

# VAX APL

---

## User's Guide

AA-P142E-TE

**June 1991**

This manual describes the VAX APL interpreter and the environment in which it operates.

**Revision Update Information:** This is a revised document.

**Operating System and Version:** VMS Version 5.4

**Software Version:** VAX APL Version 4.0

**Digital Equipment Corporation  
Maynard, Massachusetts**

---

The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

**Restricted Rights:** Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

© Digital Equipment Corporation 1982, 1984, 1985, 1987, 1988, 1991.

All Rights Reserved.

The Reader's Comments form at the end of this document requests your critical evaluation to assist in preparing future documentation.

The following are trademarks of Digital Equipment Corporation: DEC, DECnet, DECwindows, Digital, GIGI, VAX, VAXcluster, VAXstation, VAX DATATRIEVE, VAX FORTRAN, VMS, VT320, VT330, VT340, and the Digital logo.

HDSAVT, HDS201 and HDS221 are trademarks of Human Design Systems, Inc.  
Tektronix is a trademark of Tektronix, Inc.

---

# Contents

Preface .....	xiii
---------------	------

## 1 The VAX APL Operating Environment

1.1 Terminal Support .....	1-2
1.1.1 APL Terminals .....	1-2
1.1.2 Non-APL Terminals .....	1-4
1.2 APL Character Set .....	1-4
1.3 Starting an APL Session .....	1-11
1.3.1 Initialization File .....	1-12
1.3.2 Command Line .....	1-12
1.3.3 Initialization Qualifiers .....	1-13
1.4 Order of Processing .....	1-20
1.5 Terminal Designators .....	1-21
1.5.1 Terminal Designator Values .....	1-22
1.5.2 Character Sets .....	1-23
1.5.3 Overstruck Characters .....	1-25
1.5.4 $\square TLE$ and $\square GAG$ Settings .....	1-25
1.5.5 Font Files .....	1-26
1.6 APL Operating Modes .....	1-27
1.7 Keyboard Editing .....	1-28
1.8 APL Workspaces .....	1-29
1.8.1 Workspace Types .....	1-30
1.8.2 Workspace Names and File Specifications .....	1-31
1.8.2.1 VMS File Specification Format .....	1-31
1.8.2.2 Workspace Name Defaults .....	1-31
1.8.3 Workspace Passwords .....	1-31
1.8.4 The CONTINUE Workspace .....	1-32
1.8.5 Groups .....	1-32
1.8.6 The State Indicator .....	1-32
1.8.7 Workspace Size .....	1-33
1.9 Interrupting APL .....	1-33
1.10 Ending an APL Session .....	1-35

1.11	Character Sets . . . . .	1-36
1.11.1	Character Sets Used by APL Terminals . . . . .	1-37
1.11.2	Character Set Used by Non-APL Terminals . . . . .	1-38
1.11.3	Composite Character Set . . . . .	1-42
1.11.4	Digital Multinational Character Set . . . . .	1-44
1.11.5	ASCII Character Set . . . . .	1-45
1.11.6	Elements of $\square A V$ . . . . .	1-46

## 2 VAX APL Language Concepts

2.1	Array Types . . . . .	2-1
2.2	Array Structure . . . . .	2-2
2.2.1	Rank of an Array . . . . .	2-3
2.2.2	Shape of an Array . . . . .	2-4
2.2.2.1	Shape and Reshape Functions . . . . .	2-6
2.2.3	Depth of an Array . . . . .	2-8
2.2.4	Shape Domains of Primitive Function Arguments . . . . .	2-9
2.3	Scalar Product and Singleton Extension . . . . .	2-10
2.4	Empty Arrays . . . . .	2-12
2.4.1	Array Prototypes . . . . .	2-14
2.4.2	Fill Items in Arrays . . . . .	2-15
2.5	APL Expressions . . . . .	2-16
2.5.1	Identifiers . . . . .	2-16
2.5.2	Wildcards . . . . .	2-17
2.5.3	Constants . . . . .	2-18
2.5.3.1	Numeric Contants . . . . .	2-18
2.5.3.2	Character Constants . . . . .	2-19
2.5.4	Vector Notation . . . . .	2-20
2.5.5	Functions . . . . .	2-21
2.5.6	Operators . . . . .	2-21
2.5.7	Spaces and Tabs . . . . .	2-22
2.5.8	Evaluating Expressions . . . . .	2-22
2.5.9	Statements . . . . .	2-24
2.5.10	Lines . . . . .	2-25
2.5.11	Comments . . . . .	2-25
2.6	Forming Arrays . . . . .	2-25
2.7	Editing Variables . . . . .	2-27
2.7.1	Editing Character Variables . . . . .	2-28
2.7.2	Editing Numeric Variables . . . . .	2-30
2.7.3	Editing Variables with the DECwindows Interface Editor . . . . .	2-30
2.7.4	The Character-Cell Interface Editor . . . . .	2-32
2.7.5	The $\text{)EDIT}$ System Command Editor . . . . .	2-34
2.8	Indexing Arrays . . . . .	2-34

2.8.1	Index Origin . . . . .	2-35
2.8.2	Selecting One Array Item . . . . .	2-36
2.8.3	Selecting More Than One Array Item . . . . .	2-37
2.8.4	Selecting All Items Along an Array Axis . . . . .	2-38
2.8.5	Indexing Constants and Expressions . . . . .	2-39
2.8.6	Using an Expression to Generate Indexes . . . . .	2-39
2.8.7	Shape of an Indexing Result . . . . .	2-40
2.8.8	Replacing Selected Items in Arrays . . . . .	2-42
2.9	Error Handling . . . . .	2-42
2.9.1	Order of Error Checks . . . . .	2-43
2.9.2	Errors in User-Defined Functions and Operators . . . . .	2-44

### 3 User-Defined VAX APL Operations

3.1	Defining Operations . . . . .	3-1
3.2	Operation Header . . . . .	3-2
3.2.1	Function Header Form . . . . .	3-2
3.2.2	Operator Header Form . . . . .	3-3
3.2.3	Operation Result . . . . .	3-4
3.2.4	Local Symbol List . . . . .	3-4
3.3	Operation Body . . . . .	3-5
3.4	Symbolic Names in Operations . . . . .	3-5
3.4.1	Local Symbols . . . . .	3-6
3.4.2	Global Symbols . . . . .	3-6
3.4.3	Localizing Function and Operator Names . . . . .	3-6
3.4.4	Precedence of Local Symbols . . . . .	3-8
3.5	Branching Within An Operation . . . . .	3-10
3.5.1	Unconditional Branches . . . . .	3-10
3.5.2	Conditional Branches . . . . .	3-11
3.5.3	Labels . . . . .	3-12
3.5.4	Examples of Branching . . . . .	3-13
3.6	Comment Lines . . . . .	3-14
3.7	Locking an Operation . . . . .	3-14
3.8	Executing User-Defined Functions . . . . .	3-15
3.9	Executing User-Defined Operators . . . . .	3-16
3.10	Printing Operations . . . . .	3-16
3.11	Editing Operations . . . . .	3-17
3.11.1	Support Considerations . . . . .	3-17
3.11.2	The DECwindows Interface Editor . . . . .	3-18
3.11.3	The Character-Cell Interface Editor . . . . .	3-20
3.11.4	The )EDIT System Command Editor . . . . .	3-22

3.11.5	The Line-Editor . . . . .	3-23
3.11.5.1	Line Editing Commands . . . . .	3-24
3.11.5.2	Displaying Operation Lines . . . . .	3-28
3.11.5.3	Search and Replace Strings . . . . .	3-29
3.11.5.4	Editing the Operation Header . . . . .	3-31
3.11.5.5	Character Editing . . . . .	3-32
3.11.5.6	Editing Lines That Contain Control Characters . . . . .	3-36
3.11.5.7	Editing in Immediate Mode . . . . .	3-36
3.12	Examples of Defined Operations . . . . .	3-37
3.12.1	Niladic Function . . . . .	3-37
3.12.2	Monadic Function . . . . .	3-38
3.12.3	Dyadic Function . . . . .	3-38
3.12.4	Ambivalent Function . . . . .	3-39
3.12.5	Function with Axis . . . . .	3-39
3.12.6	Dyadic Operator . . . . .	3-41
3.13	Debugging Operations . . . . .	3-42
3.13.1	Suspending Operation Execution . . . . .	3-42
3.13.2	Examining the State Indicator . . . . .	3-44
3.13.3	The Trace Vector . . . . .	3-46
3.13.4	The Stop Vector . . . . .	3-48
3.14	Examples of Error Trapping . . . . .	3-49
3.14.1	System Variable Change . . . . .	3-49
3.14.2	User-Defined Error Messages . . . . .	3-50
3.14.3	Execute Trap Expression . . . . .	3-53
3.15	Programming Considerations for VAX APL . . . . .	3-54
3.15.1	Speed Optimizations in VAX APL Primitives . . . . .	3-54
3.15.2	Space Considerations in VAX APL . . . . .	3-57
3.15.3	Efficient Uses of VMS Subprocesses . . . . .	3-58

## 4 The Report Formatter

4.1	Format Phrases . . . . .	4-2
4.1.1	Too Few or Too Many Format Phrases . . . . .	4-3
4.1.2	Treatment of Empty Arguments . . . . .	4-4
4.1.3	Format Phrase Types . . . . .	4-5
4.1.3.1	Type A—Character . . . . .	4-5
4.1.3.2	Type E—Floating-Point with Exponent . . . . .	4-6
4.1.3.3	Type F—Fixed-Point . . . . .	4-8
4.1.3.4	Type G—Pattern Data . . . . .	4-9
4.1.3.5	Type I—Integer . . . . .	4-12
4.1.3.6	Type Y—Byte Data . . . . .	4-13
4.1.3.7	Type T—Absolute Tab . . . . .	4-15
4.1.3.8	Type X—Relative Tab . . . . .	4-16

4.1.3.9	Type Literal . . . . .	4-18
4.1.4	Format Phrase Parameters . . . . .	4-18
4.1.5	Format Phrase Qualifiers and Decorators . . . . .	4-19
4.1.5.1	B—Blank When Zero . . . . .	4-21
4.1.5.2	C—Insert Commas . . . . .	4-21
4.1.5.3	Kn—Scale Factor . . . . .	4-22
4.1.5.4	L—Left-Justify . . . . .	4-22
4.1.5.5	S—Standard Symbol Substitution . . . . .	4-22
4.1.5.6	Wn—Exponent Digits . . . . .	4-23
4.1.5.7	Z—Zero Fill . . . . .	4-23
4.1.5.8	M and N—Negative Left and Right Decorators . . . . .	4-24
4.1.5.9	P and Q—Positive Left and Right Decorators . . . . .	4-24
4.1.5.10	O—Zero Decorator . . . . .	4-24
4.1.5.11	R—Background Decorator . . . . .	4-25
4.2	Right Argument . . . . .	4-25
4.3	Result Array . . . . .	4-27
4.4	Formatting Character Data . . . . .	4-28

## 5 VAX APL Input and Output

5.1	Terminal Input and Output . . . . .	5-1
5.1.1	Terminal Input Variables . . . . .	5-2
5.1.1.1	Quad Input . . . . .	5-3
5.1.1.2	Quote Quad Input . . . . .	5-4
5.1.1.3	Qual Del Input . . . . .	5-5
5.1.2	Terminal Output . . . . .	5-6
5.1.2.1	Output Catenator . . . . .	5-8
5.1.2.2	Quad Output . . . . .	5-10
5.1.2.3	Bare Output . . . . .	5-10
5.1.3	Diverting Input and Output to Another Device . . . . .	5-11
5.2	File Input and Output . . . . .	5-13
5.2.1	Basic File Concepts . . . . .	5-14
5.2.1.1	File Access Methods . . . . .	5-15
5.2.1.2	The Next-Record Pointer . . . . .	5-16
5.2.1.3	Record Handling and Sequential Operations . . . . .	5-17
5.2.2	Associating Files with Channels . . . . .	5-17
5.2.2.1	Querying File Assignments . . . . .	5-21
5.2.2.2	Returning Channel Numbers . . . . .	5-22
5.2.3	Opening Files and Reading and Writing Records . . . . .	5-23
5.2.3.1	Writing and Reading ASCII Sequential Files . . . . .	5-23
5.2.3.2	Writing and Reading an Internal Sequential File . . . . .	5-25
5.2.3.3	Writing and Reading a Direct-Access or Relative File . . . . .	5-26
5.2.3.4	Writing and Reading a Keyed File . . . . .	5-27

5.2.4	Resetting Next-Record Pointer to Start of File . . . . .	5-30
5.2.5	Closing Files and Disassociating Files from Channels . . . . .	5-31
5.2.6	Determining Information about Files and Devices . . . . .	5-33
5.2.6.1	Returning File Organization and Open Status . . . . .	5-33
5.2.6.2	Returning File Information . . . . .	5-34
5.2.6.3	Returning Device Characteristics . . . . .	5-35
5.3	Advanced I/O Techniques . . . . .	5-37
5.3.1	Sharing Files . . . . .	5-38
5.3.1.1	Sharing Sequential Files . . . . .	5-38
5.3.1.2	Sharing Direct-Access, Relative, and Keyed Files . . . . .	5-42
5.3.1.3	Unlocking Shared Records . . . . .	5-42
5.3.1.4	Limiting Time on Read Functions . . . . .	5-43
5.3.2	Event Flags . . . . .	5-44
5.3.2.1	Associating Events Flags with Channels . . . . .	5-44
5.3.2.2	Event Flag System Functions . . . . .	5-45
5.3.3	Mailboxes . . . . .	5-46
5.3.3.1	Associating Mailboxes with Channels . . . . .	5-46
5.3.3.2	Sending and Receiving Messages . . . . .	5-48
5.3.3.3	$\square$ MBX—Mailbox System Function . . . . .	5-48
5.3.3.4	Sample Functions That Use Mailboxes . . . . .	5-48
5.3.4	Pure Data Records . . . . .	5-55
5.3.4.1	Reading Pure Data Files Sequentially . . . . .	5-58
5.3.4.2	Reading Pure Data Files Randomly . . . . .	5-60
5.3.4.3	Data Type Conversion Tables . . . . .	5-61

## 6 Calling External Routines

6.1	Linking a Routine into a VMS Shared Image . . . . .	6-2
6.2	Mapping the Routine into APL . . . . .	6-3
6.2.1	Dyadic Map . . . . .	6-3
6.2.2	Monadic Map . . . . .	6-5
6.3	Invoking External Routines . . . . .	6-16
6.4	Debugging External Routines . . . . .	6-18
6.5	Examples of Calls to External Routines . . . . .	6-19
6.5.1	Example 1: Calling RTL MTH\$DACOSD . . . . .	6-19
6.5.2	Example 2: Calling RTL LIB\$ERASE_PAGE . . . . .	6-20
6.5.3	Example 3: Calling LIB\$PUT_SCREEN . . . . .	6-20
6.5.4	Example 4: Calling RTL LIB\$GET_SCREEN . . . . .	6-21
6.5.5	Example 5: Calling VMS SORT . . . . .	6-22
6.5.6	Example 6: Calling VAX FORTRAN . . . . .	6-24
6.5.7	Example 7: Calling VAX DATATRIEVE . . . . .	6-25
6.5.8	Example 8: Using $\square$ MAP with /VALUE . . . . .	6-32
6.5.9	Example 9: Calling a VMS System Service . . . . .	6-33

6.5.10	Example 10: Calling SMG\$ Routines . . . . .	6–35
--------	--	------

## A VAX APL Workspace Interchange Standard

A.1	Converting VAX APL Workspaces to WSIS-Formatted Workspaces . . . . .	A–1
A.2	Converting WSIS-Formatted Workspaces to VAX APL Workspaces . . . . .	A–3
A.3	Sample WSIS Session . . . . .	A–5
A.4	Error Messages and Warnings Generated by WSIS Software . . . . .	A–9
A.4.1	WSOUT Messages . . . . .	A–9
A.4.2	WSIN Messages . . . . .	A–9
A.4.3	APLTAP Messages . . . . .	A–11

## Index

## Figures

1–1	The Digital LK201 APL Keyboard . . . . .	1–3
1–2	DECwindows Interface Window . . . . .	1–17
1–3	APL Session Using the Character-Cell Interface . . . . .	1–18
1–4	DECwindows Interface Commands Options . . . . .	1–34
2–1	DECwindows Interface Edit Options . . . . .	2–30
2–2	DECwindows Interface Edit New Variable Dialog Box . . . . .	2–31
2–3	DECwindows Interface Edit Session Commands Options . . . . .	2–32
2–4	Character-Cell Interface Variable Edit Example . . . . .	2–33
3–1	DECwindows Interface Operation Name Dialog Box . . . . .	3–18
3–2	DECwindows Interface Edit Session Example . . . . .	3–19
3–3	Character-Cell Interface Operation Edit Example . . . . .	3–21
5–1	APL Internal Record Format . . . . .	5–59

## Tables

1	Documentation Conventions Table . . . . .	xv
1–1	APL Single-Strike Characters . . . . .	1–5
1–2	Common APL Overstruck Characters . . . . .	1–8
1–3	APL Support for ASCII Graphics . . . . .	1–9
1–4	APL Support for ASCII Control Characters . . . . .	1–9
1–5	Support for Alternate APL Graphics . . . . .	1–10

1-6	APL Terminals and Designators . . . . .	1-22
1-7	APL Character Set Characteristics . . . . .	1-24
1-8	Overstruck Character Key . . . . .	1-25
1-9	$\square TLE$ and $\square GAG$ Settings . . . . .	1-26
1-10	Terminal Designator Font Files . . . . .	1-26
1-11	Editing Characters . . . . .	1-29
1-12	Workspace Name Defaults . . . . .	1-31
1-13	APL-ASCII Key Pairing (Typewriter Pairing) . . . . .	1-37
1-14	APL-ASCII Bit Pairing . . . . .	1-37
1-15	TTY Character Set . . . . .	1-39
1-16	APL COMPOSITE Character Set . . . . .	1-43
1-17	Digital Multinational Character Set . . . . .	1-44
1-18	ASCII Character Set . . . . .	1-45
1-19	Elements of $\square AV(\square IO \leftrightarrow 0)$ . . . . .	1-46
4-1	Summary of Format Phrase Syntax . . . . .	4-5
4-2	E Format Phrases . . . . .	4-7
4-3	F Format Phrases . . . . .	4-9
4-4	G Format Phrases . . . . .	4-11
4-5	I Format Phrases . . . . .	4-12
4-6	Y Format Phrases . . . . .	4-15
4-7	Summary of Format Phrase Parameters . . . . .	4-18
4-8	Summary of Format Phrase Qualifiers and Decorators . . . . .	4-20
4-9	Valid Qualifiers, Decorators, and Parameters for Format Types . . . . .	4-21
5-1	Character Set for $\square INPUT$ and $\square OUTPUT$ Files . . . . .	5-12
5-2	File Organization Qualifiers . . . . .	5-18
5-3	$/AS$ Input and Output Modes . . . . .	5-23
5-4	Possible $\square CHS$ Codes . . . . .	5-33
5-5	Device Characteristics Longword . . . . .	5-36
5-6	Data-Type Parameter Values . . . . .	5-57
5-7	Converting APL Internal Values to External Values . . . . .	5-61
5-8	Converting External Data Types to APL Values . . . . .	5-64
5-9	Converting APL Characters to ASCII ( $\square IO \leftrightarrow 0$ ) . . . . .	5-66
5-10	Converting from APL to Digital Multinational Characters ( $\square IO \leftrightarrow 0$ ) . . . . .	5-67
5-11	Converting from APL to Digital Multinational Characters ( $\square IO \leftrightarrow 0$ ) . . . . .	5-68

5-12	Converting from Digital Multinational Character Set to APL Characters ( $\square IO \leftrightarrow 0$ ) . . . . .	5-73
6-1	Characteristics of External Data Types . . . . .	6-5
6-2	Converting Internal Data to External Data Types . . . . .	6-7
6-3	Converting External Data Types to Internal Data . . . . .	6-12
6-4	VMS Data Structures . . . . .	6-15



---

# Preface

## Manual Objectives

This manual describes the VAX APL interpreter, including APL language and programming elements, facilities for controlling the APL environment, the interaction between APL and the VMS operating system, and APL's I/O capabilities.

## Intended Audience

The primary audience for this manual is experienced APL programmers. This manual is not a tutorial and is inappropriate for novice users. Programmers experienced with other languages such as FORTRAN or BASIC can learn VAX APL from this manual, but are advised to study it in conjunction with an APL language primer.

## Associated Documents

The *VAX APL Reference Manual* documents the VAX APL functions, operators, system variables and system commands. The *VAX APL Installation Guide* contains instructions for installing VAX APL on the VMS operating system. The *VAX APL Installation Guide* also explains how to install QAPL, the execute only version of VAX APL that is available, license-free to sites using VAX APL without vector processing.

This manual contains information about the VMS operating system, RMS files, the VAXTPU editor, VAX DATATRIEVE and VAX FORTRAN. The *VMS DCL Dictionary* and the *Introduction to VMS System Management* provide detailed information you may need to know to use some of the features of VAX APL. To find out more about the VMS system, refer to the VMS system documents listed in the *Introduction to VMS* or use the help utility by typing help at the system prompt (\$). The *VMS Record Management Services Manual* and *VMS Analyze/RMS\_File Utility Manual* contain more information about RMS files. For more information about VAXTPU, consult the *VMS Guide to VAXTPU/EVE Programming* and the *VAX Text Processing Utility Manual*. The

*VAX FORTRAN Language Reference Manual* and the *VAX FORTRAN User Manual* describe VAX FORTRAN. Information about VAX DATATRIEVE can be found in the *VAX DATATRIEVE Handbook*, the *VAX DATATRIEVE User's Guide*, the *VAX DATATRIEVE Reference Manual* and the *VAX DATATRIEVE Guide to Programming and Customizing*.

## Product References

In this document, VMS software products are sometimes referenced according to the following list:

- VAX APL is referred to as APL.
- VAX BASIC is referred to as BASIC.
- VAX BLISS is referred to as BLISS.
- VAX C is referred to as C.
- VAX DATATRIEVE is referred to as DATATRIEVE.
- VAX FORTRAN is referred to as FORTRAN.
- VAX LISP is referred to as LISP
- VAX PASCAL is referred to as PASCAL.
- VAX PL/1 is referred to as PL/1.

# Conventions Used in This Document

The following conventions are used in this manual.

**Table 1 Documentation Conventions Table**

Conventions	Meaning
Default values used in examples	The default value for the index origin ( $\Box I O$ ) is 1, unless explicitly stated to be 0. Numeric print precision ( $\Box P P$ ) is 10 digits. Enclosed arrays are displayed with boxes around enclosed items and with all values in the top left corner of the display areas. This is done using: $\Box D C \leftarrow \left( \begin{smallmatrix} - & 1 & 2 & 3 \end{smallmatrix} \right) ' + + + +     - - '$
Delimiting pairs	This manual uses $\wedge$ <i>text</i> $\wedge$ ; other delimiting pairs may be any of the following pairs: $\Box \Box$ $\cdots \cdots$ $\Box \Box$ $< >$ $< >$
UPPERCASE	Uppercase words and letters indicate that you should type the word or letter exactly as shown.
$A$ $B$ $K$	The APL characters $A$ , $B$ , and $K$ are used in generic descriptions of command formats. $A$ represents a left argument, $B$ represents a right argument, and $K$ represents an axis argument.
<i>italics</i>	Italicized lowercase words and letters, used in format examples, indicate that you are to substitute a word or value of your choice. <i>italics</i> should not be confused with <i>APL CHARACTERS</i> .
Quotation mark (' )	The term quotation mark refers to the APL single quotation mark (' ).
$\leftrightarrow$	The equivalence symbol means “is equivalent to”.
$\ll \gg$	The double square brackets indicate that the item or string of items inside the brackets is optional. Individual items within a string of items are delimited by the   character, which indicates that you may choose only one item from the string.
[ ]	Single square brackets that appear in the format specification for a language element are required syntax for the element being described.

(continued on next page)

**Table 1 (Cont.) Documentation Conventions Table**

Conventions	Meaning
{ }	Braces are used to enclose lists from which one item must be chosen. The items in such a list are delimited by the   character. For some user-defined operation headers, the braces are required syntax (this requirement is described in Chapter 3).
Color	Color in examples shows user input. Note that all examples in the manual are executable, and that comments beginning with the lamp (a) symbol are part of the examples; comments surrounded by parentheses are not part of the examples.
<CR> and <LF>	The <CR> and <LF> symbols indicate the presence of a control sequence representing a Carriage Return and a Line Feed.
Ctrl/X	The Ctrl/X symbol indicates that you must press the key labeled Ctrl while you simultaneously press another key, for example, Ctrl/C, Ctrl/Y, Ctrl/O.
xxx	A symbol such as [xxx] indicates that you press a key on the terminal. For example, the [Return] symbol represents a single stroke of the Return key on a terminal; the [Compose] symbol represents a single stroke of the Compose key.

Unless otherwise noted:

- All numeric values are represented in decimal notation.
- You terminate commands by pressing the Return key.
- You can use the equal sign delimiter (=) in place of the colon (:)

---

## The VAX APL Operating Environment

VAX APL (A Programming Language) is a compact and versatile programming language that is especially suited for handling array-structured data containing numbers or characters. APL is ideal for solving engineering and scientific problems, and it has proved to be an efficient general data processing language as well.

APL is easy to learn; however, it differs considerably from other commonly used languages such as FORTRAN, COBOL, and PASCAL. The distinctive characteristics of APL are immediately apparent:

- APL is an interpreter, not a compiler, so lines may be executed immediately after they are entered.
- APL has an extensive character set including Greek letters, such as  $\rho$ ,  $\Delta$ , and  $\iota$ , and a variety of other special symbols, such as  $\rightarrow$ ,  $\Box$ , and  $\Gamma$ .
- APL evaluates expressions from right to left; for instance, the result of the expression  $8-4+1$  is  $3$ .
- APL has built-in editors. You do not need to go outside the language to write or change programs.

APL users do not have to know much about their host operating system to be highly productive. The interpreter supplies virtually everything that will be needed during a terminal session. In addition to providing a built-in editor, APL provides debugging aids, system communication facilities, and a file system.

The APL interpreter is shareable and reentrant. Each user has a private copy of his or her programs and data, but many users may share one copy of the interpreter.

## The VAX APL Operating Environment

### 1.1 Terminal Support

#### 1.1 Terminal Support

APL language functions and operators are represented by a variety of special characters. The way these characters are supported depends on the type of terminal you have. On APL terminals, you can enter these special characters directly; on non-APL terminals, you must substitute ASCII or TTY mnemonics.

##### 1.1.1 APL Terminals

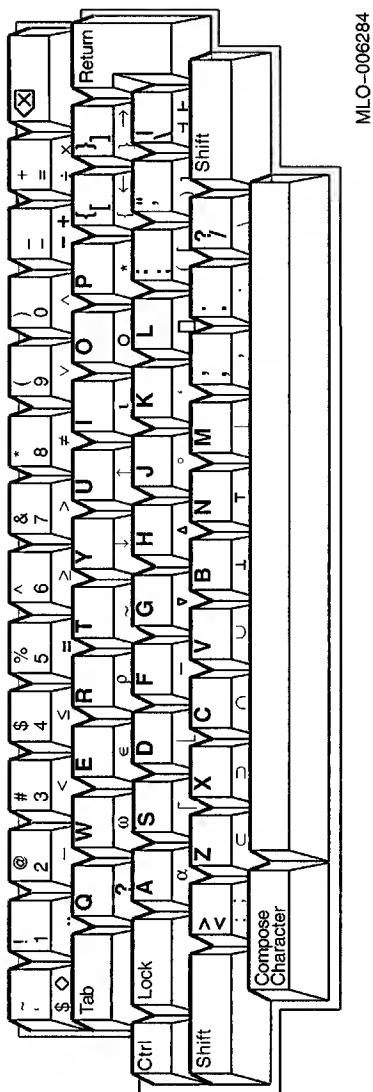
APL terminals support the full APL character set, which may be entered directly from keyboards similar to the one illustrated in Figure 1–1. Note that letters, numbers, and some of the special characters appear in the conventional typewriter keyboard positions. You can set most APL terminals to ASCII mode when you want to use the standard ASCII character set, and then switch to APL mode when you want to use the APL character set.

APL terminals operating in APL mode use one of three APL character sets: APL key-paired (also called typewriter-paired), APL bit-paired, or APL COMPOSITE character set. The APL COMPOSITE character set is a superset of the APL key-paired character set. The terminal designator you specify when you invoke APL, determines which character set your terminal uses (see Section 1.5). The key-paired and bit-paired character sets are presented in Section 1.11.1. The APL COMPOSITE character set is presented in Section 1.11.3.

## The VAX APL Operating Environment

### 1.1 Terminal Support

**Figure 1–1** The Digital LK201 APL Keyboard



## The VAX APL Operating Environment

### 1.1 Terminal Support

#### 1.1.2 Non-APL Terminals

Terminals that do not have an APL keyboard are known as non-APL terminals. On non-APL terminals, you can represent APL symbols using a special set of ASCII characters and mnemonics called TTY mnemonics. For example, to represent the APL rho symbol ( $\circ$ ) on a non-APL terminal, enter the mnemonic .RO.

Section 1.11.2 describes the special handling that APL uses for TTY characters. Table 1–15 in Section 1.11.2 provides an alphabetical listing of the TTY character set.

## 1.2 APL Character Set

The APL character sets supported by APL terminals have 95 printing graphics plus 32 ASCII control characters. The full APL character set, however, has more than 95 printing characters. These additional characters (known as overstruck characters) must be constructed by combining two characters from the APL terminal's character set. For example, the  $\boxtimes$  symbol combines the  $\square$  and  $\leftarrow$  symbols.

Different terminals form overstrikes in different ways. Some terminals allow you to enter the first character, a Backspace (or F12) and the second character on top of the first. Other terminals allow you to press the Compose key (or Ctrl/D) and then to enter the two characters. On these terminals, only the resulting overstruck character is displayed. The order in which the two characters are combined is not significant. On non-APL terminals, overstruck characters are represented by regular characters or ASCII mnemonics. For more information on terminal types and their method of forming overstruck characters, see Section 1.5.

For example, key-paired and bit-paired APL terminals use the Backspace technique. To construct the logarithm symbol ( $\circledast$ ), you would type the circle symbol ( $\circ$ ), Backspace, and the exponentiation symbol ( $\star$ ). (You could also type the symbols in the reverse order.) Note that the dollar sign (\$) may be a single-strike or an overstruck character, depending on the type of APL terminal you are using.

An overstruck character that is not in the APL character set is known as an illegal overstrike; if you enter an illegal overstrike, APL signals *CHARACTER ERROR*, unless the overstrike is part of a character constant, comment, quote quad input, or quad del input.

## The VAX APL Operating Environment

### 1.2 APL Character Set

APL generally allows you to combine two characters that create an overstruck character that looks like a valid APL single-strike character. For example, if you enter + then Backspace, and then - APL accepts the overstruck character as a valid + symbol. Similarly, a character overstruck with itself or with a space is a valid representation of that character. Also, an overstruck character that is overstruck with either of its constituent characters remains valid. (For example, o Backspace \* Backspace o is a valid representation of o.) A character overstruck with a tab, however, is an illegal overstrike (but it is legal to create a tab character by overstriking a tab with a space or with another tab).

The elements of the APL character set are listed in this section . Included are the ASCII mnemonics for TTY users, the characters used in individual APL overstruck characters, and the names commonly associated with each character.

The elements of the APL character set are arranged in the following groups:

- APL single-strike characters
- Common APL overstruck characters
- APL support for ASCII graphics
- APL support for ASCII control characters
- Support for alternate APL graphics

The APL single\_strike characters, Table 1–1, are the 95 printing graphics that are in the key-paired, bit-paired and COMPOSITE character sets.

**Table 1–1 APL Single-Strike Characters**

APL Set	TTY Set	TTY Mnemonic	APL Name
A – Z	A – Z		letters
0 – 9	0 – 9		numbers
+	+		plus
-	-		minus
×	#		times
÷	%		divide
*	*		star

(continued on next page)

# The VAX APL Operating Environment

## 1.2 APL Character Set

**Table 1-1 (Cont.) APL Single-Strike Characters**

APL Set	TTY Set	TTY Mnemonic	APL Name
-		.NG	high minus (NeGation)
-		.US	Under Score
,	,		comma
.	.		period
:	:		colon
;	;		semicolon
\$	\$		dollar sign
'	'		quote
?	?		question mark
~	~	.NT	tilde (NoT)
		.AB	stile (ABsolute value)
α		.AL	ALpha
□		.BX	quad (BoX)
..		.DD	Dieresis (Double Dot)
€		.EP	EPsilon
ι		.IO	IOta
ω		.OM	OMega
ρ		.RO	RhO
^	&		and
∨		.OR	OR
<	<		less than
>	>		greater than
=	=		equal
≠		.NE	Not Equal
≤		.LE	Less than or Equal
≥		.GE	Greater than or Equal
(	(		left parenthesis
)	)		right parenthesis
[	[		left bracket

(continued on next page)

# The VAX APL Operating Environment

## 1.2 APL Character Set

**Table 1–1 (Cont.) APL Single-Strike Characters**

APL Set	TTY Set	TTY Mnemonic	APL Name
]	]		right bracket
{	{	.LB	Left Brace
}	}	.RB	Right Brace
\	\		backslash
/	/		slash
⌈		.CE	CEiling
⌊		.FL	FLoor
◦		.SO	jot (Small O)
◦		.LO	circle (Large O)
Δ		.LD	delta (Lower Delta)
◊		.DM	DiaMond
▽		.DL	DeL
←	—		left arrow
→		.GO	right arrow (GO to)
↑	^		up arrow
↓		.DA	Down Arrow
←		.LU	Left U
→		.RU	Right U
↑		.UU	Up U
↖		.DU	Down U
↗		.LK	Left tack
↘		.RK	Right tack
↑		.EN	up tack (ENcode)
↓		.DE	down tack (DEcode)

The overstruck characters provided in most APL implementations, Table 1–2, must be constructed by combining two characters from the APL terminal character set.

## The VAX APL Operating Environment

### 1.2 APL Character Set

**Table 1–2 Common APL Overstruck Characters**

APL Set	Characters to Combine	TTY Set	APL Name
$\underline{A} - \underline{Z}$	$A - Z$	$_$	.ZA - .ZZ underscored letters
$\underline{\Delta}$	$\Delta$	$_$	.UD Underscored Delta
$\mathbf{!}$	$!$	$.$	!
$\mathbf{a}$	$a$	$\circ$	" lamp
$\mathbf{I}$	$T$	$\perp$	I-Beam
$\mathbf{x}$	$\perp$	$\circ$	.XQ hydrant (eXecute)
$\mathbf{f}$	$T$	$\circ$	.FM thorn (ForMat)
$\mathbf{d}$	$\square$	$\div$	.DQ domino (Divide Quad)
$\mathbf{i}$	$\square$	$\leftarrow$	.IQ Input Quad
$\mathbf{o}$	$\square$	$\rightarrow$	.OQ Output Quad
$\mathbf{q}$	$\square$	$'$	.QQ Quote Quad
$\mathbf{v}$	$\square$	$\triangledown$	.QD Quad Del
$\mathbf{u}$	$\Delta$	$ $	.GU Grade Up
$\mathbf{w}$	$\triangledown$	$ $	.GD Grade Down
$\mathbf{x}$	$\triangledown$	$\sim$	.PD Protected Del
$\mathbf{y}$	$\vee$	$\sim$	.NR NoR
$\mathbf{z}$	$\wedge$	$\sim$	.NN NaNd
$\mathbf{s}$	$\circ$	$*$	.LG LoGarithm
$\mathbf{r}$	$\circ$	$ $	.RV ReVerse
$\mathbf{k}$	$\circ$	$\backslash$	.TR TTranspose
$\mathbf{e}$	$\circ$	$-$	.CR Column Reverse
$\mathbf{c}$	$,$	$-$	.CC Column Comma
$\mathbf{f}$	$/$	$-$	.CS Column Slash
$\mathbf{h}$	$\backslash$	$-$	.CB Column Backslash
$\mathbf{l}$	$c$	$-$	.SS SubSet
$\mathbf;m$	$d$	$-$	.CO COntains
$\mathbf{n}$	$=$	$-$	.MT MaTch
$\mathbf{0}$	$[$	$]$	.SQ Squish Quad

Table 1–3 lists the overstruck characters provided by this APL implementation to represent certain ASCII graphics.

# The VAX APL Operating Environment

## 1.2 APL Character Set

**Table 1–3 APL Support for ASCII Graphics**

ASCII Graphic	APL Set	Characters to Combine		TTY Set	Name
'	'	0	/	.AG	Accent Grave
@	@	C	◦		AT sign
"	"	4	/	.QU	double QUote
#	#	=		.PS	Pound Sign
%	%	:	/	.PC	PerCent sign
&	&	3	/	.AP	AmPersand
^	^	6	/	.CF	CircumFlex
a – z	a – z	A – Z	/	.JA – .JZ	lowercase letters

The overstruck characters used to represent ASCII control characters in literals are listed in Table 1–4.

**Table 1–4 APL Support for ASCII Control Characters**

ASCII Set	APL Set	Characters to Combine		TTY Set	Name
NUL	¤	0	\	.WN	Ctrl/\ (NULL)
SOH	¤	A	\	.KA	Ctrl/A (Start Of Header)
STX	¤	B	\	.KB	Ctrl/B (Start of TeXt)
ETX	¤	C	\	.KC	Ctrl/C (End of TeXt)
EOT	¤	D	\	.KD	Ctrl/D (End Of Transmission)
ENQ	¤	E	\	.KE	Ctrl/E (ENQiry)
ACK	¤	F	\	.KF	Ctrl/F (ACKnowledge)
BEL	¤	G	\	.KG	Ctrl/G (BELL)
BS	¤	H	\	.KH	Ctrl/H (BackSpace)
HT	¤	I	\	.KI	Ctrl/I (Horizontal Tab)
LF	¤	J	\	.KJ	Ctrl/J (Line Feed)
VT	¤	K	\	.KK	Ctrl/K (Vertical Tab)
FF	¤	L	\	.KL	Ctrl/L (Form Feed)

(continued on next page)

## The VAX APL Operating Environment

### 1.2 APL Character Set

**Table 1–4 (Cont.) APL Support for ASCII Control Characters**

ASCII Set	APL Set	Characters to Combine	TTY Set	Name
CR	M	M \	.KM	Ctrl/M (Carriage Return)
SO	N	N \	.KN	Ctrl/N (Shift Out)
SI	O	O \	.KO	Ctrl/O (Shift In)
DLE	P	P \	.KP	Ctrl/P (Data Link Escape)
DC1	Q	Q \	.KQ	Ctrl/Q (Device Control 1)
DC2	R	R \	.KR	Ctrl/R (Device Control 2)
DC3	S	S \	.KS	Ctrl/S (Device Control 3)
DC4	T	T \	.KT	Ctrl/T (Device Control 4)
NAK	U	U \	.KU	Ctrl/U (Negative AcKnowledge)
SYN	V	V \	.KV	Ctrl/V (SYNchronize)
ETB	W	W \	.KW	Ctrl/W (End-of-Transmission Block)
CAN	X	X \	.KX	Ctrl/X (CANcel)
EM	Y	Y \	.KY	Ctrl/Y (End of Medium)
SUB	Z	Z \	.KZ	Ctrl/Z (Substitute)
ESC	3	3 \	.WE	Ctrl/[ (ESCAPE)
FS	4	4 \	.WF	Ctrl/\ (File Separator)
GS	5	5 \	.WG	Ctrl/] (Group Separator)
RS	6	6 \	.WR	Ctrl/^ (Record Separator)
US	7	7 \	.WU	Ctrl/_ (Unit Separator)
DEL	8	8 \	.WD	DEL (DELetE)

Table 1–5 lists the overstruck characters that allow you to enter characters that are not available on some APL keyboards.

