    1      1
    4     63
    56     7
    76     7
    98     1
    99     1
```

7.8 ⎕*ENQ* AND ⎕*DEQ* - SYNCHRONIZING SHARED FILE ACCESS

Format

```
     ⎕ENQ channel lock number share bit
     ⎕DEQ channel lock number share bit
```

The ⎕*ENQ* and ⎕*DEQ* system functions allow you to synchronize access to shared direct-access and binary-access files. These functions should be used only by advanced APL users familiar with the ENQ and DEQ monitor calls.[1] In addition, the issuing of ⎕*ENQ* and ⎕*DEQ* functions should be restricted to a cooperating group of users who require shared access to common files and who have decided upon a mutual protocol for synchronizing file usage.

Synchronization of file access proceeds as follows: A user assigns a file and specifies that it is to be available for shared access (*/SHARE*). However, to protect the file while a read or write is being performed, some method is needed for temporarily locking the file from access by other users. ⎕*ENQ* performs this lock function. The file remains under the control of the user who issued ⎕*ENQ* until that user releases it for continued shared access with the ⎕*DEQ* function. The arguments to both functions specify the degree to which the file is locked against other users.

The first two arguments, channel number and lock number, are considered a lock pair. The first element identifies the channel number on which the file to be shared is assigned. The second element represents the particular type of lock being performed. Depending on the conventions adopted by file users, this number might be a file record number, a range of record numbers, or some other representation. The lock number must be in the range 0 through 2*33-1. It has no inherent meaning to APL; its only significance is that agreed upon by cooperating users. The share bit is discussed later in Section 7.8.3.

The channel number is not particularly significant. The file to be shared must be associated with the specified channel, but several users can have the same file assigned to several different channels. This is illustrated in the next example. The chronological order in which operations are performed is significant and proceeds from top to bottom.

```
         USER[4,204]         USER[4,205]

      ⎕ASS '12  FRT/BU/SHARE'          ⎕ASS '11  FRT[55,12]/BU/SHARE
12                                11
      ⎕ENQ 12 99
0
      FILE OPERATIONS ON 12            ⎕ENQ 11 99
               .                       USER ENTERS A WAIT STATE
               .                                   .
               .                                   .
      ⎕DEQ 12 99                                   .
0                                 0
                                       FILE OPERATIONS ON 11
                                                   .
                                                   .
                                                   .
                                       ⎕DEQ 11 99
                                  0
```

[1]See the <u>TOPS-20 Monitor Calls Manual</u> (version 3A or later) or the <u>TOPS-10 Monitor Calls Manual</u> (version 6.03A or later).

You issue *⎕ENQ* and *⎕DEQ* on lock numbers as well as on channel numbers. An *⎕ENQ* succeeds if another user is not already enqueued on the specified lock. If the lock is currently enqueued, the second user enters the wait state until the first user issues a *⎕DEQ* to free the locked file. When you close a file, all locks on the channel associated with the file are dequeued.

A successful *⎕ENQ* or *⎕DEQ* returns a 0 as the function result. If an error is encountered, a scalar value representing the error condition is returned.[1] A return code of ¯1 means that the system does not support the use of *⎕ENQ* and *⎕DEQ*.

Only the channel number is required in an *⎕ENQ* or *⎕DEQ* specification. The function:

```
      ⎕ENQ 6
0
```

is equal to:

```
      ⎕ENQ 6 0
0
```

You can specify a matrix as an argument.

For example:

```
      ⎕ENQ M
0
```

M is a matrix in which each row specifies a lock pair. APL enqueues on all locks specified in this matrix. All locks must be successful for an *⎕ENQ* to succeed.

If you include only a channel number with *⎕DEQ*, APL clears all locks on the file. If you include a null vector as a channel number, APL clears all locks on all channels. For example, the *⎕DEQ* expression:

```
      ⎕DEQ 3 1⍴ 5 6 7
0
```

clears all locks on channels 5, 6, and 7.

[1]See the TOPS-10 Monitor Calls Manual (version 6.03A or later) for an explanation of value errors returned under TOPS-10.

7.8.1 File Locks

The lock pair:

> ⎕*ENQ* channel 0
> ⎕*DEQ* channel 0

has special significance in APL. This lock pair is considered to be the file lock and is used primarily by the file system itself to control access to shared files. APL automatically issues internal ⎕*ENQ* and ⎕*DEQ* functions on the file lock for the following organizations:

> */DA* and */DA/SHARE*
> */DI*
> */BU/SHARE*

These functions are performed to ensure efficient access to the in-core disk buffers maintained by APL.

When you specify a */DA* file without sharing, APL issues an ⎕*ENQ* on the file lock. When the file is closed, an automatic ⎕*DEQ* is performed. If */DA/SHARE*, */DI*, or */BU/SHARE* is specified, APL issues an ⎕*ENQ* on the file lock when the file access is first issued. It issues an automatic ⎕*DEQ* on the file lock whenever a terminal input request is expected. Before performing the ⎕*DEQ* function, APL writes out all output buffers and clears all input buffers.

You can explicitly issue an ⎕*ENQ* function on the file lock. If you do this, APL will not issue an automatic ⎕*DEQ* until you issue a corresponding explicit ⎕*DEQ* on the file lock or until the file is closed. You can issue an explicit ⎕*DEQ* function on the file lock at any time, thus causing APL to clear all in-core buffers for the corresponding file. For certain applications, it is more efficient to issue explicit ⎕*ENQ*'s and ⎕*DEQ*'s.

7.8.2 Determining Lock Numbers

The group of APL users who will be sharing files and issuing ⎕*ENQ* and ⎕*DEQ* functions for these files should agree upon the significance of the lock-number arguments included in the function lock pairs. As previously mentioned, this argument can be any number in the range 0 through 2*33-1, where 0 has a special meaning. The only meaning associated with a particular lock number is that agreed upon by the group of cooperating file users. The following function provides an example of users cooperating in sharing a file. This function can be executed by several users simultaneously. It controls access to the file by locking individual records of the file while they are being accessed by file users. A lock-number specification here refers to a file record number.

```
              ∇F[⎕]∇
          ∇   NCMF←CMFNUM F CHAN;CMF
    [1]       →(0≠⎕ENQ CHAN,CMFNUM)/ENQFAIL
    [2]       CMF←⍞[CMFNUM]CHAN
    [3]       →(0≠⎕DEQ CHAN,0)/DEQFAIL
    [4]       NCMF←PROCESS CMF
    [5]       NCMF←NCMF⍞[CMFNUM]CHAN
    [6]       →(0≠⎕DEQ CHAN)/DEQFAIL
    [7]       →0
```

```
[8]    ENQFAIL:'ENQ FAILED' ⎕SIGNAL 501
[9]    DEQFAIL:'DEQ FAILED' ⎕SIGNAL 502
     ∇

       ⎕ F ⎕ASS'PAYROLL,FEB/DA/SHARE'
⎕:
       12
```

7.8.3 Specifying a Share Bit

The third argument in the ⎕*ENQ* and ⎕*DEQ* system function, the share bit, can be specified if you are willing to share access to a lock. If the share bit is set to 1, sharing of a lock is established; if it is set to 0, you have exclusive use of the lock. The default share bit is 0. If several users specify a share bit 1 in an ⎕*ENQ* function, it is possible for all of the ⎕*ENQ*s to succeed at the same time. Only one user can have exclusive access to a lock at any one time.

The following example illustrates the interaction of four users accessing the same file. The chronology of the functions issued and executed is shown in the time component. Note that requests are queued in a first-in-first-out fashion.