**Table 1–5 Support for Alternate APL Graphics**

APL Set	Alternate Graphic	Characters to Combine
\$	\$	\$
◊	◊	^ v

(continued on next page)

**Table 1–5 (Cont.) Support for Alternate APL Graphics**

APL Set	Alternate Graphic	Characters to Combine	
◊	⌘	<	>
{	£	[	◦
}	¤	]	◦
↶	€	(	-
↷	ƒ	)	-

## 1.3 Starting an APL Session

To access APL, you first log in to the VMS operating system. When you receive the system prompt, you are ready to enter the DCL command line to start APL.

When APL first starts executing, it looks for qualifiers and parameters in two places, first in the initialization file, and then on the DCL command line that called APL. The initialization file and the DCL command line are called **initialization streams**.

**Qualifiers** modify the action taken by the command. APL qualifiers are available to do the following:

- Specify the APL interface. (Use in the command line only.)
- Invoke the APL run-time system. (Use in command line only.)
- Display an informational file.
- Execute a file of APL statements.
- Suppress the printing of start-up messages.
- Identify your terminal type.
- Turn on the vector processor.

The optional **parameter** specifies what a command acts upon; an APL parameter specifies the name of a workspace to be loaded at the start of an APL session instead of the *CONTINUE* or *CLEAR* workspace that APL loads by default when the session begins.

## The VAX APL Operating Environment

### 1.3 Starting an APL Session

#### 1.3.1 Initialization File

The first place APL looks for qualifiers and a parameter is in the initialization file pointed to by the VMS logical name APL\$INIT. There are two steps to creating an APL initialization file.

1. Edit a file to include the desired qualifiers and parameter.

The initialization file may contain qualifiers that are described in Section 1.3.3 except for the following:

- /EXECUTE\_ONLY
- /INTERFACE
- /EDIT

APL looks for qualifiers and a parameter only in the first record of the initialization file; thus, anything in the file after the first <CR><LF> is ignored.

In the following example, the first qualifier specifies the terminal type and the second qualifier suppresses the printing of the startup messages (see Section 1.3.3). The parameter specifies that the workspace *PAYROLL* be loaded upon startup.

```
$type startapl.ini  
/terminal=decterm/silent=all payroll
```

2. Assign the logical name APL\$INIT to the file.

Use the DCL command ASSIGN to create the logical name APL\$INIT and assign the name of the initialization file to that logical name.

The command string in the following example shows how to assign the logical name APL\$INIT to the initialization file DBA2:[USER]STARTAPL.INI.

```
$assign dba2:[user]startapl.ini apl$init
```

The *VMS User's Manual* provides more information on associating a file specification with a logical name.

#### 1.3.2 Command Line

The command line to invoke the APL interpreter consists of the command APL, any of the optional qualifiers listed in Section 1.3.3, and optionally a parameter to indicate the workspace to be loaded upon startup. The format is:

```
$ APL [[NO]qualifier] [wsname]
```

**dollar sign (\$)**

is the VMS operating system prompt.

**qualifiers**

are the optional qualifiers described in Section 1.3.3. Note that for each qualifier that initiates an action, there is a corresponding qualifier that negates the action.

**wsname**

is the optional name of the workspace to be loaded instead of the *CONTINUE* or *CLEAR* workspace that APL loads by default when the session begins. The workspace name may appear at any position in either of the initialization streams; that is, qualifiers are legal both before and after the workspace name. If a workspace name is specified in both initialization streams, APL loads the workspace specified on the command line.

### 1.3.3 Initialization Qualifiers

Qualifiers can be used in both the DCL command line and the initialization file.

If the same qualifier is specified more than once in the same initialization stream, APL uses the qualifier specified last, or the rightmost value. For example, the /NOINPUT qualifier prevents the execution of a file that was specified by a /INPUT qualifier earlier (to the left) in the same initialization stream.

If the same qualifier is specified in the initialization file and the command line, APL processes the qualifiers as if those that appear on the command line are appended to those that appear in the initialization file. Then, APL discards all but the rightmost value. The /NOHI qualifier is an exception; APL will process the rightmost /NOHI qualifier, which identifies a file to be displayed when APL starts, from both the initialization streams.

You may abbreviate qualifier names to the shortest unambiguous length. Spaces or tabs are allowed between qualifiers or between a slash (/) and the beginning of a qualifier name.

Note that a colon (:) or the equal sign (=) may be used to specify keyword values to the qualifiers. (See the specifications and example shown below.)

In the following example the DECwindows interface is used, the vector processor is not used and no startup messages are displayed.

## The VAX APL Operating Environment

### 1.3 Starting an APL Session

```
$show logical apl$init  
"APL$INIT" = "APLSTART.INI" (LNMS$PROCESS_TABLE)  
$type aplstart.ini  
/silent/novector  
$apl/int=dec
```

The APL qualifiers are listed below. Qualifiers and keywords with **[NO]** are negatable; the negated form does not take values.

#### **/EXECUTE\_ONLY**

Invokes QAPL, the execute only version of APL; it does not support the interactive sessions or features necessary for program development. You can use QAPL to run APL applications on VMS operating systems that are not licensed to support VAX APL. QAPL can be copied to any valid VMS system free of charge. Instructions to set up the QAPL environment are included in the *VAX APL Installation Guide*. QAPL cannot be run with the vector processor support unless the APL-HPD License is purchased and installed.

You must specify the terminal designator on the command line and you may specify any other qualifiers except /NOTERMINAL. For example:

```
$apl/execute/ter=vt220 payroll
```

Because QAPL does not support interactive sessions, a workspace name must be specified by the command line. If a workspace is not provided, DCL prompts you for one. If you specify a workspace that does not exist, QAPL signals *FILE NOT FOUND*, then *IMMEDIATE MODE IS NOT AVAILABLE*, and then exits to the DCL command level.

QAPL treats all operations in the workspace as locked operations. When QAPL signals an error within an operation, **ERROR** contains the line on which the error occurred and the line containing the caret (^) symbol. If your QAPL application does not trap the error (using **TRAP**), QAPL displays the error message (including the line of the operation causing the error) and exits to DCL. This behavior overrides the lock protection on operations in APL in order to facilitate error handling. (The *VAX APL Reference Manual* shows the APL error messages.)

There are three system functions and three system commands that allow you to bring objects into a QAPL workspace. These are **QPC**, **QCO**, **QLD**, **COPY**, **PCOPY**, and **LOAD**. QAPL locks all operations in the workspace when the workspace is loaded or copied. QAPL also clears stop, trace, and monitor bits for all operations and watchpoints for all variables.

The following features of VAX APL are not included in QAPL.

- Immediate mode
- Quad Input

## The VAX APL Operating Environment

### 1.3 Starting an APL Session

- The DECwindows and Character-Cell interfaces
- The system variables and functions listed below:

<code>□AUS</code>	<code>□BREAK</code>	<code>□CR</code>
<code>□FX</code>	<code>□MONITOR</code>	<code>□STOP</code>
<code>□TRACE</code>	<code>□VR</code>	<code>□WATCH</code>

- All system commands except special cases of the following:

<code>)COPY</code>	<code>)EDIT</code>	<code>)LOAD</code>	<code>)PCOPY</code>
--------------------	--------------------	--------------------	---------------------

#### **/[[NO]EDIT=(TPU\_values)**

Valid only with the /INTERFACE=CHARACTER\_CELL qualifier. Specifies the qualifiers and parameters to be used with the TPU-based interface for Character-Cell terminals. Any of the TPU qualifiers and parameters may be used. For example:

```
$apl/int=cha/edit=(sec=$aplgrp:[apluser]edit.sec,com=$aplgrp:[apluser])
```

If a specific set of values are frequently used, you may want to modify the APL symbol.

This qualifier is not available for use with QAPL sessions or in initialization files.

The *VMS Guide to VAXTPU/EVE Programming* and the *VAX Text Processing Utility Manual* contain more information on TPU qualifiers and parameters.

#### **/[[NO]HI=(vmsfilespec,APL) (vmsfilespec,KEY) (vmsfilespec,BIT) (vmsfilespec,TTY) (vmsfilespec,COMPOSITE)**

*vmsfilespec* must include at least a file name. (see Section 1.8.2.1.)

Specifies a hi file to be displayed during apl initialization. You can use the /HI qualifier in both initialization streams; thus, two HI files may be printed. The file contents are interpreted in the specified character set and are read in quote-quad input mode. (See Chapter 5.)

## The VAX APL Operating Environment

### 1.3 Starting an APL Session

```
/[[NO]]INPUT=(vmsfilespec,APL)
          (vmsfilespec,KEY)
          (vmsfilespec,BIT)
          (vmsfilespec,TTY)
          (vmsfilespec,COMPOSITE)
```

*vmsfilespec*, described in Section 1.8.2.1 must include at least a file name.

Specifies a file to be automatically executed in )*INPUT* fashion when the APL session begins. (See Section 5.2.3 and the *VAX APL Reference Manual*.)

```
/[[NO]]INTERFACE=interface
          CHARACTER_CELL
          DECwindows
          LINE
```

Selects the type of APL interface to use. The default is LINE. This qualifier is not available for use with the /EXECUTE\_ONLY qualifier or in initialization files.

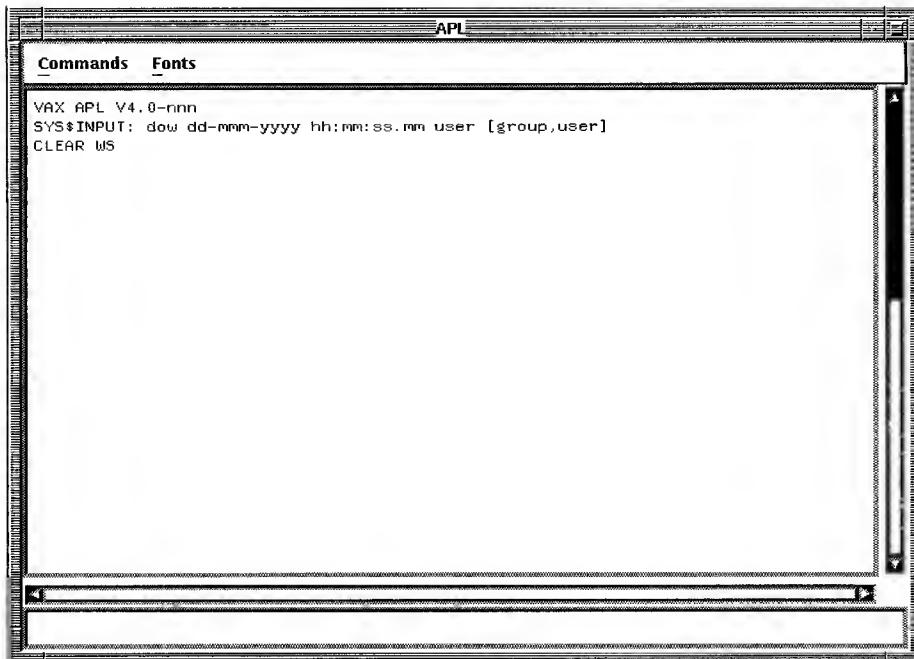
The DECwindows value invokes full DECwindows support of the APL product to more easily develop applications interactively. In addition to the initial APL DECwindow, you can open one or more sessions to edit user-defined operations and variables. *VMS DECwindows Motif User's Guide* and *VMS DECwindows Motif Applications Guide* have more information on using the DECwindows environment.

The DECwindows interface opens a window, shown in Figure 1–2, with the APL title bar, two scroll bars (vertical and horizontal), a command line at the bottom for input and the large center transcript area to record the session.

## The VAX APL Operating Environment

### 1.3 Starting an APL Session

Figure 1–2 DECwindows Interface Window



If the INTERFACE=DECwindows qualifier is included on the command line, the APL session ignores any value assigned to the /TERMINAL qualifier.

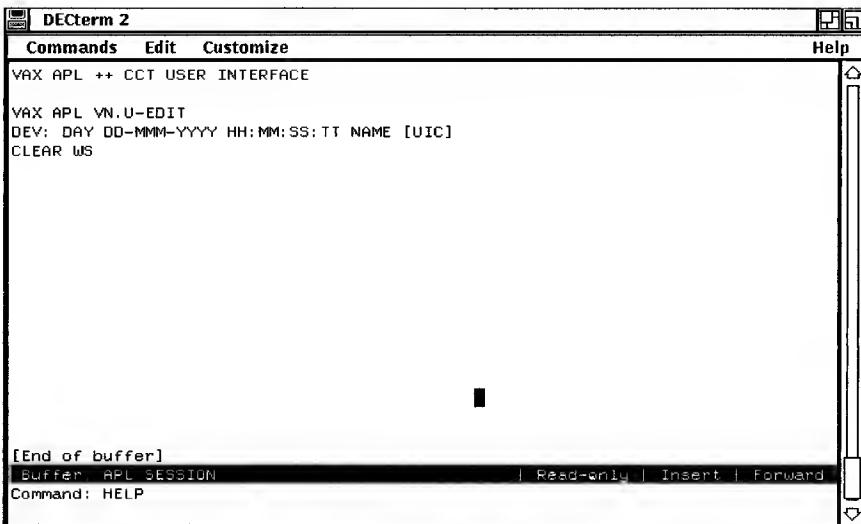
The CHARACTER\_CELL value causes the invocation of a VAXTPU-based interface, which makes windows available to more easily develop applications interactively. The /EDIT qualifier should be used to specify the TPU options preferred to enhance this APL interface.

VAXTPU uses a **buffer**, a temporary holding area, to manage your APL session. The contents of the APL interactive session are shown in an area of the screen that is called a window. The End of Buffer message defines the end of the workspace. It is only visible on the screen and is not interpreted by APL. A highlighted status line, located at the bottom of the window, shows the buffer name (APL SESSION), current mode (insert or overstrike), and the current direction (forward or reverse). Figure 1–3 shows an APL session using the Character-Cell interface.

# The VAX APL Operating Environment

## 1.3 Starting an APL Session

Figure 1–3 APL Session Using the Character-Cell Interface



The VAXTPU manages the APL buffers with commands that do the following:

- List all of the buffers used in this APL session
- Delete a specified buffer
- Change the buffer displayed in the window
- Create a new buffer that contains the contents of a specified file
- Write the contents of a buffer to a specified file

The Character-Cell interface allows you to view more than one window on your terminal screen at the same time. For example, you can have the interactive session in one window and an edit buffer in another window. To help you manage the APL windows, VAXTPU commands are available to do the following:

- Split the screen into more than one window
- Put the cursor in the next, previous or other window
- Restore the current window as a single, large window
- Enlarge or shrink the current window by a specified number of lines

## The VAX APL Operating Environment

### 1.3 Starting an APL Session

For more information about windows, buffers and the VAXTPU commands, access the online Help utility. Press the Do or PF4 key or enter Ctrl/B to reveal the Command: prompt and enter HELP. (See Figure 1-3.)

#### **/[[NO]]SILENT[=(*silentmodes*)]**

ALL  
HI  
BANNER  
WSMESSAGE  
NONE

Controls whether the APL banner line (see Section 1.4), HI files, and initial workspace message (see Section 1.4) are printed. The default is /NOSILENT, which indicates that the APL banner line, the HI files, if specified, and the initial workspace message will be printed. You can specify more than one keyword to the /SILENT qualifier by separating the desired keywords with commas and surrounding the argument with parentheses. APL evaluates the keywords cumulatively; thus, the specification /SILENT=(ALL,HI) is the same as /SILENT=ALL.

#### **/[[NO]]TERMINAL=*terminal***

*terminal* specifies the terminal designator as listed in Table 1-6.

Specifies the terminal type. If you do not specify a terminal type in the initialization file or the command line, APL prompts you for one. If you specify /TERMINAL but omit the *terminal* value, APL signals MISSING QUALIFIER OR KEYWORD VALUE and returns you to the DCL command level.)

/NOTERMINAL negates a previously specified terminal designator. If the negated terminal qualifier, /NOTERMINAL, is the last occurrence of a terminal qualifier in the initialization stream (see Section 1.3.3), APL will prompt for the terminal type (see Section 1.5).

#### **/[[NO]]VECTOR[=*threshold*]**

*threshold* specifies the minimum data size for APL to use vector processing hardware. A value of 1, indicates that the vector processor will always be used; a value of 0 indicates that the vector process will never be used.

Note that the VAX APL HPO license must be loaded before VAX APL can use the vector processor.

When you invoke a session, APL determines whether a vector processor is available. If no vector processor is available, the vector processor threshold value will be set to 0. If a vector processor is available, it will be set to the threshold value specified with this qualifier, or if this qualifier is not included

## The VAX APL Operating Environment

### 1.3 Starting an APL Session

in an initialization stream, the threshold value will be set to the non-negative, near-integer default.

## 1.4 Order of Processing

APL initialization processing proceeds in the following sequence of steps:

1. Sessions facilitated by the Character-Cell or DECwindows interface start the interface first, then pass the APL qualifiers and parameters to start the APL session. The Character-Cell interface uses a TPU-based screen and displays a session manager message in the following form:

*VAX APL -- CCT User Interface*

2. The initialization stream qualifiers are checked for syntax errors, and an error message identifying any invalid qualifiers is displayed. If an error is detected, APL returns you to DCL. (Refer to Section 1.3.3.)
3. APL determines whether there is a vector processor.
4. The user is prompted for a terminal type (if the /TERMINAL qualifier was not specified). (See Section 1.5.)
5. The APL banner line is displayed.

If a vector processor is not available, the APL banner line has the form:

*VAX APL lv.u-edit  
dev:day dd-mmm-yyyy hh:mm:ss.tt name [uic]*

The APL banner line has the form if a vector processor is not available:

*VAX APL HPO/VMS lv.u-edit  
dev:day dd--mmm--yyyy hh:mm:ss.tt name [uic]*

*l* is the support letter.

*v* is the version number.

*u* is the update number.

*edit* is the edit number.

*dev:* identifies your terminal device.

*day* is the day of the week.

*dd--mmm--yyyy and hh:mm:ss.tt* are the date and time.

*name* is your VMS user name.

*uic* is your vms user identification code.

6. APL HI files are displayed. (See Section 1.3.3.)

## The VAX APL Operating Environment

### 1.4 Order of Processing

7. The file named by the /INPUT qualifier is opened, thus establishing the file, rather than the terminal, as the default source of input.

Note that the default source of input is changed from the terminal to the /INPUT file before the workspace is loaded. This could be important if the value of  $\square LX$  (see  $\square LX$ , Latent Expression, the *VAX APL Reference Manual*) in the workspace to be loaded is not empty. If  $\square LX$  initiates processing that calls for input, APL takes the input from the /INPUT file.

8. The applicable workspace is loaded, and the appropriate message is displayed.

The workspace that APL loads is determined by the following criteria.

- a. The last workspace specified in an initialization stream.
- b. The *CONTINUE* workspace saved from a previous APL session, if one exists in your default area (see Section 1.8.1).
- c. A clear workspace, if neither of the first two choices applies.

If a clear workspace is loaded, APL displays the following message:

*CLEAR WS*

If the *CONTINUE* workspace or a workspace specified in an initialization stream is loaded, APL displays a standard *)LOAD* message, such as:

*SAVED THURSDAY 16:32:14.28 4-APR-1979 14 BLKS WAS FOO*

9. The file named by the /INPUT qualifier is read.

After displaying the appropriate load message, APL indents six spaces to signify that it is ready to accept input, unless  $\square LX$  causes execution of an expression or unless there is an operation executing at the top of the *)SI* stack.

Some steps in the sequence may be eliminated, depending on the settings of the initialization stream qualifiers.

## 1.5 Terminal Designators

When you begin an APL session, you must tell APL what type of terminal you are using. Your APL working environment is set up according to the terminal designator value you specify.

# The VAX APL Operating Environment

## 1.5 Terminal Designators

### 1.5.1 Terminal Designator Values

You can specify the terminal type in either the command line or the initialization file; if you do not, APL prompts you for it with the following:  
terminal.. .

You respond with one of the terminal designators listed in Table 1–6. If you are unsure of how to respond, enter a question mark (?) and press the Return key. APL then lists the possible designators and prompts you again. For example:

```
terminal..?
one of the following:
    apl      la      key      bit      tty      4013      4015      gigi
    vt102    composite      vt220    vt240    vt320    vt330    vt340
    hds201   hds221   hdsavt   vs       decterm
terminal.. .
```

If you enter Ctrl/Z or Ctrl/C at the terminal prompt, APL returns you to DCL.

Table 1–6 lists the terminal devices and designators supported by VAX APL.

**Table 1–6 APL Terminals and Designators**

Terminal	Designator
Bit-paired ASCII/APL terminal	BIT
Key-paired ASCII/APL terminal	KEY or APL
terminal using APL COMPOSITE character set	COMPOSITE
Digital LA12, LA34, LA36, LA37, LA38, LA100, LA120, LS120	LA
Digital VK100 (GIGI) terminal	GIGI
Tektronix 4013	4013
Tektronix 4015	4015
HDS201	HDS201
HDS221	HDS221
HDSAVT	HDSAVT
Digital VT102-PA/RA with APL feature <sup>1</sup>	VT102
Digital VT220	VT220
Digital VT240	VT240

<sup>1</sup>If you do not have the optional APL feature for the VT102 terminal, specify TTY as your terminal designator.

(continued on next page)

## The VAX APL Operating Environment

### 1.5 Terminal Designators

**Table 1–6 (Cont.) APL Terminals and Designators**

Terminal	Designator
Digital VT320	VT320
Digital VT330	VT330
Digital VT340	VT340
Digital VAXstation using VMS Workstation Software	VS
Digital VAXstation using DECwindows	DECTERM
Any terminal without APL character set	TTY[/ <i>terminal</i> ] ]

Note that you can use any APL terminal in ASCII mode as a non-APL terminal by specifying TTY as your terminal designator when you invoke APL, or you can set  $\square TT\leftarrow 2$  once inside APL.

The TTY designator takes an optional qualifier, */terminal*, which can be any one of the other terminal designators in Table 1–6. This qualifier is relevant only when you use the *)INPUT* and *)OUTPUT* system commands, or the  $\boxplus$  and  $\boxminus$  file system functions (see Chapter 5). The default value for */terminal* is */key*.

If you respond to the terminal prompt with TTY, followed by any character other than a <CR> or a slash (/) with a legal terminal designator, APL displays the following:

```
type ? for help
terminal..
```

If you enter *tty?* at the terminal prompt, APL displays the list of possible designators that can follow *TTY/*. If you attempt to use the */terminal* qualifier on a designator other than TTY, APL displays the following message before continuing:

```
qualifier ignored for non tty terminal
```

Note that *TTY/terminal* is not available when you specify your designator in an initialization stream. It is available only in response to the terminal prompt.

#### 1.5.2 Character Sets

When you invoke APL, the APL interpreter selects the character set to be used with your terminal based on the terminal designator you specified.

If you specified the BIT terminal designator, APL uses the Bit-paired character set.

## The VAX APL Operating Environment

### 1.5 Terminal Designators

The terminal designator also determines whether the APL character set is loaded automatically by APL or manually by the user.

Table 1-7 shows the APL character set characteristics for each terminal designator.

**Table 1-7 APL Character Set Characteristics**

Terminal Designator	Character Set	Qualifier Mode	Map
APL	Bit-paired	Manual	
BIT	Bit-paired	Manual	
KEY	Key-paired	Manual	
LA	Key-paired	Automatic	
GIGI	Key-paired	Manual	{ G0 - 7-bit ASCII G1 - APL Character Set }
4013	Key-paired	Automatic	
4015	Key-paired	Automatic	
HDS201	Key-paired	Automatic	
HDS221	Key-paired	Automatic	
HDSAVT	Key-paired	Automatic	
VT102	Key-paired	Automatic	{ G0 - 7-bit ASCII G1 - APL Character Set }
COMPOSITE	APL COMPOSITE	Manual	
{ VT220 VT240 VT320 VT330 VT340 DECTERM VS }	APL COMPOSITE	Automatic	{ G0 - 7-bit ASCII G1 - APL Character Set G2 - DEC Supplemental G3 - Special Graphics }
TTY	TTY	Does not map an APL character set	

Note that the LA, 4013, or 4015 terminal users could specify key as their terminal designator, and then load character sets manually.

### 1.5.3 Overstruck Characters

The key used to create overstruck characters depends upon the terminal designator specified in the initialization stream. Table 1-8 shows which key is used for each terminal designator.

For example, to create an APL overstruck character on terminals using the Backspace key, enter the first character, press the Backspace key and enter the second character. For example, Shift/l Backspace Shift/k produces quote\_quad (¶). The constituent characters may be entered in either order.

Terminals using the TTY designator use TTY mnemonics and do not need to create overstruck characters.

**Table 1-8 Overstruck Character Key**

Backspace	Ctrl/D	Compose
APL	VT220	VS
LA	VT240	
KEY	VT320	
BIT	VT330	
4013	VT340	
4015	DECTERM	
GIGI		
VT102		
HDS201		
HDS221		
HDSAVT		

### 1.5.4 $\square TLE$ and $\square GAG$ Settings

When you invoke APL, the terminal characteristics for line editing,  $\square TLE$ , and the characteristic for broadcasts,  $\square GAG$ , are set by APL. These settings are determined by the terminal designator value you specified in the initialization file or command line and the VMS terminal characteristics.

Table 1-9 shows the default settings for each terminal designator.

You can change these settings by assigning new values to  $\square TLE$  or  $\square GAG$ . (See the *VAX APL Reference Manual*.)

## The VAX APL Operating Environment

### 1.5 Terminal Designators

Table 1-9  $\square TLE$  and  $\square GAG$  Settings

Terminal Designator	$\square TLE$	$\square GAG$
{ APL LA KEY BIT 4013 4015 GIGI VT102 HDS201 HDS221 HDSAVT }	0 (noline_edit)	Inherits terminal characteristics; 1 (refuse messages) 2 (trap,translate, and display message)
{ TTY COMPOSITE VT220 VT240 VT320 VT330 VT340 DECTERM VS }	Inherits terminal characteristics; 0 (noline_edit) 1 (line_edit)	Inherits terminal characteristics 0 (display message) 1 (refuse message)

Because  $\square TLE+1$  means the Backspace key sends the cursor to the beginning of the line, you must use the arrow keys to change the position of the cursor on an input line.

#### 1.5.5 Font Files

APL character support for the designators listed in Table 1-10 use font files provided with the APL software. Logical names are used to find the associated font file. You can define these logical names to point to your own font files. Otherwise, APL uses the font files installed with VAX APL.

Table 1-10 Terminal Designator Font Files

Designator	80-column Mode Logical	132-column Mode Logical
VT220	APL\$VT220_FONT	APL\$VT220_FONT
VT240	APL\$VT240_FONT	APL\$VT240_FONT_132

(continued on next page)

**Table 1–10 (Cont.) Terminal Designator Font Files**

Designator	80-column Mode Logical	132-column Mode Logical
VT320	APL\$VT220_FONT	APL\$VT320_FONT_132
VT330	APL\$VT330_FONT	APL\$VT330_FONT_132
VT340	APL\$VT340_FONT	APL\$VT340_FONT_132

## 1.6 APL Operating Modes

There are two operating modes in APL:

- Immediate mode
- Function-definition mode

In immediate mode (sometimes called execution mode), a line is executed immediately after you enter it and press the Return key. You must be in immediate mode to execute APL statements. You enter function-definition mode only to define or edit an operation (for more information, see Chapter 3).

In immediate mode, APL clearly differentiates between what it prints out and what you enter. APL displays its output at the left margin, but indents six spaces before echoing your input; thus, the data APL prints out begins in column 1, and the data you enter in begins in column 7. The six spaces APL indents can be thought of as an input prompt. For example:

```
2 + 2
4
15
1 2 3 4 5
```

The DECwindows interface echoes input on the command line. When you press the Return key, APL transfers the input to the transcript window and executes the command.

Text can be copied from the transcript window or from another window on the display to the command line. To copy text, position the mouse pointer at the beginning of the text, hold down Mouse Button 1 (MB1) and drag the mouse until all of the text you want to Cut is highlighted. Release MB1. Position the mouse pointer on the command line and press Mouse Button 2 (MB2) to copy the text onto the command line. Press Return to execute the command. Input on the command line can be a mixture of entered and pasted text. For example, you could enter *A*, copy *←2 3 ♂ 12* from a previous command shown in the transcript window, and press Return.

## The VAX APL Operating Environment

### 1.6 APL Operating Modes

APL sessions using the Character-Cell interface can copy lines previously entered in the current session. Use the up arrow key or the mouse to position the cursor on the line you want to copy and press the Return key. The line is copied to the input line and TPU displays a message indicating that the copy was successful. The line is executed when you press the Return key.

## 1.7 Keyboard Editing

The order in which you enter characters in an input line is insignificant; regardless of how you enter the line, APL evaluates it exactly as it appears on the terminal.

If you discover an error in a line before you enter it (in other words, before you press Return), and  $\square TLE$  is set to 0 (noline\_edit), backspace to the error and press the Line Feed (LF) key. Everything from the <LF> to the right is ignored by APL. You can then complete the line by entering the correct text directly below the part in error.

If  $\square TLE$  is set to 1 (line\_edit), use the arrow keys or the Delete key to change the position of the cursor on the input line to correct the error. Press Return to execute the command. APL inputs the entire line, regardless of the location of the cursor.

If you are using the DECwindows and Character-Cell interfaces, the mouse can be used to point to and click on a new cursor position in the input line. Enter new text or edit the existing text, and press Return to execute the command.

You can use the  $\text{v}$  editor to edit the last immediate mode line that was executed; for details, see Section 3.11.5.7.

The VMS operating system recognizes several ASCII control characters as keyboard editing characters. Table 1–11 lists some of these characters and their meanings. One particularly important control character, the attention signal, is referred to throughout this manual; use it to interrupt APL operation execution and expression evaluation. For more details about the attention signal, see Section 1.9. For more details about how other control characters are implemented, see the *VMS DCL Dictionary*.

**Table 1–11 Editing Characters**

<b>Character</b>	<b>Meaning</b>
Backspace	Positions the cursor one character to the left. If the VMS terminal line-editing feature is enabled, Backspace positions the cursor at the left margin. $\square TLE$ controls the setting of the terminal line-editing feature (see the <i>VAX APL Reference Manual</i> ).
Ctrl/C	Interrupts APL operation execution and expression evaluation. One Ctrl/C is the weak attention signal; two, the strong attention signal. For details, see Section 1.9.
Ctrl/O	Suppresses output to the terminal. Entering a second Ctrl/O resumes output to the terminal, if the program is still executing.
Ctrl/R	Performs a line feed and displays the corrected line starting at column 1.
Ctrl/T	Causes APL to display an informational message. The contents of this message are discussed in Section 1.5 under the description of HDS terminals.
Ctrl/U	Deletes the current input line and reprompts for input.
Ctrl/X	Deletes all lines that you have entered but that have not yet been executed.
Ctrl/Y	Panic exit; returns you to DCL. Depending on your terminal type, you may or may not be able to return to your APL session with the DCL command CONTINUE.
Delete	Deletes the previous character. On an LA terminal, echoes as a $\leftarrow$ character.
Tab	Advances cursor to the next horizontal tab stop. APL cannot change tab stops. Tabs are passed to the operating system for interpretation.

## 1.8 APL Workspaces

APL uses a block of storage called a workspace to store the following:

- Operations, variables, and their values
- Information about the status of operations
- Group descriptions
- Temporary results obtained while executing APL statements
- Other work completed in an APL session

## The VAX APL Operating Environment

### 1.8 APL Workspaces

In this manual, the term **workspace** refers to either the active workspace or a version of an active workspace that is being saved on secondary storage and thus is inactive.

As an APL user, you have extensive control over the activity and characteristics of the workspaces in the system. You can clear, save, load, name, and delete workspaces, and you can copy operations and variables from a saved workspace into an active workspace.

#### 1.8.1 Workspace Types

There are three workspace types:

- Clear workspace
- Active workspace
- Inactive (or saved) workspace

A **clear workspace** is just what the name implies: a workspace that is entirely clear of operations, variables, and other signs of work done during APL sessions. You receive a clear workspace at the beginning of an APL session, unless a *CONTINUE* workspace is available or unless you ask for a specific workspace when you invoke APL. (For a complete description of the characteristics of a clear workspace, see the *VAX APL Reference Manual*).

The **active workspace** is the workspace you are currently using. When you begin to do work in a clear workspace, it becomes the active workspace (although it retains the name *CLEAR WS* until you explicitly name it). The effects of the work you do during an APL session are temporarily stored in your active workspace.

When you save an active workspace with the *)SAVE* command, the work you did while it was active is permanently stored on a secondary storage device, usually in your default area. The version of the workspace that you save is known as an **inactive workspace**. The saved work includes operations, variables, the APL symbol table and state indicator, some system variable settings, and so forth. An inactive workspace can be reloaded into memory and can become the active workspace once again, thus reproducing the environment that was in effect when the workspace was saved. (Note that files left open when the workspace was saved are now closed, but not deassigned, and a few system variable settings may have changed.)

## 1.8.2 Workspace Names and File Specifications

Each APL workspace defined in your disk area must have a unique name. The formats for workspace names are the same as those for other operating system file specifications.

### 1.8.2.1 VMS File Specification Format

Under the VMS operating system, the format for file specifications is as follows:

*node::device:[directory]filename.filetype;version*

An example of a complete VMS workspace name is:

*NODEA::DB1:[APLGRP]SAMPLE.APL;6*

The maximum length for a VMS file specification is 255 characters. For more details about VMS file specifications, see the *VMS User's Manual*.

### 1.8.2.2 Workspace Name Defaults

You do not have to specify all the parts of a file specification. The defaults are summarized in Table 1-12.

**Table 1-12 Workspace Name Defaults**

Component	Default
Node	The computer node you are using.
Device name	Your default device.
Directory	Your default directory.
File name	Generally must be specified, but sometimes defaults to the name of the active workspace.
File type	.APL
Version	For input, the highest version number. For output, the highest version number plus one.

## 1.8.3 Workspace Passwords

You may use the *)SAVE*, *)WSID*, or *)PASSWORD* system commands (see the *VAX APL Reference Manual*) to assign a password to an APL workspace.

Passwords are eight characters long. If you specify a password that is longer, it is truncated after the first eight characters; if you specify one that is shorter, it is padded on the right with blanks. For example, the following command assigns the password *SESAME* to the workspace *FOO*:

```
)SAVE FOO/PASSWORD=SESAME
```

## The VAX APL Operating Environment

### 1.8 APL Workspaces

The password *SESAME* is padded on the right with two blanks.

The characters that are valid in password names are the same as those that are valid in identifiers.

When a workspace has a password associated with it, you must specify that password before APL allows you to retrieve the workspace or copy objects from it. By default, a workspace has no password (the password is eight blanks).

#### 1.8.4 The CONTINUE Workspace

When you end an APL session with the *)CONTINUE* command, APL saves the active workspace in your default directory and names it *CONTINUE.APL*.

If files named *CONTINUE.APL* already exist in your directory, the new *CONTINUE* workspace will have a version number that is one higher than the next most recent version. Just like any inactive workspace, the *CONTINUE* workspace is an image of the active workspace as it existed when it was saved.

If a *CONTINUE* workspace exists in your default directory when you access APL, it is loaded as your active workspace (unless the command line or the initialization file specifically asks for another workspace).

#### 1.8.5 Groups

Selected user-defined operations and variables in a workspace can sometimes be easier to work with when they are treated as elements in a single logical collection, called a group. APL provides system commands that allow you to define groups, obtain a list of groups, list the members of a group, add members to or delete members from a group, and erase or disband a group. For details, see the *VAX APL Reference Manual*.

#### 1.8.6 The State Indicator

APL workspaces contain a status vector, known as the state indicator, which stores information about the execution of operations within the workspace. You can use the *)SI* system command (see the *VAX APL Reference Manual*) to list the contents of the state indicator. The list identifies suspended operations (user-defined functions or operators that have stopped executing for some reason) and pendent operations (user-defined functions or operators that have called other operations, and are waiting for them to complete).

If the state indicator has no value, no operations are currently suspended or pendent. For more information about the state indicator, see Chapter 3.

### 1.8.7 Workspace Size

The size of an APL workspace is dynamic; its maximum size is 2 million pages (512 bytes per page). Note, however, that the maximum size depends on your operating system resources—you may not be able to access APL’s limit of 2 million pages. The default maximum workspace size is 512 pages. You can use the `)MAXCORE` system command to change this value.

## 1.9 Interrupting APL

You can interrupt APL execution by entering any of the three forms of the attention signal:

- Ctrl/C—The weak attention signal. It means suspend execution of the current operation after executing the current statement and return control to immediate mode. During terminal output, this signal acts as a strong attention signal, immediately stopping the output. When this signal is used during a delay caused by the execution of `DL` (see the *VAX APL Reference Manual*), the delay is canceled but attention is not signaled.
- Ctrl/C Ctrl/C—The strong attention signal. It means suspend the current operation as soon as possible, even in the middle of the statement, and return control to immediate mode.
- Ctrl/Y—The panic exit. It means suspend the current operation immediately and give control to the operating system. After a panic exit, you can return to where you left off by executing the VMS operating system command `CONTINUE`. If you enter the panic exit while an operation is executing, the operation is suspended; if you then enter `CONTINUE`, the operation resumes execution at the point where it was interrupted.

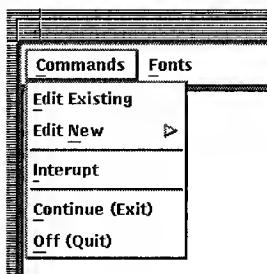
If you are using the DECwindows interface, you can use the the **Interrupt** option that is displayed if you click on **Commands** located in the menu bar of the transcript window. (See Figure 1–4.) The **Interrupt** option sends a Ctrl/C signal to the APL session.

The **abort input signal** allows you to escape to immediate mode when APL is waiting for input. The abort input signal is particularly useful when APL is executing `↓`, `↑`, or `↔` input, or when APL is in the `v` editor or super-edit mode. (You can also use the abort input signal in immediate mode if you do not want to enter a line that you have entered.) In all cases, APL cancels the current input request.

## The VAX APL Operating Environment

### 1.9 Interrupting APL

Figure 1–4 DECwindows Interface Commands Options



When you use the abort input signal to escape from a `□`, `■`, or `■` input that is inside an operation, the operation is suspended (unless it is locked), and APL returns you to immediate mode.

When you use the abort input signal to escape from the `■` editor, APL restores the operation to the definition it had before the edit session began. If you escape from super-edit mode, APL prompts you to begin entering the current line again.

Different terminal types form the abort input signal differently as follows:

Signal Form	Terminal Designator
<i>o</i> Backspace <i>U</i> Backspace <i>T</i>	APL, BIT,HDS201, HDS221, HDSAVT, KEY, LA, 4013, 4015
O Backspace U Backspace T	TTY
o Backspace u Backspace t	TTY
.OU or .ou	tty
Ctrl/D <i>o</i> <i>U</i>	VT220, VT240, VT320, VT330, VT340, DECTERM
Compose <i>o</i> <i>U</i>	VS
<i>o</i> Backspace(F12) <i>U</i>	DECterm using DECwindows interface

For terminals that form the abort input signal with *o* Backspace *U* Backspace *T*, you must enter the five keystrokes in the order shown, with no embedded spaces or tabs.

For a non-APL terminal, note that the dot (.) of .OU (or .ou) must appear in the first column of a line or must be preceded by a space. For example, in TTY mode:

```
"First, the .OU is part of a filetype
)OUTPUT DATA.OUT
      "Now, the .OU is the abort input signal
)OUTPUT DATA .OUT
51 INPUT ABORTED
      )OUTPUT DATA .OUT
          ^
```

For terminals that form the abort input signal with Ctrl/D *o* *U* or Compose *o* *U* the order in which you enter the *o* and *U* does not matter.

## 1.10 Ending an APL Session

You can end an APL session and return to operating system command level by entering an )OFF or )CONTINUE system command, or by typing Ctrl/Z. The )OFF and )CONTINUE system commands also give you the option of logging off without first returning to the DCL level. APL sessions using the DECwindows interface can also use the Continue (Exit) and Off (Quit) options to end a session. Click on the Commands option shown in the menu bar of the transcript window to display these options. (See Figure 1-4.)