```
Time       User 1           User 2          User 3          User 4


12:00...⎕ENQ 12 1 0

12:00...obtains
        exclusive
        access

12:02..................⎕ENQ 12 1 1

12:04..................waits..........⎕ENQ 12 1 1

12:06.................................waits...........⎕ENQ 12 1 0

12:08................................................waits

12:10

12:12...⎕DEQ 12 1.......obtains........obtains
                        shared access  shared access

12:14..................⎕DEQ 12 1

12:16...⎕ENQ 12 1 1...................⎕DEQ 12 1 0......obtains
        waits                                         exclusive
                                                      access

12:18...obtains.......................................⎕DEQ 12 1
        exclusive
        access
```

7.9 *)INPUT* AND *)OUTPUT* HANDLING I/O FROM NON-TERMINAL DEVICES

Format

```
    )INPUT⟦filespec⟧⟦/character set⟧
    )OUTPUT⟦filespec⟧⟦/character set⟧
```

The *)INPUT* and *)OUTPUT* system commands allow you to divert immediate mode and quad I/O to devices other than your terminal. A file specification can be included in the command to indicate the device and filename to be used for input or output. The file specification has the same format as a workspace name, and, as with a workspace name, you need not include all five parts of the file specification. See Section 2.1.4. When you omit parts of the name, the default values in Table 7-8 are assumed. If you omit the file specification, APL defaults to the terminal.

Table 7-8
File Specification Defaults

Component	Default
Device name	*DSK:*
Filename	Input for *)INPUT* Output for *)OUTPUT*
File extension or type	*.AAS*
File protection	Installation dependent
File owner ID	User's directory

You can specify an optional parameter (the character-set switch) to be used in handling the data being read or written. Legal values are */APL* for the APL character set and */TTY* for the TTY character set. The default is the character set of the user's terminal.

For TTY terminals, */APL* means use the APL character set specified when you first accessed APL. When APL prompts with *TERMINAL..*, you have the option of specifying a particular APL terminal, for example, TTY/4013. The default is LA36. See Table 1-1.

)INPUT and *)OUTPUT* are typically used to divert input and output requests to devices other than the terminal.

For example:

```
    )OUTPUT LPT:
```

This response intercepts all output normally directed to the terminal and routes it instead to the line printer. For example:

```
      )OUTPUT DSK:APL
```

This command causes all normal terminal input requests to come from a disk file named *APL.AAS*.

If you use an *)OUTPUT* command to divert output from the terminal, input is echoed to the output file as well, so that the output file has the appearance of a normal terminal sheet. This alleviates the potential confusion involved in trying to match up input and output requests. Special processing is also performed to help you synchronize input and output in the following two cases:

1. input from the terminal and output to another device

2. input from another device and output to the terminal

In the first case, APL displays the usual six spaces at the terminal to signal the completion of the last output request. In the second case, the names of the functions whose definitions appear in the input file are listed on the terminal upon successful closing of the function.

If errors occur in a function definition, the number of errors encountered is displayed along with the function name. If the APL system encounters an I/O error when *)INPUT* and *)OUTPUT* commands have diverted both input and output from the terminal, I/O in the direction of the error reverts to the terminal. For example, if an error occurs on input, subsequent input is directed to the terminal, but output continues to be sent to the output device.

APPENDIX A

ERROR MESSAGES

If an error is detected during the evaluation of an expression, APL (1) displays an appropriate error message from the list included below and (2) the line in which the error occurred. A null array with the shape 0 *ERROR NUMBER* is returned as the value of the expression that produced the error when executed with ∊. The following example of a null array indicates an end-of-file error condition:

```
      A+B
 11 VALUE ERROR
       A+B
          ^
      C←∊'A+B'
 11 ∊ VALUE ERROR
       A+B
          ^
      ⍴C
0 11
```

The meaning of error-number values is summarized below.

Error Number	Meaning/Explanation/Action
0	*IMPROPER LIBRARY REFERENCE* Attempt to *)SAVE* a disk area that is not your own and not a public library area.
1	*WS NOT FOUND* No workspace or file with the name found in the disk area specified.
2	*SYSTEM ERROR* Internal APL system error. Please report this error to your software specialist.
3	*WS FULL* The active workspace cannot retain all the information requested, nor can it expand further. Erase unneeded objects, issue a *)MAXCORE* command to enlarge the workspace, or do a *)SAVE*, *)CLEAR* and *)COPY* sequence on the needed information.
4	*NOT A VALID SV IDENTIFIER* Attempt to use a shared variable not supported by this APL implementation.

Error Number	Meaning/Explanation/Action
5	*DEFN ERROR* Improper function definition syntax (function name may have been defined elsewhere) or improper edit request syntax. Function may be locked.
6	*LABEL ERROR* Improper use of a colon or improper variable name.
7	*SYNTAX ERROR* Invalid syntax, such as two variables without an intervening operator, a function call with missing arguments, or an unmatched parenthesis.
8	*INDEX ERROR* Index value out of range, for example trying to reference the tenth item of a 9-element vector.
9	*RANK ERROR* Ranks of two operands are not conformable.
10	*LENGTH ERROR* Shapes of two operands are not conformable.
11	*VALUE ERROR* Value for the variable in question has not been previously specified, or a function with an explicit result did not return a value.
12	*RANGE ERROR* Value of result exceeds capacity of machine word.
13	*POSSIBLE SI DAMAGE* A function in the state indicator has been erased or edited.
14	*DEPTH ERROR* Too many right brackets or parentheses on a line.
15	*DOMAIN ERROR* Function not defined for given values of arguments.
16	*UNBALANCED DELIMITER* Execute string does not contain a closing quote or, function definition does not contain a closing 'del' character.
17	*EDIT ERROR* Improper line editing request.
18	*ATTENTION SIGNALED* Attention signal detected during function execution (not all attention signals produce this message). Attention is signaled on ASCII terminals by two CTRL/C characters and by the ATTN key on 2741-style terminals.
19	*DEVICE DOES NOT EXIST* Improper device specification.
20	*DEVICE NOT AVAILABLE* The desired device is already assigned to another job.

Error Number	Meaning/Explanation/Action
21	*INCORRECT COMMAND* A system command is incorrectly spelled.
22	*INCORRECT PARAMETER* Improper command syntax for a recognized system command.
23	*WS LOCKED* An improper password (or none at all) has been given to access a workspace with a)*LOAD*,)*COPY*, etc., command.
24	*NOT GROUPED, NAME IN USE* The group-name specified has been defined elsewhere. The objects in the group-member-list have not been grouped.
25	*EXECUTE ERROR*
31	*PROTECTION FAILURE* Attempt to)*LOAD* or)*SAVE* a read-protected workspace from disk area other than your own, or a directory is full.
32	*FILE BEING MODIFIED* Two users are trying to)*SAVE* the same workspace simultaneously, or a file is already in use (by another user) during direct-access file I/O.
33	*UNEXPECTED FILE ERROR*
35	*DIRECT I/O ERROR* An error has occurred during a directory read or write.
39	*NO SUCH DIRECTORY*
41	*NO ROOM ON THIS FILE STRUCTURE OR QUOTA EXCEEDED* File structure is full or disk allocation is exceeded. In the latter case, files must be deleted from the user's disk area before more files can be added.
42	*WRITE-LOCK ERROR* Device is physically write-protected (usually a magnetic tape). Write-enable the device.
43	*NOT ENOUGH TABLE SPACE IN MONITOR* The system has run out of space to perform certain functions for the user. See the systems programmer at your installation.
44	*PARTIAL ALLOCATION ONLY* Entire space request for a disk file allocation could not be fulfilled. The space that was available has been allocated.
45	*BLOCK NOT FREE ON ALLOCATED POSITION* A disk block that the monitor allocated to APL as free is not available. See the systems programmer at your installation.
46	*MESSAGE TOO LONG* The maximum message length for the *HI* message has been exceeded. Maximum length is 384 characters.

Error Number	Meaning/Explanation/Action
47	*LINE TOO LONG TO EDIT* Line editing is not permitted on multiple-line statements (such as statements that overflow to the following line or multiple-line quoted strings). It is sometimes possible to edit the line by changing the)*WIDTH* parameter, to set the whole statement on one single line.
48	*INPUT LINE TOO LONG*
49	*FILE CONTAINS A DAMAGED WS*
50	*ERROR IN GARBAGE COLLECTION* Internal APL system error. Please report this error to your software specialist. Workspace damage is probable.
51	*ERROR IN COPY* An error has occurred during a)*COPY* command. Please report this error to your software specialist. Workspace damage is probable.
52	*LINKAGE ERROR* Internal APL error. Workspace damage has been detected. Please report this error to your software specialist.
53	*NOT ENOUGH CORE AVAILABLE* Not enough memory is available for the task requested. This error results when the user is within the limit specified by the)*MAXCORE* command, but the system itself does not have enough memory to allow the workspace to expand. Revise your needs for memory, use virtual storage facilities, or try to run at a time when more memory is available.
54	*STACK OVERFLOW* Internal APL error. There is not enough room on the stack for APL operations to continue. Please report this error to your software specialist. Workspace damage is probable.
55	*LOGICAL NAME DSK DOES NOT REFER TO PHYSICAL DISK* APL has determined that the logical name DSK does not refer to a physical disk structure. Reassign the name DSK to a disk (necessary for direct access I/O).
56	*INCORRECT MODE FOR DEVICE* The I/O mode for the action requested is improper for the chosen device (e.g., trying to)*SAVE* to a terminal).
57	*FILE DOES NOT CONTAIN A WORKSPACE* Attempt to)*LOAD* or)*COPY* a file that does not contain an APL workspace.
58	*I/O ERROR* A data-transmission error was detected during input or output. This message is usually associated with a non-recoverable device error.
59	*FILE ALREADY EXISTS WITH GIVEN NAME* Attempt to)*SAVE* a workspace with the same filename as an existing file that is not a workspace. Either rename the existing file on disk or change the)*WSID* of the APL workspace.

Error Number	Meaning/Explanation/Action
60	*WS NOT SAVED, THIS WS IS* Attempt to)*SAVE* a workspace with the same filename as an existing workspace, without specifying the)*WSID* first. This error message is to prevent inadvertent overwriting of previously saved workspace.
61	*RENAME ERROR* An error has occurred during file deletion or protection alteration. This frequently occurs when a file or workspace is already protected and cannot be renamed.
62	*NOT A PROPER DEVICE* Improper device selection, for example trying a)*SAVE* to a device which supports dump-mode I/O but which is not a DSK.
64	*CHANNEL NOT ASSIGNED* The channel specified in a file operation has not been previously associated with a file via a ⎕*ASS* system function.
65	*CANNOT DO BOTH INPUT AND OUTPUT* Either input or output, but not a combination of both, is allowed to a sequential file. Close the file and reopen it to perform the desired operation.
66	*CANNOT INPUT FROM FILE* The user has tried to input from an output-only device, such as a line printer.
67	*CANNOT OUTPUT TO FILE* The user has attempted to output to an input-only device, such as a card reader.
68	*FILE LOCKED* An improper password has been given for a direct-access file.
69	*FILE FORMAT NOT DIRECT ACCESS* An attempt has been made to perform direct-access I/O to a non-direct access file.
70	*FILE FORMAT NOT INTERNAL SEQUENTIAL* An attempt has been made to perform internal sequential I/O to an non-internal sequential access file.
71	*IMPROPER MODE OR SOFTWARE CHECKSUM ERROR* A file operation is attempting to use a mode that is improper for the device specified in an ⎕*ASS* system function. Issue an ⎕*ASS* to a device that supports the necessary mode.
72	*DEVICE ERROR* Physical device error during file I/O. Report this error to your operations staff.
73	*DEVICE DATA ERROR* A checksum or parity error during file I/O has occurred. The file is possibly recorded incorrectly on the specified device.

Error Number	Meaning/Explanation/Action
74	*BLOCK TOO BIG* A data-transfer error has occurred during I/O. Specifically, the last user has attempted to read a block of data that is too large.
75	-------- End-of-file (EOF) detected (no message is printed; execution continues).
78	*END OF TAPE* End-of-reel on a magnetic tape (MTA) detected.
79	*SYSTEM FUNCTION ILLEGAL IN EXECUTE OR IMMEDIATE MODE*

APPENDIX B

SUMMARY

This appendix contains the following items in the form of tables:

Table B-1
Primitive Scalar Functions
(Section 3.2)

Monadic		Dyadic	
Function	Meaning	Function	Meaning
+*Y*	*Y*	*X*+*Y*	Add *X* to *Y*
¯*Y*	Negative of *Y*	*X*-*Y*	Subtract *Y* from *X*
×*Y*	Sign of *Y*[1]	*X*×*Y*	Multiply *X* and *Y*
÷*Y*	Reciprocal of *Y*	*X*÷*Y*	Divide *X* by *Y*
⋆*Y*	*E* to the *Y*th power	*X*⋆*Y*	*X* to the *Y*th power
|*Y*	Magnitude of *Y*	*X*|*Y*	*X* residue of *Y*
⌈*Y*	Ceiling of *Y*	*X*⌈*Y*	Maximum of *X* and *Y*
⌊*Y*	Floor of *Y*	*X*⌊*Y*	Minimum of *X* and *Y*
⍟*Y*	Natural logarithm of *Y*	*X*⍟*Y*	Log of *Y* to the base *X*
!*Y*	Factorial of *Y*	*X*!*Y*	Binomial coefficient (number of combinations of *Y* things taken *X* at a time)
?*Y*	A random integer of ⍳*Y*	*X*?*Y*	*X* number of random integers in the range 1 through *Y*
○*Y*	Pi times *Y*	*X*○*Y*	Trigonometric operators (*Y* is in radians. See Table B-2.)

[1]Definition: ×*Y* is -1 if *Y*<0
×*Y* is 0 if *Y*=0
×*Y* is 1 if *Y*>0

Table B-2
The Dyadic Circle Function

Expression	Result	Expression	Result
0○*X*	(1-*X*⋆2)⋆.5		
1○*X*	sine *X*	-1○*X*	arcsin *X*
2○*X*	cosine *X*	-2○*X*	arccos *X*
3○*X*	tangent *X*	-3○*X*	arctan *X*
4○*X*	(1+*X*⋆2)⋆.5	-4○*X*	(-1+*X*⋆2)⋆.5
5○*X*	sinh *X*	-5○*X*	arcsinh *X*
6○*X*	cosh *X*	-6○*X*	arccosh *X*
7○*X*	tanh *X*	-7○*X*	arctanh *X*

The functions in Table B-3 return 1 if the relationship is true, and 0 if it is false.

Table B-3
Relational and Logical Functions
(Sections 3.2.1 and 3.2.2)

Function	Meaning
X<*Y*	*X* less than *Y*
X≤*Y*	*X* less than or equal to *Y*
X=*Y*	*X* equal to *Y*
X≥*Y*	*X* greater than or equal to *Y*
X>*Y*	*X* greater than *Y*
X≠*Y*	*X* not equal to *Y*
X∧*Y*	*X* and *Y*
X∨*Y*	*X* or *Y*
X⍲*Y*	*X* nand *Y* (not both *X* and *Y*)
X⍱*Y*	neither *X* nor *Y*
~*Y*	not *Y*

Table B-4
Primitive Mixed Functions

Function	Section	Meaning
X,*Y*	3.3.1	Catenate *X* to *Y* along the last dimension of *X*
X/*Y*	3.3.2	*X* (logical) compression along the last dimension of *Y*
X/[*N*]*Y*	3.3.2	*X* (logical) compression along the *N*th dimension of *Y*
X⌿*Y*	3.3.2	*X* (logical) compression along the first dimension of *Y*
X?*Y*	3.3.3	Deal *X* integers selected randomly in range 1 through *Y* without duplication
X⊥*Y*	3.3.4	Decode the representation of *Y* in number system *X*
X↓*Y*	3.3.5	For *X*>0, drop first *X* elements of *Y* - for *X*<0, drop last |*X* elements of *Y*
X⊤*Y*	3.3.6	Encode *Y* in number system *X*
X\Y	3.3.7	*X* (logical) expansion along the last dimension *Y*
X\\[*N*]*Y*	3.3.7	*X* (logical) expansion along the *N*th dimension of *Y*
X⍀*Y*	3.3.7	*X* (logical) expansion along the first dimension of *Y*
X⍒*Y*	3.3.8	Generate an index vector such that *X*[⍒*Y*] is in descending order
X⍋*Y*	3.3.9	Generate an index vector such that *X*[⍋*Y*] is in ascending order

Table B-4 (Cont.)
Primitive Mixed Functions

Function	Section	Meaning
⍳*Y*	3.3.10	Generate the first *Y* consecutive integers from current origin
X⍳*Y*	3.3.11	Find the first occurrence of *Y* in vector *X*
X,[*N*]*Y*	3.3.12	Laminate *X* to *Y* along the *N*th dimension of *X*