## The VAX APL Operating Environment

### 1.10 Ending an APL Session

When you enter Ctrl/Z, APL responds as though there were an )OFF in the input line. Thus, Ctrl/Z works anywhere that the )OFF system command works. Anything that appears on the input line before the Ctrl/Z is executed before the session is terminated. For example:

```
1+1  ANOW ENTER CTRL/Z AFTER THIS COMMENT
2
TWA4: WEDNESDAY 23-JAN-1991 16:30:43.63
CONNECTED 00:01:21.62 CPU TIME 00:00:00.62
0 STATEMENTS 0 OPERATIONS
148 PAGE FAULTS 383 BUFFERED IO 32 DIRECT IO
$
```

Ctrl/Z works inside the v editor even during character-editing (superedit) mode:

```
vF
[1] C←2 3 30 42 77 53 75 38
[2] A←C+24
[3]           ⏐USER ENTERS CTRL/Z
EXIT
TWA4: WEDNESDAY 23-JAN-1991 16:30:43.63
CONNECTED 00:01:21.62 CPU TIME 00:00:00.62
0 STATEMENTS 0 OPERATIONS
148 PAGE FAULTS 383 BUFFERED IO 32 DIRECT IO
$
```

For more information about the )OFF and )CONTINUE commands, see the *VAX APL Reference Manual*.

---

#### Note

---

If you end an APL session by disconnecting a dialed-in terminal's telephone connection, the active workspace is lost unless your terminal is defined as a virtual terminal to the VMS operating system. For more information, see the *VMS DCL Dictionary*.

---

### 1.11 Character Sets

When you access APL, your terminal uses one of the character sets listed in the following subsections: APL key-paired, APL bit-paired, APL COMPOSITE, or TTY. Be careful not to confuse the character set your terminal uses with the character set the APL language uses internally (see Table 1-19 in Section 1.11.6).

### 1.11.1 Character Sets Used by APL Terminals

Most terminals that have an APL keyboard use either the key-paired (Table 1–13) or bit-paired (Table 1–14) character set. Digital LA, GIGI, and VT102 terminals use the key-paired character set. The HDS201, HDS221, HDSAVT, 4013, and 4015 terminals also use the key-paired character set.

**Table 1–13 APL-ASCII Key Pairing (Typewriter Pairing)**

decimal	00	16	32	48	64	80	96	112
0	NUL	DLE	SP	0	—	*	◊	P
1	SOH	DC1	“	1	α	?	A	Q
2	STX	DC2	)	2	⊥	ρ	B	R
3	ETX	DC3	<	3	∩	Γ	C	S
4	EOT	DC4	≤	4	L	~	D	T
5	ENQ	NAK	=	5	€	↓	E	U
6	ACK	SYN	>	6	—	υ	F	V
7	BEL	ETB	]	7	▽	ω	G	W
8	BS	CAN	∨	8	Δ	▷	H	X
9	HT	EM	^	9	ι	↑	I	Y
10	LF	SUB	≠	(	◦	⊂	J	Z
11	VT	ESC	÷	[	‘	←	K	{
12	FF	FS	,	;	□	↪	L	→
13	CR	GS	+	×		→	M	}
14	SO	RS	.	:	τ	≥	N	\$
15	SI	US	/	\	ο	—	O	DEL

**Table 1–14 APL-ASCII Bit Pairing**

decimal	00	16	32	48	64	80	96	112
0	NUL	DLE	SP	0	←	*	→	P
1	SOH	DC1	“	1	α	?	A	Q
2	STX	DC2	—	2	⊥	ρ	B	R
3	ETX	DC3	<	3	∩	Γ	C	S

(continued on next page)

## The VAX APL Operating Environment

### 1.11 Character Sets

Table 1–14 (Cont.) APL-ASCII Bit Pairing

decimal	00	16	32	48	64	80	96	112
4	EOT	DC4	≤	4	L	~	D	T
5	ENQ	NAK	=	5	€	†	E	U
6	ACK	SYN	≥	6	—	υ	F	V
7	BEL	ETB	>	7	ν	ω	G	W
8	BS	CAN	≠	8	Δ	▷	H	X
9	HT	EM	∨	9	ι	↑	I	Y
10	LF	SUB	)	]	◦	¤	J	Z
11	VT	ESC	(	[	‘	‘	K	¬
12	FF	FS	,	;	□	◊	L	\$
13	CR	GS	+	—		{	M	}
14	SO	RS	.	:	τ	×	N	÷
15	SI	US	/	\	◦	^	O	DEL

#### 1.11.2 Character Set Used by Non-APL Terminals

Terminals that cannot enter APL characters must use the TTY character set, which uses ASCII mnemonics to represent APL characters.

Note that when you enter lowercase letters from a non-APL terminal, APL converts them to uppercase letters (except in literals or comments). Unknown TTY mnemonics, however, are not changed to uppercase. For example, notice the error message returned when the user tries to load a file that does not exist:

```
)load myfile.1st
1 FILE NOT FOUND (FILE NOT FOUND)
)LOAD MYFILE.1ST
^
```

APL changed everything except .ls to uppercase because it did not recognize .ls as a character.

APL recognizes the TTY mnemonics for the characters { } ~ ' and | and for the lowercase letters (.JA – .JZ) only if your terminal is not capable of producing the actual characters or lowercase letters.

# The VAX APL Operating Environment

## 1.11 Character Sets

Backspaces are allowed on non-APL terminals, but the only permissible overstruck characters are characters overstruck with themselves or with spaces. If you attempt to enter any other overstrikes, APL signals *CHARACTER ERROR*.

Table 1–15 lists the TTY characters in alphabetical order and shows their ASCII and APL equivalents.

**Table 1–15 TTY Character Set**

TTY Set	Name	ASCII Set	APL Set	Characters to Combine
.AB	stile (ABsolute value)			
.AG	Accent Grave	'	'	0 /
.AL	ALpha		α	
.AP	AmPersand	&	&	3 /
.BX	quad (BoX)		□	
.CB	Column Backslash	\	\	-
.CC	Column Comma	,	,	-
.CE	CEiling	⌈	⌈	
.CF	CircumFlex	^	^	6 /
.CO	COntains	⊇	⊇	-
.CR	Column Reverse	⊖	⊖	-
.CS	Column Slash	/	/	-
.DA	Down Arrow	↓		
.DD	Dieresis	..		
.DE	base (DEcode)	⊥		
.DL	DeL	▽		
.DM	DiAMond	◊		
.DQ	Divide Quad	◻	◻	÷
.DU	Down U	□		
.EN	represent (ENcode)	⊤		
.EP	EPsilon	ε		
.FL	FLoor	L		

(continued on next page)

# The VAX APL Operating Environment

## 1.11 Character Sets

**Table 1–15 (Cont.) TTY Character Set**

TTY Set	Name	ASCII Set	APL Set	Characters to Combine
.FM	thorn (ForMat)	þ	Τ	◦
.GD	Grade Down	ψ	▽	
.GE	Greater than or Equal	≥		
.GO	right arrow (GO to)	→		
.GU	Grade Up	▲	▽	
.IB	I-Beam	▀	Τ	±
.IO	IOta	ι		
.IQ	Input Quad	▀	□	←
.JA – .JZ	lowercase letters	a – z	ⓐ – ⓝ	A – Z
.KA	Ctrl/A	SOH	Ã	A
.KB	Ctrl/B	STX	Ã	B
.KC	Ctrl/C	ETX	Ã	C
.KD	Ctrl/D	EOT	Ã	D
.KE	Ctrl/E	ENQ	Ã	E
.KF	Ctrl/F	ACK	Ã	F
.KG	Ctrl/G (Bell)	BEL	Ã	G
.KH	Ctrl/H (BackSpace)	BS	Ã	H
.KI	Ctrl/I (Horizontal Tab)	HT	Ã	I
.KJ	Ctrl/J (Line Feed)	LF	Ã	J
.KK	Ctrl/K (Vertical Tab)	VT	Ã	K
.KL	Ctrl/L (Form Feed)	FF	Ã	L
.KM	Ctrl/M (Carriage Return)	CR	Ã	M
.KN	Ctrl/N (Shift Out)	SO	Ã	N
.KO	Ctrl/O (Shift In)	SI	Ã	O
.KP	Ctrl/P	DLE	Ã	P
.KQ	Ctrl/Q	DC1	Ã	Q
.KR	Ctrl/R	DC2	Ã	R
.KS	Ctrl/S	DC3	Ã	S

(continued on next page)

# The VAX APL Operating Environment

## 1.11 Character Sets

**Table 1–15 (Cont.) TTY Character Set**

TTY Set	Name	ASCII Set	APL Set	Characters to Combine
.KT	Ctrl/T	DC4	¶	T
.KU	Ctrl/U	NAK	¤	U
.KV	Ctrl/V	SYN	¶	V
.KW	Ctrl/W	ETB	¤	W
.KX	Ctrl/X	CAN	¶	X
.KY	Ctrl/Y	EM	¶	Y
.KZ	Ctrl/Z	SUB	¶	Z
.LB	Left Brace	{	{	
.LD	delta (Lower Del)	Δ		
.LE	Less than or Equal	≤		
.LG	LoGarithm	⊗	○	*
.LK	Left tack	⊣		
.LO	circle (Large O)	○		
.LU	Left U	⌚		
.MT	MaTch	≡	=	-
.NE	Not Equal	≠		
.NG	high minus (NeGation)	-		
.NN	NaNd	^K	^	~
.NR	NoR	^K	∨	~
.NT	tilde (NoT)	~	~	
.OM	OMega	ω		
.OQ	Output Quad	◻	□	→
.OR	OR	∨		
.PC	PerCent sign	%	%	:
.PD	Protected Del	⋮	▽	~
.PS	Pound Sign	#	#	=
.QD	Quad Del	◻	□	▽
.QQ	Quote Quad	◻	□	'

(continued on next page)

## The VAX APL Operating Environment

### 1.11 Character Sets

Table 1–15 (Cont.) TTY Character Set

TTY Set	Name	ASCII Set	APL Set	Characters to Combine
.QU	double QUote	"	"	4 \
.RB	Right Brace	}	}	
.RK	Right tacK	-		
.RO	RhO	o		
.RU	Right U	»		
.RV	ReVerse	◊	○	
.SO	jot (Small O)	◦		
.SQ	Squish Quad	□	[ ]	
.SS	SubSet	≤	≤	—
.TR	TRanspose	¤	○	\
.UD	Underscored Delta	Δ	Δ	—
.US	UnderScore	—		
.UU	Up U	υ		
.WD	DEL (DELete)	DEL	§	8 \
.WE	Ctrl/[ (ESCAPE)	ESC	§	3 \
.WF	Ctrl/\	FS	↳	4 \
.WG	Ctrl/]	GS	§	5 \
.WN	Ctrl/@ (Null)	NUL	§	0 \
.WR	Ctrl/^	RS	§	6 \
.WU	Ctrl/_	US	✗	7 \
.XQ	hydrant (eXecute)	‡	⊥	◦
.ZA - .ZZ	underscored letters	A - Z	A - Z	—

#### 1.11.3 Composite Character Set

DIGITAL VT220, VT240, VT320, VT330, VT340, DECterms and VAXstations use the APL COMPOSITE character set listed in Table 1–16.

The VAX APL Operating Environment  
1.11 Character Sets

**Table 1–16 APL COMPOSITE Character Set**

dec	0	32	64	96	128	160	192	224
0	NUL	SP	@	'	unused	unused	◊	◊
1	SOH	!	A	¤	unused	“	¬	¤
2	STX	”	B	¤	unused	≤	△	¤
3	ETX	#	C	¤	unused	∨	¤	¤
4	EOT	\$	D	¤	IND	^	¤	¤
5	ENQ	%	E	¤	NEL	≠	¤	¤
6	ACK	&	F	¤	SSA	÷	¤	¤
7	BEL	'	G	¤	ESA	×	¤	¤
8	BS	(	H	¤	HTS	-	¤	¤
9	HT	)	I	¤	HTJ	¤	¤	¤
10	LF	*	J	¤	VTS	±	¤	¤
11	VT	+	K	¤	PLD	¤	¤	¤
12	FF	,	L	¤	PLU	¤	¤	¤
13	CR	-	M	¤	R1	€	¤	¤
14	SO	.	N	¤	SS2	¤	¤	¤
15	SI	/	O	¤	SS3	¤	¤	¤
16	DLE	0	P	¤	DCS	¤	¤	¤
17	DC1	1	Q	¤	PU1	¤	¤	¤
18	DC2	2	R	¤	PU2	¤	¤	¤
19	DC3	3	S	¤	STS	¤	¤	¤
20	DC4	4	T	¤	CCH	¤	¤	¤
21	NAK	5	U	¤	MW	¤	¤	¤
22	SYN	6	V	¤	SPA	¤	¤	¤
23	ETB	7	W	¤	EPA	↓	¤	¤
24	CAN	8	X	¤	unused	¤	¤	¤
25	EM	9	Y	¤	unused	¤	¤	¤
26	SUB	:	Z	¤	unused	¤	¤	¤
27	ESC	;	[	{	CSI	¤	¤	¤
28	FS	<	\		ST	¤	¤	¤

(continued on next page)

# The VAX APL Operating Environment

## 1.11 Character Sets

Table 1–16 (Cont.) APL COMPOSITE Character Set

dec	0	32	64	96	128	160	192	224
29	GS	=	] }		OSC	+	¤	¤
30	RS	>	^ ~		PM	→	¤	¤
31	US	?	—	DEL	APC	≥	¤	¤

Note that in column 1 dec is an abbreviation for decimal.

### 1.11.4 Digital Multinational Character Set

Table 1–17 shows the DIGITAL Multinational Character Set (MCS).

Table 1–17 Digital Multinational Character Set

dec	00	32	64	96	128	160	192	224
00	NUL	SP	@	'	unused	unused	À	à
01	SOH	!	A	a	unused	í	Á	á
02	STX	"	B	b	unused	¢	Â	â
03	ETX	#	C	c	unused	£	Ã	ã
04	EOT	\$	D	d	IND	unused	Ä	ä
05	ENQ	%	E	e	NEL	¥	Å	å
06	ACK	&	F	f	SSA	unused	Æ	æ
07	BEL	'	G	g	ESA	§	Ç	ç
08	BS	(	H	h	HTS	"	È	è
09	HT	)	I	i	HTJ	©	É	é
10	LF	*	J	j	VTS	¤	Ê	ê
11	VT	+	K	k	PLD	«	Ë	ë
12	FF	,	L	l	PLU	unused	Ì	ì
13	CR	-	M	m	R1	unused	Í	í
14	SO	.	N	n	SS2	unused	Î	î
15	SI	/	O	o	SS3	unused	Ï	ï
16	DLE	0	P	p	DCS	°	unused	unused
17	DC1	1	Q	q	PU1	±	Ñ	ñ

(continued on next page)

Table 1–17 (Cont.) Digital Multinational Character Set

dec	00	32	64	96	128	160	192	224
18	DC2	2	R	r	PU2	²	Ò	ò
19	DC3	3	S	s	STS	³	Ó	ó
20	DC4	4	T	t	CCH	unused	Ô	ô
21	NAK	5	U	u	MW	µ	Õ	õ
22	SYN	6	V	v	SPA	¶	Ö	ö
23	ETB	7	W	w	EPA	.	×	÷
24	CAN	8	X	x	unused	unused	Ø	ø
25	EM	9	Y	y	unused	¹	Ù	ù
26	SUB	:	Z	z	unused	²	Ú	ú
27	ESC	;	[	{	CS1	»	Û	û
28	FS	<	\		ST	¼	Ü	ü
29	GS	=	]	}	OSC	½	Ý	ý
30	RS	>	^	~	PM	unused	unused	unused
31	US	?	_	DEL	APC	¿	ß	unused

Note that in column 1 **dec** is an abbreviation for **decimal**.

### 1.11.5 ASCII Character Set

You can set most APL terminals to ASCII mode when you want to use the standard ASCII character set, then switch to APL mode when you want to use the APL character set. Table 1–18 shows the ASCII character set.

Table 1–18 ASCII Character Set

dec	00	16	32	48	64	80	96	112
0	NUL	DLE	SP	0	@	P	'	p
1	SOH	DC1	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3	#	3	C	S	c	s
4	EOT	DC4	\$	4	D	T	d	t

(continued on next page)

# The VAX APL Operating Environment

## 1.11 Character Sets

**Table 1–18 (Cont.) ASCII Character Set**

dec	00	16	32	48	64	80	96	112
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
10	LF	SUB	*	:	J	Z	j	z
11	VT	ESC	+	;	K	[	k	{
12	FF	FS	,	<	L	\	l	
13	CR	GS	-	=	M	]	m	}
14	SO	RS	.	>	N	^	n	~
15	SI	US	/	?	O	_	o	DEL

Note that in column 1 **dec** is an abbreviation for **decimal**.

### 1.11.6 Elements of $\Box{AV}$

$\Box{AV}$  contains a vector of the 256 characters known to APL. Table 1–19 shows the characters and their positions in the vector based on an index origin of 0.

**Table 1–19 Elements of  $\Box{AV}(\Box{IO+0})$**

dec	0	32	64	96	128	160	192	224
0	NUL	SP	-	◊	'	Δ	✉	□
1	SOH	“	α	Α	α	Α	✉	□
2	STX	)	ι	Β	β	Β	✉	□
3	ETX	<	η	C	c	₵	□	□
4	EOT	≤	λ	D	d	₵	□	□
5	ENQ	=	ε	E	e	Ⓔ	Ⓐ	□
6	ACK	>	-	F	f	Ⓕ	ߵ	□
7	BEL	▮	߻	G	g	Ⓖ	ߵ	□
8	BS	߻	߻	H	h	Ⓗ	߻	□

(continued on next page)

The VAX APL Operating Environment  
1.11 Character Sets

**Table 1–19 (Cont.) Elements of  $\square AV(\square IO \leftarrow 0)$**

dec	0	32	64	96	128	160	192	224
9	HT	^	ι	I	i	<u>I</u>	*	□
10	LF	≠	◦	J	j	<u>J</u>	⊗	□
11	VT	÷	‘	K	k	<u>K</u>	ϕ	□
12	FF	,	□	L	l	<u>L</u>	¤	□
13	CR	+		M	m	<u>M</u>	¤	□
14	SO	.	τ	N	n	<u>N</u>	‐	□
15	SI	/	◦	O	o	<u>O</u>	†	□
16	DLE	0	*	P	p	<u>P</u>	χ	□
17	DC1	1	?	Q	q	<u>Q</u>	≤	□
18	DC2	2	ρ	R	r	<u>R</u>	≥	□
19	DC3	3	Γ	S	s	<u>S</u>	≡	□
20	DC4	4	~	T	t	<u>T</u>	^	□
21	NAK	5	↓	U	u	<u>U</u>	□	□
22	SYN	6	υ	V	v	<u>V</u>	□	□
23	ETB	7	ω	W	w	<u>W</u>	□	□
24	CAN	8	▷	X	x	<u>X</u>	□	□
25	EM	9	↑	Y	y	<u>Y</u>	□	□
26	SUB	(	◁	Z	z	<u>Z</u>	□	□
27	ESC	[	↔	{	@	:	□	□
28	FS	;	¬	¬	“	„	□	□
29	GS	×	♂	}	#	‡	□	□
30	RS	:	≥	\$	%	‡	□	□
31	US	\	-	DEL	&	¤	□	□

The index of a character in  $\square AV$  is the sum of its row and column numbers.

Note that in column 1 **dec** is an abbreviation for **decimal**.



# 2

## VAX APL Language Concepts

In the VAX APL language, you form expressions by applying functions to arrays, and the APL interpreter evaluates the expressions. An **array** is a collection of one or more data elements called items. An **item** may be a scalar data element, or it may include many data elements. An item may be numeric, character, or a combination of the two.

This chapter discusses arrays, describes the syntax of APL expressions, and explains how expressions are evaluated by the APL interpreter.

### 2.1 Array Types

There are two types of arrays in APL: simple and enclosed. In a simple array, each item is a single data element. In an enclosed array, one or more items are an array. For example:

```
          ⎕CREATE B, SIMPLE ARRAY, 5 SCALAR ITEMS
B←23 15 9 83 99
23 15 9 83 99
          ⎕CREATE C, ENCLOSED ARRAY, 5 ITEMS
C←(2 4 98) 7 88 29
13 +---+ 7 88 29
| 2 4 98 |
+---+
```

There are two types of data elements: character and numeric. Any given array may contain either or both of these types. When the items of an array are entirely character or entirely numeric, the array is called **homogeneous**. When the items are a mixture of character and numeric data elements, the array is called **heterogeneous**. For example:

## VAX APL Language Concepts

### 2.1 Array Types

```
      ⎕CREATE SIMPLE HOMOGENEOUS
      ⎕B←23 15 9 83 99
      23 15 9 83 99
      ⎕CREATE SIMPLE HETEROGENEOUS
      ⎕C←'P' 31 'N' 1 'C' 'D'
      P 31 N 1 CD
      ⎕CREATE ENCLOSED HOMOGENEOUS
      ⎕D←'X' 'APPLE' 'L' ('N' 'E' 'S')
      X +---+ L +---+
      |APPLE| |NES|
      +---+ +---+
      ⎕CREATE ENCLOSED HETEROGENEOUS
      ⎕F←'P' (31 1 24) 'N' 1 ('CRIB')
      P +-----+ N 1 +---+
      |31 1 24| |CRIB|
      +-----+ +---+
```

A character array may include any value from the atomic vector returned by  $\square AV$  (the niladic system function whose value is the 256 characters known to APL; see the *VAX APL Reference Manual*.) To designate that an array constant is of type character, you enclose it in single quotation marks. For more information on character arrays, see Section 2.5.3.2.

The numeric data type can be subdivided into the following:

- Boolean—a 1 or a 0.
- Integer—the positive and negative integers and zero.
- Near-integer—a number equal to an integer within the tolerance defined by  $\square CT$  (see the *VAX APL Reference Manual*).
- Floating-point—the integers and real numbers.

Although you cannot control the internal precision of numeric representation, you do have some control over numeric output representation. The  $\square PP$  system variable (see the *VAX APL Reference Manual*) allows you to specify the output precision of floating-point numbers, and you can use the APL format functions ( $\square FMT$  and  $\ast$ ) to control the output precision of specific arrays.

For more information on numeric arrays, see Section 2.5.3.1.

### 2.2 Array Structure

Arrays are also characterized by their structure. The structure of an array is the way in which the array's items are arranged. Specifically, an array's structure is defined by three properties: rank, shape, and depth.

### 2.2.1 Rank of an Array

The **rank** of an array is determined by the number of axes (sometimes called dimensions or coordinates) along which its items are arranged. For practical purposes, there is no intrinsic limit on the number of axes in an APL array. As long as the size of the array does not exceed the size of your workspace, you can have up to 65,535 axes. The special terms associated with arrays of rank 0, 1, and 2 are, respectively, scalar, vector and matrix.

A rank 0 array, or **scalar**, is a single numeric or character value that has zero axes (thus, it cannot be indexed). APL considers any of the following, when entered, to be scalars:

```
1
32.28
'A'
99999
'Z'
'0'
'5'
5
```

It is possible for a single data element to have a structure, that is, one or more axes. In this case, the item is not a scalar (which never has an axis) but a **singleton**. For example, any of the values in the preceding list could be converted to nonscalar singletons with the reshape ( $\rho$ ) or the ravel (, ) function. The following subsections include more information on singleton arrays.

A rank 1 array, or **vector**, consists of any number of numeric or character values arranged along one axis. A singleton vector is a vector containing only one value. The following are all vectors:

```
1 2 3 4 5
'VECTOR'
23 197 6 2543 14 29 11 2
23 +-----+ +-----+
|197 6 2543| |14 29 11 2|
+-----+ +-----+
'THIS IS A CHARACTER STRING'
```

Note that the items in a numeric vector must be separated by at least one space. The spaces are not part of the vector, but they are necessary in order to delimit the end of one item and the beginning of the next. In a character vector, as in any character array, spaces that are embedded between quotation marks are part of the vector.

## VAX APL Language Concepts

### 2.2 Array Structure

A rank 2 array, or **matrix**, consists of any number of numeric or character values arranged along two axes, commonly called rows and columns. A singleton matrix is a matrix containing one row and one column. The following matrix has three rows, and four columns:

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

In an array of rank 3 or more, the leading axes are known as planes. For example, the following array has a rank of 3. There are two planes, three rows, and four columns. Note that the planes are separated by a blank line.

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

The rank of an enclosed array is not affected by the rank of any of the individual items. In the following example, *R* is an enclosed vector with length 3. The first item is a scalar (rank 0); the second item is a matrix (rank 2); and the third item is a vector (rank 1). The rank of *R* is 1. Note that the enclosed items of *R* are created with a combination of parentheses and the enclose function (*c*).

```
□←R+317,(c 3 5p'ALPHABET SOUP'),c3 28 317  
317 +-----+ +-----+  
|ALPHA| |3 28 317|  
|BET | +-----+  
|SOUP |  
+-----+  
      pR          □QUERY FOR SHAPE OF R  
3  
      p oR         □QUERY FOR RANK OF R  
1
```

#### 2.2.2 Shape of an Array

The shape of an array is the number of items along each of its axes. An array is known as an empty array if the length of any of its axes is 0.

Scalars do not have a shape because they have no axes. Vectors have one axis, and the shape (also known as the length) is represented by a single value. For example:

## VAX APL Language Concepts

### 2.2 Array Structure

```
1 2 3 4 5          aLENGTH = 5
'VECTOR'           aLENGTH = 6
23 197 6 2543 14 29 11 2    aLENGTH = 8
23 (197 6 2543) (14 29 11 2) aLENGTH = 3
'THIS IS A CHARACTER STRING' aLENGTH = 26
```

Matrices have two axes, and the shape is expressed as two values. The first value is the number of the rows. The second value is the number of the columns. As the following example shows, a matrix is always rectangular; the product of the number of rows times the number of columns equals the total number of items in the array.

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

Singletons can have any number of axes, but none of the axes can have a length other than 1. To reveal the shape of a singleton, you would use the shape ( $\rho$ ) function, which is discussed briefly in the next section.

The shape of an enclosed array is not affected by the shape of any of the individual items. In the following example,  $R$  is an enclosed vector array. The first item is a scalar (no shape); the second item is a matrix (shape 3 5); and the third is a vector (shape 3). The shape of  $R$  is 3. To show the shapes of the individual items of  $R$ , the example uses the each ("") operator with the shape ( $\rho$ ) function as a left operand. The "" operator applies its operand to each item in  $R$ .

```
□←R+317,(c 3 5ρ'ALPHABET SOUP'),c3 28 317
317 +---+ +-----+
|ALPHA| |3 28 317|
|BET  | +-----+
|SOUP |
+---+
ρR          aQUERY FOR SHAPE OF R
3
ρ"R          aQUERY FOR SHAPE OF EACH ITEM IN R
++ +---+ +-+
|| |3 5| |3|
++ +---+ +-+
```

## VAX APL Language Concepts

### 2.2 Array Structure

#### 2.2.2.1 Shape and Reshape Functions

The monadic  $\rho$  function (known as shape) takes an array as its argument and displays the current shape of the array. The display is a vector that contains one number for each axis in the array. The values of the numbers represent the lengths of the axes.

The dyadic  $\rho$  function (known as reshape) builds an array of a specified shape. The left argument determines the new shape vector. The right argument contains the items that will populate the new array. When you use the  $\rho$  function for this purpose, you are *reshaping* the array.

Scalars do not have a shape because they have no axes. If you query for the shape of an array that is scalar, APL returns an empty numeric vector, as follows:

```
B←5      ⍝CREATE B, A NUMERIC SCALAR
⍴B      ⍝QUERY FOR THE SHAPE OF B
        (APL outputs a blank line)
C←'A'
⍴C      ⍝CREATE C, A CHARACTER SCALAR
        ⍝QUERY FOR THE SHAPE OF C
        (APL outputs a blank line)
```

Vectors do have a shape because they have one axis. The value of the result is the length of the axis as follows:

```
B←5 3 1    ⍝CREATE B, A 3-ITEM VECTOR
⍴B      ⍝QUERY FOR THE SHAPE OF B
3
```

Matrices have two axes, and the shape vector contains two values as shown in the following example. The first value is the number of the rows. The second value is the number of the columns.

```
⍝CREATE M, A MATRIX OF 3 ROWS AND 7 COLUMNS
⍝NOTE SPACE CHARACTER IN 2ND ROW 7TH COLUMN
M←3 7⍪'GEORGIE PORGIE EATSPIE'
GEORGIE
PORGIE
EATSPIE
⍴M      ⍝QUERY FOR SHAPE OF M
3 7
```

An array arranged in three planes would have a shape vector of three values. For example, an array with three planes, four rows, and five columns has the following shape:

```
3 4 5
```

## VAX APL Language Concepts

### 2.2 Array Structure

A singleton can have any number of axes, but none of the axes can have a length other than 1. This means that a value that appears to be a scalar can be multidimensional. For example:

```
162 ⎵←J←1⍴162      ⋊CREATE J, A VECTOR SINGLETON
1       ⍵J          ⋊QUERY FOR SHAPE OF J
1
A       ⎵←K←1 1⍴'A'  ⋊SHAPE VECTOR HAS 1 VALUE
          ⋊CREATE K, MATRIX SINGLETON
1       ⍵K          ⋊QUERY FOR SHAPE OF K
1 1
8       ⎵←M←1 1 1⍴8  ⋊SHAPE VECTOR HAS 2 VALUES
          ⋊CREATE M, RANK 3 SINGLETON
1       ⍵M          ⋊QUERY FOR SHAPE OF M
1 1 1
          ⋊SHAPE VECTOR HAS 3 VALUES
```

Because the monadic shape ( $\rho$ ) function returns a vector that contains one number for each axis in the array, the shape of the shape of an array ( $\rho \rho$ ) returns the number of axes, or the rank, of the array. For example:

```
1       ⋊ASSIGN AND DISPLAY A MATRIX G
1 2 3
4 5 6
7 8 9
3       ⍵G          ⋊QUERY FOR SHAPE OF G
3 3
2       ⋊⍳G          ⋊QUERY FOR THE RANK OF C
5
5       ⋊S←5          ⋊ASSIGN AND DISPLAY A SCALAR S
0       ⍵S          ⋊QUERY FOR THE SHAPE OF S
          (APL outputs a blank line)
0       ⍵⍳S          ⋊QUERY FOR THE RANK OF S
```

## VAX APL Language Concepts

### 2.2 Array Structure

The following examples use reshape (dyadic  $\rho$ ):

```
V←'RANK2ARRAY'  
2 5 ρ V  
RANK2  
ARRAY  
H←4 3ρ↑12  
ρ H  
4 3  
[]←H←3 4 ρ H  
1 2 3 4  
5 6 7 8  
9 10 11 12  
ρ H  
3 4  
      ↗NEW SHAPE OF H
```

When displaying arrays that have three or more axes, APL inserts one blank line between each plane and one additional blank line for each additional axis. For example:

```
A←'ABCDEFGHIJKLMNPQRSTUVWXYZ12345'  
2 2 2 4ρA  
ABCD  
EFGH  
  
IJKL  
MNOP  
  
QRST  
UVWX  
  
YZ12  
345A
```

For more information on the  $\rho$  functions, see shape and reshape in the *VAX APL Reference Manual*.

#### 2.2.3 Depth of an Array

Depth refers to the levels of nesting that occur in an array. A simple array has one level of nesting (zero if the array is a simple scalar). An enclosed array has a depth of at least two.

The monadic  $\equiv$  function (known as depth) takes an array as the argument and returns an indicator of the deepest level of nesting among all items of the array. (The depth function is described in greater detail in the *VAX APL Reference Manual*.) For example:

```

9      ⌊<B<9           ⋀SIMPLE SCALAR
0      ⌊B                 ⋀DEPTH OF B
0
1      ⌊<C< WHERE ARE YOU GOING? ⋀SIMPLE VECTOR ARRAY
WHERE ARE YOU GOING?
1      ⌊C                 ⋀DEPTH OF C
1
1      ⌊<D<2 10⍴C     ⋀SIMPLE MATRIX ARRAY
WHERE ARE
YOU GOING?
1      ⌊D                 ⋀DEPTH OF D
1
1      ⌊<E<1 (5 6 7) 11 12 ⋀ENCLOSED ARRAY, 1 NESTING LEVEL
1 +----+ 11 12
|5 6 7|
+----+
1      ⌊E                 ⋀DEPTH OF E
2
2      ⌊<F<1 (5 6 7 (8 9 10)) 11 12 ⋀ENCLOSED ARRAY WITH MORE NESTING
1 +-----+ 11 12
|5 6 7 +---+ |
| |8 9 10|| |
| +---+ |
+-----+
1      ⌊F                 ⋀DEPTH OF F
3

```

## 2.2.4 Shape Domains of Primitive Function Arguments

Many APL primitive functions restrict their argument domain to arrays of a particular shape. They may require arguments to be:

- Singletons—1-item arrays of any rank (note that this includes scalars); can be thought of as the scalar domain.
- In the vector domain—a singleton or a vector.
- In the matrix domain—a singleton, a vector, or a matrix.

The following are formal definitions for the scalar, vector, and matrix domains:

**Scalar domain:**  $(0 \geq \rho \rho ARG) \vee 1 = \times / \rho ARG$

**Vector domain:**  $(1 \geq \rho \rho ARG) \vee 1 = \times / \rho ARG$

**Matrix domain:**  $(2 \geq \rho \rho ARG) \vee 1 = \times / \rho ARG$

## VAX APL Language Concepts

### 2.2 Array Structure

From these, a more general definition may be induced:

$N$ -array domain:  $(N \geq \rho \rho ARG) \vee = \times / \rho ARG$

where  $N$  is the rank of the domain to be defined.

### 2.3 Scalar Product and Singleton Extension

Many **dyadic functions**, functions with two arguments, require the shapes of their arguments to conform to each other in some way. For example, the scalar functions (such as  $+ - \times \div$ ) require that the left and right arguments have the same shape. When you execute a scalar function, APL applies the function over each corresponding pair of items. This process is known as **scalar product**.

```
          ⋀SCALAR FUNCTIONS EXTEND OVER
3 8 + 7 12      ⋀ EACH SUCCESSIVE PAIR OF ITEMS
                 ⋀ THIS IS A SCALAR PRODUCT
```

Scalar product is applied pervasively , that is, at all depths (levels of nesting) of an array. For example:

```
B←2 3 (4 5 6 (1 4 2)) 5 (8 8)      ⋀ ENCLOSED VECTOR
C←5 1 (2 9 2 (4 0 2)) 1 (0 7)      ⋀ C = SAME SHAPE
B                                         ⋀ DISPLAY B
2 3 +-----+ 5 +---+
| 4 5 6 +---+ |  | 8 8 |
|             |  | +---+
|             |  |
|             |  |
+-----+  +---+
C                                         ⋀ DISPLAY C
5 1 +-----+ 1 +---+
| 2 9 2 +---+ |  | 0 7 |
|             |  | +---+
|             |  |
+-----+  +---+
B + C                                     ⋀ ADD APPLIED AT ALL DEPTHS
7 4 +-----+ 6 +---+
| 6 14 8 +---+ |  | 8 15 |
|             |  | +---+
|             |  |
+-----+  +---+
```

Some of the dyadic functions permit one argument to be in the singleton domain (either a singleton or a scalar) while the other argument has a different shape. APL reshapes the singleton (or scalar) to conform to the shape of the other argument. This reshaping is known either as **singleton extension** or **scalar extension**. For example:

## VAX APL Language Concepts

### 2.3 Scalar Product and Singleton Extension

```
      ⌊EXAMPLE OF SINGLETON EXTENSION
      ⌋B←1          ⌋CREATE AND DISPLAY B, A SCALAR
1
      ⌋C←2 2⍴4      ⌋CREATE AND DISPLAY C, A MATRIX WITH SHAPE 2 2
1 2
3 4
      B + C        ⌋APL RESHAPES B TO CONFORM TO C
2 3
4 5
```

Singleton extension is applied pervasively at all depths of an array. For example:

```
      ⌋B←2 3 (4 5 6 (1 4 2)) 5 (8 8)      ⌋CREATE B, AN ENCLOSED VECTOR
2 3 +-----+ 5 +---+
| 4 5 6 +----+ | 8 8 |
|   | 1 4 2 || +---+
|   +----+ |
+-----+
      D←2          ⌋D IS A SCALAR
      B + D        ⌋D GETS RESHAPED, THEN ADDED
4 5 +-----+ 7 +----+
| 6 7 8 +----+ | 10 10 |
|   | 3 6 4 || +----+
|   +----+ |
+-----+
```

When the argument to be extended is a singleton, APL makes the singleton a scalar (by removing all of its axes) and then reshapes the scalar. For example:

```
(1 1 1⍴1) + 2 2⍴4 ⌋SINGLETON PLUS MATRIX
```

becomes

```
1 + 2 2⍴4           ⌋SCALAR PLUS MATRIX
```

and finally

```
(2 2⍴1) + 2 2⍴4 ⌋MATRIX PLUS MATRIX
```

If both arguments are singletons, APL reshapes them in a way that satisfies the particular function's conformance rules. In the following example, the  $\phi$  function (known as rotate) requires the left argument to have a rank that is one less than the right argument:

```
(1 1 1⍴2) ϕ 1 1 1⍴3 ⌋TWO SINGLETONS
```

becomes

```
(1 1⍴2) ϕ 1 1 1⍴3 ⌋FUNCTION'S CONFORMANCE RULES SATISFIED
```

## VAX APL Language Concepts

### 2.3 Scalar Product and Singleton Extension

If both arguments are singletons, but the particular function has no applicable conformance rule, the result is a singleton whose rank is the larger of the argument ranks. For example:

```
(1 1p2) + 1 1 1p3  aTWO SINGLETONS WITH DIFFERENT RANKS
```

becomes

```
(1 1 1p2) + 1 1 1p3  aSMALLER RANK ARGUMENT CONFORMS TO LARGER
```

and the result is

```
1 1 1p5          aRESULT RANK CONFORMS TO LARGER ARGUMENT
```

Note that singleton extension is not limited to the scalar functions; any mixed function whose argument domain permits a singleton shape will perform the extension. For example, the monadic index generator (`i`) takes a singleton argument and returns a vector regardless of the number of the argument's axes.

```
aEXAMPLE OF SINGLETON AND
a  A MIXED PRIMITIVE FUNCTION
S←5           aS IS A SCALAR
T ← 1 1pS      aT IS A MATRIX SINGLETON
U ← 1 1 1 1 1pS  aU IS A RANK 5 SINGLETON
                  aALL 3 ARGUMENTS RESHAPED
                  a  BY SINGLETON EXTENSION
iS
1 2 3 4 5
iT
1 2 3 4 5
iU
1 2 3 4 5
```

## 2.4 Empty Arrays

**Empty arrays** are arrays that have shape, depth, and type, but have no items. APL often returns an empty array—displayed as a blank line—when no other result is appropriate. For example, the shape of a scalar is an empty vector:

```
W←p5          aQUERY FOR SHAPE OF A SCALAR AND ASSIGN
W              aDISPLAY EMPTY
                  (APL outputs a blank line)
pW            aQUERY FOR SHAPE OF THE EMPTY ARRAY
0
p pW          aQUERY FOR RANK
1
```

Note that the rank of this empty array is 1. Empty arrays may have any rank except 0. They may be simple or enclosed, and their type may be homogeneous or heterogeneous. Generally, the type of an empty array is disregarded. However, in some expressions, the type of an empty array can determine the type of the result (even an empty result). This applies to the following operations:

catenate ( , )	replication ( <i>A</i> / )
disclose ( $\triangleright$ )	reshape ( $\rho$ )
drop ( + )	reverse ( $\phi$ )
enclose ( $\subset$ )	take ( $\dagger$ )
expansion ( <i>A</i> \ )	transpose ( $\wp$ )
first ( $\dagger$ )	union ( $\cup$ )
indexing ([ <i>K</i> ])	unique ( $\cup$ )
intersection ( $\cap$ )	without ( $\sim$ )
ravel ( , )	

For the catenate ( , ), union (  $\cup$  ), intersection (  $\cap$  ), without (  $\sim$  ), and indexing ([*K*]) operations, the left argument determines the type of the result; for the other functions, the right argument is the controlling argument. For example:

```
(10) , ''  $\leftrightarrow$  10      @LEFT ARG IS NUMERIC, EMPTY IS NUMERIC
'' , 10  $\leftrightarrow$  ''      @LEFT ARG IS CHARACTER, EMPTY IS CHARACTER

@10  $\leftrightarrow$  10      @RIGHT ARG IS NUMERIC, EMPTY IS NUMERIC
@''  $\leftrightarrow$  ''      @RIGHT ARG IS CHARACTER, EMPTY IS CHARACTER

1 @ 10  $\leftrightarrow$  10      @RIGHT ARG IS NUMERIC, EMPTY IS NUMERIC
1 @ ''  $\leftrightarrow$  ''      @RIGHT ARG IS CHARACTER, EMPTY IS CHARACTER
```

The rule is particularly significant for the take (  $\dagger$  ), disclose (  $\triangleright$  ), expansion (*A* \ ), and replication (*A* / ) operations, which produce fill items based on the prototypes (described in Section 2.4.1) of their right arguments:

```
0 \ 10  $\leftrightarrow$  0
0 \ ''  $\leftrightarrow$  ' '
1 -2 0 2/1 2 3  $\leftrightarrow$  1 0 0 3 3      @NUMERIC FILL ITEMS (ZEROS)
1 -2 0 2/'ABC'  $\leftrightarrow$  'A'  CC'      @CHARACTER FILL ITEMS (BLANKS)

1 ^ 10  $\leftrightarrow$  0
1 ^ ''  $\leftrightarrow$  ' '
```

## VAX APL Language Concepts

### 2.4 Empty Arrays

In all other cases, the type of an empty array is disregarded, and the result type is the type of the defined result domain of the function. For example:

```
' + 10 ↔ 10          @BECAUSE THE RESULT OF + IS NUMERIC  
' ⌊1 ↔ (10) ⌊1 ↔ 1      @NOTE: ⌊10 ↔ 1
```

You may find empty arrays useful when you write user-defined operations. For instance, you can use empty arrays in conditional branching or to initialize an array (see Chapter 3).