X∊*Y*	3.3.13	Determine the membership of *X* in array *Y*
,*Y*	3.3.14	Return the ravel of *Y* (make *Y* a vector)
⌽*Y*	3.3.16	Reverse along the last dimension of *Y*
⌽[*N*]*Y*	3.3.16	Reverse along the *N*th dimension of *Y*
⊖*Y*	3.3.16	Reverse along the first dimension of *Y*
⍴*X*	3.3.18	Return the shape of *X*
X⍴*Y*	3.3.15	Reshape *Y* to make dimension *X*
X⌽*Y*	3.3.17	Rotate by *X* along the last dimension of *Y*
X⌽[*N*]*Y*	3.3.17	Rotate by *X* along the *N*th dimension of *Y*
X⊖*Y*	3.3.17	Rotate by *X* along the first dimension of *Y*
X↑*Y*	3.3.19	For *X*>0, take first *X* elements of *Y* - for *X*<0, take last |*X* elements of *Y*
⍉*Y*	3.3.20	Transpose the dimensions of *Y* (for a matrix, exchange the rows and columns)
X⍉*Y*	3.3.21	Transpose array *Y* according to *X*

Table B-5
Extended Functions

Function	Section	Meaning
⌹*Y*	3.4.1	Invert the matrix *Y*
X⌹*Y*	3.4.2	Perform matrix division, solve linear equations, find a least squares solution
∊*Y*	3.4.3	Execute the character string *Y*
⊥*Y*	3.4.3	Execute the character string *Y*
⍎*Y*	3.4.4	Execute the character string *Y*
X$*Y*	3.4.5	Format data (see below)
⍕*Y*	3.4.6	Format character array *Y*
X⍕*Y*	3.4.7	Format character array *Y* with width and precision specified by *X*
⊤*Y*	3.4.8	For numeric *Y*, convert to a character string - for character string *Y*, if *Y* is not a variable or function name, return the null vector; if *Y* is a variable to a character string; if *Y* is a function to a character string consisting of the lines of *Y* separated by a carriage return/line feed

In the following table, f and g stand for any primitive scalar dyadic function.

Table B-6
Operators

Operator	Section	Meaning
*X*f.g*Y*	3.5.1	Inner product
X∘f*Y*	3.5.2	Outer product
f/*Y*	3.5.3	The f reduction along the last dimension of *Y*
f/[*N*]*Y*	3.5.3	The f reduction along the *N*th dimension of *Y*
f⌿*Y*	3.5.3	The f reduction along the first dimension of *Y*
f\Y	3.5.4	The f scan along the last dimension of *Y*
f\\[*N*]*Y*	3.5.4	The f scan along the *N*th dimension of *Y*
f⍀*Y*	3.5.4	The f scan along the first dimension of *Y*

The format function is used with the following syntax:

FMT $ *V*;*V*2;...*VN*

or

FMT $ *V*

where

V can be any variable or expression

FMT must be a character vector containing one or more format fields chosen from the following list:

Format	Meaning
'*MAW*'	Character data; cannot be used with numeric values
'*MEW.D*'	Floating-point numeric data with exponent
'*MQ*F*W.D*'	Fixed-point numeric data
'*MQ*I*W*'	Integer numeric data with automatic rounding
'*M*X*W*'	Blanks inserted in edited line
'*M*⍞text⍞'	Literal text inserted in edited line

where

M is the optional repetition factor.

W is the field width.

D is the number of decimal positions.

Q is any number of qualifiers chosen from the following list:

Qualifier	Meaning
B	Blank field if value is 0
C	Insert commas
L	Left-justify
Z	Zero-fill
M⍞text⍞	Insert text left of negative result
N⍞text⍞	Insert text right of negative result
P⍞text⍞	Insert text left of nonnegative result
Q⍞text⍞	Insert text right of nonnegative result
R⍞text⍞	Insert text in background

You must separate format fields with commas. Up to eight significant digits can be specified.

The symbol ᴀ can be used instead of ⍞ (" for TTY terminals).

Table B-7
System Variables

Variable	Section	Meaning
Can be reset:		
⎕*AUS*	4.2.5	Automatically backs up the active workspace if value is 1.
⎕*CT*	4.2.7	Sets the degree of tolerance or relative fuzz to be applied in performing comparisons, value must be in exponent form; range 0 through 1*E*¯8.
⎕*ERROR*	4.2.9	Contains three lines describing the error that occurred.
⎕*GAG*	4.2.10	Inhibits messages sent from other users.
⎕*IO*	4.2.11	Changes the setting of the index origin to 0 or 1.
⎕*LX*	4.2.13	Defines an expression to be executed automatically when the workspace is activated.
⎕*PP*	4.2.15	Sets the precision of non-integer output. Legal values are integers in range 1 through 18.
⎕*PW*	4.2.16	Sets the maximum number of characters that can appear in an output line. Legal values are integers in range 30 through 390.
⎕*RL*	4.2.17	Determines a link in the chain of random numbers used in the roll and deal functions.
⎕*SF*	4.2.18	Sets a new prompt or signal message for evaluated input.
⎕*TIMELIMIT*	4.2.19	Sets a limit to the amount of time you have to respond to a quote-quad input request.
⎕*TRAP*	4.2.21	Expression to execute when an error occurs within a user-defined function.

Table B-7 (Cont.)
System Variables

Variable	Section	Meaning
Retain system-specified values:		
⎕*AI*	4.2.1	Stores account information on the current APL session, including user identification and CPU connect, and keying time.
⎕*ALPHA*	4.2.2	Contains a vector of 27 characters ∆ and *A* through *Z*.
⎕*ALPHAU*	4.2.3	Contains a vector of 27 underlined characters ⍙ and $\underline{A}$ through $\underline{Z}$.
⎕*ASCII*	4.2.4	Contains 128 ASCII characters.
⎕*AV*	4.2.6	Contains a vector of every character in APL.
⎕*CTRL*	4.2.8	Contains a vector of 32 characters listed in Table 4-3.
⎕*LC*	4.2.12	Stores a vector of line numbers in the APL workspace state indicator, arranged in order of most recently suspended function first.
⎕*NUM*	4.2.14	Contains a vector of the 10 digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
⎕*TIMEOUT*	4.2.20	Reports whether a user ran out of time during a quote-quad input request.
⎕*TS*	4.2.22	Stores the current data and time in base format.
⎕*TT*	4.2.23	Determines the time of terminal being used in current session.
⎕*UL*	4.2.24	Stores the system job number associated with the current APL session in base 10 format.
⎕*WA*	4.2.25	Determines the maximum amount that the active workspace can increase.

Table B-8
System Functions

Function	Section	Meaning
⎕*BREAK*	4.3.1	Suspends function execution and returns you to immediate mode.
⎕*CR*	4.3.2	Obtains a canonical representation of a defined function whose name is the character string you specify. The function lines are reformatted to be of equal length, thus allowing the resulting function to be treated as data.
⎕*DL*	4.3.3	Delays the execution of the function in which it is included by the number of seconds specified.
⎕*EX*	4.3.4	Erases an existing use of a name in the workspace. It will not erase a label, a group, a suspended or pendent function, or a system variable.
⎕*FI*	4.3.5	Takes a character argument and converts it to a numeric, placing zeroes in each position that does not correspond to a valid number.
⎕*FX*	4.3.6	Reverses the operation of the ⎕*CR* system function.
⎕*NC*	4.3.7	Returns the classification of a name or list of names.
⎕*NL*	4.3.8	Constructs a list of named objects residing in the active workspace.
⎕*QLD*	4.3.9	Loads a workspace.
⎕*QCO*	4.3.9	Copies a workspace.
⎕*QPC*	4.3.9	Copies a workspace with certain protection.
⎕*SIGNAL*	4.3.10	Signals an error up the stack one level to the caller.
⎕*VI*	4.3.11	Converts a character vector into a numeric vector.

Table B-9
File System Functions

Function	Meaning	Section
data⎕*APPEND* channel ⟦,header ⟦,type ⟦,length⟧⟧⟧	Allows you to write after the end-of-file.	7.6.3 7.6.6
⎕*ASS*'channel filespec ⟦password⟧⟦/fileorg⟧⟦/share⟧⟦/dump⟧ '	Assigns a file on the specified channel number and effectively opens the file for input or output.	7.4.1
⎕*CHS* channel(s)	Returns the file organization and the open status of the files on one or more channels.	7.7.1
⎕*CIQ* ⟦header⟦,type⟧⟧	Unpacks data accumulated for storage with ⎕*COQ*.	7.6.7
⎕*CLS* channel(s)	Closes the files on one or more channels.	7.4.3
data ⎕*COQ* ⟦header ⟦,type⟧⟧	Packs data of different types into a variable for storage as one logical record.	7.6.7
⎕*DAS* channel(s)	Deassigns the files on one or more channels.	7.4.2
⎕*DEQ* channel lock number	Synchronizes shared access to multiple-user files by releasing the locked file.	7.8
⎕*DVC* channel(s)	Returns device characteristics word and blocksize for one or more files.	7.7.2
⎕*ENQ* channel lock number share bit	Synchronizes shared access to multiple-user files by issuing a request to enqueue on a lock pair.	7.8
⎕*FCM* channel	Returns information about the records in a direct-access file.	7.7.4
⎕*FLS* channel	Returns the sharing status and other information about files on one or more channels.	7.7.3

Table B-9 (Cont.)
File System Functions

Function	Meaning	Section
channel ⎕*MTP* operations	Performs a variety of magnetic tape operations including rewinding, setting density, and returning data mode.	7.6.8.2
'filespec'⎕*RENAME* channel	Renames a currently assigned file.	7.4.4

File System Arguments

Argument	Meaning
channel	An integer in range 1 to 12 inclusive specifying the channel on which the file being referenced is assigned.
data	The value to be input or output by the I/O function.
filename	A standard filename in the following format: device:name.extension<protection>[directory] All fields except the name itself are optional. If you include the protection, enclose it in angle brackets. If you include the directory enclose it in square brackets. Names are a maximum of six letters and/or numbers. An extension consists of a period or comma followed by a maximum of three letters and/or numbers. Defaults are the following:
file org	One of the following:

Component	Meaning
device	*DSK*:
name	No default
extension (file type)	File-org dependent
protection	Installation-dependent
user ID	User's directory

Switch	Default Ext	Meaning
/AS	*.AAS*	ASCII sequential
*/AS**	*.AAS*	ASCII sequential; opened for append
/IS	*.AIS*	Internal sequential
*/IS**	*.AIS*	Internal sequential; opened for append

Table B-9 (Cont.)
File System Functions

File System Arguments

Argument	Meaning		
file org (cont.)	Switch	Default Ext	Meaning
	/DA	*.ADA*	Direct-access; read and write
	/DI	*.ADA*	Direct-access; read only (default)
	/BS	*.ABI*	Binary-access; read or write
	*/BS**	*.ABI*	Binary-access; opened for append
	/BU	*.ABI*	Binary-access; read and write; one user at a time
header	A specification indicating whether or not a header is included to indicate the type and number of values in the input or output; 1 is the default and indicates a header; 0 indicates no header.		
length	The number of characters or digits to be written in one operation.		
lock-number	A number in range 0 through 2*33-1 used to synchronize shared file access by multiple users.		
operations	A vector containing codes indicating particular magnetic tape operations; identical to the meaning of codes used by the TAPOP UUO on TOPS-10.		
password	Up to eight characters preceded by a hyphen (-). The null and default password is the hyphen.		
share bit	A specification indicating whether or not a user is willing to share access to a lock on a file; 1 indicates that sharing is desired; 0 is the default and indicates exclusive use of the lock.		
type	Type of value to be input or output; one of the following:		

Value	Type
1	Integer
2	Boolean
3	Single-precision floating-point
4	Double-precision floating-point
5	APL 9-bit
6	ASCII

Table B-10
Keyboard I/O Functions

Function	Section	Meaning
X←⎕	2.5.1	Quad (evaluated) input from keyboard
X←⍞	2.5.2	Quote-quad (character) input from keyboard, up to but not including carriage return
X←⍔	2.5.3	Quad-del (unedited) input from keyboard
⎕←*X*	2.5.5	Quad output (display value of *X*)

Table B-11
File I/O Functions

Function	Section	Meaning
⍇ ⟦{[mode] / [record] / [word]}⟧ channel	7.3	File input
data⍈ ⟦{[record] / [word]}⟧ channel	7.3	File output (write value of data)

The arguments in the file functions have the following meaning:

channel	is the channel number (1-12) on which the file is assigned.
data	is the information written to the file.
mode record word	mode of input or output (ASCII sequential file). record is the record to be read or written (direct-access file). word is the word to be read or written (binary-access file).

For ASCII sequential files, legal values for modes are:

Input Type	Character Set	Mode
⎕	TTY	1
⍞	TTY	2
⍌	TTY	3
⎕	APL	4
⍞	APL	5
⍌	APL	6

Table B-12
System Commands

Command	Section	Meaning
)*BLOT* ⟦n⟧	5.7.1	Generates a mask in a random pattern of length n for concealing confidential input. If n is not specified, the default length is 25.
)*C* ⟦n⟧ file spec.	5.6.1	Ends current session after saving active workspace; returns you to operating system command level and runs program specified. The default device searched is *SYS*:.
)*CALL* ⟦n⟧ file spec.	5.6.1	Same as)*C* except the default device searched is *DSK*:.
)*CHARGE*	5.7.2	Displays a record of activity during the current APL session. Information is installation-dependent, but includes connect time, CPU time, and the number of APL statements and operations executed.
)*CLEAR*	5.2.1	Replaces the active workspace with the clear workspace.
)*CONTINUE* ⟦*HOLD*⟧	5.6.2	Saves the currently active workspace as the continue workspace and exits from APL. Unless *HOLD* is specified, the job is logged off the system. On disk, the workspace appears with the name *CONTIN.APL* in your disk area.

Table B-12 (Cont.)
System Commands

Command	Section	Meaning
)*COPY* wsname ⟦password⟧ ⟦named-object-list⟧	5.4.1	Copies objects identified in the named-object-list from username into the current workspace. If the list is omitted, variables, functions and groups are copied.
)*CREATE* filesize ⟦filespec password⟧ ⟦blocking factor⟧	7.6.1	Creates a direct-access file and associates a password, blocking factor, and number of records with the file.
)*DIGITS* ⟦n⟧	5.5.1	Displays or changes the number of significant digits displayed on output. The maximum number is 18. The default is 10.
)*DROP* wsname ⟦switchlist⟧	5.2.2	Deletes the workspace username from your disk area. Information specified by switch-list is displayed as the files are deleted.
)*ECHO* ⟦{*ON* / *OFF*}⟧	5.5.2	Allows or suppresses the display of error lines. The default setting is *ON*.
)*ERASE* name-list	5.4.2	Erases the objects identified in name-list from the active workspace.
)*FNS* ⟦letter⟧	5.4.3	Displays an alphabetical list of function names in the current workspace. If letter is included, the list begins at the specified letter.
)*GROUP* group-name ⟦group-number-list⟧	5.4.4	Collects named objects in the group-member-list into the groups specified by the group-name. If you omit the list, the group-name is dispersed.
)*GRP* group-name	5.4.5	Lists the members of the group identified by the group-name.
)*GRPS* ⟦letter⟧	5.4.6	Displays an alphabetical list of group-names. If letter is included, the list begins at the specified letter.

Table B-12 (Cont.)
System Commands

Command	Section	Meaning
)*INPUT* [[file spec]] [[/character set]]	7.9	Specifies an alternate device and filename from which APL input is to be taken; if you omit the file spec, input is accepted from the terminal. The character set is either APL or TTY.
)*LIB* wsname [[switch-list]]	5.2.3	Displays the names of workspaces. If you omit wsname, all workspaces in your disk area are listed. The switch-list argument controls the display of additional information about the workspaces.
[[)*LOAD* magtape-position]] wsname [[password]]	5.2.4	Retrieves a workspace from a secondary storage device. If you include a password, it must match the password of the file.
)*MAXCORE* [[{P-of-memory / K-of-memory}]]	5.3.1	Displays or changes the current setting for the maximum workspace size. The standard default is 20K words on TOPS-10 and 40P words on TOPS-20 for the data segment. The maximum value for K-of-memory is the smaller of 176K words or the system memory limit. For P-of-memory, the maximum value is 352P.
)*MINCORE* [[{P-of-memory / K-of-memory}]]	5.3.2	Displays or changes the current setting for the minimum workspace size. The minimum and standard default on both systems is 0.
)*MODE* [[{KEYWORD / ESCAPE}]]	5.5.3	Displays or changes the current mode of output for terminals that do not have an APL character set. The default is *KEYWORD*. The mode setting does not affect input from the keyboard and either mode is acceptable on input.
)*MON*	5.6.3	Returns you to operating system command level, leaving your workspace intact. While at command level, you can issue any command that does not alter your memory image. You can subsequently return to APL by typing the *CONTINUE* monitor command.
)*OFF* [[*HOLD*]]	5.6.4	Ends the current APL session. Unless you specify *HOLD*, your job is also logged off the system.

Table B-12 (Cont.)
System Commands

Command	Section	Meaning
)*ORIGIN* [[n]]	5.5.4	Displays or changes the index origin for the currently active workspace. n can be 0 or 1. The default setting is 1. The origin is preserved with the workspace when it is saved.
)*OUTPUT*[[file spec]] [[/character set]]	7.9	Redirects terminal output to another device and filename. If you omit the filespec, output is sent to the terminal. The character set parameter is either APL or TTY.
)*OWNER*	5.3.3	Displays the directory of the user who created the currently active workspace, the date on which it was created, and the terminal number of the device at which it was created.
)*PASSWORD* [[password]]	5.2.5	Displays or changes the password of the currently active workspace.
)*PCOPY* wsname [[password]] [[named-object-list]]	5.4.7	Copies objects identified in the named-object-list from wsname to the current workspace, protecting names already in use. If you omit the list, all variables, functions, and groups are copied.
)*R* [[n]] file spec.	5.6.5	Ends the current APL session and runs the specified program. If n is specified, the value is added to the starting address of the program to be run. The file specified must contain a ready-to-run program (that is, an *.EXE* file). The default device searched is *SYS*:.
)*RUN* [[n]] file spec	5.6.5	Same as)*R* except that the default device searched is *DSK*:.
)*SAVE* [[magtape position]][[wsname]] [[password]]	5.2.6	Saves a copy of the currently active workspace on a secondary storage device, under the name and password specified. If you omit the password, the current password is assumed. If you omit both wsname and password, the current workspace is used ()*WSID*).

Table B-12 (Cont.)
System Commands

Command	Section	Meaning
)*SEAL* [[{*ON* / *OFF*}]]	5.3.4	Displays the current setting of the workspace seal or turns the seal on or off. When the seal is on, only the user who turned the seal on can copy objects from the workspace or can turn the seal off. The default is off. This command has no effect on the)*LOAD* command.
)*SI*	5.4.8	Displays the workspace state indicator which reports on the progress of function execution.
)*SIV*	5.4.9	Displays the workspace state indicator, along with local variable names at each level.
)*SIZE*	5.3.5	Displays the size of the currently active workspace, in P-of-memory on TOPS-20 and K-of-memory on TOPS-10. It also displays the number of pages (TOPS-20) or the number of blocks (TOPS-10) the workspace would occupy if saved on disk.
)*TABS* [[n]]	5.5.5	Displays or changes the increment between tab settings for APL output. The default tab setting is 0. This command is designed to be used with terminals that have physical tab stops.
)*TIME*	5.3.6	Displays connect and CPU time accumulated while the current workspace has been active.
)*VARS* [[letter]]	5.4.10	Displays an alphabetical list of global variables in the currently active workspace. If you include letter, the list begins at the specified letter.
)*VERSION*	5.3.7	Displays the APL version number with which the currently active workspace was saved.
)*WIDTH* [[n]]	5.5.6	Displays or changes the maximum width of the output line; n must be an integer in the range 30 through 390.
)*WSID* [[wsname]] [[password]]	5.2.7	Displays or changes the name of the currently active workspace; optionally changes the password associated with the workspace but does not display it.

Arguments

Argument	Meaning
Blocking factor	One of the following values: 8, 16, 32, 64, or 128. The default is 16.
Character set	The identifier APL or TTY, representing the character set of a user's terminal.
filename	Same format as wsname, except that the name itself has no default and the default extension depends on the type of file.
file size	An integer specifying the maximum number of records that a direct-access file can have.
file spec	Same format as filename.
group-name	An identifier that names a group of variables, functions, or other groups.
group-member-list	A list of variables, functions, or group-names separated by spaces.
identifier	Any sequence of letters or numbers beginning with a letter. Only the first 31 characters in an identifier are significant.
K-of-memory	An integer value representing the number of 1K-word blocks of memory. Users of virtual memory systems should note that 1K is equal to two pages of memory.
letter	One of the characters *A*-*Z*, ∆, or the understruck characters $\underline{A}$-$\underline{Z}$, or $\underline{\Delta}$.
magtape-position	An integer that specifies that the action of the command is to take place following the nth end-of-file mark on the magnetic tape. If no position is specified, the action takes place with the tape in its current position.
n	An integer value.
name-list	A list of identifiers that name variables and/or functions, separated by spaces.
named-object-list	A list of identifiers that name variables, functions, and/or groups, separated by spaces.
number	One of the digits, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
password	Up to eight characters preceded by a hyphen (-). The null and default password is the hyphen (-).
P-of-memory	An integer value representing the number of pages of memory.

Arguments

Argument	Meaning
switch-list	A list of switches in which each switch consists of a slash(/) followed by one of the letters *A*, *B*, *C*, *L*, *M*, *N*, *P*, or *T*. Valid switches include:
	/A Access: the date the file was last read (disk only)
	/B Blocks: the number of blocks required for the file
	/C Creation: the creation date of the file
	/L Long: equal to typing */B/P/C*
	/M Mode: the mode in which the file was written (disk only)
	/N No header: suppresses the printing of the display header line
	/P Protection: the protection code associated with the file (disk only)
	/T Time: the creation time of the file (disk only)
wsname	A standard name in the following format:
	device:name.extension<prot>[directory]
	All fields are optional. If you specify a protection, also type the angle brackets. If you specify a directory, enclose it in square brackets. Names are a maximum of six letters and/or numbers. An extension (or filetype) consists of a period (or comma in TTY mode) followed by a maximum of three letters and/or numbers. Defaults are the following:

Component	Default
device	*DSK*:
name	name of active workspace (from)*WSID*)
extension	.APL
protection	installation-dependent
directory	user's project-programmer number

APPENDIX C

⌶-BEAMS

⌶-beams are another aid for reporting statistics about the system. The following list shows the type of information returned by the 17 ⌶-beams described in this section:

1. Time of day or date

2. CPU time, APL sign-on time, or keying time

3. State indicator line numbers

4. System job jumber of user's project-programmer number

Some ⌶-beams report on general system characteristics (for example, date) and others return information relevant only to a particular user and session (for example, number of APL operations performed). Some of the ⌶-beams have the same functionality as the system variables and system functions described in Chapter 4. This redundancy is preserved in the current version of APL to promote the compatibility of APL programs written under previous versions of the language. However, where there are equivalent ⌶-beams and system functions, we recommend that you use the system functions.

An ⌶-beam consists of the ⌶ character and an integer scalar. You type the ⌶ character by overprinting the encode character (⊤) with the decode character (⊥).

The following paragraphs list ⌶17 through ⌶33 along with the type of information they return:

⌶17 Returning Symbol Table Information