#### 2.4.1 Array Prototypes

During certain operations, APL inserts fill items as it builds a new array. APL defines the shape and kind of fill items based on the prototype of an array. The **prototype** of an array *B* is an array with the same shape as the first item in *B*. The contents of the prototype are character blanks in positions corresponding to characters and numeric zeros in positions corresponding to numbers. You can determine the prototype of an array with the expression  $\text{↑}0\text{p}B$ . For example:

```
⍴+LIS←12 13 15  
12 13 15          @FIRST ITEM = SIMPLE NUMERIC SCALAR  
↑0pLIS           @SHOW THE PROTOTYPE OF LIS  
0  
⍴+AL←'ACE'  
ACE              @FIRST ITEM = SIMPLE CHARACTER SCALAR  
↑0pAL            @PROTOTYPE OF AL IS A BLANK CHARACTER  
  
GIE←(2 2p'Y'3 5'X') 352 'ABC'  
GIE              @DISPLAY GIE, A 3-ITEM VECTOR  
+---+ 352 +---+  
| Y 3|      |ABC|  
| 5 X|      +---+  
+---+  
0              @FIRST ITEM = ENCLOSED, HETEROGENEOUS MATRIX  
↑0pGIE          @SHOW THE PROTOTYPE OF GIE  
0
```

## 2.4.2 Fill Items in Arrays

A **fill item** is an array (consisting of spaces, zeros, or a combination of the two) that APL inserts into another array. A **fill element** is a scalar data element inside a fill item. There are two kinds of fill elements: character blanks and numeric zeros. The prototype (described in Section 2.4.1) of an array determines the shape of a fill item and the kind of each element in the fill item.

The derived function expand ( A\B ) is an example of an operation where APL inserts fill items into an array. For example:

```
LIS←12 13 15
↑0pLIS      ⋊SHOW THE PROTOTYPE OF LIS
0
    Ⓛ←BOO←1 0 1 0 1
1 0 1 0 1
    ⋊ZEROS IN BOO DECIDE LOCATION OF FILL ITEMS
    BOO\ LIS      ⋊EXPAND LIS
12 0 13 0 15
```

In the preceding example, the shape of the fill item is a simple scalar and the fill element is a numeric zero. In the next example, the fill item is again a simple scalar, but the fill element is a character blank.

```
Ⓐ←AL←'ACE'
ACE
↑0pAL      ⋊PROTOTYPE OF AL IS A CHARACTER BLANK
BOO←1 0 1 0 1
    BOO\ AL      ⋊EXPAND AL, FILL ITEMS ARE BLANKS
A C E
```

In the next example, the shape of the fill item is an enclosed matrix of shape 2 2 and the fill elements are heterogeneous (a combination of zeros and blanks).

```
GIE←(2 2⍩'Y'3 5'X') 352 'ABC'
↑0pGIE      ⋊SHOW THE PROTOTYPE OF GIE
0
    BOO←1 0 1 0 1
    BOO\ GIE      ⋊EXPAND GIE, NOTE SHAPE/KIND OF FILL ITEMS
+---+ +---+ 352 +---+ +---+
| Y 3 | | 0|      | 0| | ABC|
| 5 X | | 0|      | 0| +---+
+---+ +---+      +---+
```

## VAX APL Language Concepts

### 2.4 Empty Arrays

Expand is not the only operation that requires APL to insert fill characters. There is also the derived function replicate ( $A/B$ ), the take ( $\uparrow$ ) and disclose ( $\downarrow$ ) primitive functions, and the  $\Box$ ,  $\EXP$ , and  $\REP$  system functions.

Note that APL always uses scalar blanks as the fill items for  $\Box$  because the  $\Box$  argument is always a simple character array. (Expand, replicate, take, disclose,  $\Box$ ,  $\EXP$ , and  $\REP$  are described in the *VAX APL Reference Manual*.)

## 2.5 APL Expressions

The line is the basic unit of work in APL. It consists of one or more statements which, in turn, consist of one or more expressions.

An APL expression is:

- A variable or constant standing alone.
- A function and its arguments (arguments are represented by variables, constants, or expressions).
- An operator and its operands (operands are represented by variables, constants, functions, or expressions) and the arguments to its derived function.
- An expression enclosed inside of parentheses.

The following sections define identifiers, variables, and constants, explain the use of spaces in expressions, describe how APL evaluates expressions, and then discuss APL lines and comments.

### 2.5.1 Identifiers

APL has several kinds of **identifiers**:

- Variable names—Names that represent values that can be changed.
- Label names—Names that represent line numbers in user-defined operations (see Chapter 3).
- System variable and system function names—Names that are predefined and begin with the quad symbol ( $\Box$ ).
- User-defined function or user-defined operator names—Names you assign to programs you write (see Chapter 3).
- Group names—Names that represent collections of names (see the *VAX APL Reference Manual*).

The rules for forming all but the system identifiers are as follows:

- The maximum length is 31 characters. Identifiers longer than 31 characters are truncated on the right without any message.
- Allowable characters include *A* through *Z*, *A* through *Z*, *a* through *z*, *\_*, *Δ*, and 0 through 9. Embedded spaces are not allowed.
- The first character must be *A* through *Z*, *A* through *Z*, *a* through *z*, *Δ*, or *Δ*; in other words, the first character may not be 0 through 9 or *\_*.

For example:

Legal Identifiers	Illegal Identifiers
ABC63b8	1AC75 (Begins with a number)
Δ74	Z 9436 (Contains an embedded space)
ΔG956HΔ	Pv742B (Contains invalid character v)

Note that you can use upper and lower case alphabetics interchangeably when forming identifiers. APL recognizes the lower case characters and converts them to upper case.

## 2.5.2 Wildcards

When you specify an identifier as an argument in a system command or system function, you can often substitute the \* and ÷ wildcards for all or part of the identifier's name. The star (\*) symbol represents 0 or more characters, and the divide (÷) symbol represents a single character. If you use the TTY character set, use the asterisk (\*) and percent (%) symbols, respectively.

For example, if you specify a variable *TR\**, you refer to all variables of any length that begin with the letters *TR*. If you specify a variable *TRAC÷*, you refer to all variables that begin with the letters *TRAC* and end with one additional character.

The system commands and system functions that use identifiers with wildcards in their arguments are as follows:

)COPY	)NMS	□QCO
)ERASE	)OPS	□QPC
)FNS	)PCOPY	
)GROUP	)VARS	
)GRPS		

For more details about the \* and ÷ wildcards, see the *VMS DCL Dictionary*.

## VAX APL Language Concepts

### 2.5 APL Expressions

#### 2.5.3 Constants

A **constant** is a numeric or character item whose value is literally the constant itself; it is not a symbol for some other value.

##### 2.5.3.1 Numeric Contants

A **numeric constant** is either of the following:

- One or more decimal digits with an optional decimal point.
- A decimal quantity followed by *E* and the power of 10 by which the quantity is to be multiplied (no embedded spaces are allowed).

For example, all the following constants are valid representations of the same value:

712        712.0        7120E<sup>-1</sup>        07.12E2

Whenever possible, APL prints numbers without decimal points and exponents:

```
□←A←712 712.0 7120E-1 07.12E2  
712 712 712 712
```

To represent a negative number in APL, you enter a numeric constant preceded by a negative sign (<sup>-</sup>). Embedded spaces are not permitted. Note that the negative sign, or high minus sign, is not the same character as the minus sign (-), which is used to indicate the subtraction function.

APL signals an error if a constant is not well-formed:

```
1E.  
7 SYNTAX ERROR (ILL FORMED NUMERIC CONSTANT)  
1E.  
^  
1E99  
27 LIMIT ERROR  
1E99  
^  
1B1              EVALUATED AS 1 B1  
11 VALUE ERROR  
1B1              EVALUATED AS 1 B1  
^
```

### 2.5.3.2 Character Constants

A character constant is one or more characters from the atomic vector  $\Box V$  (including spaces, carriage returns, line feeds, control characters, and so on) enclosed in quotation marks. For example:

```
'ABCDEFG'  
'((10V)=V1V)/V'  
'THIS IS A CONSTANT.'  
'12345'
```

When APL prints a character constant, it omits the enclosing quotation marks. If you want APL to output a quotation mark, enter one extra quotation mark next to the one you want printed. Do not precede the extra quotation mark with a space. If you do, you create two separate items instead of a single item. For example:

```
B←'TONY''S TENNIS RACQUET'      ⍝ CREATE SIMPLE  
                                ⍝ CHARACTER VECTOR  
B  
TONY'S TENNIS RACQUET  
C←'TONY' 'S TENNIS RACQUET'      ⍝ CREATE 2-ITEM  
                                ⍝ CHARACTER VECTOR  
C  
+---+ +-----+  
|TONY| |S TENNIS RACQUET|  
+---+ +-----+
```

Because carriage returns and line feeds may be items of a character constant, you can have a vector composed of several lines. In the following example, *B* is a vector:

```
B←'THIS IS A  
MULTIPLE LINE  
LITERAL.'  
B  
THIS IS A  
MULTIPLE LINE  
LITERAL.  
pB  
44
```

If it is included inside a character constant, an illegal overstruck character does not generate an error. It is recognized as three characters within the character constant. Note that a valid overstruck character within a character constant is recognized as one character.

## VAX APL Language Concepts

### 2.5 APL Expressions

---

#### Note

---

If you enter a character constant with an unbalanced number of quotation marks, APL interprets that you are still defining the constant when you press the Return key to enter the line. Consequently, APL includes a <CR><LF> as part of the constant. You can spot this error by noticing that APL does not indent six spaces after you press the Return key. APL continues to treat everything you enter as part of the constant until you enter a closing quotation mark, enter a weak attention signal (see Section 1.9), or enter more than the maximum number of characters allowed in a line (see Section 2.5.9).

---

#### 2.5.4 Vector Notation

**Vector notation** (also known as strand notation) is a method of combining a list of arrays into a single vector. When you specify a list of arrays separated by spaces, APL joins them together. If the value of each array in the list is a simple scalar, the arrays are combined into a simple vector. If the values are not all simple scalars, the result is an enclosed vector. For example:

```
    ⎕←SHO←1 2 3      ⋊3 SCALARS BECOME SIMPLE VECTOR OF 3 ITEMS
1 2 3
B←43
C←24
D←65
    ⎕←SHO←B C D      ⋊RESULT IS 3-ITEM SIMPLE VECTOR
43 24 65
B←43 10
C←24
C←65
    ⎕←SHO←B C D      ⋊RESULT IS 3-ITEM ENCLOSED VECTOR
+---+ 65 65
| 43 10 |
+---+
```

You can use parentheses in a strand to create more deeply nested vectors. In the following example, *SHO* is a 3-item vector whose third item is a 2-item vector, and this 2-item vector is composed of a 3-item vector and a scalar. The overall depth is 3.

```
    ⎕←SHO←1 2 ((3 4 5) 6)
1 2 +---+
| +---+ 6 |
|| 3 4 5 | |
| +---+   |
+---+
```

### 2.5.5 Functions

APL performs operations by evaluating functions that are applied to arrays. There are several kinds of functions in APL:

- Primitive functions are functions provided by APL that implement the basic operations of the APL language. The names of primitive functions are symbols from the APL character set (see the *VAX APL Reference Manual*).
- System functions are functions provided by APL that affect the APL environment. The names of system functions are valid APL identifiers that start with the `□` symbol (see the *VAX APL Reference Manual*).
- User-defined functions are programs defined by the APL user to implement algorithms. The names of user-defined functions conform to the rules for the formation of identifiers. User-defined functions may be APL code or external routines (see Chapter 3 and Chapter 6.)
- Derived functions are functions created by APL operators (see the *VAX APL Reference Manual*).

In addition, functions are divided into four groups called niladic, monadic, dyadic, and ambivalent functions, which take 0, 1, 2, or either 1 or 2 arguments, respectively.

### 2.5.6 Operators

APL can also apply operators to functions and arrays to produce derived functions, which may then be applied to arrays.

There are two kinds of operators:

- Primitive operators include slash (/), backslash (\), each (``), and dot (.). For more information, see the *VAX APL Reference Manual*.
- User-defined operators are operators defined by the APL user to implement algorithms. The names of user-defined operators conform to the rules for the formation of identifiers. User-defined operators are formed with APL expressions. For more information, see Chapter 3.

In addition, operators are divided into two groups called monadic and dyadic operators, which take 1 and 2 operands, respectively.

## VAX APL Language Concepts

### 2.5 APL Expressions

#### 2.5.7 Spaces and Tabs

Spaces and tabs generally are not significant in APL. They have meaning when you use them to separate identifiers and constants from each other, but they do not have meaning when you use them to separate primitive functions from their arguments or from each other. For example, note the following groups of equivalent expressions:

```
B←      35
B← 35

C←16
C←          16

A+B+1-C
A ← B + 1 -C

X←qB
X←q B

A←1↑ρ+/B
A ← 1 ↑ ρ + / B

D←1↑ 2
D←1 ↑ 2
D←1 ↑ 2
```

You cannot set tab stops in APL. APL passes tabs to the operating system for interpretation.

In this manual, any combination of spaces and tabs is referred to as white space.

#### 2.5.8 Evaluating Expressions

APL evaluates unparenthesized expressions in strict right-to-left order, regardless of the particular functions in the expression. Unlike some languages, which perform multiplication and division before addition and subtraction, APL has no explicit function precedence. For example, APL evaluates the expression  $3 \times 4 + 5$  from right to left, and the result is 27, rather than 17:

```
3×4+5
27
```

You can control the order in which individual functions are evaluated by enclosing parts of an expression in parentheses. To cause the expression in the preceding example to evaluate to 17, you would enter the following:

```
(3×4)+5
17
```

Note, however, that parentheses do not take absolute precedence; APL evaluates the line from right to left for as long as possible. For example:

```

A←1
(A←2)+A
3
A
2

```

The right-to-left evaluation rule does not explain how APL evaluates expressions in all situations. There is also the concept of binding strength, which refers to how APL groups objects for evaluation. The relative binding strengths for various objects are listed below in descending order:

Object	Binding Strength
Brackets	To what is on the left
Left assignment	To the identifier on the left
Right operand	To dyadic operator
Strand	Array to array
Left operand	To the operator
Left argument	To the function
Right argument	To the function
Right assignment	To the value on the right

Left assignment and right assignment both refer to the left and right arguments of the assignment function ( $\leftarrow$ ).

Note that the binding strength of parentheses depends on the evaluation of their contents. Also note that brackets and monadic operators do not bind to the right.

As a result of the binding hierarchy, operators have a long left scope, and functions have a long right scope.

Brackets are at the top of the list, and they bind most strongly to the object to the left than to anything else. Thus, APL evaluates brackets and the object to their left first. For example, APL evaluates  $A+B[I]$  as  $A+(B[I])$  and not as  $(A+B)[I]$ .

A strand is a list of arrays separated by spaces. APL binds the arrays more strongly to themselves than it binds a function to its arguments or an operator to its left operand. For example, in the expression  $A\ B\ F\ 3$  where  $F$  is a function, APL binds the  $A$  and  $B$  together and uses the result as the left argument to  $F$ .

## VAX APL Language Concepts

### 2.5 APL Expressions

However, note that arrays do not bind more strongly to themselves than do dyadic operators to their right operands. For example, note the following expression:

*LOPERAND DYADICOPERATOR G H K*

APL does not bind the strand *G H K* together, because the binding of the right operand to its operator ranks higher in the binding hierarchy. APL evaluates the preceding expression as follows:

*(LOPERAND DYADICOPERATOR G) H K*

Note that the expression  $1\ 2\ 3[2]$  results in an error. This is because the binding strength of brackets and the object to their left is greater than that of strands, and APL evaluates the expression as  $1\ 2\ (3[2])$  not as  $(1\ 2\ 3)[2]$ .

#### 2.5.9 Statements

**Statements** consist of one or more expressions executed as a unit. You can include more than one statement on a line if you separate the statements with the diamond character ( $\diamond$ ). For example:

```
A←64 ⋮ B←92 ⋮ A÷B  
0.6956521739
```

Statements separated by diamonds are executed from left to right. (However, the expressions that make up the statements between the diamonds are still evaluated from right to left.) For example:

```
⍝A←64 ⋮ ⍝B←92 ⋮ A÷B  
64  
92  
0.6956521739
```

If APL encounters an error in a multistatement line, statements to the right of the statement in error are not executed. For example, the identifier *B* is undefined in the following multistatement line:

```
)CLEAR
CLEAR WS
      A←1 ⋀ B ⋀ C←3
11 VALUE ERROR
      A←1 ⋀ B ⋀ C←3
          ^
          A
1
      C
11 VALUE ERROR
      C
          ^
```

Do not confuse the purpose of the semicolon with that of the diamond character: the semicolon is an output catenator, not a statement separator. For details, see Section 5.2.

### 2.5.10 Lines

An APL line consists of the statement or statements that you enter beginning at column 6 and ending when you press the Return key (unless you are inside a character constant). APL does not begin to evaluate the expressions in a line until you complete the line by pressing the Return key.

The maximum length of a line in APL is 2048 keystrokes. APL automatically echoes a long single line as several lines on your terminal. So, when you reach the end of a line on your terminal, you should continue typing; you do not have to enter a special character to indicate that the line is being continued.

If you enter more than 2048 characters, APL signals *INPUT LINE TOO LONG*.

### 2.5.11 Comments

Comments begin with the `a` character, and you may position them at the end of lines containing APL expressions or you may write them on separate lines. APL ignores everything to the right of the `a` character.

## 2.6 Forming Arrays

To form a scalar array or a vector array containing two or more items, you simply enter the data. To form matrix arrays or vector arrays containing one item (a vector singleton), you must use a function such as dyadic reshape, which was introduced in Section 2.2.2.1. When you enter a single number or character, you create a scalar. When you enter more than a single number or character, you create a vector.

## VAX APL Language Concepts

### 2.6 Forming Arrays

The following expressions create and display simple arrays. Note that numbers are separated with spaces, pairs of parentheses, or characters. Characters can be grouped together or can be separated, but they are always delimited with single quotation marks. Note that adjacent quotation marks are evaluated as part of a character string and not as delimiters; when you want delimiters, separate the marks with a space.

```
A CREATE A SCALAR AND 3 VECTORS
5 ⋄ 5 3 7 ⋄ (5)(3)(7) ⋄ 'A'5'B'3'C'7
5
5 3 7
5 3 7
A 5 B 3 C 7

A CREATE A SCALAR AND 2 VECTORS
'A' ⋄ 'ABC' ⋄ 'A' 'B' 'C'
A
ABC
ABC

A SHOW QUOTATION MARKS AS NON-DELIMITERS
A ONLY THE OUTERMOST MARKS ARE DELIMITERS
'A''B''C''D'
A'B'C'D
```

To create enclosed arrays, use parentheses and single quotation marks to group and separate items. For numeric data, use parentheses. For example:

```
A ENCLOSED ARRAY OF LENGTH 4
◻←ENC← 5 (9 3) 241 (3 (45 84) 2)
5 +---+ 241 +-----+
| 9 3 | | 3 +----+ 2 |
+---+ | | 45 84 | |
| +----+ |
+-----+
≡ENC           A DEPTH = 3, TWO LEVELS OF NESTING
3
A PARENTHESES GROUP, DO NOT SEPARATE
◻←G←(1 2 3)   A G IS NOT ENCLOSED
1 2 3
A THIRD ELEMENT GROUPED SEPARATELY
◻←E←'A' 'B' 'CD'   A ENCLOSED VECTOR, LENGTH 3
A B +---+
| CD |
+---+
A IN F, THE CHARACTERS FORM
A A SEPARATE GROUP
◻←F←512 'DOUBLED' 1024      A F IS ALSO LENGTH 3
512 +-----+ 1024
| DOUBLED |
+-----+
```

To enclose a value that cannot be grouped and separated, such as a simple vector, you would use the monadic enclose (`c`) function (for more details, see the *VAX APL Reference Manual*). Enclosing a scalar has no effect, but enclosing a singleton does. For example:

```

    ⌊←R← c214      ⋓ ATTEMPT TO ENCLOSE SCALAR
214
    ≡R                  ⋓ DEPTH = 0 SIMPLE SCALAR
0
    ⌊←Q← c,214      ⋓ ENCLOSURE RAVEL (,) OF SCALAR
+---+
|214|
+---+
    ≡Q                  ⋓ DEPTH = 2 ENCLOSED
2
    ⌊←R← c'ABCD'    ⋓ ENCLOSURE A SIMPLE VECTOR
+----+
|ABCD|
+----+
    ≡R                  ⋓ QUERY FOR DEPTH
2

```

## 2.7 Editing Variables

Frequently arrays are assigned to variables. After the array is created and assigned to a variable, that array can be included in commands by entering the variable name. For example:

```

    ⌊←A←4 9 p 239 12 32 3 12 9 43 2 84 23 3 23 1 4 2 54 342
239 12 32 3 12 9 43 2 84
    23 3 23 1 4 2 54 342 239
    12 32 3 12 9 43 2 84 23
    3 23 1 4 2 54 342 239 12
        A×4
956 48 128 12 48 36 172 8 336
    92 12 92 4 16 8 216 1368 956
    48 128 12 48 36 172 8 336 92
    12 92 4 16 8 216 1368 956 48
    ⌊←C←6 6 p A+3
242 15 35 6 15 12
    46 5 87 26 6 26
    4 7 5 57 345 242
    15 35 6 15 12 46
    5 87 26 6 26 4
    7 5 57 345 242 15

```

Variables can be edited if you are using the DECwindows or Character-Cell interface. You can also use the `)EDIT` system command to edit variables. You cannot edit system variables.

## VAX APL Language Concepts

### 2.7 Editing Variables

Depending on the specific attributes of any given variable, the shape information may change when the variable enters the editor and again when it is reestablished in the APL environment.

If an object has a rank greater than two, it is displayed in the VAXTPU session as a matrix of the following shape:

$(\times / \neg 1 + \rho object), (\neg 1 + \rho object)$

When you exit from VAXTPU, the object remains rank two; it does not return to its original shape. When you quit from VAXTPU, the object returns to APL with its original shape intact.

If an object has a rank of two or less, its rank remains intact both in the VAXTPU session and when it returns to APL. The exception to this rule is the scalar object, which returns to APL as a vector.

If you edit a new or empty variable, APL generates an empty temporary file for VAXTPU. If the VAXTPU file is empty (contains 0 records) when it returns from VAXTPU, it arrives in APL as an empty vector (if it did not previously exist in APL), or as an empty object (if it did previously exist in APL). In this case, the rank of the returned object is the same as the original unless the orginal was greater than two, in which case the object's shape is 0 0.

#### 2.7.1 Editing Character Variables

When a character variable is a vector containing embedded <CR><LF>s, it is written out to VAXTPU as two records:

Vector Data in APL	Data in VAXTPU
'aa<CR><LF>aa'	'aa
aa<CR><LF>aa	aa'
	aa
	aa

When a vector returns to APL, it enters as the ravel of the catenation of all the records in the VAXTPU file. APL embeds a <CR><LF> at the end of each record. If any single record contains a <CR><LF>, you must use the system command editor, )EDIT and specify /MODE: 3 for it to remain intact; in mode 2 and with the DECwindows and Character-Cell interface editors, the <LF> will not return. (/MODE: 3 is the only method that allows you to return embedded <CR><LF>s from VAXTPU to APL.) If any records are empty, APL marks them with <CR><LF>s. If the last record in the vector is empty, it is marked by a <CR><LF>; if it is not empty, it is not followed by a <CR><LF>.

## VAX APL Language Concepts

### 2.7 Editing Variables

Vector Data in VAXTPU	Data in APL	
	Mode 3	Mode 2
'aa	'aa<CR><LF>aa'	'aa<CR><LF>aa'
aa'		
aa	aa<CR><LF>aa	aa<CR><LF>aa
aa		
'aa<CR><LF>aa'	'aa<CR><LF>aa'	'aa<CR>aa'
aa<CR><LF>aa	aa<CR><LF>aa	aa<CR>aa

When a row of a character variable matrix contains embedded <CR><LF>s inside quotation marks, it is written to VAXTPU as a single record (and the shape of the matrix is preserved).

Matrix Data in APL	Data in VAXTPU
'aa<CR><LF>aa'	'aa<CR><LF>aa'
'aa<CR><LF>aa'	'aa<CR><LF>aa'
'aa<CR><LF>aa'	'aa<CR><LF>aa'

When a matrix returns to APL, it has one row for each record. All of the rows will have the same number of columns as the longest record (shorter records are extended with blanks). Any empty records return as rows filled with blanks. If any single record contains a <CR><LF>, and you specify /MODE:3, APL treats the <CR><LF> as the end of a row and forms a new record; if you specify /MODE:2, the <LF> will not return.

Matrix Data in VAXTPU	Data in APL	
	Mode 3	Mode 2
'aa	'aa	'aa
aa'	aa'	aa'
aa	aa	aa
aa	aa	aa
'aa<CR><LF>aa'	'aa	'aa<CR>aa'
'aa<CR><LF>aa'	aa'	'aa<CR>aa'
		'aa
		aa'
aa<CR><LF>aa	aa	aa<CR>aa
aa<CR><LF>aa	aa	aa<CR>aa
		aa
		aa

Note that in the preceding examples, the <CR><LF> symbol does not appear visually in the APL environment, although it does cause data to be displayed on a new line.

## VAX APL Language Concepts

### 2.7 Editing Variables

#### 2.7.2 Editing Numeric Variables

When you edit a numeric variable that is initially of rank two or greater, VAXTPU returns a numeric matrix of the following shape:

$(\rho \text{ matrix}) = (\text{no. records in file}), (\text{no. numeric values in first record})$

When a numeric matrix returns to APL, all records in the matrix must be numeric, and all must have the same number of items as there are in the first record. Empty records are not allowed unless the entire matrix is empty.

If you edit a numeric variable that is initially a vector, VAXTPU returns a vector that is the ravel of all the records in the VAXTPU file. The returning array may not contain any nonnumeric data. Empty records are ignored.

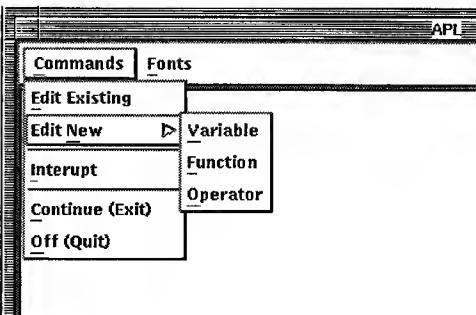
#### 2.7.3 Editing Variables with the DECwindows Interface Editor

The DECwindows interface provides full DECwindows support of the APL product. In addition to the interactive area in the initial APL DECwindow, you can open one or more sessions to edit user-defined operations and variables. (See Section 3.11.2 for more information on editing user-defined operations.)

Defining a new variable is similar to editing an existing variable. One difference is that you have to specify information about the variable if you are defining a new variable. Follow these steps to start an edit session.

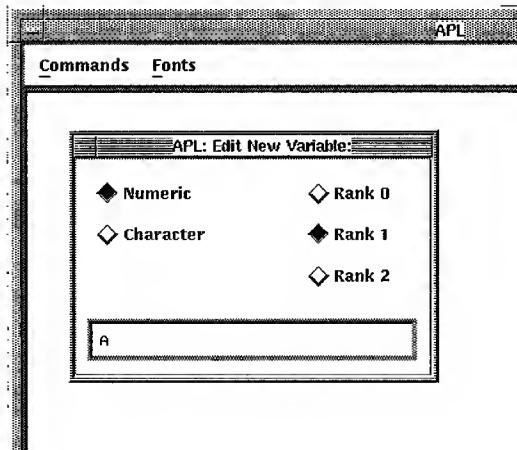
1. Click on the Commands option located on the Menu Bar in the transcript session to expose the Commands menu. (See Figure 2-1.)

**Figure 2-1 DECwindows Interface Edit Options**



2. Select the Edit Existing or Edit New option. If you select the Edit New option, another menu will display a choice of object types. Select the Variable option.
3. Select the appropriate options to describe the variable you want to edit from the Edit New Variable dialog box. Figure 2-2 displays this box. Click on either the Numeric or Character option depending on the type of variable you are editing. Select the rank by clicking on the appropriate rank option. Finally, click in the input area at the bottom of the dialog box and enter the name of the variable.

**Figure 2-2 DECwindows Interface Edit New Variable Dialog Box**



4. The Title Bar in the edit session is the name of the variable. If you are editing an existing variable, the array will be displayed.  
You can enter text to edit the variable or you can use the mouse to copy text from one edit session to another edit session. The mouse can be used to copy text from the transcript window, to an edit window. You cannot copy text into the transcript window.
5. When you have finished editing the variable, select one of the edit session options that are displayed when you click on the Commands option in the menu bar of the edit session. Figure 2-3 shows these options.

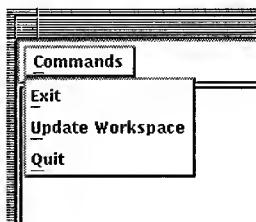
**DECLIT AA VAX P142E**

**VAX APL user's guide**

## VAX APL Language Concepts

### 2.7 Editing Variables

**Figure 2-3 DECwindows Interface Edit Session Commands Options**



You can update the transcript session with the new array by choosing the Update Workspace option. When you return to the transcript session, you can enter commands to call the variable. You do not have to close the edit session to return to the transcript session.

If you select the Exit option from the edit session Commands menu, the transcript session is updated with any changes to the variable, and the edit session is closed. The Quit option closes the edit session without saving changes.

#### 2.7.4 The Character-Cell Interface Editor

The Character-Cell interface provides a VAXTPU based window environment for APL sessions on the Digital VT220, VT240, VT320, VT330, VT340 and DECterm terminals. This environment inserts the text of the operation you are editing into a temporary holding area, a buffer. You can display more than one buffer on the screen at one time and edit more than one variable during an APL session. (See Section 1.3.3.)

To edit a character variable, press the Do or PF4 key or enter Ctrl/B to display the Command: prompt. Enter the following command, substituting the name of the variable being edited for *variable*.

*GET variable← ↵*

To edit a numeric variable, enter the following command at the Command: prompt, substituting the name of the variable for *variable*.

*GET variable← □*

The variable is displayed in the window of the new buffer. The end-of-file marker defines the end of the buffer. It is only visible on the screen and is not saved as part of the variable. The status line shows the name of the buffer.

You can edit the variable by using one of the following methods:

- Entering the APL characters from the keyboard

- Inserting text copied from the APL SESSION Buffer or another edit buffer
- Including entire files

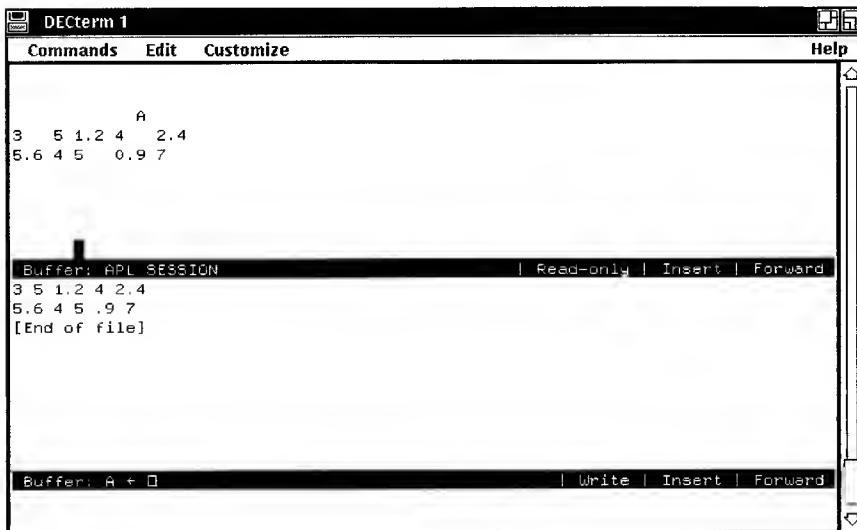
The VAXTPU Help utility has specific information about copying text. Enter *HELP* at the Command: prompt and look at copy, cut, paste and restore. Enter *HELP INCLUDE FILE* at the Command: prompt to get more information about including files. (Press the Do or PF4 key or enter Ctrl/B to reveal the Command: prompt.)

To update the interactive apl session with the variable, use the command *WRITE* in the following form in response to the VAXTPU Command: prompt:  
***WRITE [variable]***

The name of the variable is optional. If you do not specify the name of the variable, the name associated with that buffer is used.

Figure 2-4 displays a session using two windows, the APL SESSION and an edit session.

**Figure 2-4 Character-Cell Interface Variable Edit Example**



To return to the interactive session, enter *BUFFER APL SESSION* at the Command: prompt. (See Section 1.3.3.) Alternatively, if you are using a split screen with the interactive session in one window and an edit session in

## VAX APL Language Concepts

### 2.7 Editing Variables

another, you can return to the interactive session by entering *OTHER* at the Command: prompt or, if you are using a mouse, by clicking on the window.

#### 2.7.5 The )EDIT System Command Editor

The )EDIT system command allows you to edit APL objects with the VAXTPU editor. The default object type is a function. To edit a variable, you must specify a value 2 for the name class qualifier (/NC:2).

Use the form following form to edit an APL object:

```
)EDIT operation[[qualifiers]]
```

***operation***

Is the name of the APL operation you want to edit.

***qualifier***

Is one or more of the optional qualifiers. See the *VAX APL Reference Manual* for more documentation on the )EDIT system command.

Depending on the value you specify for /NG when you invoke )EDIT , negative numbers appear in VAXTPU preceded by a high minus sign ( - ), a minus sign ( - ), or an ASCII minus sign ( - ), which appears as an APL plus sign ( + ). If you do not specify the /NG qualifier, APL uses the current setting for  $\square NG$  as the default.

The display of numbers passed from APL to VAXTPU is dependent on  $\square PP$ . Therefore, you could lose precision when editing numbers. You can use the /PP qualifier when you invoke )EDIT to specify a different value for the print precision.

Depending on the  $\square PW$  setting in effect when you invoke )EDIT , some numbers may be segmented such that the digits span more than one record. If these numbers are not repaired in the VAXTPU file before returning to APL, then each segmented number will return as two separate values.

## 2.8 Indexing Arrays

To access an individual item stored in an array, you must know its position, or index value, within the array. The number of index values, or indexes, needed for an array depends on the array's rank. In general, the number of indexes must match the number of axes of the array; thus, a vector requires one index, a matrix requires two indexes, an array with three axes requires three indexes, and so forth. Scalars may not be indexed.

To index an array, specify the array name, followed by the indexes enclosed in square brackets and separated with semicolons. Note that the number of semicolons in an index specification is equal to one less than the rank of the array being indexed. Each index must be a simple near-integer array (which may be empty), or an expression that evaluates to a simple near-integer array.

Examples:

```
V←5 7 9 11 13 15      aV IS A VECTOR
V[3]      a1 AXIS MEANS 1 INDEX VALUE, NO SEMICOLON
9
M←2 3p5 7 9 4 6 8      aM IS A MATRIX
M[2;2]    a2 AXES MEANS 2 INDEX VALUES, 1 SEMICOLON
6
```

In some cases, you must use parentheses when you want to group a vector of values. Note the difference in the following expressions.

```
(7 6 5 4 3 2 1)[2 4]      aPARENTHESES REQUIRED
6 4
7 6 5 4 3 2 1[2 4]      aINDEX BINDS TO ITEM ON THE LEFT
36 INDEX RANK ERROR (CANNOT INDEX A SCALAR)
7 6 5 4 3 2 1[2 4]      aINDEX BINDS TO ITEM ON THE LEFT
^
```

In the previous example, the first expression contains a vector that is grouped in parentheses. The index expression binds to the entire group and selects the second and fourth items. There is no grouping in the second expression. It is illegal because the index binds only to the scalar value 1.

### 2.8.1 Index Origin

The first position along each axis of the values stored in an array is called the index origin; it may be 1 or 0. If the index origin is 1, the indexes of arrays begin with position 1; thus, members of a 3-item vector named *A* would be numbered *A*[1], *A*[2], *A*[3]. Similarly, if the index origin is 0, items would be numbered *A*[0], *A*[1], *A*[2].

The index origin setting applies to all arrays in a workspace. The default index origin in a clear workspace is 1, but you can change it by setting the *IO* system variable. See the *VAX APL Reference Manual* for details.

For all examples in this manual, it is assumed that the index origin is 1, except where an example explicitly states that it is 0.

## VAX APL Language Concepts

### 2.8 Indexing Arrays

#### 2.8.2 Selecting One Array Item

To select a single item from an array, specify the array name and the indexes that refer to the item's position in the array. (This may also be done with the pick ( $\triangleright$ ) function; see the *VAX APL Reference Manual* for more information.) For example, to access the first item stored in vector  $A$ , specify the name of the vector followed by the index 1:

```
A[1]
```

If  $A$  is the following vector, then  $A[3]$  is 25, as shown:

```
0←A←72 91 25 46 87      ⋀CREATE AND DISPLAY A
72 91 25 46 87
A[3]
25
```

If the array is a matrix, you must specify two indexes. The first index designates the row, and the second index designates the column. Use semicolons (;) to separate the indexes as follows:

```
0←B←2 4⍴18      ⋀CREATE AND DISPLAY B
1 2 3 4
5 6 7 8
B[2;3]          ⋀SELECT ITEM IN ROW 2, COLUMN 3
7
```

When you select an enclosed item from an array, the item remains enclosed. You can disclose the item by using the first or disclose function with your index expression. First ( $\dagger$ ) selects the first item of an array and discloses it. Disclose ( $\triangleright$ ) removes one level of nesting from an array each time it is used. For example:

```

      ⌊+T+2 3@1 2 3 4 5 (6 2 3)    ⋄CREATE A NESTED ARRAY
1 2 3
4 5 +----+
| 6 2 3|
+----+
      ⌊+Z← (⌐'TABLE'),(⌐T),'LIST',22 25 753    ⋄Z CONTAINS T
+----+ +-----+      L I S T   22 25 753
| TABLE| | 1 2 3 |
+----+ | 4 5 +----+
|     | | 6 2 3 |
|     +----+
+-----+      ⋄QUERY FOR THE DEPTH OF Z
3
      Z[2]
+----+
| 1 2 3 |
| 4 5 +----|
|     | | 6 2 3 |
|     +----|
+-----+      ⋄USE FIRST TO DISCLOSE ONE LEVEL
1 2 3
4 5 +----+
| 6 2 3|
+----+
      ↑(+Z[2])[2;3] ⋄USE FIRST TO DISCLOSE TWO LEVELS
6 2 3
      ⌈>Z[2]        ⋄USE > TWICE TO DISCLOSE BOTH LEVELS
1 0 0
2 0 0
3 0 0

4 0 0
5 0 0
6 2 3

```

For more information on the first and disclose functions, see the *VAX APL Reference Manual*.

### 2.8.3 Selecting More Than One Array Item

By specifying the indexes in the form of an array, you can access more than one array item at a time. Again, use semicolons (;) to separate the indexes.

## VAX APL Language Concepts

### 2.8 Indexing Arrays

```
    ⌊←V←32 44.6 71 .8 65 97.2
 32 44.6 71 0.8 65 97.2
    V[3 5 6]      ⋀INDEX 3 TIMES FROM V
 71 65 97.2
    ⌊←M←2 4⍳8
 1 2 3 4
 5 6 7 8
    M[2;1]      ⋀INDEX 1 ITEM FROM M
 5
    M[1 2;2 3]   ⋀INDEX 2 ITEMS FROM M
 2 3
 6 7
```

Indexing is not affected by the type of the array—simple or enclosed, character, numeric, and heterogeneous arrays are all indexed in the same way. Also, note that you can duplicate an item by specifying its position more than once. For example:

```
A←'ABCDEFGHIJKLMNPQRSTUVWXYZ'
⋀ELEMENT 27 IS BLANK
BABY←A[10 5 14 14 9 6 5 18 27 12 25 14 14 5]
BABY
JENNIFER LYNNE
```

#### 2.8.4 Selecting All Items Along an Array Axis

To select all items along an array axis, you omit the index for that axis in the index specification. You must include a semicolon to indicate which index has been omitted. For example:

```
    ⌊←A←4 3⍳12    ⋀CREATE AND DISPLAY A
 1 2 3
 4 5 6
 7 8 9
10 11 12
    A[1;]    ⋀OMIT SECOND AXIS, SELECT ROW 1, ALL COLUMNS
 1 2 3
    A[;2 3]  ⋀OMIT FIRST AXIS, SELECT COLUMNS 2 AND 3
 2 3
 5 6
 8 9
11 12
```

Note that it is legal to omit all the indexes (but not the semicolons) from an index specification; the value returned is the array itself. For example:

```

    ⌊←A←15          ⋓CREATE AND DISPLAY A
1 2 3 4 5
    A[]
1 2 3 4 5
    ⌊←B←2 4⍴8      ⋓CREATE AND DISPLAY B
1 2 3 4
5 6 7 8
    B[;]            ⋓OMIT ALL INDEXES
1 2 3 4
5 6 7 8

```

## 2.8.5 Indexing Constants and Expressions

The array being indexed need not be a variable. It can be a constant set of values or an expression enclosed in parentheses.

```

(7 6 5 4 3 2 1)[2 4]    ⋓PARENTHESES REQUIRED
6 4
(2 4 8 16*2)[1 2]        ⋓POWER FUNCTION EXTENDS TO
4 16                      ⋓ EACH ITEM OF 2 4 8 16

```

When a set of values is not enclosed in parentheses, the index binds only the rightmost value, and APL signals an error because it cannot index a scalar. For example:

```

    ⋓INDEX BINDS TO ITEM ON THE LEFT
7 6 5 4 3 2 1[2 4]
36 INDEX RANK ERROR (CANNOT INDEX A SCALAR)
7 6 5 4 3 2 1[2 4]
^

```

## 2.8.6 Using an Expression to Generate Indexes

The index can be an expression which is evaluated to generate the item positions. For example:

```

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

```

In this example, *V* and *K* are both vectors. The expression *V[K+1]* accesses the items of *V* referenced by *K+1*, that is, the third, fifth, and sixth members of vector *V*.

## VAX APL Language Concepts

### 2.8 Indexing Arrays

#### 2.8.7 Shape of an Indexing Result

When a variable is indexed, the shape of the result is equal to the catenation of the shapes of all the indexes. If the variable is indexed using the following form:

$Z \leftarrow X[K_1; K_2; K_3; \dots; K_n]$

then

$(\rho Z) = (\rho K_1), (\rho K_2), (\rho K_3), \dots, (\rho K_n).$

For example:

```
V←'ABCDEF'  
     AA VECTOR INDEXED WITH A VECTOR IS A VECTOR  
V[6 5 4 3 2 1]  
FEDCBA  
     AA VECTOR INDEXED WITH A MATRIX IS A MATRIX  
M←2 3⍳2 5 4 6 5 4  
V[M]  
BED  
FED  
     AA MATRIX INDEXED WITH TWO 2-DIMENSIONAL  
     A INDEXES RESULTS IN A 4-DIMENSIONAL ARRAY  
M←2 2⍳1 2 2 1  
M  
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♂16
    M
1 2 3 4
5 6 1 2
    ♂SCALAR RESULT
    A+M[1;1]
    A
1
    ♂A
                (APL outputs a blank line)
    ♂EMPTY ARRAY
    ♂M[10;]
0 4
    ♂1-ITEM VECTORS
    B←M[1;1]
    B2←M[,1;1]
    B
1
    B2
1
    ♂B
                (APL outputs a blank line)
    ♂B2
1
    ♂2-ITEM VECTOR
    C←M[;2]
    C
2 6
    ♂C
2
    ♂A 2-BY-1 MATRIX
    D←M[;,2]
    D
2
6
    ♂D
2 1

```

## VAX APL Language Concepts

### 2.8 Indexing Arrays

#### 2.8.8 Replacing Selected Items in Arrays

You can combine indexing with the specification function (see Section 1.5) to change the values of items already stored in an array. For example:

```
0←A←2 3⍴6      ⋄CREATE AND DISPLAY A
1 2 3
4 5 6
      A[1;2 3]←7 9  ⋄REPLACE AT COLUMNS 2 AND 3 OF ROW 1
      A
1 7 9
4 5 6
      A[2;1 2]←9    ⋄SINGLETON EXTENSION OF 9
      A
1 7 9
9 9 6
      A[1;1]←12     ⋄REPLACE AT COLUMN 1 OF ROW 1
      A
12 7 9
9 9 6
      A[1;1 1]←2 3  ⋄INDEX ASSIGNMENT WORKS LEFT TO RIGHT
      A
3 7 9
9 9 6
```

### 2.9 Error Handling

Once APL evaluates a primitive function, it does one of two things: determines a result, or signals an error. When APL signals an error, it prints a 3-line error message that includes:

- A description of the error.
- The text of the line in which the error occurred.
- A caret (^) approximately underneath the particular point at which the error was discovered.

For example, if you try to display the variable *B*, and *B* has not been assigned a value, APL displays the following:

```
11 VALUE ERROR
      B
      ^
```

The description of the error consists of an error number, a primary error message (such as *VALUE ERROR*), and, if applicable, a parenthesized secondary error message that provides more information about what caused the error. (You can prevent the display of secondary error messages by setting the

system variable  $\square TERSE$  to 1; see the *VAX APL Reference Manual* for more information.)

If the operating system detects an error, its message—if it provides any useful information—is displayed as a secondary error message.

The display of error messages is dependent on  $\square PW$ . (See the *VAX APL Reference Manual*.)

- If the description of the error is longer than  $\square PW$  characters, it is truncated; it is not continued on the next line.
- If the text of the line in which the error occurred is longer than  $\square PW$  characters, the text is truncated, except when the truncated part includes the particular point in the line at which the error was discovered. In that case, APL truncates enough characters from the beginning of the line to allow the location of the error to be displayed.

The entire message generated by the last error that occurred is always stored as the value of the system variable  $\square ERROR$  even if part of the message was truncated when it was displayed.

For a complete list and description of APL error messages, see the *VAX APL Reference Manual*.

### **2.9.1 Order of Error Checks**

When APL evaluates expressions, it usually checks for errors in the same order regardless of whether the evaluation involves a primitive function, a system function, a primitive operator, or an assignment to a system variable. In general, this order is as follows:

*SYNTAX*  
*VALUE.*  
*AXIS RANK*  
*AXIS LENGTH*  
*AXIS DOMAIN*  
*INDEX RANK*  
*INDEX LENGTH*  
*INDEX DOMAIN*  
*RANK*  
*LENGTH*  
*DOMAIN*  
*LIMIT*

If an expression contains more than one error, APL reports only the error that is discovered first.

## VAX APL Language Concepts

### 2.9 Error Handling

#### 2.9.2 Errors in User-Defined Functions and Operators

When an error occurs during execution of a user-defined function or user-defined operator, APL suspends operation execution (unless the function or operator is locked; see Section 3.6 for details) and prints the appropriate 3-line error message. Line 2 of the message includes, in addition to the text of the line in which the error occurred, the name of the function or operator and the line number (and statement number, if applicable) of the line displayed.

APL allows you to write your own routines to handle errors that occur during execution of user-defined functions and operators. For details, see Chapter 3.

---

## User-Defined VAX APL Operations

**User-defined functions** and **user-defined operators** in VAX APL are equivalent to what are known as programs in other languages. In APL, these programs are called functions and operators because you use them the same way that you use APL's primitive functions and primitive operators.

Together, the user-defined functions and operators are known as **user-defined operations**. They allow you, in effect, to extend the APL language. You can define your own operations and store them with workspaces. Then, you can call on them as easily as you call on primitive operations.

### 3.1 Defining Operations

There are two operating modes in APL: immediate mode and function-definition mode. In **immediate mode**, APL lines are executed immediately after you enter them. In **function-definition mode**, the lines you enter are considered to be part of the body of a function or operator. It is in function-definition mode that you develop, edit, and save user-defined functions and operators.

User-defined operations consist of two parts: a header and a body. The header provides a name for the operation and indicates its valence, how many arguments or operands the operation takes. The body is a sequence of statements that defines the actions to be performed by the operation when it is executed.

Examples in the VAX APL manuals that include an operation definition number each line and use the del character (v) to mark the beginning and the ending of a definition. When you define an operation, you do not need to enter line numbers. You enter the del character only if you are using the function-definition mode of the line-editor. For example:

## User-Defined VAX APL Operations

### 3.1 Defining Operations

```
    v A FOO B
[1]  i(A+B)
[2]  v
      3 FOO 5
1 2 3 4 5 6 7 8
```

## 3.2 Operation Header

The operation header provides a name for the operation, indicates how many arguments or operands it takes (the valence) and whether it returns a result, indicates whether the operation accepts an axis argument, and identifies any local symbols.

### 3.2.1 Function Header Form

When the user-defined operation is a function, the type of the function, niladic, monadic, dyadic or ambivalent depends on the number of arguments it takes. The forms of the headers for the function types are as follows:

Type	Form
Niladic	<code>[[result-id ← ]]<i>function-name</i> [[([k])]] [[;loc-id; . . . ]]</code>
Monadic	<code>[[result-id ← ]]<i>function-name</i> [[([k])]] arg [[;loc-id; . . . ]]</code>
Dyadic	<code>[[result-id ← ]]<i>arg2 function-name</i> [[([k])]] <i>arg1</i> [[;loc-id; . . . ]]</code>
Ambivalent	<code>[[ result-id ← ]]<i>{arg2}function-name</i> [[([k])]] <i>arg1</i>[[;loc-id; . . . ]]</code>

The *arg*, *arg1*, and *arg2* in the headers are dummy arguments. They serve as placeholders for the actual arguments (values) you supply when you call the function. The number of dummy arguments in the header indicates whether the function is niladic (no arguments), monadic (one argument), dyadic (two arguments), or ambivalent (either one or two arguments). Note that you use braces (`{}`) around *arg2* to distinguish between a dyadic and ambivalent function.

The optional result identifier (*result-id*) in the header is also a dummy argument; it is a placeholder for the value that is the result of the function.

The optional list of local symbol identifiers (*loc-id*) indicates identifiers that are local to the function. (The list items are separated by semicolons.)

The optional axis argument (`([[k]])`) must be enclosed with braces and brackets. For an example of axis in a user-defined operation, see Section 3.12.5.

You can use any valid variable name to represent a dummy argument; in fact, *result-id* and one of the *args* may have the same name. In this case, *result-id* assumes an initial value when the function is called. However, you cannot use the same name for both arguments of a dyadic or ambivalent function. Also, *function-name* cannot be the same as *result-id*, *k*, either of the *args*, or the name of any label inside the function. (It may be the same as a *loc-id*.)

### 3.2.2 Operator Header Form

User-defined operators differ from user-defined functions in the following ways:

- An operator can be monadic or dyadic, but not niladic or ambivalent.
- The result (not *result-id*) of an operator is a derived function.
- The derived function can be monadic, dyadic, or ambivalent, but not niladic.

The headers for user-defined functions and operators are very similar. The forms of the headers for operators are as follows:

Type of Derived Function	Form of Operator Header
<b>Monadic Operators</b>	
Monadic	<code>[[result-id+]] (lop op-name [[{[k]}]]) arg [[;loc-id; ...]]</code>
Dyadic	<code>[[result-id+]] arg2 (lop op-name [[{[k]}]]) arg1 [[;loc-id; ...]]</code>
Ambivalent	<code>[[result-id+]] {arg2} (lop op-name [[{[k]}]]) arg1 [[;loc-id; ...]]</code>
<b>Dyadic Operators</b>	
Monadic	<code>[[result-id+]] (lop op-name [[{[k]}]] rop) arg [[;loc-id; ...]]</code>
Dyadic	<code>[[result-id+]] arg2 (lop op-name [[{[k]}]] rop arg1 [[;loc-id; ...]])</code>
Ambivalent	<code>[[result-id+]] {arg2} (lop op-name [[{[k]}]] rop) arg1 [[;loc-id; ...]]</code>

All user-defined operators have a left operand (*lop*). If you specify a right operand (*rop*), the operator is dyadic. If you do not specify the right operand, the operator is monadic. The parentheses are required around the name of the operator (*op-name*), the operands, and the axis argument (if specified).

The resulting derived function always has a right argument (either *arg* or *arg1*). If you specify a left argument (*arg2*), the derived function is dyadic. If you place braces ({{}}) around the left argument, the derived function is ambivalent.

## User-Defined VAX APL Operations

### 3.2 Operation Header

Note that *arg*, *arg1*, *arg2*, *result-id*, *loc-id*, and *k* in the headers serve the same purposes for user-defined operators as they do for user-defined functions. (For more information, see the previous section.) However, note that *result-id* is the result of the derived function and not of the operator.

You can use any valid variable name to represent a dummy argument; in fact, *result-id* and one of the *args* (or operands) may have the same name. In this case, *result-id* assumes an initial value when the function is called. However, you cannot use the same name for both arguments of a dyadic or ambivalent function. Also, *op-name* cannot be the same as *result-id*, *k*, either of the *args*, either of the operands, or the name of any label inside the function. (It may be the same as a *loc-id*.)

#### 3.2.3 Operation Result

If you include *result-id* in the operation header, the function (or derived function) returns an explicit result that is temporarily stored in the result identifier. Then, by including the operation's name in an expression, you can use the value of the function (or derived function) as an argument to other operations. Note that the function (or derived function) returns an explicit result only if you assign a value to *result-id* in the body of the operation. (If you use an operation's name in an expression when there is no *result-id* specified in the header, or if you do not give a value to *result-id* in the body of the function, APL signals *VALUE ERROR* when the function is executed.)

If you do not include *result-id* in the header, the function (or derived function) cannot return an explicit result. An operation that does not return an explicit result may print some data when you execute it, but you cannot directly use that data (and thus, the operation) as an argument to another operation or in any situation that requires a value.

#### 3.2.4 Local Symbol List

Headers may also include names of local symbols used in a user-defined operation. You place these symbols to the right of the rest of the operation header, separating them from the rest of the header and from each other by semicolons. For example, the following operation header indicates that *X*, *LOC*, and *G* are local symbols:

*NAME;X;LOC;G*

Dummy arguments and operands are also considered to be local symbols, so the names that appear in the local list must not be the same as dummy arguments and operands in the same definition. The operation name itself, however, may appear in the local list. For more details about local symbols, see Section 3.4.1.

### 3.3 Operation Body

An operation can have up to 9,999 APL lines and a header. All valid APL statements are valid within an operation, including calls to other operations. You can include system commands, but they must be arguments to the execute function (`⊣`). If you are using the line-editor to define or edit an operation APL executes system commands immediately, provided the right parenthesis that begins the system command is the first nonblank character after the line prompt or edit command. For example:

```
    v FOO
[1]  A⊣XQ')LOAD MYWS'
[2]  )WIDTH
80
[2]
```

Here, the system command in line [1] is an argument to the `⊣XQ` execute function; it will be executed when line [1] of the function is executed. Before entering line [2], the user entered the system command `)WIDTH`. APL returned the value 80, and then prompted for the next line of the function definition.

If you enter a system command in the middle of an expression, APL signals *SYNTAX ERROR* when you attempt to execute the operation. Notice what happens if you try to execute a system command immediately, but the right parenthesis that begins the command is not the first nonblank character after the line prompt:

```
    v FOO
[1]  B⊣XQ')LOAD MYWS'
[2]  B )WIDTH      ^Incorrect syntax
[3]
```

Whenever an operation line is entered with unbalanced parentheses or brackets, APL signals *SYNTAX ERROR*. In this example, the user began to enter line [2], and then decided to check the terminal width. APL accepted `)WIDTH` as part of function line [2]. When `FOO` is executed, APL will reject line [2] because of the incorrect syntax.

### 3.4 Symbolic Names in Operations

Each APL workspace contains an area, called the symbol table, that includes the names and values of all the variables, functions, operators, and groups you defined in the workspace. The symbol table is saved with a workspace; in a clear workspace, it is empty.

## User-Defined VAX APL Operations

### 3.4 Symbolic Names in Operations

Symbols are classified as being either local or global, depending on how their values are treated before, during, and after the execution of a user-defined function or operator.

#### 3.4.1 Local Symbols

Local symbols have significance only during the execution of a particular operation. To specify a symbol as being local, include it at the end of the operation header.

You establish local symbols when you call an operation; you cannot bring values into the operation using localized symbols. There is one exception to this rule: system variables retain their current value when the operation is invoked. Any local values are lost when you exit from an operation. If the operation changes the value of a system variable, the change is local only and the system variable reverts to its original setting when the operation terminates. If you use a local symbol before assigning it a value, APL signals *VALUE ERROR*.

Operation-line labels are treated as local variables. They are initialized when you call the operation; however, you cannot assign values to them. Dummy arguments and operands are also treated as local variables, and they get their values when the operation is activated.

#### 3.4.2 Global Symbols

Symbols that you use in the operation body, but that you do not include in the local symbol list in the operation header, are considered to be global symbols. They have the same value inside and outside of the operation; thus, if execution of an operation changes the value of a global symbol, the value of that symbol remains changed after execution of the operation completes.

Because naming conventions for functions, operators, and variables are the same, you cannot duplicate names among your global functions, global operators, and global variables. You can, however, have a local and global symbol with the same name. In that case, certain rules apply for determining which value takes precedence, global or local. See Section 3.4.4 for details.

#### 3.4.3 Localizing Function and Operator Names

Although an operation header's local symbol list is generally used to localize variable names, you can localize function and operator names as well. Like local variables, local functions and operators have significance only during the execution of a particular operation.

## User-Defined VAX APL Operations

### 3.4 Symbolic Names in Operations

One use of local functions and operators is to execute operations that are stored on an external device. This technique allows you to execute a particular operation whenever necessary without having to keep it in the workspace permanently. For instance, to execute a function LOC stored in canonical form (see  $\square CR$  in the *VAX APL Reference Manual*) in a file called EXTLOC.AIS, you could code the following:

```
          a FIRST DEFINE THE FUNCTION LOC AND
          a STORE ITS CANONICAL REPRESENTATION
          a IN A FILE CALLED EXTLOC.AIS
          a
          v LOC
[1]   'THIS IS THE FUNCTION LOC'
[2]   v
      CHAN $\leftarrow$ ASS 'EXTLOC/IS'
      ( $\square CR$ 'LOC')  $\boxtimes$  CHAN
      LOC
      THIS IS THE FUNCTION LOC
       $\square$ DAS CHAN
      )ERASE LOC
          a NOW DEFINE A FUNCTION THAT EXECUTES THE
          a FUNCTION YOU STORED.
          a
          v EXLOC;LOC;CHAN;CRLOC
[1]   CHAN $\leftarrow$ ASS 'EXTLOC/IS'
[2]   CRLOC $\leftarrow$   $\boxtimes$  CHAN
[3]   LOC $\leftarrow$ DFX CRLOC
[4]   LOC          a EXECUTE LOCAL FUNCTION LOC
[5]    $\square$ DAS CHAN
[6]   v
      EXLOC
      THIS IS THE FUNCTION LOC
      LOC
      11 VALUE ERROR
      LOC
      ^
```

You can accomplish the same objective by including the operation's own name in its local symbol list. The header for the preceding example would be as follows:

```
LOC ;LOC;CHAN;CRLOC
```

Then, the 5-line function LOC that is permanently in your workspace establishes and executes the local function LOC , which is stored on disk and could consist of many more than five lines.

## User-Defined VAX APL Operations

### 3.4 Symbolic Names in Operations

When an operation's own name is localized, it does not necessarily have to be used as the name of a local function or operator. For instance, the function *FOO* uses the symbolic name *FOO* as a local variable name:

```
    ∇ FOO ;FOO
[1]   FOO←5
[2]   ∇
```

The global value of *FOO* is the name of the function; the local value of *FOO* within the function is 5 (or whatever other value you assign to it). Note that if you localize an operation's own name in this manner, you cannot call the operation recursively.

#### 3.4.4 Precedence of Local Symbols

During operation execution, the value of a local symbol shadows (supersedes) the value of a global symbol with the same name. Also, depending on the particular operation being executed, a local symbol can shadow another local symbol with the same name.

In the following example, APL uses the value of a local variable *A* during operation execution; then, when APL exits from the function, *A* retains its global value:

```
A←0      ⎕DEFINE GLOBAL A
∇F;A      ⎕DEFINE LOCAL A
[1]   ⎵←A+1 ⎕PRINT VALUE OF LOCAL A
[2]   ∇
      F
1
      ⎕LOCAL A SHADOWS GLOBAL A DURING
      ⎕OPERATION EXECUTION
      A
0
      ⎕GLOBAL A UNSHADOWED AFTER FUNCTION DEFINITION
```

If two user-defined operations have a local variable with the same name, APL uses the value from the operation in which it is currently executing. For example:

```
∇FUNC1;B      ⎕DEFINE LOCAL B
[1]   ⎵←B+10     ⎕PRINT VALUE OF LOCAL B
[2]   FUNC2       ⎕CALL FUNC2
[3]   ⎵←B         ⎕PRINT VALUE OF B AGAIN
[4]   ∇
      ∇FUNC2;B      ⎕DEFINE LOCAL B
[1]   ⎵←B+25     ⎕PRINT VALUE OF B
[2]   ∇
```

## User-Defined VAX APL Operations

### 3.4 Symbolic Names in Operations

```
10          FUNC1           A EXECUTE FUNC1
25
10
11 VALUE ERROR
B
A
```

Here, the local variable *B* has a value of 10 in *FUNC1* and a value of 25 in *FUNC2*. While executing *FUNC1*, APL uses 10; while executing *FUNC2*, APL uses 25. When APL returns to *FUNC1*, *B* reassumes the value 10. Finally, when APL exits from *FUNC1*, *B* has no value.

Local values act like global values within nested operations. For example:

```
VFUNC1;A      ADEFINE LOCAL A
[1]  A←10
[2]  A
[3]  FUNC2
[4]  A
[5]  ∇
    VFUNC2
[1]  A
[2]  A←2 × A
[3]  ∇
    FUNC1
10
10
20
A
11 VALUE ERROR
A
A
```

The local variable *A* in *FUNC1* is global with respect to its nested function, *FUNC2*. However, once the execution of *FUNC1* is complete, *A* has no global value.

System variables can be localized within operation definitions. In the following example, the function *F1* localizes the value of the system variable  $\square I_0$ . After *F1* executes,  $\square I_0$  retains its global value. The function *F2*, however, does not localize the value of  $\square I_0$ . Thus, after *F2* executes,  $\square I_0$  retains the value it was assigned in *F2*.

## User-Defined VAX APL Operations

### 3.4 Symbolic Names in Operations

```
VF1 ;IO
[1] IO←0
[2] ⌊5 ⌈
VF2
[1] IO←0
[2] ⌊5 ⌈
      ⌊5      ⋓IO IS 1
1 2 3 4 5
      F1      ⋓IO IS 0 WITHIN F1
0 1 2 3 4
      ⌊5      ⋓BUT GLOBAL VALUE IS STILL 1
1 2 3 4 5
      F2      ⋓FUNCTION F2 DOES NOT LOCALIZE IO
0 1 2 3 4
      ⌊5      ⋓SO GLOBAL VALUE OF IO IS NOW 0
0 1 2 3 4
```

## 3.5 Branching Within An Operation

Normally, APL lines in operations are executed in the order of their line numbers; execution begins at the first line following the operation header and ends with the last line in the operation. You can modify this standard order of execution by using the branch function, which changes the sequence of execution by transferring control to another line in the operation.

The branch function is monadic; its argument array must be in the vector domain (or APL signals *RANK ERROR*), and its first value must be a near-integer (or APL signals *DOMAIN ERROR*). When the argument array is not a singleton, APL takes the array's first item as the object of the branch.

There are two types of branches: unconditional and conditional.

### 3.5.1 Unconditional Branches

Unconditional branches consist of a branch symbol (+), followed by a representation of the number of the operation line to which you want to transfer control. For example, the following statement causes an unconditional branch from line [5] (the current line) to line [1], thus making line [1] the next statement to be executed:

```
[5]+1
```

The argument you specify after the + can be a label, a constant, a variable, or an expression. If its value (or, if it is a vector, the value of its first item) is equivalent to an integer line number within the current definition, execution continues at that line. If the integer does not reference a line number in the current operation, the branch statement exits the operation and returns you to immediate mode or to the calling operation.

## User-Defined VAX APL Operations

### 3.5 Branching Within An Operation

APL users often deliberately specify out-of-range numbers to stop execution. Line number [0] , the operation header, is considered to be an out-of-range number; therefore, when you specify  $\rightarrow 0$ , you force an exit from the operation and a return to immediate mode or to the calling operation.

If the object of the branch is an empty array, the branch function is ignored and the normal order of execution continues (control passes to the next statement, which is not necessarily on the next line).

The possible line number specifications that can be the arguments of a branch function, and the effects of each, are summarized below.

Line Number Specifications	
Kind of Line Number Specification	Effect of the Branch
A line number within the operation	Execution continues at that line.
Zero or a line number NOT within the operation	Operation execution ends and control returns to immediate mode or the calling operation.
An empty array	The branch is ignored; execution continues at the next statement.
No argument (bare branch)	Terminates a suspended operation and all preceding pendent operations.

#### 3.5.2 Conditional Branches

Conditional branches execute as the result of the evaluation of an expression, not in response to any specific IF logic. There are several popular ways to write conditional branches. One is to use the following form:

$\rightarrow \text{line-number} \times_1 \text{logical-expression}$

For example:

[1]  $\rightarrow 9 \times_1 A > B$

Here, APL evaluates the logical expression that is the right argument of  $\times_1$ . Logical expressions return either a 1 (true) or a 0 (false); therefore, if  $A$  is greater than  $B$ , the value of the expression is  $9 \times_1 1$  , and control passes to line number [9]. If  $A$  is less than or equal to  $B$  , then the value of the expression is  $9 \times_1 0$  , and control passes to the next statement.

Another conditional branch form is:

$\rightarrow \text{logical-expressions} / \text{line-numbers}$

## User-Defined VAX APL Operations

### 3.5 Branching Within An Operation

For example:

```
[1] → ((A>B),(A<B),A=B)/7 8 9
```

In this type of conditional branch, several line numbers are specified as possible branch destinations. Each line number is associated with a logical expression. When the statement executes, APL transfers control to the first line number whose corresponding expression evaluates to 1.

You can also use this form with one logical expression and one line number. For example, in the expression  $\rightarrow(A < B)/13$ , if the logical expression  $A < B$  evaluates to 1, then control passes to line number [13]; if it evaluates to 0, then control passes to the next statement (because the expression  $0/13$  returns 0).

A third conditional branch form is as follows:

```
→(line-numbers)[K]
```

For example:

```
[1] → (LABEL1, LABEL2, LABEL3) [K]
```

Here, the value of  $K$  is used as an index for selecting a branch destination from among a group of labels representing line numbers.

### 3.5.3 Labels

Because APL automatically renames operation lines as consecutive integers when it exits from function-definition mode, problems can occur when branches refer to explicit line numbers. Alternatively, 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:

```
[4] INCR: X←X+1  
.  
.  
.  
[8] → (X<IMAX)/INCR
```

As shown, a label consists of a distinct identifier followed by a colon (:). When you specify the label in the branch function, 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: its value is local to the operation in which it appears. Label values are internally respecified upon each exit from function-definition mode. You cannot explicitly define a value for a label, and you cannot place the name of a label in the operation header.

### 3.5.4 Examples of Branching

The following are examples of user-defined functions that use branching (user-defined operators would use branching in an identical manner). In the first example, the conditional branch in line [2] controls the number of times that lines [3], [4], and [5] are executed. When  $N$  is greater than 0, the value of the expression  $0 \times 10 = N$  is an empty array, so control passes to the next statement. When  $N$  equals 0, the value of the expression is 0, so APL exits from the function.

```
v R← FACTORIAL N
[1] R←1
[2] LOOP: + 0×10=N
[3] R←R×N
[4] N←N-1
[5] + LOOP v
      FACTORIAL 5
120
```

In the next example, the unwinding back through the recursive calls of the function begins when the branch function on line [1] transfers control to line [5] :

```
v Z← FAC N
[1] + NZERO×1N=0
[2] Z+N×FAC N-1      A NOTICE THAT RECURSIVE
[3]                               DEFINITIONS ARE PERMITTED
[4] + 0
[5] NZERO: Z← 1 v
      FAC 5
120
```

A third example shows that when the argument to a branch function is an empty array, control passes to the next statement (which is not necessarily on the next line):

```
v NEXTS
[1] 1 ⋄ + 10 ⋄ 2
[2] 3 v
      NEXTS
1
2
3
```

## User-Defined VAX APL Operations

### 3.6 Comment Lines

You can include comments in an operation definition at the end of the operation header, at the end of lines containing APL statements, or on separate lines.

The first character in a comment must be a lamp character (`~`), which is formed with `\n` and `\0`. APL treats the text to the right of this symbol as a comment; this text has no effect on the execution of the function. The text of a comment can consist of any combination of valid APL characters; an illegal APL overstruck character is not invalid within a comment, because it is considered to be three valid APL characters.

Note that a comment cannot extend across line boundaries. If you want a multiline comment, you must repeat the lamp character at the beginning of each line of the comment.

### 3.7 Locking an Operation

APL allows you to lock operation definitions so that they cannot be displayed or changed. To create a locked operation, or to lock an existing operation, you open or close the operation with a del-tilde (`\~`) character (known as protected del) rather than with a simple del (`\~`). Del-tilde (`\~`) is formed with `\~` and `\~`.

The following example shows the locking of a previously unlocked function definition:

```
\~ TRIG  
.  
.  
.  
[19] \~
```

Locked operations have the following characteristics:

- They can be erased.
- They cannot be unlocked.
- They cannot be displayed or edited in any way.
- Trace, stop, and monitor vectors cannot be set for them; any trace, stop, or monitor vectors in effect at the time a function is locked are automatically cleared.

If an error occurs during execution of a locked operation, APL does not suspend the operation; instead, it exits from the locked operation and from all locked pending operations until the operation on the top of the *SI* stack is not locked. If all operations on the *SI* stack are locked, APL clears the *SI* stack.

Note that the second line of the resulting error message includes the operation name in which the error occurred, but not the line number nor the contents of the line. A  $\forall$  appears inside brackets ([ ]), instead of the line number, to indicate that the operation is locked.

You should be cautious about calling an unlocked operation from a locked operation; if the unlocked operation becomes suspended, the environment of the locked operation is available for examination.

For example, if a locked function called *PAY* uses a local variable called *SALARY*, the value of *SALARY* cannot be displayed if an error occurs during the execution of *PAY*, because APL would exit from *PAY*. However, if *PAY* calls an unlocked function called *PRINT*, and *PRINT* becomes suspended, the value of *SALARY* in *PAY* may be displayed (unless *PRINT* also has a local variable called *SALARY*).

## **3.8 Executing User-Defined Functions**

When you call a function that has arguments, you must supply the values for APL to use during execution. You include the values described in the calling syntax of the operation name. For example, the following function header indicates that the function *CALC* has two dummy arguments:

*A CALC B*

Thus, when you execute *CALC*, you must supply two values, one for *A* and one for *B*:

*25 CALC 42*

Then, when APL executes *CALC* the values 25 and 42 are used wherever *A* and *B* appear in the body of the function.

Operations that do not return explicit results, such as *CALC*, must be either the only operation or the leftmost operation in a statement. For example, the following statement is legal because the function *CALC* is the last function executed in the statement:

*25 CALC 42 + 1 + 2*

(There is no output)

## User-Defined VAX APL Operations

### 3.8 Executing User-Defined Functions

Note that APL signals *VALUE ERROR* if *CALC* is not the last operation executed. For example:

```
(25 CALC 42) + 1 + 2
11 VALUE ERROR
(25 CALC 42) + 1 + 2
^
```

If the header includes a dummy argument for axis ( $\{[k]\}$ ), you can optionally include an axis specification when you call the function. The axis argument is specified in the same manner for user-defined operations as for APL primitive functions; it follows the operation name and is enclosed in brackets ([ ]). The braces ({}) are part of the header definition and are not allowed when you call the operation.

## 3.9 Executing User-Defined Operators

User-defined operators are called in the same manner as user-defined functions. You must supply the operator name and all required arguments and operands.

If the header includes a dummy argument for axis ( $\{[k]\}$ ), you can optionally include an axis specification when you call the operator. The axis argument is specified in the same manner for user-defined operations as for APL primitive functions; it follows the operation name and is enclosed in brackets ([ ]). The braces ({}) are part of the header definition and are not allowed when you call the operation.

## 3.10 Printing Operations

The APL workspace SYS\$LIBRARY:WSPRINT.APL, contains a function which can be used to print APL objects on an LN03 printer.

To use the function, copy the workspace into the APL session and execute the function using the following form:

**[queue] WSPRINT[pw] object**

**queue**

Is the name of the LN03 print queue. If you do not specify the queue, the file will be sent to the default print queue SYS\$PRINT.

**pw**

Is an optional value to specify the print width. The APL object will be printed in portrait mode if *pw* is 80 or less. If *pw* is greater than 80 the object will be printed in landscape mode.

***object***

Is the name of the APL object you want to print.

For example:

```
)COPY SYS$LIBRARY:WSPRINT.APL  
'LN03_PRINT' WSAPRINT [132] 'FOO'
```

## 3.11 Editing Operations

APL provides four tools to allow you to define and edit operations.

- The DECwindows interface invokes full DECwindows support to more easily develop operators, functions and variables. (See Section 3.11.2.)
- The Character-Cell interface provides a TPU-based window environment to facilitate development of operators, functions and variables. (See Section 3.11.3.)
- The `)EDIT` system command allows you to edit global APL objects with the VAX TPU editor. (See Section 3.11.4.)
- Line-edit mode can be used in any APL session. (See Section 3.11.5.)

### 3.11.1 Support Considerations

When edited operations are returned to the interactive session, the lines in the operation are automatically renumbered sequentially, beginning with line [1]. Therefore, lines you insert with fractional numbers retain those numbers only while the operation is open for editing.

APL preserves leading and embedded white space (blanks or tabs) in function lines so you can format the operation (using indentation) to make it more readable. APL does not force any special spacing on labels, numeric constants, or comments when displaying the operation. Trailing white space is removed from lines to conserve storage.

Note that `PP` (Print Precision) is an implicit argument when the an editor displays numeric constants. If you are using the Character-Cell interface editor or the system command `)EDIT`, numeric constants in the operation could lose precision when displayed in VAXTPU.

## User-Defined VAX APL Operations

### 3.11 Editing Operations

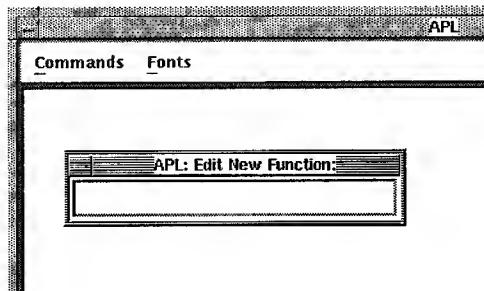
#### 3.11.2 The DECwindows Interface Editor

The DECwindows interface provides full DECwindows support of the APL product. In addition to the interactive area in the initial APL DECwindow, the transcript session, you can open one or more sessions to edit user-defined operations and variables. (See Section 2.7.3 for more information on editing variables.)

Defining a new operation, is similar to editing an existing operation. One difference is that you have to specify the type if you are defining a new operation. Follow these steps to start an edit session.

1. Click on the Commands option located on the Menu Bar in the transcript session to expose the Commands menu.
2. Select the Edit Existing or Edit New option. If you select the Edit New option, you must also select the object type. Select either the Function or Operator option.
3. Click on the input area of the dialog box, shown in Figure 3-1, and enter the name of the operation.

**Figure 3-1 DECwindows Interface Operation Name Dialog Box**



4. The Title Bar in the edit session is the name of the operation. If this is a new operation, only the header will be displayed. If you are editing an existing operation, the definition will be displayed.

You can enter text to edit the header or body of the operation or you can use the mouse to copy text from one edit session to another edit session. The mouse can be used to copy text from the transcript window, to an edit window. You cannot copy text into the transcript window.

## User-Defined VAX APL Operations

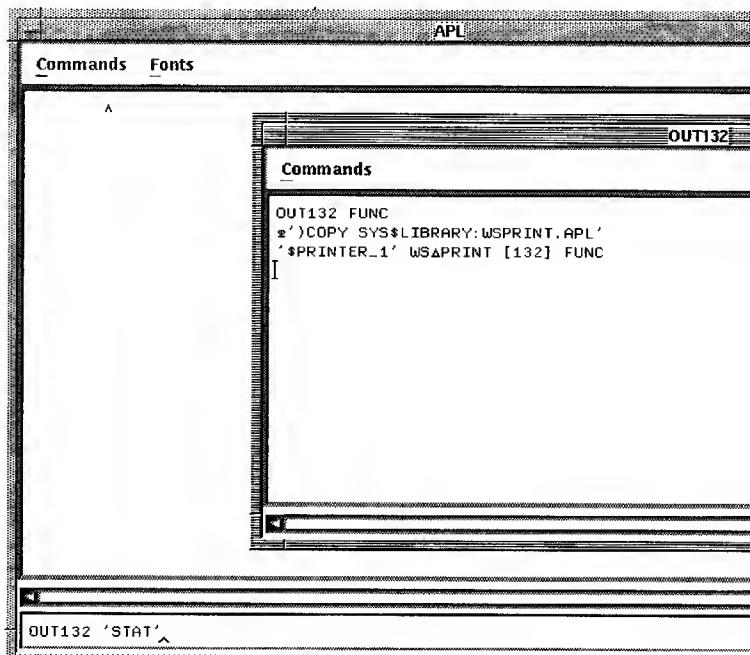
### 3.11 Editing Operations

5. Finish defining the function by selecting Exit, Update Workspace or Quit from the Commands option on the menu bar in the edit session. Figure 2–3 shows the options that are available after you click on the Commands option in the edit session.