```
      ⎕←⌶17
16 3
```

The ⌶17 returns information about the symbol table associated with your workspace. This ⌶-beam returns a 2-element vector in which the first element is the symbol table size in words and the second is the number of symbol table entries in use in the workspace.

I18   Returning the Condition of the Workspace

```
      I18
0
```

The I18 returns the condition of the active workspace. It returns a value of 0 to indicate that the workspace is intact or a nonzero number to indicate that the workspace has suffered some kind of damage. If I-beam 18 returns a nonzero value, APL attempts to correct the damage.

I19   Returning the Keying Time

```
      I19
1321181
```

The I19 calculates the amount of time that the keyboard has been unlocked awaiting input during the current APL session. Time is expressed in 60ths of a second. I-beam 19 is useful for instructional programs that time the response of students' answers. It returns one component of the information available from the ⎕*AI* system variable (Section 4.2.1).

I20   Returning the Time of Day

```
      I20
3446581
```

The I20 returns the current time of day. It returns the time from midnight in 60ths of a second. To request this number in hours, minutes, and seconds, specify the following:

```
      3↑24 60 60 60⊤I20
15 57 40
```

I20 returns one component of the information available from the ⎕*TS* system variable (Section 4.2.22).

I21   Returning the CPU Time

```
      I21
3496
```

The I21 returns the CPU time used since you signed on in the current APL session. Time is expressed in 60ths of a second. I-beam 21 is useful in comparing the execution times of different programs. You can include I21 in a function and make the execution of that function dependent on the compute time used so far in the session. I21 returns one component of the information available from the ⎕*AI* system variable (Section 4.2.1).

I22 Returning Workspace Availability