When you return to the transcript session, you can enter commands to call the operation. You do not have to close the edit session to return to the transcript session. For example, in Figure 3–2 the edit session window has not been closed. The command entered on the input line uses the operation recently defined in the edit session.

If you select the Exit option from the edit session Commands menu, the definition is written to the workspace and the edit session is closed. The Quit option closes the edit session without saving changes in the operation definition.

**Figure 3–2 DECwindows Interface Edit Session Example**



## User-Defined VAX APL Operations

### 3.11 Editing Operations

#### 3.11.3 The Character-Cell Interface Editor

The Character-Cell interface provides a TPU/EVE based window environment for APL sessions on the Digital VT220, VT240, VT320, VT330, VT340 and DECterm terminals. This environment inserts the text of the operation you are editing into a temporary holding area, a buffer. You can display more than one buffer on the screen at one time and edit more than one operation during an APL session. (See Section 1.3.3.)

You can view operations that are suspended or pendent, but you cannot modify them. If you attempt to edit an operation that is suspended or pendent, APL puts the appropriate message in the TPU/EVE message buffer.

To start an edit session, press the Do or PF4 key or enter Ctrl/B to display the Command: prompt. Enter the following command, substituting the name of the operation being edited for *operation*.

*GET operation← v*

The contents of the operation are displayed in the window of the new buffer. The end-of-file marker defines the end of the buffer. It is only visible on the screen and is not saved as part of the operation. The status line shows the name of the buffer, *operation← v*. (See Figure 3-3.)

You can edit the header and the body of the operation. Text can be added by using one of the following methods:

- Entering the APL characters from the keyboard
- Inserting text copied from the APL SESSION Buffer or another edit buffer
- Including entire files

The EVE Help Utility has specific information about copying text. Enter *HELP* at the Command: prompt and look at copy, cut, paste and restore. Enter *HELP INCLUDE FILE* at the Command: prompt to get more information about including files. (Press the Do or PF4 key or enter Ctrl/B to reveal the Command: prompt.)

To write the contents of a buffer to the interactive APL session, use the following form:

*WRITE [[operation ← v]]*

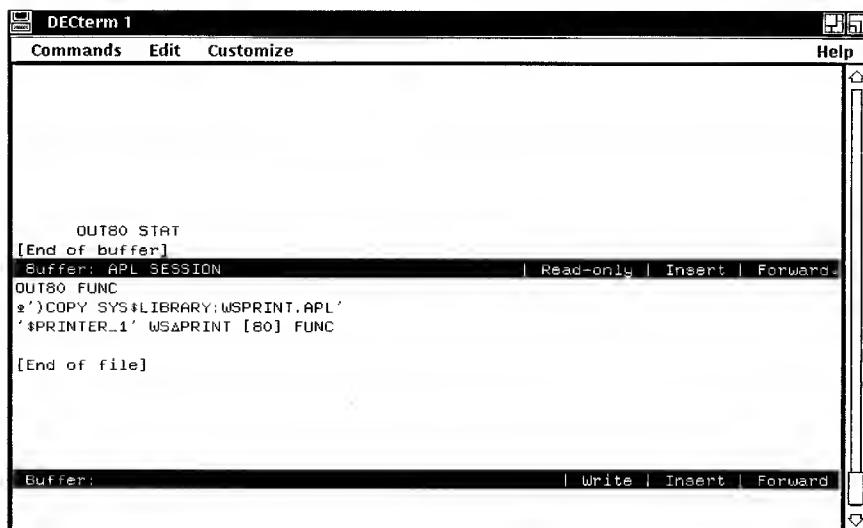
The name of the operation is optional. If you do not specify the name of the operation, EVE uses the name associated with that buffer.

## User-Defined VAX APL Operations

### 3.11 Editing Operations

Figure 3–3 shows an APL session using the Character-Cell interface. In the figure, two windows are open. The top edit window shows the user-defined operation *OUT80*. The bottom window is the interactive session *APL SESSION*. The command on the command line updates the interactive session with the contents of the edit window.

**Figure 3–3 Character-Cell Interface Operation Edit Example**



To return to the interactive session, enter *BUFFER APL SESSION* at the Command: prompt. Alternatively, if you are using a split screen with the interactive session in one window and an edit session in another, you can return to the interactive session by entering *OTHER* at the Command: prompt. If you are using a mouse and multiple windows, you can position and click the mouse on the interactive session to make that the active session.

When an operation returns to APL, APL embeds a <CR><LF> at the end of each line. If there is a <CR><LF> inside any single record, APL treats the <CR><LF> as the end of an operation line and forms a new record (note that if the resulting record contains an unbalanced delimiter, APL signals an error). When a line of an operation contains embedded <CR><LF> characters, it is written out to VAXTPU as two records. For example:

## User-Defined VAX APL Operations

### 3.11 Editing Operations

```
(Defined with line-editor in APL)
  v FOO
[1]  Z←'AA
AA'v
      (Opened for edit with Character-Cell interface editor)
FOO
Z←'AA
AA'
      (This returns to APL)
  v FOO [] v
  v FOO
[1]  Z←'AA
[2]  AA'
v
```

#### 3.11.4 The )EDIT System Command Editor

The )EDIT system command allows you to edit APL operations with the VAXTPU editor. The default object type is a function. If you want to edit an operator, you must specify a value 4 for the name class qualifier (/NC:4).

When you invoke )EDIT, APL creates a temporary file containing the object you want to edit and then invokes VAXTPU. When you exit VAXTPU, APL reads the edited file from VAXTPU into the workspace. You can view operations that are suspended or pendent, but you cannot modify them. APL returns you to the VAXTPU editor if an error occurs as the file reenters the workspace.

Use the form following form to edit an APL object:

)EDIT *operation*[[*qualifiers*]]

#### *operation*

Is the name of the APL operation you want to edit.

#### *qualifier*

Is one or more of the optional qualifiers. See *VAX APL Reference Manual* for more documentation on the )EDIT system command.

When you invoke )EDIT for an existing operation, you can specify the /LC qualifier if you want to view the line numbers associated with the lines of the operation. However, these line numbers may change since APL reassigns them when the file returns to the APL environment. If you add new lines to the operation during the VAXTPU session, it is not necessary to specify new line numbers. If you do not specify /LC when you invoke )EDIT, APL sends the canonical representation of the operation (no line numbers) to VAXTPU.

The display of a user-defined operation passed from APL to VAXTPU is dependent on  $\square PP$ . Therefore, numeric constants in the operation could lose precision when displayed in VAXTPU. You can use the  $/PP$  qualifier when you invoke  $)EDIT$  to specify a different value for the print precision.

When you invoke  $)EDIT$  for a new or empty operation, APL generates an empty temporary file for VAXTPU. If the file is empty (contains 0 records) when it returns to APL, it arrives as an empty operation and has the same name you specified when you used the  $)EDIT$  expression. By definition, an empty operation contains no lines and has a header that includes only the name of the operation; the header does not include specifications for local variables.

When a line of an operation contains embedded  $<CR><LF>$  it is written out to VAXTPU as two records. When an operation returns to APL, APL embeds a  $<CR><LF>$  at the end of each line. If there is a  $<CR><LF>$  inside any single record and you specify  $/MODE: 3$ , APL treats the  $<CR><LF>$  as the end of an operation line and forms a new record. If the resulting record contains an unbalanced delimiter, APL signals an error. If you specify  $/MODE: 2$ , the  $<LF>$  will not return.

Data in VAXTPU	Line in APL	
'aa	Mode 3 'aa<CR><LF>aa'	Mode 2 'aa<CR><LF>aa'
aa'		
'aa<CR><LF>aa'	APL signals EDIT COMMAND ERROR	'aa<CR>aa'
aa<CR><LF>aa	aa	aa
	aa	aa

Note that the  $<CR><LF>$  symbol does not appear visually in the APL environment, although it does cause data to be displayed on a new line.

#### 3.11.5 The Line-Editor

APL provides a line editor that allows you to add, delete, and change definition lines, alter the operation header, or even edit individual characters in a line.

You must be in function-definition mode to edit an operation. To open an existing function or operator for editing, type the del ( $\nabla$ ) character followed by the operation name; do not type the entire operation header. To edit a new operation, type the del character and the full operation header.

You cannot modify a pendent operation.

## User-Defined VAX APL Operations

### 3.11 Editing Operations

APL responds with a display of a line number (in brackets). If this is an existing operation, the number will be equal to the last line number in the operation plus one. For new operations, the number will be 1. (The operation header is on line 0.) In turn, you respond, on the same line, with an edit command or with the text of a line that you want to append to the operation definition. For example, the following deletes line [5] from the function *STAT* :

```
    v STAT  
[7]  [Δ5]v
```

Alternatively, you could enter an editing command immediately following the operation name. Thus, the following also deletes line [5] in *STAT* :

```
v STAT[Δ5]v
```

Once you have control of the editor, you may enter as many editing commands as you need. When you complete the editing session, you type a *v* to close the operation and return to immediate mode. You may type the closing *v* on the same line as an edit command.

You can execute system commands from within function-definition mode, and APL interprets them as if they were entered in immediate mode. If you desire the system command to be executed as part of a user-defined operation, you can use any of the execute functions (see the *VAX APL Reference Manual*). *)EDIT* and *)STEP* cannot be executed from inside function-definition mode.

Note that if you type a closing *v* immediately after entering the header for a new operation, you create an empty operation, that is, an operation that has a header but no body.

If you want to abort an editing session, enter the abort input signal. APL returns you to immediate mode and leaves the operation as it was before the editing session. If you abort an editing session while defining a new operation, the new operation is not created.

#### 3.11.5.1 Line Editing Commands

You use line editing commands to add, change, or delete lines in an operation definition. The function editor commands are summarized below.

Form	Meaning
v <i>operation-name</i>	Open operation for editing.
v	End operation editing.
v	End operation editing and lock operation.

Form	Meaning
[n]	Define or change line <i>n</i> .
[Δ n]	Delete line <i>n</i> .
[Δ n,l]	Delete <i>l</i> lines beginning at line <i>n</i> .
[Δ ]	Delete the current line.
[n□]	Display line <i>n</i> .
[□ n]	Display all lines, beginning with line <i>n</i> .
[□ n,l]	Display <i>l</i> lines beginning at line <i>n</i> .
[□ ]	Display all lines of function.
[n\$/s1//s2/] [m□n]	Beginning at line <i>n</i> search for string <i>s1</i> and replace it with string <i>s2</i> . Do character editing of line <i>m</i> ; begin at character position <i>n</i> .
[◦□n]	Open last entered line for character editing, beginning at character position <i>n</i> .
[◦□ ]	List last entered line.
/	In character editing, type beneath each character you want to delete.
<i>n</i>	In character editing, insert <i>n</i> (a numeric digit) spaces before current character.
<i>letter</i>	In character editing, insert multiples of 5 spaces— <i>A</i> inserts 5 spaces, <i>B</i> inserts 10 spaces, and so on.
, <i>text</i>	In character editing, insert <i>text</i> before the current character.
line feed	Delete everything on the line to the right of the line feed.

You can enter more than one editing command at a time. For example, the following directs APL to open the function *FOO* for editing, delete line [8], display the entire function, and then begin character editing of line [5] at character position 16:

```
V FOO[Δ8][□][5□16]
```

If you enter an invalid editing command, APL signals *DEFN ERROR* (definition error). APL often includes a secondary error message that provides specific information about what caused the error. When you open an operation for editing, you can add lines in response to the bracketed line numbers. For example, the function named *STAT* exists in form shown (an incorrect algorithm) in the example.

## User-Defined VAX APL Operations

### 3.11 Editing Operations

```
V 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] V
V STAT
[6] ♂FUNCTION RETURNS VALUE OF STANDARD
[7] ♂ DEVIATION OF X
[8] STANDX<VARX*0.5
[9] V
```

To replace existing lines in an operation definition, specify the number (in brackets) of the line you want to replace, followed by the new text for the line.

For example, you can replace line [1] of the function *STAT* as follows:

```
V STAT
[9] [1] SUMX++/X
[2] V
```

The line number you specify must be a nonnegative number less than 10,000. If the line does not currently exist, it is inserted.

To insert a new line between existing lines of an operation definition, specify the new line number (in brackets) followed by the text of the new line. The line numbers you insert must be positive numbers up to but not including 10,000, with or without a decimal point, and with no more than five decimal places.

For example, to insert a line between lines [5] and [6], you could specify any number from [5.00001] to [5.99999]; to insert a line before line [1], you could use any number from [0.00001] through [0.99999], and so on. In the following example, new lines are inserted between existing lines [0] and [1], and [5] and [6], respectively:

```
V STAT
[1] [0.5] ♂SUM ITEMS OF ARRAY X
[0.6] [5.5] VARX<SUMX2÷NSUBJ)-MEANX*2
[5.6] V
```

After each insertion, APL prompts with the next line number after the inserted line. To determine the next line number, APL truncates trailing zeros and then increments the current line number by  $1E^{-D}$ , where  $D$  is the number of decimal places in that line number. Thus, the next line after [0.5] is [0.6], the next line after [5.5] is [5.6], and the next line after [8.29] is [8.3].

---

**Caution**

---

Note that line number [6] follows [5.9], and then line number [7], not [6.1], follows [6].

---

To delete existing lines in an operation definition, type a delta ( $\Delta$ ) and a line number or range of line numbers within square brackets.

For example, to delete line [5] of *STAT*, you would type the following:

```
v STAT
[11]  [Δ5]
[6]   v
```

To delete a range of line numbers, specify two numbers after the  $\Delta$ : the first number identifies the starting point for the deletions, and the second number specifies the total number of lines to be deleted. Thus, the delete command [ $\Delta$  3,4] deletes four lines beginning with line [3].

When you specify a range of numbers, only existing lines are counted; for example, if during an editing session the existing lines in the function are lines [1], [2], [4], [5], and [6], then the delete command [ $\Delta$  2,3] deletes lines [2], [4], and [5]. Note, however, that the initial line is still counted even if it does not exist; that is, if the previous delete command had been [ $\Delta$  3,3], then only two lines ([4] and [5]) would have been deleted.

If the number of lines remaining in the operation following the starting point is less than the number of lines specified by the second argument in the delete command, all the remaining lines are deleted. After the deletions, APL prompts with the next available line number.

You can delete the current line of the operation by responding to the line number prompt with [ $\Delta$ ]. In the following example, lines [5], [6], and [7] are deleted:

```
v STAT
[11]  [Δ5]
[6]   [Δ]
[7]   [Δ]
[8]   v
```

APL displays the number of the next line after the deleted one. You can then type a new line, edit another line, or exit from function-definition mode by typing a  $v$ . After you close the operation, APL renumerbs the lines.

## User-Defined VAX APL Operations

### 3.11 Editing Operations

#### 3.11.5.2 Displaying Operation Lines

APL allows you to display individual operation lines, all of an operation's lines, a range of an operation's lines, or all of an operation's lines from a specified line to the end of the operation.

To display an individual line, type the line number and a quad (`□`) in square brackets. For example, to display line [3] of function `STAT`, type:

```
v STAT [3□]  
[3]  SUMX2++/(X*2)
```

APL prints the number of the line just displayed to give you the opportunity to specify a new version of the line. You can then perform any editing operation, or you can exit from function-definition mode by typing a `v`.

To display an entire operation definition, type the quad (in brackets) without a line number. The following example displays the entire function named `STAT` (now a correct algorithm):

```
v STAT[□]v  
v STANDX+NSUBJ STAT X  
[1]  aSUMX ITEMS OF ARRAY X  
[2]  SUMX++/X  
[3]  SUMX2++/(X*2)  
[4]  aCOMPUTE MEAN, VARIANCE, STANDARD DEVIATION  
[5]  MEANX←SUMX+NSUBJ  
[6]  VARX←(SUMX2+NSUBJ)-MEANX+2  
[7]  aFUNCTION RETURNS VALUE OF STANDARD  
[8]  a DEVIATION OF X  
[9]  STANDX←VARX*0.5  
v
```

The `v` characters preceding the operation header and following line [9] delimit the operation and identify its name. The characters do not change the mode as the operation prints. APL displays the number of the next line after the final line of the operation to give you the opportunity to add new text, perform any other editing operation, or exit from function-definition mode by typing a `v`.

To display the operation definition from a particular line number to the end, type the quad character and the line number at which you want the display to begin. For example:

```
v STAT[□8]v  
[7]  aFUNCTION RETURNS VALUE OF STANDARD  
[8]  a DEVIATION OF X  
[9]  STANDX←VARX*0.5  
v  
[10]
```

APL displays the number of the next line after the final line of the operation definition, in this case [10], to give you the opportunity to add more text. You can then perform any editing operation or exit from function-definition mode by typing a `v`.

To display a range of line numbers, specify two numbers after the `o`: the first number identifies the starting line number, and the second number specifies the total number of lines to be listed. Thus, the list command `[o 5,3]` displays three lines beginning with line [5].

When you specify a range of numbers, only existing lines are counted; for example, if during an editing session the existing lines in the function are lines [1], [2], [4], [5], and [6], then the display command `[o 2,3]` lists lines [2], [4], and [5]. Note, however, that the initial line is still counted even if it does not exist; that is, if the previous display command had been `[o 3,3]`, then only two lines ([4] and [5]) would have been listed.

If the number of lines remaining in the operation following the starting point is less than the number of lines specified by the second argument in the display command, the list ends with the last line of the operation. After the display, APL prompts with the next available line number.

Another way to exit from function-definition mode after displaying all or part of an operation is to type the closing `v` on the same line as the display request. For example:

```
v STAT[o 8]v
[7]   ⎕FUNCTION RETURNS VALUE OF STANDARD
[8]   ⎕ DEVIATION OF X
[9]   STANDX←VARX×0.5
v
```

Because you have already typed the closing `v`, APL does not prompt you for the next line of the function.

### 3.11.5.3 Search and Replace Strings

To replace one string with another in an operation, you can specify a search and replace expression using the following form:

```
[ [[line-number]] $d[[search-string]] d[[d[[replacement-string]] d]] ]
```

#### *line-number*

Specifies the operation line where you want the search to begin. The search continues from this line to the end of the operation or until an occurrence of the search string is found. If you do not specify *line-number*, the search begins at the current line. If you specify the `o` symbol, the search occurs on the most recently entered immediate mode line.

## User-Defined VAX APL Operations

### 3.11 Editing Operations

#### **a**

Specifies a delimiter that is any character you choose to separate the fields of the search and replace expression; it cannot be a carriage return. Once you declare the delimiter in any given expression, it must not change. The important consideration when choosing a character for a delimiter is that it does not appear in either the search string or the replacement string.

#### ***search-string***

Is the string of characters that you want to search for. APL searches for exact matches on each line of the function; a match that spans more than one line is not considered a match. If you do not specify a value for *search-string*, APL searches for the string specified in the previous search and replace expression. If there is no previous *search-string* value, APL signals *DEFN ERROR (NO PREVIOUS SEARCH STRING)*. If APL cannot find the string, it prompts you for the next line at the end of the function.

Although *search string* is an optional parameter, the before and after delimiters are not. You must specify these delimiters.

#### ***replacement-string***

Is the string of characters that you want to put in place of the search string. When APL finds an occurrence of the search string, it is deleted and replaced with the value of *replacement-string*. If you specify a value for *replacement-string*, the delimiters are required. If you do not specify a value for *replacement-string*, you do not have to specify its before and after delimiters. However, the meaning of the search and replace expression changes depending whether these delimiters are specified in this optional situation. If you specify them, the search string is deleted. If you omit them, APL finds the search string (if possible) but does not delete it. For further description of this behavior, see the following list of form variations.

Note that the combined length of *search-string* and *replacement-string* should be less than the current *OPW* setting.

Since both the search and replace strings are optional, there are six variations of the search and replace expression. The forms for the variations are shown below. Note that the slash (/) symbol is used as the delimiter, *s1* equals *search-string*, and *s2* equals *replacement-string*:

Variation	Meaning
<code>[\$/s1//s2/]</code>	Replace <i>s1</i> with <i>s2</i> .
<code>[\$//s2/]</code>	Replace next occurrence of the previous search string with <i>s2</i> .

Variation	Meaning
[\$/s1///]	Delete s1.
[\$/s1/]	Search for s1, but do not delete it.
[\$////]	Delete next occurrence of the previous search string.
[\$//]	Search for the previous s1, but do not delete it.

Note that APL begins each search on the current line unless you specify a value for *line-number*. When you are doing successive searches using the simple search form (no deletes), you must specify the current line number plus one after you have found the first occurrence of the string.

#### 3.11.5.4 Editing the Operation Header

You can edit the name or arguments of an operation header by accessing line number [0]. You can display and replace the operation header just as you can any other line in the operation, but you cannot delete the header using [ $\Delta$  0]. You must have specified a valid operation header before leaving function-definition mode. If an error occurs while you are changing the header, the original header is retained.

The following example displays the operation header:

```
v STAT
[12] [0[]]
      v STANDX←NSUBJ STAT X
[0]  v
```

Notice that the header is displayed without a line number. When you specify a character position in the header, APL types the header with line number [0] and without the v. For example:

```
v STAT
[10] [0[]7]
[0]  STANDX←NSUBJ STAT X
      ↑
```

Note that the up-arrow in this example indicates the position of the cursor or terminal head. It does not appear on your terminal.

## User-Defined VAX APL Operations

### 3.11 Editing Operations

#### 3.11.5.5 Character Editing

In addition to providing a way to edit operation definitions line by line, APL provides a way to edit lines character by character.

To begin character editing, specify the line number of the line to be edited and the estimated character position at which editing is to begin using the following form:

[*line-number*] [*character-position*]

For example:

```
    V DIESEL
[7]  [10]10]
[1]  A←R * GAMMA - 1
      ↑
```

APL displays the line, performs a <CR><LF>, and positions the cursor or terminal head at the position you specified (the up-arrow in the example indicates the position specified; it does not appear on the terminal). Once you are in the desired position, you can enter the commands to delete unwanted characters and insert blanks in the line where new text is needed or inserting text. Use the keys in the following table to enter the commands.

Note that if you want to add characters to the end of a line without changing any of the rest of the line, you should specify character position 0. APL will place the cursor just after the end of the line, and will be ready to insert the text. editing commands. Thus, you may immediately make your additions to the end of the line.

Character	Meaning
Space	Moves the cursor one character to the right.
Backspace	Moves the cursor one character to the left (provided that terminal line editing is turned off; see □ <i>TLE</i> in the <i>VAX APL Reference Manual</i> ).
Slash(/)	Deletes the character above the slash.
Digit	Inserts, preceding the character above the digit, a number of spaces equal to the value of the digit.

## User-Defined VAX APL Operations

### 3.11 Editing Operations

Character	Meaning
Letter	Inserts, preceding the character above the letter, a number of spaces equal to 5 times the relative position of the letter. Typing an <i>A</i> inserts 5 spaces; typing a <i>B</i> inserts 10 spaces, and so forth.
Comma ( , ) <i>text</i>	Inserts, preceding the character above the comma, the text that follows the comma. (APL does not process other editing characters to the right of the comma editing command.)
Any other	Gives an error.

If the number of spaces and text (minus the number of slashes) that you add extends the length of the current line to exceed the length of the terminal line (the value of  $\square PW$ ), APL signals *OUTPUT LINE TOO LONG (PAGE WIDTH EXCEEDED)*.

If the character position is negative or greater than either the page width ( $\square PW$ ) or 255, APL signals *EDIT ERROR (COLUMN POSITION OUT OF RANGE)*.

When you finish entering your commands, press the Return key; APL responds by displaying the line without the deleted characters and with the inserted spaces and text. Then, APL positions the cursor at the first inserted space on the line to be edited, or, if you did not insert spaces, at the end of the line. If necessary, you can enter new text in the spaces or at the end of the line. If you used the comma command, APL positions the cursor after the text inserted by the comma and not at the first inserted space.

Press the Return key to return to function-definition mode. APL replaces the existing line in the operation with the new one and prompts with the next line number. If you used a comma, you can perform another character-edit operation. To return to function-definition mode after using the comma, press the Return key twice.

If you change the line number while you are editing the line, any edits you make are associated with the new line number; the original line remains unchanged.

If you edit the header (line 0) and change the operation's name, then when you exit from function-definition mode, a new operation is created with the new name. The new operation is an edited version of the old one; the old operation still exists in its original form.

You cannot delete a line by typing a slash (/) beneath all of its characters.

## User-Defined VAX APL Operations

### 3.11 Editing Operations

While you are in character editing mode, APL keeps track of your current line with an internal pointer mechanism. If you enter a string of edit commands, APL interprets the commands sequentially from left to right and updates the current line after each one. For example,  $\left[\Delta 2\right]\left[\Delta 4\right]\left[\Box\right]$  deletes lines 2 and 4 of an operation, and then displays the operation. After APL deletes line 2, the current line is line 3. After APL deletes line 4, the current line is line 5. After APL displays the operation, the current line is the last line plus one.

If you append data to a series of edit commands, the last edit command in the series determines the current line that is created by the data. The following example illustrates the need to foresee the current line when you append data in this manner:

```
    V F
[1] 1
[2] [1]7]
[1] 1
        4
[1] [ ]21
[2] [ ]
    V F
[1] 1
[2] 21
    V
```

The following example illustrates how character editing is used to correct line [1] of a function named FRYER:

```
[1] T+(LETTR=STRING/18P,STRING
```

There are several errors in this line:

- The word *LETTER* is misspelled *LETTTR*.
- The right parenthesis is missing after the first occurrence of *STRING* .
- The 8 should not appear at all.
- The *P* should be *P* .

You could edit this line as follows:

```
    ^ FRYER
[5] [1]14]
[1] T+(LETTR=STRING/18P,STRING
        1           1 //1
[1] T+(LETT R=STRING /1 ,STRING
```

## User-Defined VAX APL Operations

### 3.11 Editing Operations

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. (On some terminals, using the space bar to move over text that you want to preserve will cause the text to disappear from the video screen.) The new line looks like this:

```
[1] T←(LETTER=STRING)/\p,STRING
```

When you press the Return key, the new line replaces the existing line [1] in your function definition.

You could also use the comma character-editing command to correct line [1] of FRYER:

```
^ FRYER
[5] [1\14]
[1] T←(LETTR=STRING/\18P,STRING
      ,E
[1] T←(LETTER=STRING/\18P,STRING
      ,
      ) (move cursor, add command and text)
[1] T←(LETTER=STRING/\18P,STRING
      //,\p
      ) (move cursor, add commands and text)
[1] T←(LETTER=STRING)/\p,STRING
[2]
```

You cannot include an unbalanced quotation mark when using comma editing. When there is an unbalanced quotation mark, APL waits for the matching mark just as it would when you enter a quoted string outside of comma editing mode.

Note that when you use the comma character-editing command after repositioning the cursor with the Backspace key, you may insert unexpected blank spaces:

```
\ F
[1] THIS IS A LINE
[2] [1\15]
[1] THIS IS A LINE
      ↑ (APL positions cursor here)
[2] , X
      (User types 8 Backspaces, Space X Space)
      (New line contains 4 extra spaces)
[1] X THIS IS A LINE
      ↑
```

## User-Defined VAX APL Operations

### 3.11 Editing Operations

The extra spaces are inserted because of the original placement of the cursor. APL does not recognize whether the spaces belong to the comma string or not, and so they are included in it.

You can cancel the revision of the operation line by typing a deliberate character error (such as an illegal overstrike) after a character-editing display.

When APL encounters a character error, it displays both an error message and the line, with a caret (^) pointing at the illegal character. The old line is retained; you may edit the new line in immediate mode.

You can escape from character-editing mode at any time by entering the abort input signal. APL cancels any changes you have made to the line. You may then enter another editing command or exit from the function editor by typing  $\nabla$ .

Note that if you are in character-editing mode and you want to abort the editing session entirely, you must enter the abort input signal twice: once to escape from character-editing mode and once to abort the session and return to immediate mode. APL ignores any changes you have made during the session.

#### 3.11.5.6 Editing Lines That Contain Control Characters

In character-editing mode, APL outputs ASCII control characters differently. Normally, APL sends control characters to the terminal for interpretation by the terminal. However, if the control character is part of a literal that you are editing in character-editing mode, it is displayed as the overstruck character (see Table 1-4 and is not sent to the terminal for interpretation).

This way of displaying control characters is particularly important when a literal in an operation definition contains a <CR><LF>. If the <CR><LF> were output for interpretation by the terminal, it would be impossible to edit the literal. Instead, APL displays the <CR><LF> as  $\text{M} \text{ X}$ ; then, you can edit the literal using the normal character editing techniques.

#### 3.11.5.7 Editing in Immediate Mode

The way you edit lines in immediate mode is similar to the way you edit them in function-definition mode. In immediate mode, line edits affect the last nonblank line entered from the keyboard (not including any previous immediate-mode or  $\nabla$  editor editing commands). Because immediate-mode lines do not have line numbers, to initiate editing you type, in brackets: a jot character ( $\circ$ ), a quad character ( $\square$ ), and an integer representing the character position at which editing is to begin. For example:

```

ACRON+INIT1, INIT2[ INIT3
11 VALUE ERROR
ACRON+INIT1, INIT2[ INIT3
          ^
[.025]
ACRON+INIT1, INIT2[ INIT3
          /
1
ACRON+INIT1, INIT2, INIT3

```

You could perform this immediate-mode edit with the comma-insertion command. In the following example, the user recalls the last line, enters a slash (/) to delete a character, follows the slash with the comma-insertion command, and enters a second comma as the inserted character. After the user enters a Return, the line is displayed again with the cursor underneath the character following the inserted comma. After a second Return, the line is displayed with the cursor at the end of the line (awaiting additional modification). Finally, APL executes the line after the third Return.

```

[.025]
ACRON+INIT1, INIT2 [INIT3
          /,
ACRON+INIT1, INIT2, INIT3
          ^
ACRON+INIT1, INIT2, INIT3
(APL executes line)

```

Then, immediate-mode editing proceeds exactly as character editing in function-definition mode. Note, however, that after you press the Return key to conclude the final edits, APL executes the line. If you merely want to display (but not execute) the last line entered from the keyboard, type [.0] (with no character position).

## 3.12 Examples of Defined Operations

This section gives an example of niladic, monadic, dyadic, and ambivalent user-defined functions as well as a function that takes an axis. There are also examples of a dyadic operator and a monadic operator that take an axis.

### 3.12.1 Niladic Function

The function *AVG* is niladic; it takes no arguments. *AVG* does not return an explicit result; thus, it cannot be used as an argument to another operation. Also note the value of *VECTOR*, both as a global variable outside the definition of *AVG* and as a local variable inside *AVG*:

## User-Defined VAX APL Operations

### 3.12 Examples of Defined Operations

```
V AVG;VECTOR
[1]  ⌂ ← 'ENTER THE VECTOR TO BE AVERAGED: '
[2]  'THE RESULT IS ' ;(+/VECTOR)÷ρ,VECTOR+□XQ⌂
[3]  V
    VECTOR←'ABCD'
    AVG
ENTER THE VECTOR TO BE AVERAGED: 3 5 4 6 7
THE RESULT IS 5
VECTOR
ABCD
100×AVG
ENTER THE VECTOR TO BE AVERAGED: 3 5 4 6 7
THE RESULT IS 5
11 VALUE ERROR (FUNCTION DOES NOT RETURN A RESULT)
100×AVG
^
```

#### 3.12.2 Monadic Function

The function *AVERAGE* is monadic; it takes one argument, which is placed to the right of the function name. *AVERAGE* returns an explicit result; thus, it can be used as an argument to another function (in this case, the multiplication function).

```
V ANS←AVERAGE VEC
[1] ANS←(+/VEC)÷ρ,VEC
[2] V
    AVERAGE 3 4 5 6 7
5
100×AVERAGE 3 4 5 6 7
500
```

#### 3.12.3 Dyadic Function

The function *IN* is dyadic; it takes both a right and a left argument:

```
V LETTER IN STRING;T
[1]  ⌈RETURNS NUMBERIC POSITION WHERE LETTER
[2]  ⌈APPEARS IN STRING
[3]  T←(LETTER=,STRING)/\:ρ,STRING
[4]  →END×\:0=ρT
[5]  →\:0,□←T
[6]  END: 'NO OCCURRENCES'
[7]  V
    LETTER←'C'
    T←'GLOBAL'
    LETTER IN 'ABCACBC'
3 5 7
```

## User-Defined VAX APL Operations

### 3.12 Examples of Defined Operations

```
LETTER IN 'LMNOP'  
NO OCCURRENCES  
T  
GLOBAL  
LETTER  
C
```

#### 3.12.4 Ambivalent Function

The following function is ambivalent: it may take either one or two arguments. If two values are supplied, the function adds them together; if a single value is supplied, the function adds one to the value:

```
∨ Z←{A} PLUS B  
[1] →(0=□NC 'A')/MONAD      □NC=0 MEANS NAME NOT IN USE  
[2] Z+A+B  
[3] +0  
[4] MONAD: Z←1+B  
[5] ∨  
    5 PLUS 6  
11  
    PLUS 7  
8
```

#### 3.12.5 Function with Axis

The following example defines a dyadic function that accepts an axis argument when invoked. In the header, the dummy argument for the axis is enclosed by brackets ([ ]) and braces ({ }). The brackets indicate an axis argument, the braces indicate that the axis is optional.

The *FIRST\_ROTATE* function performs the same operation as the primitive *⊖* function except that it counts the axes in reverse order. You can call *FIRST\_ROTATE* just as you would call a similar primitive function.

```
∨ Z←A FIRST_ROTATE {[K]} B  
[1] →(0≠□NC 'K')/L ◊ K+□IO  
[2] L: Z←A⊖[(□IO+ppB)-K+~□IO]B  
[3] ∨  
    □IO+1  
    X←3 ⌊ 'ABCDEFGHI'  
    X  
ABC  
DEF  
GHI  
    1 2 3⊖X  
DHC  
GBF  
AEI
```

## User-Defined VAX APL Operations

### 3.12 Examples of Defined Operations

```
1 2 3 FIRST_ROTATE X
BCA
FDE
GHI
1 2 3 e[1]X
DHC
GBF
AEI
1 2 3 FIRST_ROTATE[1]X
BCA
FDE
GHI
1 2 3 e[2]X
BCA
FDE
GHI
1 2 3 FIRST_ROTATE[2]X
DHC
GBF
AEI
    IO←0
1 2 3 eX
DHC
GBF
AEI
1 2 3 FIRST_ROTATE X
BCA
FDE
GHI
1 2 3 e[0]X
DHC
GBF
AEI
1 2 3 FIRST_ROTATE[0]X
BCA
FDE
GHI
1 2 3 e[1]X
BCA
FDE
GHI
1 2 3 FIRST_ROTATE[1]X
DHC
GBF
AEI
```

If you specify an axis when you call a user-defined function that is not defined as accepting an axis argument, APL signals *AXIS DOMAIN ERROR (INCORRECT OPERATION)*. If you omit the optional axis argument of such a function, and the body of the function attempts to reference the axis value when the function

## User-Defined VAX APL Operations

### 3.12 Examples of Defined Operations

is called, APL signals *VALUE ERROR* and suspends the function at the point of the reference. For example:

```
    ∇ Z ← F X
[1]  Z←X
[2]  ∇
      F[1] 5          aSPECIFY AXIS
  30 AXIS DOMAIN ERROR (INCORRECT OPERATION)
      F[1] 5          aSPECIFY AXIS
      ^
      ∇ Z ← G{[K]} X
[1]  Z← φ[K] X
[2]  ∇
      G 6          aNO AXIS
  11 VALUE ERROR
G[1] Z← φ[K] X
      ^
      )SI
G[1] *
```

#### 3.12.6 Dyadic Operator

The following example defines a user-defined operator named *BASE\_N*, which applies a function on arguments of different number system bases.

*BASE\_N* is a dyadic operator. When *BASE\_N* is invoked, the user specifies a function as the left operand (*FNC*) and the number system base of the arguments as the right operand (*N*). The resulting derived function is ambivalent.

```
    ∇ Z ← {A} (FNC BASE_N N) B ;X
[1]  → (0=□NC 'A') / MON      aCHECK FOR DERIVED FUNC VALENCE
[2]  Z← (N↑A) FNC N↓B        aDYADIC EVALUATION
[3]  L0: → (1≠X ← φ,N) /L1    aCHECK FOR 1-ITEM BASE
[4]  X← 「N@」/,Z            aCOMPUTE LEFT ARG TO ENCODE
[5]  L1: Z←(XφN)↑Z   ◊ → 0    aCONVERT BACK TO BASE N
[6]  MON: Z←FNC N↓B   ◊ → L0  aMONADIC EVALUATION
[7]  ∇
      aADD 5 TO 6 IN BINARY GIVING 11
      1 0 1 (+ BASE_N 2) 1 1 0
1 0 1 1
      a10 TIMES 28 IN OCTAL
      1 2 (× BASE_N 8) 3 4
4 3 0
      a5 FEET 3 INCH MINUS 2 FEET 9 INCH
      5 3 (- BASE_N 3 12) 2 9
2 6
```

## User-Defined VAX APL Operations

### 3.13 Debugging Operations

#### 3.13 Debugging Operations

APL provides several features that help you find logic errors in the operations you write. These features include the following:

Feature	Description
Status Vector	Reports on the state of active operations in your workspace (see <code>)SI</code> in the <i>VAX APL Reference Manual</i> )
Trace Vector	Reports on the execution of designated lines of operations (see <code>□TRACE</code> in the <i>VAX APL Reference Manual</i> )
Stop Vector	Halts operation execution before designated lines are executed (see <code>□STOP</code> in the <i>VAX APL Reference Manual</i> )
<code>□MONITOR</code>	Reports on execution counts and CPU times of designated lines of operations (see the <i>VAX APL Reference Manual</i> )
<code>□WATCH</code>	Reports references and modifications to variables (see the <i>VAX APL Reference Manual</i> )
<code>)STEP</code>	Executes individual lines of an operation one at a time (see the <i>VAX APL Reference Manual</i> )

##### 3.13.1 Suspending Operation Execution

When you debug an operation, you often need to be able to stop its execution and examine its environment before the normal completion point. Operation execution can be suspended before normal completion by means of any of the following:

- An error report generated by APL (see `□ERROR` in the *VAX APL Reference Manual*)
- An attention signal generated by the user
- The `□STOP` system function (see the *VAX APL Reference Manual*)
- The `□BREAK` system function (see the *VAX APL Reference Manual*)

When operation execution is suspended, APL displays the name of the suspended operation and the line (and statement) that was being executed. APL then begins a new line, indents six spaces, and awaits input in immediate mode.

While an operation is suspended, its local variables remain active. (Note that the arguments and operands to a user-defined operation are also local variables.) You can examine those variables, or you can specify new values for them by using an immediate-mode assignment.

## User-Defined VAX APL Operations

### 3.13 Debugging Operations

To determine the contents of an operand whose value is a function, use the `□VR` system function. `□VR` returns a character representation of the value of its argument (for more information, see the *VAX APL Reference Manual*). If you attempt to display an operand without using `□VR`, APL executes it. If the operand is a variable, APL displays the value. If the operand is a non-niladic function (system or user-defined), APL signals *SYNTAX ERROR* because it expects one or more arguments to accompany the function on the command line. For example:

```
    v Z ← (LO UOP) B   aMONADIC OPERATOR
[1]   Z← LO B           aUOP APPLIES LO TO B
[2]   v
                  a0 LOG 0 IS UNDEFINED
      * / UOP 0 0
      15 DOMAIN ERROR
      UOP[1] Z← LO B           aUOP APPLIES LO TO B
          ^
          B                   aDISPLAY B
      0 0
                  aATTEMPT DISPLAY OF OPERAND
      LO
      7 SYNTAX ERROR (NON-NILADIC FUNCTION HAS NO ARGUMENTS)
      LO
          ^
          □VR 'LO'
      * /
```

You can use any editor to display an unlocked pendent, suspended or monitored operation, but you cannot use an editor to modify an operation while it is pendent. However, you can use the `v` editor to modify a suspended or monitored operation, with the following restrictions:

- You cannot do any editing that will cause any line number to change.
- You cannot edit the operation header.
- You cannot add, delete, or change any labels.