```
      I22
18219
```

The I22 returns the maximum amount to which the active workspace can increase. The size is given in words and is obtained by subtracting the current data segment size from the maximum data segment size. I-beam 22 can be used in a function whose execution is dependent on the space available in the workspace. It is similar to the ⎕*WA* system variable (Section 4.2.25); however, I22 returns the number of words available, instead of the number of bytes (where 4 bytes = 1 word).

I23 Returning the System Job Number

```
      I23
17
```

The I23 returns the system job number associated with the current APL session. The job number is returned in base 10 notation. To request this number in octal, specify:

```
      10⊥(3⍴8)⊤I23
21
```

I-beam 23 is equal to the ⎕*UL* system variable (Section 4.2.20).

I24 Returning the APL Sign-on Time

```
      ⎕←I24
2058600
```

The I24 returns the time when you began the current APL session. It returns the time from midnight in 60ths of a second. I-beam 24 returns one component of the information available from the ⎕*AI* system variable (Section 4.2.1).

I25 Returning the Current Date

```
      I25
70579
```

The I25 returns the current date. The date is displayed in the form MMDDYY in base 10 notation. To format a 3-element vector representing the date, specify the following:

```
      (3⍴100)⊤I25
7 5 79
```

I-beam 25 returns one component of the information available from the ⎕*TS* system variable (Section 4.2.22).

⌶26 Returning a Line Number

```
→⌶26
```

The ⌶26 returns the line number of the statement currently being executed or about to be executed. The scalar returned by I-beam 26 is the first element of the vector returned by ⌶27 and is the first line number in the state indicator. This number represents the line at which the innermost function in the state indicator was suspended or is currently executing.

⌶26 is a particularly helpful function when used in branch statements. You resume execution by specifying →⌶26 rather than entering the line number displayed at the time the last function was suspended. ⌶26 returns one component of the information available from the ⎕*LC* system variable (Section 4.2.12).

⌶27 Returning a Vector of Line Numbers

```
⌶27
```

The ⌶27 returns a vector of function line numbers currently in the state indicator. The first element of the vector is the line number returned by ⌶26 and represents the line at which the innermost function was suspended or is currently executing. If an empty vector is returned, this indicates that no functions are suspended or executing.

⌶27 can be used as an aid in resuming function execution without including a specific line number at which the function was suspended. For example, you can define function *RES* as follows:

```
      ∇A←RES
[1]   A←(⌶27)[2]
[2]   ∇

      ⍝RESUME EXECUTION WITH

      →RES
```

⌶27 returns the same information available from the ⎕*LC* system variable (Section 4.2.12).

⌶28 Returning the Terminal Character Set

```
      ⌶28
0
```

The ⌶28 returns the character set of the output device associated with the workspace. This device is the user's terminal unless otherwise specified by the)*OUTPUT* system command (Section 7.9). The integer scalar returned by ⌶-beam 28 is one of the following:

Value	Meaning
0	APL character set
1	TTY character set

⌶28 is related to the ⎕*TT* system variable (Section 4.2.22).

⌶29 Returning the User's Project-Programmer Number

```
      ⌶29
4 132
```

The ⌶29 returns the project-programmer number associated with the current session. The number is returned as a 2-element vector in base 10 notation. To format this number in octal, specify the following:

```
      10⊥(6⍴8)⊤⌶29
4 204
```

⌶29 returns one component of the information now available from the ⎕*AI* system variable (Section 4.2.1).

⌶30 Clearing the State Indicator

```
⌶30
)SI
```

The ⌶30 clears the state indicator. It has the same effect as typing a series of right arrows (→), one for each suspended function. See Section 5.4.9 for a description of state indicator clearing techniques. ⌶30 removes from the system all pendent and suspended functions calls. As the above example indicates, an)*SI* command issued after the clear request results in the display of a blank line.

⌶31 Returning the Number of APL Statements

```
      ⌶31
519
```

The ⌶31 returns the number of APL statements that have been executed since the current session began. This function is useful in evaluating the performance of programs in the workspace.

I32 Returning the Number of APL Operations

```
      I32
837
```

The I32 returns the number of APL operations that have been executed since the current session began. There may be several operations performed in each APL statement. For example:

```
A←3+4
```

This statement contains two operations: addition and assignment. Like I-beam 31, this function is useful in evaluating program performance.

I33 Returning the Time Used

```
I33
```

On TOPS-10, the I33 returns the number of kilo-core-seconds since sign-on. If the GETTAB UUO is privileged in the system, I33 returns a value of zero. On TOPS-20, I33 also returns 0.

APPENDIX D

SPECIFYING TOPS-20 DIRECTORIES

TOPS-20 provides two ways for you to access another user's directory. The first way is with a logical name in place of the device name; the second way is with a project-programmer number instead of a directory name. You can use either method with APL; however, the use of logical names is recommended.

NOTE

> When you see a project-programmer number (for example, [4,204] in this manual or in an error message, use the TRANSL command to find out its corresponding directory name. Refer to Section D.2.1.

For more information about referencing other users' files, refer to the TOPS-20 User's Guide.

D.1 USING LOGICAL NAMES

To use a logical name in accessing another user's directory:

1. Give the DEFINE SYSTEM command to define a logical name (of no more than six characters) as the other user's directory name.

2. Use the logical name in place of the device name when typing the file specification.

D.1.1 Giving the DEFINE Command

To give the DEFINE command:

1. Type DEFINE and press the ESC key; the system prints (LOGICAL NAME).

```
@DEFINE (LOGICAL NAME)
```

2. Type the logical name (an ending colon is optional) and press the ESC key. The system prints (AS).

```
@DEFINE (LOGICAL NAME) BAK: (AS)
```

3. Type the structure and the directory name (enclosed in angle brackets) and press the RETURN key. The system prints an @.

```
@DEFINE (LOGICAL NAME) BAK: (AS) DATA:<SCHULERT>
```

To check the logical name, specify the INFORMATION (ABOUT) LOGICAL-NAMES system command.

```
@INFORMATION (ABOUT) LOGICAL-NAMES (OF)
BAK: => DATA:<SCHULERT>
```

D.1.2 Using the Logical Name

Once you define a logical name, you can then include it in an APL expression in place of a device name.

The following example shows how to load a workspace from the directory named DATA:<SCHULERT>. (Remember, you have already defined the logical name BAK: as DATA:<SCHULERT>.)

```
      )LOAD BAK:TEST
SAVED  15:45:03 24-OCT-78 5F
```

D.2 USING PROJECT-PROGRAMMER NUMBERS

To use a project-programmer number in accessing another user's directory:

1. Give the TRANSL command to find the corresponding project-programmer number for the desired directory name.

2. Include the project-programmer number after the file type.

You do not have to define a logical name when using a project-programmer number. However, project-programmer numbers may not remain constant over time; therefore, use logical names whenever possible.

D.2.1 Using the TRANSL Command

To run the TRANSL command:

1. Type TRANSL and press the ESC key. The system prints TRANSLATE (DIRECTORY).

```
@TRANSLATE (DIRECTORY)
```

2. Type the structure, the directory name, and press the RETURN key. The default structure is your currently connected structure. The system prints the corresponding project-programmer number.

```
@TRANSLATE (DIRECTORY) PS:<SCHULERT>
PS:<SCHULERT> (IS) PS:[4,75]
@
```

You can also use TRANSL to verify that a project-programmer number is correct. To do this, replace the directory name with the project-programmer number.

```
@TRANSLATE (DIRECTORY) [4,75]
BASIC:[4,75] (IS) BASIC:<TST1>
@
```

D.2.2 Using the Project-Programmer Number

To use a project-programmer number in APL, include it in an expression after the file type.

The following example shows how to load an APL workspace from the directory named SCHULERT, using a project-programmer number. (Remember, you have already translated the directory name.)

```
      )LOAD TEST[4,75]
SAVED  13:52:23 27-FEB-79 5P
```

APPENDIX E

TERMINAL SESSION

The following is a sample APL terminal session. The sign-on and sign-off may be different at each installation, but the APL statements will be the same.