A suspended operation remains active until you terminate or clear it from the state indicator. You can resume execution at any time by entering:

`→line-number`

where *line-number* identifies the line at which execution is to be continued. Note that the `□LC` system function contains the line number of the line where execution was suspended. Therefore, `→□LC` restarts the suspended operation at the beginning of the line that was interrupted, `→□LC+1` restarts the operation at the line following the line that was interrupted, and so on.

## User-Defined VAX APL Operations

### 3.13 Debugging Operations

To exit from a suspended operation (and to remove it from the state indicator), enter a branch to line 0, as follows:

```
→0
```

To terminate a suspended operation and all preceding pendent operations, enter a bare branch (the branch function without a line number):

```
→
```

You can also use `)SIC` or `RESET` to clear the state indicator. However, these commands clear all suspended and pendent operations while bare branch clears only the most recent suspended operation and its accompanying pendent operations.

#### 3.13.2 Examining the State Indicator

The state indicator is a vector in your active workspace that includes information about the status of all the active operations in the workspace. You can examine the state indicator by specifying the `)SI` system command (see the *VAX APL Reference Manual* for more information).

You can use the `)SINL` or `)SIS` system commands (see the *VAX APL Reference Manual*) to obtain a more extensive display of the state indicator. In addition to returning the information displayed by `)SI`, `)SINL` returns a list of local and dummy variables for each operation, as well as the current line being executed by the execute function. `)SIS` displays the line that is currently being executed and the argument expression of any pendent execute functions. For example:

```
    ∇ Z ← A F B;□IO;Q
[1] L:                      □LABELLED LINE
[2] □BREAK 'LINE 2 OF F'
[3] ∇
    ∇ R
[1] S
[2] ∇
    ∇ S;F          □NOTE F IS A LOCAL VARIABLE IN S
[1]           A
[2]           A
[3] F+T :5
[4] ∇
    ∇ F+T X        □NOTE F IS A LOCAL PARAMETER IN T
[1]           A
[2] F+÷5-ρX      □LINE 2 WILL CAUSE ERROR
[3] ∇
```

## User-Defined VAX APL Operations

### 3.13 Debugging Operations

```
      aEXECUTE F (WHICH WILL SUSPEND) AND
      a THEN R (WHICH EXECUTES S WHICH EXECUTES)
      a T WHICH SUSPENDS)

1 F 2
LINE 2 OF F
R
15 DOMAIN ERROR (DIVISION BY ZERO)
T[2] F←÷5-ρX      aLINE 2 WILL CAUSE ERROR
      ^
      )SI
T[2] *
S[3]
R[1]
F[2] *
      )SINL
T[2] * F X
S[3]   F
R[1]
F[2] * Z Q □IO B A L
      )SIS
T[2] * F←÷5-ρX      aLINE 2 WILL CAUSE ERROR
S[3]   F←T 15
R[1]   S
F[2] * □BREAK 'LINE 2 OF F'
```

The state indicator listing displays global and local operations in the order in which they were most recently active, and, for each operation, the number of the last line executed. Thus, in the preceding example, function *T* was suspended during the execution of line [2], which was called in line [3] of function *S*, which was called in line [1] of function *R*. In addition, prior to this sequence of calls, function *F* was suspended during execution of line [2].

Suspended operations in the *)SI* display are marked by a star (\*) following the name and line number. The other operations in the list are pendent; that is, they are awaiting return from another operation.

In the above example, note that the current value of the symbol *F* is a value that is local to the function *T*; the variable *F* in function *S* and the global function named *F* are currently inaccessible:

The *)SI* listing also indicates pending quad-input requests, as well as execute operations (*□XQ* or *‡*) that have been invoked. Continuing the preceding example:

## User-Defined VAX APL Operations

### 3.13 Debugging Operations

```
SI+[]  
[]:  
  []XQ ')SI'  
[]XQ  
[]  
T[2] *  
S[3]  
R[1]  
F[2] *
```

You can clear the state indicator by ending the execution of the suspended operations in the list. There are several ways to accomplish this:

- Use the `)SIC` system command.
- Type a bare branch (`→`) for each operation marked by a star.
- Use the `RESET` system function (see the *VAX APL Reference Manual*) to clear the state indicator completely.
- Save the active workspace, then clear and copy it again.

When the state indicator is clear, there is no output when `)SI` is executed.

#### 3.13.3 The Trace Vector

When debugging, you may find it helpful to obtain a printout of intermediate results of operation execution. The trace vector allows you to print the values computed by one or more operation statements each time those statements execute.

To set the trace vector, use the `TRACE` system function (see the *VAX APL Reference Manual*). You can set the trace vector either in immediate mode or within an operation. Each time a statement on one of the line numbers you specify executes, the following information is displayed in the order shown:

- The operation name.
- A bracketed statement line number, for the first or only statement on the line; for subsequent statements, the line number, a `◇`, and the statement number, all in brackets.
- The final value returned by the statement.

You may set the trace vector within an operation to aid in selective tracing. You may, for instance, want to initiate tracing if certain conditions are in effect and disable it when a specified value exceeds a defined maximum.

## User-Defined VAX APL Operations

### 3.13 Debugging Operations

In the following example, lines [1], [2], and [3] of function *F* are traced:

```
v F
[1] 12 ⋄ 44
[2] →3
[3] 26 ⋄ 55
[4] v
      1 2 3 ⌊TRACE 'F'
1
      F
F[1] 12
F[1\d2] 44
F[2] →3
F[3] 26
F[3\d2] 55
```

If the statement being traced is a branch statement, the value printed is the line number to which control is passed by the branch.

To trace all the statements of an operation, enter the following:

```
(\N)⌊TRACE 'F'
```

where *N* is a number at least as large as the number of statements in *F*.

You can also trace the end of an operation—that is, after the last statement executes but before the operation exits—by specifying line number [0], as follows:

```
0 ⌊TRACE 'F'
```

To disable the trace vector, enter the following:

```
(\0)⌊TRACE 'F'
```

To examine an operation's trace vector, use ⌊TRACE in its monadic form; APL returns an integer vector representing the line numbers being traced. For example:

```
⌊TRACE 'F'
4 6 7
```

The trace vector setting is saved with the workspace. Note that the trace vector setting is associated with lines of APL code, not with specific line numbers; the lines to be traced are not affected by the *v* editor. Operation lines set to be traced before the editor is invoked are still set to be traced after the editing session is completed (provided the line has not been modified), even though some of the lines to be traced may have new line numbers. Note that each of the following system commands or system functions clears the trace

## User-Defined VAX APL Operations

### 3.13 Debugging Operations

vector: `FX`, `MAP`, and `)EDIT`. Locking or replacing an operation for which a trace vector is defined or clears the trace vector.

#### 3.13.4 The Stop Vector

The stop vector allows you to suspend execution of an operation at predetermined points. While the operation is suspended, you can examine its environment. To set the stop vector, use the `STOP` system function (see the *VAX APL Reference Manual*).

The syntax of the stop vector is similar to that of the trace vector. For example, the following stops execution of the function *F* before lines [4], [6], and [7]:

```
4 6 7 ⎕STOP 'F'  
1
```

You can set the stop vector either in immediate mode or within an operation. When you execute the operation, the stop vector suspends execution at the first line number you specify, and then displays the operation name and the line number. You can resume execution by typing a branch to the desired line number or to `LC`. The stop vector then suspends execution at the next line you have specified.

Each time operation execution is suspended because of a stop bit, APL signals `STOPSET`. Be careful if you want to set stop bits in operations that use `TRAP` (see the *VAX APL Reference Manual*); the `STOPSET` error will initiate execution of the `TRAP` expression.

To examine an operation's stop vector, use `STOP` in its monadic form; APL returns an integer vector representing the line numbers at which execution is to be stopped. For example:

```
⎕STOP 'F'  
4 6 7
```

The stop vector setting is saved with the workspace. Note that the stop vector setting is associated with lines of APL code, not with specific line numbers; thus, the lines to be stopped are not affected by the `v` editor. Operation lines set to be stopped before the editor is invoked are still set to be stopped after the editing session is completed (provided the line has not been modified), even though some of the lines to be stopped may have new line numbers. Note that each of the following commands or functions clears the stop vector: `FX`, `MAP`, and `)EDIT`. Locking or replacing an operation for which a stop vector is defined clears the stop vector.

## 3.14 Examples of Error Trapping

Error trapping allows you to handle errors in user-defined operations in the same way that APL handles errors in primitive functions, that is, by informing the caller that the function failed and explaining why.

Normally, if APL detects an error while executing an operation, it suspends execution and prints the error information on the terminal. Error trapping enables you to gain control when an error occurs so that you can prepare alternatives to the default error processing.

The following system variables and system functions help you create your own error-handling routines. They allow you to determine where errors occur and why, and to halt operation execution, check the logic flow, and then resume execution.

- The `□TRAP` system variable—You use `□TRAP` to gain control when an error occurs. It contains an APL expression to be executed if an error occurs during execution of a user-defined operation. The expression to be executed often is a branch to an error-handling routine.
- The `□SIGNAL` system function – Indicates that an error occurred. You can use it in your error-handling routines to display a standard APL error message or a user-defined error message.
- The `□ERROR` system variable – Contains the text of the error message for the last error that occurred.
- The `□BREAK` system function – Suspends operation execution, prints the value of its argument, and returns control to immediate mode. Often used as the last statement in an error-trapping routine, to be executed if the error is not one of the errors that the routine checks for. Note that `□TRAP` cannot trap an operation suspended with `□BREAK`.

For more details on these system functions and system variables, see the individual descriptions in the *VAX APL Reference Manual*. The following sections give three examples of error-trapping techniques.

### 3.14.1 System Variable Change

In the first example, a trap is set to ensure that the index origin is set to 1. If an error occurs during operation execution, APL executes the expression associated with `□TRAP`. In this case, the expression transfers control to label `IO`, where APL checks whether `□IO` is equal to 0. If `□IO` equals 0, APL proceeds to label `SET` and resets `□IO` to 1. Control is then transferred to label `DIV`, where APL executes the expression again, this time with `□IO` set to 1.

## User-Defined VAX APL Operations

### 3.14 Examples of Error Trapping

```
v Z←A DIVIDE B; ⎕TRAP
[1]      ⎕THIS PROGRAM TAKES A NUMER, A,
[2]      A AND DIVIDES IT BY ⎵B.
[3]      ⎕EXAMPLE: 2 DIVIDE 3 WILL RETURN A
[4]      A 3-ITEM VECTOR OF (2÷1), (2÷2), 2÷3
[5]      ⎕TRAP←'IO'
[6]      DIV:Z←A÷⍵B
[7]      →0
[8]      IO:→(0=⎕IO)/SET
[9]      ⎕BREAK 'DIVIDE ERROR'
[10]     SET:⎕IO←1
[11]     + DIV
[12] 
v
⎕IO←0
25 DIVIDE 5
25 12.5 8.333333333 6.25 5
⎕ERROR
15 DOMAIN ERROR (DIVISION BY ZERO)
DIVIDE[6] DIV:Z←A÷⍵B
^
⎕IO
1
```

Notice in this example that APL executed the function even though ⎕IO was set to 0. ⎕ERROR contains the error that occurred, but the trap handled the error condition. Note that ⎕IO is now set to 1.

The value of ⎕IO is all that is checked by the error-trapping routine. If another error occurs, the ⎕BREAK system function is executed (line 8), thus suspending execution and returning control to immediate mode. For example:

```
'A' DIVIDE 5
DIVIDE ERROR
⎕ERROR
15 DOMAIN ERROR (INCORRECT TYPE)
DIVIDE[6] DIV:Z←A÷⍵B
^
```

⎕BREAK prints the argument you supplied and suspends execution.

#### 3.14.2 User-Defined Error Messages

The next example involves three functions: *MASTER*, *ERROR* , and *SNIGGLE*. *MASTER* takes a scalar or a vector argument, adds 50 to each item, and passes the vector or scalar to *SNIGGLE*. *SNIGGLE* tries to turn the argument into a square matrix. If the argument cannot be squared (its shape does not have an integer square root), *SNIGGLE* signals the user-defined error 550 NOT SQUARE and exits back to *MASTER*. *MASTER* has a trap set to send errors to the function *ERROR* . Here are the functions:

## User-Defined VAX APL Operations

### 3.14 Examples of Error Trapping

```
V Z←MASTER VECTOR;[]TRAP
[1] []TRAP←'ERROR ⌊ →[]LC'
[2]     ⌈MASTER ADDS 50 TO EACH ITEM IN VECTOR.
[3]     ⌈SNIGGLE MAKES THE VECTOR SQUARE.
[4] SNIGGLE VECTOR+50
[5]     ⌈IF IT REFUSES TO BE SQUARED, SEND IT TO ERROR.
[6] Z←MATRIX
[7] V

V SNIGGLE VEC;[]TRAP
[1]     ⌈URNS A VECTOR INTO A SQUARE MATRIX;
[2]     ⌈IF THE VECTOR WON'T GO, SIGNAL THE CALLER.
[3] []TRAP←'→SNIG'
[4] MATRIX←(((ρVEC)×0.5),(ρVEC)×0.5)ρVEC
[5] →0
[6] SNIG: 'NOT SQUARE' ⌋SIGNAL 550
[7] V

V ERROR;A
[1] →('1550'≠ 3↑[]ERROR)/BREAK
[2] A←⌈(ρVECTOR)×0.5
[3]     ⌈MAKE VECTOR VALID, THEN RETURN TO MASTER TO
[4]     ⌈ TRY AGAIN. EXECUTION RESTARTED BY SECOND
[5]     ⌈ STATEMENT IN MASTER'S TRAP ARGUMENT ([]LC).
[6] VECTOR←VECTOR,((A×2)-ρVECTOR)ρ0 ⌊ →0
[7] BREAK: ⌋BREAK 'INVALID ARGUMENT'
[8] V
```

A sample execution of *MASTER* is as follows:

```
MASTER 18
51 52 53
54 55 56
57 58 50
    []ERROR
550 NOT SQUARE
MASTER[4] SNIGGLE VECTOR+50
^
)SI
(There is no output)
```

This is what happened:

- *MASTER* is called with a vector that is not square, 18.
- *MASTER* sets up the localized *[]TRAP*.
- *MASTER* calls *SNIGGLE* with the argument 50+18.
- *SNIGGLE* sets up its localized *[]TRAP*.
- At *SNIGGLE*[4], execution of the expression at line [4] results in a *DOMAIN ERROR*, thereby invoking the error trap.

## User-Defined VAX APL Operations

### 3.14 Examples of Error Trapping

- The expression associated with  $\square TRAP$  is ' $\rightarrow SNIG$ ' , so  $\square TRAP$  redirects execution to line [6].
- At this point,  $\square ERROR$  contains the *DOMAIN ERROR*. At line [6], *SNIGGLE* uses  $\square SIGNAL$  to set  $\square ERROR$  to the user-defined error message.
- $\square SIGNAL$  terminates execution of *SNIGGLE* and forces an error at *MASTER[4]*, where *SNIGGLE* was called.
- At *MASTER[4]*, error trapping is invoked. (Note that *MASTER* is on top of the  $\square SI$  stack now.)
- $\square TRAP$  is equal to ' $ERROR \diamond \rightarrow \square LC$ ' , so  $\square TRAP$  executes the function *ERROR*. (Note that  $\square LC$  currently is line [4], where the error occurred.)
- The function *ERROR* checks whether  $\square ERROR$  contains error number 550. If it does not, *ERROR* breaks to the terminal for assistance.
- The function *ERROR* corrects error 550 by extending the argument *VECTOR* with enough zeros to make it square.
- *ERROR* returns control to *MASTER*, where the second statement in the argument to the  $\square TRAP$  transfers control to  $\square LC$ , which equals 4.
- Execution of *MASTER* resumes, this time with a valid argument. Because the error was corrected and the function was restarted, the state indicator is clear.

The following illustrates what happens when *MASTER* is executed with an invalid argument that *ERROR* cannot fix:

```
MASTER 'ABC'
INVALID ARGUMENT
  ERROR
  15 DOMAIN ERROR (INCORRECT TYPE)
MASTER[4] SNIGGLE VECTOR+50
  ^
    )SI
ERROR[7] *
^
MASTER[4] *
  -
    )SI
MASTER[4] *
```

In this example:

- *MASTER* tries to invoke *SNIGGLE* with '*ABC*' + 50. A *DOMAIN ERROR* results.
- Execution of the trap expression invokes the function *ERROR*.

- Because  $\square ERROR$  does not contain error number 550, the function *ERROR* transfers control to the label *BREAK*.
- The  $\square BREAK$  system function displays its argument and suspends the *ERROR*.
- The  $)SI$  display indicates that *ERROR* is suspended at line [7], *MASTER* is suspended at line [4], and there is an  $\downarrow$  execute (of  $\square TRAP$ ) pending.
- A bare branch removes *ERROR* and the  $\downarrow$  function from the state indicator. The trap expression is never completed; APL leaves *MASTER* suspended at line [4].

### 3.14.3 Execute Trap Expression

In the next example, the function *F* calls the function *G* to perform matrix inversion. Notice what happens when a *DOMAIN ERROR* in *G* triggers execution of the trap expression ( $\rightarrow 4$ ) in the calling function, *F*:

```

    v Z←F A
[1]  □TRAP← '→4'
[2]  Z←G A
[3]  →0
[4]  'MATRIX INVERSION ERROR, MATRIX WAS: '
[5]  A
[6]  □ERROR
[7]  v

    v Z←G A
[1]  Z←◻A
[2]  v

      A+2 2p 1E20 0 0 1E-20
      X←F A
      MATRIX INVERSION ERROR, MATRIX WAS:
      1E20  0E0
      0E0  1E-20
      11 VALUE ERROR (FUNCTION RESULT UNDEFINED)
F[2]  Z←G A
      ^
      11 VALUE ERROR (FUNCTION RESULT UNDEFINED)
X←F A
      ^
      )SI

```

There is no output

This is what happened:

- *F*[1] sets  $\square TRAP$  to go to line [4], where inversion errors are reported.
- *F*[2] calls *G* with the singular matrix *A*.

## User-Defined VAX APL Operations

### 3.14 Examples of Error Trapping

- $G[1]$  gets a domain error, which invokes error trapping.
- $\Box TRAP$  is set to the global value ' $\rightarrow 4$ ', so APL tries to do  $\star \Box TRAP$  and go to line [4].
- There is no line [4] in  $G$ , so  $G$  exits back to  $F[2]$ , without setting its return value for  $Z$ .
- APL signals the following error:

```
11 VALUE ERROR (FUNCTION RESULT UNDEFINED)
F[2] Z←G A
^
```

APL then invokes error trapping, which again does  $\star \Box TRAP$  and goes to line [4].

- The message at line [4], the value of  $A$  at line [5], and the unexpected  $\Box ERROR$  at line [6] are printed.
- $F$  exits but never sets its return value  $Z$ . Therefore, APL signals the following:

```
11 VALUE ERROR (FUNCTION RESULT UNDEFINED)
X←F A
^
```

## 3.15 Programming Considerations for VAX APL

The following subsections describe methods you may want to consider as you use APL.

### 3.15.1 Speed Optimizations in VAX APL Primitives

The behavior of the primitive functions listed below has been optimized to produce a result in the least amount of time. Sometimes the optimization occurs on a subset of all possible arguments to the primitive, based on rank (for example, vector versus general array) or internal type (Boolean, integer, floating-point, or character).

- $1\ 1\ \& B$ ,  $1\ 2\ \& B$ , and  $2\ 1\ \& B$  for any matrix  $B$

These expressions with the transpose ( $\&$ ) function are used for three purposes:

- To select the diagonal items:  $1\ 1\ \& matrix$
- To derive the identity function:  $1\ 2\ \& matrix \leftrightarrow matrix$
- To build an ordinary transpose:  $2\ 1\ \& matrix \leftrightarrow \& matrix$

### 3.15 Programming Considerations for VAX APL

- $A / B$  for Boolean singleton  $A$  and any array  $B$

This expression with the slash (/) operator is often used in branching ( $\rightarrow A / B$ ). When  $A$  is 1 (true), the control of a user-defined operation moves to the label specified in  $B$ . Otherwise, the control moves to the next statement. Two common conditional forms include the following:

- *Boolean-singleton / label*
- *Boolean-singleton ρ label*

- $A \rho B$  where  $A$  is a Boolean scalar or an integer scalar equal to 0 or 1

This expression is often used with branching; see slash (/) above for more explanation.  $1 \rho B$  selects the first item of  $B$  and  $0 \rho B$  creates an empty vector of the same type as  $B$ .

- $A * 0$  and  $A * 1$  and  $A * 2$  for any array  $A$

By definition,  $A * 0$  is always 1.  $A * 1$  requires no computation since 1 is the exponential identity item:  $(A * 1 \leftrightarrow A)$ .  $A * 2$  is computed as  $A * A$  since multiplication is faster than exponentiation.

- $A , B$  where  $A$  and  $B$  are either singletons or vectors

This expression shows the optimization of two frequent operations: catenating one item onto the front or back of a vector, and catenating two vectors together.

- $f / B$  where  $f$  is  $\wedge$  or  $\vee$  function, and  $B$  is a boolean vector

The result of these expressions is always 1 or 0. When  $B$  is a Boolean vector, the APL interpreter is often able to evaluate the result before looking at all the items contained in  $B$ .

For  $\wedge / \text{Boolean-vector}$ , the result is 0 as soon as an item in the vector is 0. When all items are 1, the result is 1.

For  $\vee / \text{Boolean-vector}$ , the result is 1 as soon as an item in the vector is 1. When all items are 0, the result is 0.

- $\wedge \backslash B$  where  $f$  is  $\wedge$ ,  $\vee$ ,  $<$ , or  $\leq$  and  $B$  is a Boolean vector

These expressions are often used for producing masks.

$\wedge \backslash \text{Boolean-vector}$  turns off all bits after the first 0.

$\vee \backslash \text{Boolean-vector}$  turns on all bits after the first 1.

$< \backslash \text{Boolean-vector}$  turns off all bits after the first 1 (only the first 1 is left turned on).

$\leq \backslash \text{Boolean-vector}$  turns on all bits after the first 0 (only the first 0 is left turned off).

## User-Defined VAX APL Operations

### 3.15 Programming Considerations for VAX APL

Since these expressions often use numeric vectors as arguments, VAX APL is optimized to perform the operation quickly.

$+ / num$  is the sum of the numbers in  $num$ .

$\times / num$  is the product of the numbers in  $num$ .

$\lceil / num$  is the largest number in  $num$ .

$\lfloor / num$  is the smallest number in  $num$ .

- $A \sqsubset B$  and  $A \in B$  where  $A$  and  $B$  are of the same type and either Boolean, character, or integer

$A \sqsubset B$  finds where  $B$  is in  $A$ .

$A \in B$  determines which items of  $A$  are in  $B$ .

Since the comparison technique is exact (and not fuzzy, see  $\square CT$  in *VAX APL Reference Manual*), the APL interpreter makes quick comparisons between the items of the arguments.

- $A \wedge . = B$  and  $A \vee . \neq B$  where  $A$  is a character matrix and  $B$  is a character vector shorter than 65,536 characters

This expression is used to look up a string in a table.

$A \wedge . = B$  determines which rows of  $A$  are equal to  $B$ .

$A \vee . \neq B$  determines which rows of  $A$  are not equal to  $B$ .

- $A \circ . f B$  where  $f$  is  $<$ ,  $<$ ,  $=$ ,  $\geq$ ,  $>$ , or  $\neq$  and  $A$  and  $B$  are either character or integer arrays

These expressions are often used for producing masks. They build tables of the comparisons of all items of  $A$  versus all items of  $B$ .

- $A \circ . + B$  and  $A \circ . \times B$  where  $A$  and  $B$  are any numeric arrays

These expressions are often used for building addition and multiplication tables.

- $A \circ . * B$  for any array  $A$  and Boolean array  $B$

This expression builds a table of  $A$  versus  $B$  that contains a 1 wherever  $B$  contains a 0 and contains the item of  $A$  where  $B$  is 1. Because of the multiplicative identity in the vacant spots (flagged by the zeros in  $B$ ), the table is useful as a mask that will be multiplied times another array.

- $\rho \rho B$  and  $A / \sqsubset B$  are handled as special cases.

$\rho \rho B$  is the rank of  $B$ .

$A / \sqsubset B$  determines the indices into  $B$  of where the 1s are in  $A$ . (See  $\square OM$  in the *VAX APL Reference Manual*.)

## User-Defined VAX APL Operations

### 3.15 Programming Considerations for VAX APL

- $A f B$  where  $f$  is  $+$ ,  $-$ ,  $<$ ,  $\leq$ ,  $=$ ,  $\geq$ ,  $>$ , or  $\neq$  and  $A$  and  $B$  are either Boolean or integer scalars
- $A \uparrow B$  and  $A \downarrow B$  where  $B$  is a scalar or vector
- $A / B$  where either  $A$  is a Boolean singleton or vector and  $B$  is any array,  $A$  is an integer singleton and  $B$  is any array, or  $A$  is an integer vector and  $B$  is a singleton
- $0 \phi B \leftrightarrow B$

In  $A \phi [K] B$  for any conforming  $A$  and  $B$ , let  $L$  be  $(\rho B)[K]$  (The length of the axis being rotated). Then, if  $L$  is 1, or  $A | L$  is 0, the result is  $B$ .

#### 3.15.2 Space Considerations in VAX APL

This section describes how APL allocates and deallocates memory from VMS. It also describes techniques for reducing the amount of memory required by a workspace.

APL allocates memory from VMS dynamically as you create objects in a workspace. The memory remains allocated when you reduce the size of an object or even when you expunge an object from a workspace. You can release this unused memory by performing the following steps:

- )SAVE the workspace. The saved version does not contain the unused memory. If you clear the SI stack (with the )SIC command) before the )SAVE, you release additional memory.
- )CLEAR the workspace. A clear workspace has no allocated memory.
- )COPY the entire workspace. The copied workspace does not contain symbols that have no values (unless a symbol is referenced in a user-defined operation).
- )SAVE the smaller version of the workspace.

In addition, there are several system variables that can occupy a great deal of storage in the workspace. These are as follows:

- $\square ERROR$ —the last error message
- $\square L$ —the name of the last variable that hit a watchpoint
- $\square R$ —the previous value of a watched variable

You can make more memory available to APL by setting each of these system variables to the empty character array (' '). This value takes up the least amount of memory possible.

## User-Defined VAX APL Operations

### 3.15 Programming Considerations for VAX APL

)*NMS* reports all of the symbols in the symbol table. If the name class of a symbol is 0, it has no previous value, even though the symbol table entry uses memory. Symbols that have a name class of 0, which are not referenced by a user-defined operation, can be released from the workspace by following the steps listed in the preceding description.

#### 3.15.3 Efficient Uses of VMS Subprocesses

There are four APL system commands that spawn VMS subprocesses: )*DO*, )*DROP*, )*LIB*, and )*PUSH*.

Each time you use )*DO*, )*DROP*, /*CONFIRM*, or )*PUSH*, APL creates a subprocess, uses the subprocess, and then stops it.

Each time you use )*LIB* or )*DROP* (without specifying the /*CONFIRM* qualifier), APL creates a subprocess that remains active for the remainder of your APL session. If you continue to use either of these commands, you do not incur the additional overhead required to spawn a new subprocess.

---

## The Report Formatter

The  $\square FMT$  system function allows you to combine and reformat character and numeric data, and to output the data as a character matrix. Typically, the character matrix represents a report—a summary of data generated when VAX APL was used to solve a problem.  $\square FMT$  edits the data as it is moved to an output field. For example,  $\square FMT$  fills or erases zeros in numeric fields; rounds numeric data; and inserts commas, dollar signs, and other text where appropriate.

The  $\square FMT$  system function has the general form:

*format-phrases*  $\square FMT$  {array | (array ; array ; . . . )}

The right argument is either one array or a list of arrays of any type or rank. If the argument is a list, it must be surrounded by parentheses and the arrays must be separated by semicolons.

The right argument may also be an enclosed array of depth 2 which is in the vector domain and its items are simple homogeneous arrays. Each item of the vector is treated in the same manner as an element of a list type argument.

The left argument is a character vector composed of one or more format phrases of the form described in Table 4-1, Section 4.1.3. The phrases must be separated by commas.

$\square FMT$  combines the data from all of the arrays in the right argument and arranges it as a single matrix whose columns are then formatted according to corresponding format phrases specified in the left argument.

## The Report Formatter

For example, `□FMT` could combine two arrays and format the result as follows:

```
□←FIRST←2 3p 1 6
1 2 3
4 5 6
□←SECOND←3 2p 0 1 6
6 5
4 3
2 1
'I2,I3,I5,I7,I8' □FMT (FIRST;SECOND)
1 2      3      6      5
4 5      6      4      3
                  2      1
```

Because the phrases in the left argument control how the columns of data are output, and because there are five target columns in this example, the left argument to the `□FMT` function contains five format phrases:

```
'I2,I3,I5,I7,I8' □FMT(FIRST;SECOND)
```

The format phrase I2 governs how the first target column from the reformatted right argument is output, phrase I3 affects the second target column, and so on (see Section 4.1 for an explanation of format phrases).

---

### Note

---

Semicolons in the right argument do not function as output catenators; they merely separate the arrays that make up the right argument.

When it contains more than one array, `□FMT`'s right argument must be surrounded by parentheses, to distinguish its semicolon list from lists containing the output catenator.

---

## 4.1 Format Phrases

The left argument to `□FMT` is a character string that consists of one or more format phrases that govern how corresponding target columns in the right argument are output. The format phrases must be separated by commas.

Format phrases have the form:

```
[rep][quals]type[width [.dig]]]
```

A number for *rep* (repetition) specifies that the format phrase is to apply to that number of consecutive target columns in the right argument. Thus, using the repetition parameter, the format phrase

'I3,I3,I3,I3,F9.2'  $\square FMT$  (NUMS;TOTALS)

can be represented as:

'5I3,F9.2'  $\square FMT$  (NUMS;TOTALS)

The *quals* (qualifiers) parameter refers to one or more of the format phrase modifiers called qualifiers and decorators (see Section 4.1.5). For example, the qualifier L means that the data fields in the target column should be left-justified in the  $\square FMT$  output:

```
TOTALS+479.59 29.99 12799.50 1444.09 325.88
'LF9.2,F9.2'  $\square FMT$  (TOTALS;TOTALS)
479.59        479.59
29.99        29.99
12799.50    12799.50
1444.09      1444.09
325.88      325.88
```

The *type* of a format phrase refers to the type of the data to be formatted. The possible types are identified in Table 4-1 (see Section 4.1.3). Some of the more commonly used types are I, for integer data; F, for fixed-point data; E, for floating-point data; and A, for character data.

The *width* is the width in the  $\square FMT$  output of the values in the corresponding target column. For example, the format phrase I5 means that each of the integers in the target column is to be five characters wide in the  $\square FMT$  output.

The *dig* (digits) parameter refers to the number of decimal places (fixed-point data) or significant digits (floating-point data) to be included in the  $\square FMT$  output. For example, the format phrase F8.2 means that the fixed-point data should have two decimal places in the  $\square FMT$  output.

#### 4.1.1 Too Few or Too Many Format Phrases

If there are more format phrases in the left argument than target columns in the right argument, the extra format phrases are ignored. If there are fewer format phrases than target columns, the format phrases are reused starting with the leftmost phrase. For example:

## The Report Formatter

### 4.1 Format Phrases

```
SUBTOT←24.29 1.27 305.25 297.98 44.59
ONHAND←112 7 43 586 1289
TOTALS←279.59 29.19 2799.50 1444.09 325.88
'F8.2,I5' ⎕FMT (SUBTOT;ONHAND;TOTALS)
24.29 112 279.59
 1.27    7   29.19
305.25   43 2799.50
297.98   586 1444.09
 44.59 1289 325.88
```

Here, the format phrase *F8.2* affected the first target column (the values in *SUBTOT*), the phrase *I5* governed the output of the second target column (the values in *ONHAND*), and then the first format phrase, *F8.2*, was reused to affect the output of the third target column (the values in *TOTALS*).

#### 4.1.2 Treatment of Empty Arguments

If ⎕FMT's left argument is an empty character array, APL signals a *LENGTH ERROR*.

In general, empty items in the right argument's semicolon list are ignored. If the right argument is an empty array, ⎕FMT's result is an empty character matrix. For example, the argument (5;;7) is equivalent to (5;7). However, if the number of columns in an empty array is not zero, the appropriate format phrase is applied, as in the following example.

```
NOCOLS←1 0⍪1
NOROWS←0 1⍪ 1
'I5' ⎕FMT (5;NOCOLS;7)
5      7
'I5' ⎕FMT (5;NOROWS;7)
5      7
⍝ 'I5' ⎕FMT ( )
0 0
⍝ 'I5' ⎕FMT (;)
0 0
⍝ 'I6' ⎕FMT 0 4⍪0
0 24
⍝ 'I6' ⎕FMT 4 0⍪0
4 0
⍝ 'I5' ⎕FMT (2 0⍪0;3 0⍪0)
3 0
```

### 4.1.3 Format Phrase Types

The format phrase types identified in Table 4–1 can be organized into two categories: those that control the output of a target column from the right argument, and those that are not associated with a target column.

The format phrase types A, E, I, F, G, and Y are in the first category. They determine how the values in a target column are output. The format phrase types T, X, and *literal* are in the second category. Types T and X affect the output positioning but not the content of data fields from the right argument, and type *literal* outputs literal text.

**Table 4–1 Summary of Format Phrase Syntax**

Phrase	Type of Data
<code>[[rep]] [[quals]] A width</code>	Character
<code>[[rep]] [[quals]] E width.dig</code>	Floating-point with exponent
<code>[[rep]] [[quals]] F width.dig</code>	Fixed-point
<code>[[rep]] [[quals]] G <math>\alpha</math> pattern<math>\alpha</math></code>	Picture
<code>[[rep]] [[quals]] I width</code>	Integer
<code>[[rep]] [[quals]] Y width</code>	Byte
<code>[[rep]] T [[col]]</code>	Absolute tab
<code>[[rep]] X [[col]]</code>	Relative tab
<code>[[rep]] <math>\alpha</math> text<math>\alpha</math></code>	Literal

#### 4.1.3.1 Type A—Character

Data fields that are in target columns formatted by a type A format phrase are placed in the output matrix right-justified in a field with a width specified by the *width* parameter. You may use the L qualifier (see Section 4.1.5.4) to left-justify the data in the output field. The R qualifier (see Section 4.1.5.11) is the only other qualifier that may be used in type A format phrases.

Type A phrases format character data only. If you try to use a type A phrase to format numeric data, APL signals *DOMAIN ERROR*.

Note that type A format phrases affect target columns that are exactly one character wide. You can use the repetition parameter (*rep*) to apply the same format phrase specification to a string of characters:

## The Report Formatter

### 4.1 Format Phrases

```
NAME←3 50 'SMITHJONESBROWN'  
AMT←1999 2345 4675  
'5A1,I6' ⓁFMT (NAME;AMT)  
SMITH 1999  
JONES 2345  
BROWN 4675
```

For more information about formatting character data, see Section 4.4.

#### 4.1.3.2 Type E—Floating-Point with Exponent

Type E format phrases output numeric data values in exponential form. The values are rounded to the number of significant digits specified by the digits parameter (*dig*), and are then written in exponential form to an output field that has a width specified by the *width* parameter.

For example, the format phrase *E8.4* means that the values in the target columns are to be converted to exponential form and written, with four significant digits, to an 8-character output field.

Exponential data values consist of two parts: the fraction and the exponent. For example, in the number  $1.254E3$ , the fraction is  $1.254$  and the exponent is  $E3$ . ⓁFMT displays the fraction in the range greater than or equal to 1 and less than 10, or 0.

ⓁFMT uses 4 as the default exponent width: 1 character for the E, 1 character for the possible negative symbol, and 2 characters for the exponent digits. You can use the *Wn* qualifier (see Section 4.1.5.6) to reduce the default number of exponent digits to 1 or to increase it to 3. Within the exponent part, the exponent digits are left-justified without leading zeros or a sign (if none is needed).

Note that the width specified (by the *width* parameter) for the output field should be large enough to accommodate the following:

- A leading negative symbol on the fraction of the number (when necessary).
- A decimal point (when necessary).
- The number of significant digits specified by the digits parameter.
- The default width of the exponent of the number (4 unless you use the *Wn* qualifier to change the number of exponent digits).

If the width specified is greater than the width of the formatted value, the value is right-justified in the output field, and leading spaces are inserted to fill the field. Note, however, that the value may not seem to be right-justified because any unused positions in the exponent are output as spaces:

## The Report Formatter

### 4.1 Format Phrases

```

 $\text{OUT} \leftarrow 'E10.4' \text{ FMT } 1234 .00000000001234$ 
1.234E3
1.234E-11
p OUT
2 10

```

Two spaces were included following the 3 in the first output value because the default field width for the exponent of the number was 4.

If the width specified is too small, the formatted value is output as a field of stars:

```

'E8.4,E9.4' FMT 1 2 p1254
*****1.254E3

```

Here, a minimum of 9 character positions was needed to write  $1254$  in exponential form: 1 for the decimal point, 4 for the significant digits, and the default 4 characters for the exponent. If the right argument had been  $-1254$ , 10 character positions would have been needed for the formatted output.

The qualifiers B, K, L, O, R, S, and W are permitted with type E format phrases. Table 4-2 lists the type E format phrases.

**Table 4-2 E Format Phrases**

Format Phrase	Value in Target Column	Formatted Result <sup>1</sup>
E10.4	24.414	$\Delta 2.441E1\Delta\Delta$
E10.4	-24.415	$\bar{-}2.442E1\Delta\Delta$
E10.5	-24.415	*****
W1E10.5	-24.415	$\bar{-}2.4415E1\Delta$
E10.2	24.414	$\Delta\Delta\Delta 2.4E1\Delta\Delta$
LE10.2	24.414	$2.4E1\Delta\Delta\Delta\Delta\Delta$
E10.3	.005555	$\Delta\Delta 5.56E-3\Delta$
E8.2	0	$\Delta 0.0E0\Delta\Delta$
BE8.2	0	$\Delta\Delta\Delta\Delta\Delta\Delta\Delta$
LE12.5	77	$7.7000E1\Delta\Delta\Delta\Delta$
K3E12.5	-77	$\Delta\bar{-}7.7000E4\Delta\Delta$
OZEROE8.2	2.5	$\Delta 2.5E0\Delta\Delta$

<sup>1</sup>Note that the delta( $\Delta$ ) represents the space character

(continued on next page)

## The Report Formatter

### 4.1 Format Phrases

**Table 4–2 (Cont.) E Format Phrases**

Format Phrase	Value in Target Column	Formatted Result <sup>1</sup>
<i>O</i> <i>T</i> <i>ZERO</i> <i>E</i> 8 . 2	0	Δ Δ Δ Δ <i>ZERO</i>
<i>E</i> 11 . 5	- 1	- 1 . 0000 <i>E</i> 0 Δ Δ

<sup>1</sup>Note that the delta(Δ) represents the space character

#### 4.1.3.3 Type F—Fixed-Point

Type F format phrases output numeric data values in fixed-point form. The data values are rounded to the number of decimal places specified by the digits parameter (*dig*) and placed in an output field that has a width specified by the *width* parameter.

For example, the format phrase *F*8 . 4 means that the values in the target columns are to be written in fixed-point form to an 8-character output field, and rounded to four digits to the right of the decimal point.

If the value to be formatted has fewer than the number of decimal places specified by the digits parameter, trailing zeros are added:

```
'F8.4'  OFMT 123.4 55  
123.4000  
55.0000
```

Note that the width specified (by the *width* parameter) for the output field should be large enough to accommodate:

- A leading negative symbol (when necessary).
- At least one digit to the left of the decimal point (the digit 0 is used if the magnitude is less than 1 or equal to 0). (A leading negative symbol (−) is used in place of the digit 0 when the output is a negative number.)
- A decimal point (when necessary).
- The number of digits (specified by the digits parameter) to the right of the decimal point.