```
@APLSF
terminal..LA
APL-20 DECSYSTEM-20 APLSF 2(407)
TTY22)  9:43:39 WEDNESDAY 27-JUN-79 MASELLA        [4,204]
CLEAR WS
      ⍝SET PAGE WIDTH TO 72
      ⎕PW←72
      ⎕PW
72
      ⍝IMMEDIATE EVALUATION
      2+3
5
      ⍝RESULT WAS NOT INDENTED
      ⍝EVALUATION FROM RIGHT TO LEFT
      2×3+4
14

      ⍝ELEMENT-BY-ELEMENT ADDITION
      4 3+2 4
6 7
      ⍝SCALAR APPLIED TO A VECTOR
      5×4 9 2
20 45 10
      ⍝DIVISION IS FLOATING POINT
      5÷3 ¯2 5
1.666666667 ¯2.5 1
      ⍝NOTE DIFFERENCE BETWEEN ¯ AND -
      8 -3 2 9
5 6 ¯1
      ⍝MONADIC DIVIDE IS RECIPROCAL
      ÷5 8 2
0.2 0.125 0.5
      ⍝EXPONENTIATION
      4 9 2*÷2
2 3 1.414213562
      ⍝INDEX GENERATOR
      ⍳5
1 2 3 4 5
      ⍝CEILING FUNCTION
      ⌈3.2 ¯4.2 7.6 ¯18.6
4 ¯4 8 ¯18
```

```
      ⍝FLOOR FUNCTION
      ⌊3.2 ¯4.2 7.6 ¯18.6
3 ¯5 7 ¯19

      ⍝RELATIONAL LESS THAN FUNCTION
      8 19 27<18
1 0 0
      ⍝COMPRESS FUNCTION
      (8 19 27<18)/8 19 27
8
      ⍝ASSIGNS VECTOR TO N
      N←2+⍳7
      ⍝
      ⍝SELECTS ELEMENTS
      1 0 1 0 1 0 1/N
3 5 7 9
      ⍝1 SELECTS
      1/2
2
      ⍝0 REJECTS
      0/2

      ⍝NULL VECTOR PRINTS AS BLANK LINE
      ⍝NUMBER OF ELEMENTS IN NULL VECTOR IS 0
      ⍴0/2
0
      ⍝RESIDUE  N MOD 3
      3|N
0 1 2 0 1 2 0
      ⍝NUMBERS NOT DIVISIBLE BY 3
      0≠3|N
0 1 1 0 1 1 0
      ⍝NUMBERS OF N = TO 3 OR NOT DIVISIBLE BY 3
      ((3=N)∨0≠3|N)/N
3 4 5 7 8
      ⍝LEAST INDEX OF FUNCTION
      5 7 9 11⍳9 6 11
3 5 4
      ⍝FIFTH AND SEVENTH ELEMENTS OF N
      N[5 7]
7 9
      ⍝CATENATE FUNCTION
      N[3 4 7],20 12 13
5 6 9 20 12 13
      ⍝DEFINE FUNCTION WITH A RESULT
      ⍝WITH 1 DUMMY ARGUMENT AND 2 LOCAL VARIABLES
      ⍝LOOP AND END ARE LABELS
      ⍝FUNCTION IS TO FIND ALL PRIMES UP TO
      ⍝AND INCLUDING N
      ∇R←PRIMES N;DONE;D
[1]   D←3
[2]   DONE←N⋆÷2
[3]   R←1+2×⍳⌊ ¯1+N÷2
[4]   LOOP:→(DONE<D)/END
[5]   R←((D=R)∨0≠D|R)/R
[6]   D←R[1+R⍳D]
[7]   →LOOP
[8]   END:R←1 2,R
[9]   ∇
      ⍝FUNCTION IS CALLED WITH 15
      PRIMES 15
1 2 3 5 7 11 13
```

```
      ⍝TRYING ANOTHER PARAMETER
      PRIMES 31
1 2 3 5 7 11 13 17 19 23 29
      ⍝ANSWER IS WRONG SINCE 31 IS PRIME
      ⍝SET TRACE ON ALL LINES OF FUNCTION
      T∆PRIMES←⍳8
      PRIMES 31
PRIMES[1] 3
PRIMES[2] 5.567764363
PRIMES[3] 3 5 7 9 11 13 15 17 19 21 23 25 27 29
PRIMES[4]
PRIMES[5] 3 5 7 11 13 17 19 23 25 29
PRIMES[6] 5
PRIMES[7] 4
PRIMES[4]
PRIMES[5] 3 5 7 11 13 17 19 23 29
PRIMES[6] 7
PRIMES[7] 4
PRIMES[4] 8
PRIMES[8] 1 2 3 5 7 11 13 17 19 23 29
1 2 3 5 7 11 13 17 19 23 29
      ⍝TRACE CAUSED THE PRINCIPAL VALUE OF THE
      ⍝TRACED LINE TO BE PRINTED EACH TIME THE
      ⍝LINE WAS EXECUTED



      ⍝NOW SET STOP AT LINE 4
      S∆PRIMES←4
      PRIMES 31
PRIMES[1] 3
PRIMES[2] 5.567764363
PRIMES[3] 3 5 7 9 11 13 15 17 19 21 23 25 27 29
PRIMES[4]
      ⍝DISPLAY VALUE OF R
      R
3 5 7 9 11 13 15 17 19 21 23 25 27 29
      ⍝R SHOULD INCLUDE THE FOLLOWING
      R←R,31
      ⍝CLEAR TRACE AND STOP VECTORS
      S∆PRIMES←0
      T∆PRIMES←0
      ⍝CHECK STATE INDICATOR
      )SI
PRIMES[4] *
      ⍝RESTART EXECUTION AT LINE 4
      →4
1 2 3 5 7 11 13 17 19 23 29 31
      ⍝OPEN FUNCTION AND EDIT LINE 3
      ∇PRIMES
[9]   [3]R←1+2×⍳⌊¯1+N÷2∇
```

```
        ∇PRIMES
[9]     [3⎕14]
[3]      R←1+2×⍳⌊¯1+N÷2

               /1
[3]      R←1+2×⍳⌈¯1+N÷2
[4]     ∇
        ⍝CALL FUNCTION
        PRIMES 31
1 2 3 5 7 11 13 17 19 23 29 31
        ⍝WRITE THE TIME FUNCTION
        ∇TIME
[1]     ∇
        ∇Z←TIME;T
[1]     Z←60 60 60⊤(T←⌶21)
[2]     TIMER←T
[3]     ∇
        ⍝INITIALIZE TO CURRENT CPU TIME
        TIMER←⌶21
        TIME
0 4 3
        ⍝FIND PRIMES ≤ 10000
        R←PRIMES 10000
        TIME
0 7 32
        ⍝NUMBER OF PRIMES ≤ 10000
        ⍴R
1230
        ⍝LIST THE LAST 5 PRIMES ≤ 10000
        ¯5↑R
9931 9941 9949 9967 9973

        ⍝LIST DEFINED VARIABLES IN THIS WORKSPACE
        )VARS
N         R         TIMER

        ⍝ERASE R
        )ERASE R
        R
  11 VALUE ERROR
         R
          ^
        ⍝VALUE ERROR BECAUSE R NO LONGER EXISTS
        ⍝LIST VARIABLES AGAIE
        ⍝LIST VARIABLES AGAIN
        )VARS
N         TIMER

        ⍝R WAS OMITTED
        ⍝SAVE THIS WORKSAPCE UNDER THE NAME MYWS,APL
        ⍝THEN LOG OFF

        )SAVE MYWS
  10:27:50 27-JUN-79 2 PGS
        )OFF
TTY22) 10:27:54 27-JUN-79
CONNECTED    0:44:15 CPU TIME    0:00:08
315 STATEMENTS 986 OPERATIONS
KILLED JOB 19, USER MASELLA, ACCOUNT APL, TTY 22,
  AT 27-JUN-79 10:27:54,  USED 0:0:9 IN 0:44:49
```

INDEX

READER'S COMMENTS

NOTE: This form is for document comments only. DIGITAL will use comments submitted on this form at the company's discretion. If you require a written reply and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

Did you find errors in this manual? If so, specify the error and the page number.

Please indicate the type of reader that you most nearly represent.

- ☐ Assembly language programmer
- ☐ Higher-level language programmer
- ☐ Occasional programmer (experienced)
- ☐ User with little programming experience
- ☐ Student programmer
- ☐ Other (please specify)__________

Name__________ Date__________

Organization__________ Telephone__________

Street__________

City__________ State__________ Zip Code or Country__________

Do Not Tear – Fold Here and Tape

digital

No Postage
Necessary
if Mailed in the
United States

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

SOFTWARE PUBLICATIONS
200 FOREST STREET MR1-2/E37
MARLBOROUGH, MASSACHUSETTS 01752

Do Not Tear – Fold Here and Tape

Cut Along Dotted Line