If the width specified is greater than the width of the formatted value, the value is right-justified in the output field, and leading spaces are inserted to fill the field.

## The Report Formatter

### 4.1 Format Phrases

If the width specified is too small, the formatted field is displayed as a field of stars:

```
'F8.4'  DFMT -123.4 55  
*****  
55.0000
```

**Table 4–3 F Format Phrases**

Format Phrase	Value in Target Column	Formatted Result <sup>1</sup>
F9.4	1254	1254.0000
F10.3	.0056	Δ Δ Δ Δ 0.006
F8.4	-24.415	-24.4150
F8.5	-24.415	*****
F10.2	24.414	Δ Δ Δ Δ 24.41
LF10.2	24.415	24.42Δ Δ Δ Δ
F10.3	.005555	Δ Δ Δ Δ 0.006
F10.6	.005555	Δ Δ 0.005555
F8.2	0	Δ Δ Δ 0.00
BF8.2	0	Δ Δ Δ Δ Δ Δ Δ
LF15.5	77	77.00000Δ Δ Δ Δ Δ
K3F15.5	-77	Δ Δ Δ -77000.00000
0ZeroF8.2	2.5	Δ Δ Δ 2.50
0ZeroF8.2	0	Δ Δ Δ Δ Zero
F8.6	-0.1	- .1000000

<sup>1</sup>Note that the delta(Δ) represents the space character

All of the qualifiers and decorators listed in Table 4–8, Section 4.1.5, are permitted with type F format phrases.

#### 4.1.3.4 Type G—Pattern Data

The data fields in target columns formatted by a type G format phrase are rounded to integers and then placed in the output field as specified by the given pattern. Each character position in the pattern corresponds to a character position in the output field.

## The Report Formatter

### 4.1 Format Phrases

The following codes may be included in the pattern:

Pattern code	Meaning
9	Put a digit in this position.
Z or z	Put a digit in this position unless the digit is a leading or trailing zero; in that case, put a blank in this position.
any character	Put the specified literal character in this position (spaces are permitted).
@	Put the @ character in this position; however, you may replace the @ character using standard symbol substitution (see Section 4.1.5.5). This is intended primarily to allow TTY users to include decimal points in the pattern.

The pattern ZZZZ99, for example, means that a digit from the target column should be placed in each of the six positions in the output field, except that any leading zeros in the leftmost four positions of the output field should be replaced by blanks.

There should be a Z or a 9 in the pattern for each digit in the target column; if there are more digits than Zs and 9s, the output field is filled with stars (\*). APL signals *DOMAIN ERROR* if the pattern does not include at least one 9 or one Z.

Note that type G format makes it easy to mix numeric and character data:

```
'G<999-99-9999>' ⎕FMT 144590701  
144-59-0701  
'G<Z9/99/99>' ⎕FMT 41784  
4/17/84
```

A Z pattern code blanks out only leading or trailing zeros, not embedded zeros; thus, the pattern ZZ99ZZZ99ZZ has the same effect as the pattern ZZ9999999ZZ.

Literal characters (including trailing blanks) that occur in the pattern to the right of Z pattern codes are inserted only if there are digits output to the left of the literal. For example:

```
'G<$Z,ZZZM>' ⎕FMT 1234 455 44 1 0  
$1,234M  
$ 455M  
$ 44M  
$ 1M  
$
```

## The Report Formatter

### 4.1 Format Phrases

You can use standard symbol substitution to change a pattern code to another character. For example, the following substitutes lowercase f for pattern code Z, thus allowing Z to be inserted as a literal character:

```
'S <Zf>G<ZZZfF9>' D FMT 78 4  
ZZZ7F8  
ZZZ 4
```

Note that although lowercase f assumed the function of pattern code Z, uppercase F did not and was placed in the output field as a literal.

TTY users who need to include decimal points in their patterns can do so by using the @ character in the pattern and then using standard symbol substitution to replace the @ with a decimal point:

```
'G<ZZ.ZZ>' .BXFMT 12      ".ZZ IS UNDERSCORED Z  
12.ZZ  
'S<@.>G<ZZ@ZZ>' .BXFMT 123  
1.23
```

The qualifiers B, K, M, N, P, Q, R, and S are permitted with type G format phrases listed in Table 4-4.

**Table 4-4 G Format Phrases**

Format Phrase	Value in Target Column	Formatted Result <sup>1</sup>
<Z9ZZ9Z>	10101	Δ 10101
G<Z9ZZ9Z>	101	Δ 00101
G<Z9ZZ9Z>	9000	Δ 0900Δ
G<Z9ZZ9Z>	543210	54321Δ
G<Z9ZZ9Z>	6543210	*****
G<Z,ZZ9.99>	47799	Δ Δ 477.99
G<Z,ZZ9.99>	38	Δ Δ Δ 0.38
G<Z,ZZ9.99>	1234567	*****
G<Z,ZZ9.99DOLLARS>	47799	Δ Δ 477.99Δ DOLLARS
G<TOTAL IS ZZZZ9>	1227	TOTALΔ ISΔ Δ 1227
G<TOTAL IS ZZZ UNITS>	428	TOTALΔ ISΔ 428Δ UNITS

<sup>1</sup>Note that the delta(Δ) represents the space character

(continued on next page)

## The Report Formatter

### 4.1 Format Phrases

**Table 4–4 (Cont.) G Format Phrases**

Format Phrase	Value in Target Column	Formatted Result <sup>1</sup>
<i>G&lt;TOTAL IS ZZZ UNITS&gt;</i>	0	<i>TOTAL ISΔΔΔΔΔΔΔΔΔΔ</i>
<i>G&lt;TOTAL IS ZZZ UNITS&gt;</i>	1234	*****
<i>G&lt;ZZ.99&gt;</i>	25.3	ΔΔΔ25
<i>K2G&lt;ZZ.99&gt;</i>	25.3	25.30

<sup>1</sup>Note that the delta(Δ) represents the space character

#### 4.1.3.5 Type I—Integer

Type I format phrases output numeric data values in integer form. The data fields in the target columns are rounded to integers and then placed in output fields that have a width specified by the *width* parameter.

For example, the format phrase *I7* means that the values in the target columns are to be converted to integers and written to a 7-character output field. If the width specified (by the *width* parameter) is greater than the width of the formatted value, the value is right-justified in the output field, and leading spaces are inserted to fill the field.

If the width is too small to accommodate all of the integer's digits (and a negative symbol, if necessary), then the entire output field is filled with stars.

**Table 4–5 I Format Phrases**

Format Phrase	Value in Target Column	Formatted Result <sup>1</sup>
<i>I8</i>	443322	ΔΔ443322
<i>I6</i>	443322	443322
<i>I5</i>	443322	*****
<i>I5</i>	221.6764	ΔΔ222
<i>I5</i>	-221.6764	Δ-222
<i>I5</i>	.00245	ΔΔΔΔ0
<i>I5</i>	0	ΔΔΔΔ0

<sup>1</sup>Note that the delta(Δ) represents the space character

(continued on next page)

**Table 4–5 (Cont.) I Format Phrases**

Format Phrase	Value in Target Column	Formatted Result <sup>1</sup>
ZI5	0	00000
BI5	0	Δ Δ Δ Δ Δ
BI5	.00245	Δ Δ Δ Δ Δ
LI5	0	0 Δ Δ Δ Δ
K4I5	5	50000
K4I5	55	* * * *
0¶ZERO¶LI8	2.5	3Δ Δ Δ Δ Δ Δ Δ
0¶ZERO¶LI8	0	ZEROΔ Δ Δ Δ

<sup>1</sup>Note that the delta(Δ) represents the space character

All of the qualifiers and decorators listed in Table 4–8, Section 4.1.5, are permitted with type I format phrases.

#### 4.1.3.6 Type Y—Byte Data

Type Y format phrases output the internal representation of the target data values in hexadecimal notation.

APL stores all data internally as one of four possible data types: Boolean, character, integer, or floating-point. Because each hexadecimal digit in the type Y formatted value represents four bits of internal data, you need to specify an output field width of at least 1 (representing 4 bits) for Boolean values, 2 (8 bits) for character values, 8 (32 bits) for integer values, and 16 (64 bits) for floating-point values. (See Table 4–6.)

When integer values are formatted, the high-order bits are the leftmost bits in the output field, and they are the first bits to be truncated if the width specification is less than 8. The formatted integer value is right-justified in the output field, and leading zeros are suppressed. For example, the integer value 47 is stored internally as the following:

0000002F

Thus:

```
'Y8' ⎕FMT 47
2F
```

## The Report Formatter

### 4.1 Format Phrases

Or, if you use the zero fill qualifier:

```
'ZY8' OFMT 47  
0000002F
```

If you specify a width smaller than 8, the internal value is truncated on the left. If the truncation would include any significant digits, the output field is replaced by stars. For example:

```
'Y2' OFMT 47  
2F  
'Y1' OFMT 47  
*
```

When floating-point (VAX D-floating) values are formatted, they are left-justified in the output field. The bytes are rearranged so that the sign and exponent appear first (on the left), followed by the fraction part with trailing zeros suppressed (unless you use the Z qualifier, which makes the value look similar to customary binary representations of floating-point data). Thus, type Y format floating-point values display the bits from the internal value in the following order: bits 15 through 0, bits 31 through 16, bits 47 through 32, and bits 63 through 48. (For details about how VAX stores D-floating values, see the *VAX MACRO and Instruction Set Reference Manual*.)

Note that if one value in an array must be stored in floating-point form—either because it was input in fixed- or floating-point form, or because it could not be stored as an integer (because it was not in the range -2147483648 to 2147483647) then all the values in the array are stored in floating-point form:

```
'Y8' OFMT 13  
1  
2  
3  
'ZY8' OFMT 1E1,13  
42200000  
40800000  
41000000  
41400000
```

The qualifiers B, L, S, Z, O, and R are permitted with type Y format phrases.

**Table 4–6 Y Format Phrases**

Format Phrase	Value in Target Column	Formatted Result <sup>1</sup>
<i>Y</i> 1	1	1
<i>Y</i> 1	0	0
<i>Y</i> 2	'A'	61
<i>Y</i> 3	'A'	Δ61
<i>Y</i> 8	-1+2*31	*****
<i>Y</i> 8	2147483647	7FFFFFFF
<i>Y</i> 8	-2147483648	80000000
<i>Y</i> 16	-1+2*31	4FFFFFFFE000004
<i>Y</i> 8	24444	ΔΔΔΔ5F7C
<i>ZY</i> 8	24444	00005F7C
<i>BLY</i> 8	24444	5F7CΔΔΔΔ
<i>BLY</i> 8	0	ΔΔΔΔΔΔΔΔ
<i>Y</i> 16	2.5E7	4CBEBC2
<i>ZY</i> 16	2.5E7	4CBEBC2000000000
<i>ZY</i> 16	-2.5E7	CCBEBC2000000000

<sup>1</sup>Note that the delta(Δ) represents the space character

#### 4.1.3.7 Type T—Absolute Tab

Type T format phrases are not associated with values in the right argument; they affect only the positioning, not the format, of the next output field in the  $\square FMT$  result.

APL has an internal pointer, called the print column pointer, that references the column that is to the right of the rightmost column in the output field that received the last formatted value of the  $\square FMT$  result. This pointer indicates the leftmost position of the next field to be output.

Type T format phrases change the value of the pointer to the column specified in the column parameter. For example:

```
'I5'  $\square FMT$  1 301234 5566 5874
1234 5566 5874
'T5,I5,T15,I5,T25,I5'  $\square FMT$  1 301234 5566 5874
1234      5566      5874
```

## The Report Formatter

### 4.1 Format Phrases

The column parameter, if specified, must be an integer in the range 0 through 255. If you omit the column parameter (or specify 0), the pointer is moved to the next column to the right of the rightmost column:

```
'T15,I5,T5,I5,T,I5' ⓁFMT 1 3p1234 5566 5874  
5566 1234 5874
```

When columns in the ⓁFMT result are overlapped, the new values overwrite any values written previously, except that blanks which occur as fill characters in the new value do not overwrite the old value. For example:

```
OLD+1 21 p'THIS IS THE OLD VALUE'  
NEW+1 2p 219 4455  
'21A1' ⓁFMT OLD  
THIS IS THE OLD VALUE  
'21A1,T5,I8,T13,I9' ⓁFMT (OLD;NEW)  
THIS IS T2190LD V4455
```

Note that the fill characters generated by the I8 and I9 format phrases did not overwrite the previously written values.

If a type T format phrase moves the pointer beyond (to the right of) the rightmost previously used column, the ⓁFMT result is extended with blanks:

```
⌚RESULT←'I5,T35' ⓁFMT 12345  
12345  
p RESULT  
1 34
```

Here, the pointer was left at column 35. Because no more values were output, column 35 was not used and the result has 34 columns.

No qualifiers are permitted with type T format phrases. If a repetition parameter (*rep*) value is specified with this format phrase, it is ignored.

#### 4.1.3.8 Type X—Relative Tab

Type X format phrases are not associated with target values in the right argument; they affect only the positioning, not the format, of the next output field in the ⓁFMT result.

APL maintains internally a print column pointer which references the column in the ⓁFMT result that is to the right of the rightmost column in the output field that received the last formatted value. This pointer indicates the leftmost position of the next field to be output.

## The Report Formatter

### 4.1 Format Phrases

Type X format phrases change the value of the pointer by the number of column positions specified in the column parameter (*col*). A negative value for the column parameter moves the pointer to the left; a positive value moves the pointer to the right. (A negative value may be indicated by either the minus or the high minus sign.) For example:

```
'I5,X10,I5,X-10,I5' DFORMAT 1 3p777 888 999  
777      999  888
```

When columns in the *DFMT* result are overlapped, the new values overwrite any values written previously, except that blanks which occur as fill characters in the new value do not overwrite the old value. For example:

```
'I5,X3,I5,I6,X-5,I5' DFORMAT 1 4p555 66666 77777 888  
555      66666 77888
```

Note that the fill characters generated when the *I5* format phrase formatted 888 did not overwrite the 7s remaining from the previously written field.

If a type X format phrase moves the pointer beyond (to the right of) the rightmost previously used column, the *DFMT* result is extended with blanks:

```
D+RESULT+I5,X35! DFORMAT 12345  
12345  
    oRESULT  
1 40
```

Here, the pointer moved from column 6 to column 41. Because no more values were output, column 41 was not used and the result has 40 columns.

If a type X format phrase specifies the pointer value to be less than 1, *DFMT* makes the pointer value 1. For example:

```
'I5,X-10,I6,X5,I5' DFORMAT 1 3p555 2222 666  
2222      666  
'I5,X-10,I6,X-10,I5' DFORMAT 1 3p555 2222 666  
6662
```

The value of the column parameter in type X format phrases must be an integer in the range -255 through 255. If *col* has the value zero, the column referenced by the print column pointer does not change. If the *col* is omitted, the effect is as if the column parameter had the value 1.

No qualifiers are permitted with type X format phrases.

## The Report Formatter

### 4.1 Format Phrases

#### 4.1.3.9 Type Literal

You can use the literal format phrase to insert literal data into the `□FMT` result. This type of format phrase is not associated with a target field in the right argument. The literal text surrounded by the delimiters is copied unchanged to the output array, in the position referenced by the current value of the print column pointer. For example:

```
'@THE VALUE IS:8,F6.3' □FMT 2.415  
THE VALUE IS: 2.415
```

The literal *text* may be empty. No qualifiers are permitted with the literal format phrase.

#### 4.1.4 Format Phrase Parameters

The format phrase type identifies the data type of the values to be formatted; the format phrase parameters described in Table 4–7, specify additional information about how the values are to be formatted. For example, format phrase parameters control the output field width, the position of the values in the output field, and the insertion of special symbols or text to the right or left of the values in the output field.

One of the parameters specified in Table 4–7, the *qual* parameter, actually refers to a series of format phrase modifiers called qualifiers and decorators. These modifiers are listed in Table 4–8, and are further described in the remaining sections of this chapter.

**Table 4–7 Summary of Format Phrase Parameters**

Parameter	Meaning
<i>rep</i>	The number of consecutive target columns to be affected by the format phrase, or the number of times a parenthesized group of format phrases is to be repeated (to a maximum of 65534).
<i>qual</i>	One or more of the format phrase qualifiers or decorators listed in Table 4–8.
<i>width</i>	The width in the result array of the formatted value from the target column in the right argument. The width must be an integer in the range 1 through 255.

(continued on next page)

**Table 4–7 (Cont.) Summary of Format Phrase Parameters**

Parameter	Meaning
<i>dig</i>	The number of decimal places (F, or fixed-point, format) or significant digits (E, or floating-point with exponent, format) to be included in the result array. The digit parameter's value must be an integer in the range 0 through 127.
<i>column</i>	For the T (absolute tab format), an integer in the range 0 through 255 (0 is the same as T by itself) that identifies the leftmost column that the next formatted value is to occupy in the result array. For the X (relative tab) format, an integer in the range -255 through 255 that identifies the number of columns to be shifted before the next formatted value is output.

#### 4.1.5 Format Phrase Qualifiers and Decorators

The format phrase qualifiers and decorators modify the actions of the basic format phrase types. Each format phrase may include multiple qualifiers and decorators specified in any order, but a particular qualifier or decorator may appear only once in the phrase. The qualifiers and decorators that are permitted with each format phrase type are summarized in Table 4–9.

When more than one qualifier or decorator is specified and their actions conflict, the following rules apply:

- The B qualifier overrides the effects of the Z, C, and O qualifiers and the M, N, P, and Q decorators.
- Zeros inserted because of a Z qualifier are not affected by an L qualifier.
- The M qualifier overwrites the minus sign usually displayed with negative values.
- The O qualifier overrides the effects of the P and Q decorators.
- The R qualifier has no effect on zeros inserted because of a Z qualifier, or on blanks in the M, N, O, P, and Q decorators. The R qualifier, however, does overwrite blanks inserted because of a B qualifier.

## The Report Formatter

### 4.1 Format Phrases

**Table 4–8 Summary of Format Phrase Qualifiers and Decorators**

Qualifiers	Meaning
B	For types I, E, F, G, and Y, if the value of the item in the target column is zero, make the field in the target column blank in the result array.
C	For types I and F, insert commas between each group of three digits in the integer part of the formatted value.
L	For types I, F, E, A, and Y, left-justify the fields in the target column.
$Kn$	For types I, F, G, and E, before formatting the fields in the target column, multiply the fields by the scale factor $10^n$ .
$S_a symbol pairs_a$	For types I, E, F, and Y, replace, in the formatted output, all occurrences of the first character in each symbol pair with the corresponding second character of the symbol pair. For type G, replace, in the pattern, all occurrences of the first character in each symbol pair with the corresponding second character of the symbol pair.
$Wn$	For type E, use $n$ exponent digits in the formatted output.
Z	For types I, F, and Y, fill leading blanks in the formatted output with zeros.
Decorators	Meaning
$M_a text_a$	For types I, F, and G, replace the sign of negative formatted values with $a text_a$ placed to the left of the value.
$N_a text_a$	For types I, F, and G, place $text$ to the right of negative formatted values.
$O_a text_a$	For types I, F, G, and Y, replace formatted zero values with $text$ .
$P_a text_a$	For types I, F, and G, place $text$ to the left of positive formatted values.
$Q_a text_a$	For types I, F, and G, place $text$ to the right of positive formatted values.
$R_a text_a$	For types I, F, E, A, G, and Y, fill unused columns in the formatted output with $text$ .

Note that  $text$  may be empty for M, N, P, and Q.

**Table 4–9 Valid Qualifiers, Decorators, and Parameters for Format Types**

<b>Format Phrase</b>	<b>Permitted Qualifiers</b>	<b>Permitted Decorators</b>	<b>Parameters</b>
A	L	R	w r
Y	B L S Z	O R	w r
I	B C L Kn S Z	M N O P Q R	w r
F	B C L Kn S Z	M N O P Q R	w d r
E	B L Kn Wn S	O Q	w d r
G	B L S	M N O P Q R	r
literal			r
T			r c
X			r c

**Key to Parameters**

w—width parameter. A value must be specified and must be a positive integer.

d—dig parameter. A value must be specified and must be a positive integer.

r—rep parameter. A value is optional, but if specified must be a positive integer or zero.

c—column parameter. A value is optional. If specified with the T format phrase, it must be a positive integer or zero. If specified with the X format phrase, it must be an integer.

**4.1.5.1 B—Blank When Zero**

The B qualifier replaces the output field with blanks if the formatted value is equal to zero. For example:

```
'F8.2' 0FMT 1 5 p249.54 0 762.27 0 6
249.54      0.00   762.27      0.00     6.00
'BF8.2' 0FMT 1 5 p249.54 0 762.27 0 6
249.54      762.27      6.00
```

The B qualifier is permitted with the type I, F, E, G and Y format phrases.

**4.1.5.2 C—Insert Commas**

The C qualifier inserts commas between each group of three digits in the integer part of a formatted value. For example:

```
'F14.1,I10' 0FMT 1 4 p12249.49 734214 91142452.15 2150
12249.5    734214 91142452.0          15
'CF14.1,I10' 0FMT 1 4 p12249.49 734214 91142452.15 2150
12,249.5    734,214 91,142,452.0        15
```

The C qualifier is permitted with the type I and F format phrases.

## The Report Formatter

### 4.1 Format Phrases

#### 4.1.5.3 Kn—Scale Factor

The **Kn** qualifier multiplies the target value by the scale factor  $10^n$  before the value is formatted. The *n* must be an integer that is positive, zero, or negative (you can use a minus or a high minus sign to make it negative). For example:

```
'F10.2,I10' 0FMT 1 4 p249.49 29 762.27 881124  
249.49      29    762.27    881124  
'K3F10.2,K2I10' 0FMT 1 4 p249.49 29 762.27 881124  
249490.00    2900 762270.00  88112400  
'K-4F10.2,K-3I10' 0FMT 1 4 p249.49 29 762.27 881124  
0.02        0     0.08      881  
'K-50F4.2,I10' 0FMT 20  
0.00
```

The **Kn** qualifier is permitted with the type I,E,F, and G format phrases. It is particularly useful with type G, which rounds to integers.

If a value of *n* is too large, it may cause the scaled value to be too large to represent:

```
'K50F10.2' 0FMT 20  
27 LIMIT ERROR (FLOATING OVERFLOW)  
'K50F10.2' 0FMT 20  
^
```

#### 4.1.5.4 L—Left-Justify

The **L** qualifier places the target value left-justified in its output field. For example:

```
'F10.2,I10' 0FMT 1 4 p249.49 29 762.27 881124  
249.49      29    762.27    881124  
'LF10.2,LI10' 0FMT 1 4 p249.49 29 762.27 881124  
249.49      29    762.27    881124
```

The **L** qualifier is permitted with the type I,F,E,A, and Y format phrases.

#### 4.1.5.5 S—Standard Symbol Substitution

The **S** qualifier replaces, in the formatted value, certain occurrences of the first character in a specified symbol pair with the second character in the pair. The symbol pairs are placed in the format phrase immediately following the **S**. The first character in each symbol pair—that is, the character to be replaced—must be a star (\*), decimal point (.), comma, (,), or zero (0). The first characters are replaced in the formatted value by the second character in each symbol pair as follows:

- \* Replaced if it occurs as an overflow indicator for a value formatted by a type Y,I,F,G, or E format phrase.
- . Replaced if it is a decimal point that occurs in a value formatted by a type E or F format phrase.
- , Replaced if it occurs in a value formatted by a type I or F format phrase that had a C qualifier.
- 0 Replaced if it occurs as a fill character in a value formatted by a type I, Y, or F format phrase that had a Z qualifier, or a type G format phrase that had a 9-pattern character.

For example:

```
'CI5' DFORMAT 1 2 p1234 12345
1,234*****
'S<,.**->CI5' DFORMAT 1 2 p1234 12345
1.234-----
'ZF8.2' DFORMAT 1 2p555.66 29.88
00555.6600029.88
'S<.,0*>ZF8.2' DFORMAT 1 2p555.66 29.88
**555,66***29,88
```

The S qualifier may also be used with the type G format phrase to substitute an alternate character for the 9, Z, or @ pattern character. For details, see Section 4.1.3.4.

#### 4.1.5.6 Wn—Exponent Digits

The Wn qualifier changes the default width of the exponent digit of an exponential value from 2 to the value specified by n. The n must be an integer in the range 1 through 3. For example:

```
'E12.4' DFORMAT 1 4 p249.49 267E-5 .23 99
2.495E2      2.670E-3     2.300E-1     9.900E1
'W1E12.4' DFORMAT 1 4 p249.49 267E-5 .23 99
2.495E2      2.670E-3     2.300E-1     9.900E1
'W3E12.4' DFORMAT 1 4 p249.49 267E-5 .23 99
2.495E2      2.670E-3     2.300E-1     9.900E1
```

The Wn qualifier is permitted with type E format phrases only.

#### 4.1.5.7 Z—Zero Fill

The Z qualifier inserts zeros into unused leading positions of values formatted by a type I, F, or Y format phrase. For example:

```
'I8,F8.2' DFORMAT 1 2p777 88.8
777     88.80
'ZI8,2F8.2' DFORMAT 1 2p777 88.8
000007770088.80
```

## The Report Formatter

### 4.1 Format Phrases

#### 4.1.5.8 M and N—Negative Left and Right Decorators

The M and N decorators insert text beside negative values formatted by a type I, F, or G format phrase. If the M decorator is used, the text is placed to the left of the value and it replaces the negative sign. If the N decorator is used, the text is placed to the right of the value and the negative sign still appears to the left of the value. For example:

```
'M<CREDIT>F13.2' 0FMT 1 2p29.99 -249.54  
29.99 CREDIT249.54  
     AEMPTY VALUE FOR M ELIMINATES SIGN  
'M<>F13.2' 0FMT 1 2p29.99 -249.54  
29.99      249.54  
'N<CREDIT>F14.2' 0FMT 1 2p29.99 -249.54  
29.99 -249.54CREDIT
```

The maximum length of the inserted text is 255 characters. Note that the field width you specify must be large enough to accommodate both the value and the inserted text. An empty text string is permitted.

#### 4.1.5.9 P and Q—Positive Left and Right Decorators

The P and Q decorators insert text beside positive values formatted by a type I, F, or G format phrase. If the P decorator is used, the text is placed to the left of the value. If the Q decorator is used, the text is placed to the right of the value. For example:

```
'P<DEBIT>F13.2' 0FMT 1 2p29.99 -249.54  
DEBIT29.99 -249.54  
'Q<DEBIT>F13.2' 0FMT 1 2p29.99 -249.54  
29.99DEBIT -249.54
```

The maximum length of the inserted text is 255 characters. Note that the field width you specify must be large enough to accommodate both the value and the inserted text. An empty text string is permitted.

#### 4.1.5.10 O—Zero Decorator

The O decorator inserts text in place of data that has been formatted as a zero value. The text is right-justified in the output field. For example:

```
'20<ZERO>F6.1,2 O<NO VALUE>I9' 0FMT 1 4p 2.5 .01445 598 .499  
2.5 ZERO      598 NO VALUE
```

The O decorator is permitted with type I, F, G, and Y format phrases. The maximum length of the inserted text is 255 characters. Note that the field width you specify must be large enough to accommodate the inserted text.

#### 4.1.5.11 R—Background Decorator

The R decorator inserts text into unused portions of a formatted value's output field. Starting at the left, the text is repeated as many times as necessary to fill the field. For example:

```
'R<-/*>I10' DFORMAT 1 30 22 333 4444
-/*-//*-/22-/*-/*-333-/*-*4444
```

The R decorator does not replace zeros inserted by the Z qualifier, nor does it replace blanks inserted by other decorators. However, the R decorator does replace blanks inserted by the B qualifier.

The R decorator is permitted with type I, F, E, A, G, and Y format phrases. The maximum length of the inserted text is 255 characters.

## 4.2 Right Argument

The right argument is a list of simple homogeneous arrays of any type or rank. The list must be surrounded by parentheses (unless there is only one array in the list), and the arrays must be separated by semicolons. Missing elements in the list do not affect the result. For example,  $(A ; B)$  is the same as  $(A ; B)$ . Alternatively, the right argument may be an enclosed vector of depth 2. In other words, the argument is in the vector domain and its items are simple homogeneous arrays. Each item of the vector is treated in the same manner as an element of a list type argument.

Before formatting the right argument to produce the result matrix, DFORMAT combines the arrays in the list and arranges them as an intermediate matrix in canonical form. The successive columns (target columns) of the intermediate matrix are formatted for the result array according to specifications in successive format phrases in the left argument. Vectors and enclosed scalars, for example, are treated as 1-column matrices:

```
TOTALS+ 479.59 29.99 12799.50 1444.09 325.88
'F8.2' DFORMAT TOTALS
479.59
29.99
12799.50
1444.09
325.88
```

In this example, the items in the vector *TOTALS* were treated as one target column of values, that is, as a matrix with the shape 5 1. Then, the values were output as specified by the format phrase *F8.2*.

## The Report Formatter

### 4.2 Right Argument

Arrays with ranks greater than 2 are treated as matrices. The matrices look the same as the original arrays would look if they were displayed without blank lines between planes. For example:

```
aSTANDARD DISPLAY OF ARRAY NUMS
◻←NUMS←2 4 5p 140
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

aNUMS AS RIGHT ARGUMENT
'I3,I4,I5,I6' ◻FMT NUMS
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
```

Note that each of the five columns of data in the right argument to  $\square FMT$  is a target column, so the left argument has five format phrases, one for each column.

After the arrays in the list are reshaped, they are placed side by side in the intermediate matrix.

For example, if both *NUMS* and *TOTALS* were included in the right argument to  $\square FMT$ , they would first be converted to matrices as described, and then placed side by side to be treated as a single matrix by  $\square FMT$ , as follows:

col 1	col 2	col 3	col 4	col 5	col 6
1	2	3	4	5	479.59
6	7	8	9	10	29.99
11	12	13	14	15	12799.50
16	17	18	19	20	1444.09
21	22	23	24	25	325.88

col 1	col 2	col 3	col 4	col 5	col 6
26	27	28	29	30	
31	32	33	34	35	
36	37	38	39	40	

Six target columns result, so the left argument to `⎕FMT` would need six format phrases:

```
TOTALS← 479.59 29.99 12799.50 1444.09 325.88
NUMS← 2 4 5 140
'13,13,13,I3,F9.2' ⎕FMT (NUMS;TOTALS)
1 2 3 4 5 479.59
6 7 8 9 10 29.99
11 12 13 14 15 12799.50
16 17 18 19 20 1444.09
21 22 23 24 25 325.88
26 27 28 29 30
31 32 33 34 35
36 37 38 39 40
```

## 4.3 Result Array

The result array is a character matrix formed according to the specifications of successive format phrases from the left argument, as applied to successive target columns from the canonical form of the right argument. When the result array is formed, however, the nonediting format phrase types T, X, and *literal* are not associated with target columns.

If there are extra target columns, the left argument is rescanned beginning with the leftmost format phrase. If there are extra format phrases, they are ignored, except that the format phrase types T, X, and *literal* will continue to be processed until APL encounters one of the following:

- A format phrase type other than T, X, or *literal*
- The end of the format string
- A left parenthesis
- A right parenthesis with an incompletely used repetition count

The result matrix has a number of rows equal to the longest column of the right argument's intermediate matrix; short columns are filled with blanks. Within each output field in the result matrix, the data values are right-justified unless you use the L qualifier (except that type Y floating-point data is normally left-justified). Negative values are output with a minus sign (APL `-`,

## The Report Formatter

### 4.3 Result Array

APL –, or ASCII –, depending on  $\square NG$ ; see the *VAX APL Reference Manual*) unless you use the M qualifier. Plus signs are not included in the formatted output unless you insert them as literal text or via decorators.

## 4.4 Formatting Character Data

When you format character data, note that the target columns in the reformatted right argument always consist of exactly one character. Because you generally want to format a string of characters rather than a single character, you will find the repetition parameter to be particularly useful. For example:

```
TEAMS+5 5p'PHILABOST.N.J. WASH.N.Y. '
'5A1' □FMT TEAMS
PHILA
BOST.
N.J.
WASH.
N.Y.
```

Here,  $\square FMT$  applied the format phrase *A1* to each of the five target columns in the right argument:

col 1	col 2	col 3	col 4	col 5
P	H	I	L	A
B	O	S	T	.
N	.	J	.	.
W	A	S	H	.
N	.	Y	.	.

You can use the width parameter to alter the spacing of the characters in the string:

```
TEAMS+5 5p'PHILABOST.N.J. WASH.N.Y. '
'A1,A2,2A1,A3' □FMT TEAMS
P HIL A
B OST .
N .J.
W ASH .
N .Y.
```

Note that *A* format right-justifies the character value in the output field.

## The Report Formatter

### 4.4 Formatting Character Data

One way to move a character string's position in the output field is to use a large value for the width parameter for the first character in the string:

```
TEAMS<5 5p'PHILABOST.N.J. WASH.N.Y. '
'A10,4A1' Dfmt TEAMS
    PHILA
    BOST.
    N.J.
    WASH.
    N.Y.
```

The characters in target column 1 (P, B, N, W, N) were right-justified (by default) in a 10-character output field.

Note that because `Dfmt` treats vectors as 1-column matrices, applying `Dfmt` to a vector of characters may not yield the result you expect. For example:

```
'5A1' Dfmt 'PHILA'
P
H
I
L
A
```

To keep the character vector on one line, convert the vector to a 1-row matrix:

```
'5A1' Dfmt 1 5p'PHILA'
PHILA
'A1' Dfmt ,[0.5]'PHILA'
PHILA
```



# 5

## VAX APL Input and Output

VAX APL is an interpreter; thus, its input and output (I/O) operations can be as simple as typing a line of input and receiving APL's immediate response with the appropriate output.

As you become more proficient with the language and begin to extend it by writing user-defined operations, you probably will want to perform I/O operations that are more complex than the I/O that occurs by default. To allow such extended I/O capability, APL provides the following:

- I/O system variables that facilitate terminal I/O operations from within user-defined operations. These variables include quad input, quote quad input, quad del input, quad output, and bare output.
- File system operations that allow you to manipulate data in external files. These include `)INPUT`, `)OUTPUT`, and others.

All forms of I/O can be used either in immediate mode or within user-defined operations; however, the I/O variables are more commonly used within user-defined operations.

### 5.1 Terminal Input and Output

Input and output operations not involving external files—the default terminal I/O and the use of I/O variables—are sometimes called terminal I/O, because the only I/O device involved is your terminal.

The default terminal I/O is straightforward; you enter input from your terminal, and APL echoes your input, beginning in column 7. If the statement you enter does not have a quiet function as the leftmost function, APL prints the result beginning at column 1 on the next line of your terminal. If the statement does have a quiet function as the leftmost function, specification (`←`), for example, APL does not display a result.

# VAX APL Input and Output

## 5.1 Terminal Input and Output

```
      ⌂+A<25      ⌂QUAD SPECIFICATION IS NOT QUIET  
25      ⌂B+64*3      ⌂SPECIFICATION FUNCTION WITHOUT QUAD IS QUIET  
        17+A ⌂ B  
42      262144
```

APL does not control wrapping of input lines. If the system setting for your terminal allows wrapping, input lines that are too long to be echoed as one line on your terminal are continued (wrapped) on subsequent lines. If the system setting for your terminal inhibits wrapping, input is not echoed on more than one line; any characters that do not fit on one line are displayed in the last column on the line. Thus, when the input line is complete, the last column contains the last character entered.

You can cancel an input line before entering it (before pressing the Return key) by entering the abort input signal. (See Section 1.9.)

### 5.1.1 Terminal Input Variables

The quad input (⌂), quote quad input (⌂) and quad del input (⌂) system variables allow you to request input from the terminal. When one of these system variables appears in an expression, APL displays a prompt; the result of any expression entered in response to this prompt becomes the value of that system variable.

Typically, these system variables are used with the specification function (←) so that the value of the input data is assigned to a variable. However, these system variables are legal in any context that requires a value.

While the system is waiting for your input, you can execute a system command or you can define or edit an operation; the input request remains pending until you supply a value. APL cancels the input request if you do any of the following: enter Ctrl/Z; execute one of the system functions ⌂RESET, ⌂BREAK, or ⌂SIGNAL; or execute a system command that changes the state of the active workspace, that is, a )LOAD, )XLOAD, )CLEAR, )OFF, )CONTINUE, )MON or )SIC command.

To escape without entering a value, either type the right-arrow (→) character, or enter the abort input signal. If you are inside an operation when you escape, the operation is suspended (unless it is locked).

Note that escaping with branch (→) is quiet; APL simply cancels the input request. When you escape from any of the input system variables with the abort input signal, however, APL cancels the input request and signals *INPUT ABORTED* (trappable with ⌂TRAP; branch (→) is not trappable).

## VAX APL Input and Output

### 5.1 Terminal Input and Output

If the input you enter contains an error, APL prints the appropriate error message and reissues the input prompt.

#### 5.1.1.1 Quad Input

Quad input (`◻`), also known as evaluated input, allows you to request input from the terminal. The default prompt is `◻:`, followed by a <CR><LF> and six spaces. You can define your own prompt with `◻SF` (see the *VAX APL Reference Manual*).

Note that using the `◻` input system variables to request input inside of an operation is convenient. For example:

```
    VR←SQUARE;A
[1]  'ENTER VALUE TO BE SQUARED'
[2]  A←◻
[3]  R←A*2  ⌂
      SQUARE
ENTER VALUE TO BE SQUARED
◻:
      5
25
```

If you enter multiple statements separated by the diamond (`◊`) character, APL evaluates them individually, beginning with the leftmost statement. APL uses the value of the rightmost statement as the value of quad input. For example:

```
    B←◻
◻:
      X←1 ◊ Y ←2 ◊ Z←3
      B
3
      X
1
      Y
2
      Z
3
```

Multiple expressions separated by the output catenator (`;`) are not allowed to quad input. For example:

```
    A←◻
◻:
      1;2;3
15 DOMAIN ERROR (SEMICOLON LIST NOT ALLOWED)
      1;2;3
      ^
◻:
```

## VAX APL Input and Output

### 5.1 Terminal Input and Output

If you enter character data in response to the `⊣` input prompt, you must use single quotation marks. For example:

```
NAME ← ⊣  
⊣:  
'JAMES CLERK MAXWELL'
```

APL reprompts if you enter an attention signal, an illegal overstruck character, or an expression with no value, such as a blank line or an operation that does not return a value.

#### 5.1.1.2 Quote Quad Input

When quote quad input (`⊣`) input appears in an expression, APL accepts the data between the current cursor position and the next carriage return as a character value. For example:

```
'2 + 3',⊣      ⍝CATENATE TWO CHARACTER VECTORS  
IS A VERY SIMPLE EQUATION.  
2 + 3 IS A VERY SIMPLE EQUATION.
```

If you enclose the input in quotation marks, the quotation marks are taken as part of the value:

```
X←⊣  
'THAT'S AMAZING'  
X  
'THAT'S AMAZING'
```

If the input is a single character, `⊣` input is a character scalar. If the input is two or more characters, `⊣` input is a character vector. If you enter only a Return or an attention signal, `⊣` input is an empty character vector.

Because whatever you type is accepted as part of a character value, you cannot execute system commands or invoke the function editor while `⊣` input is pending.

Legal overstrikes typed while `⊣` input is pending are accepted as one character. Treatment of illegal overstrikes depends on the character set being used in the session. Terminals using the APL COMPOSITE character set do not generate illegal overstrikes; instead, they create the squish quad symbol. The squish quad symbol is treated as one character. Illegal overstrikes are accepted as three characters for all other character sets. For example:

```
⊣TT      ⍝SESSION USING COMPOSITE CHARACTER SET  
19  
X←⊣
```

## VAX APL Input and Output

### 5.1 Terminal Input and Output

```
1+[]¤A  
ρX  
5  
X[3 4 5]  
[]¤A  
[]TT*2      SESSION USING TTY CHARACTER SET  
X_.QQ  
1+.XX.TRA  
ρX  
7  
X[3 4 5 6 7]  
.XX.TRA
```

#### 5.1.1.3 Qual Del Input

Quad del ([]) input is similar to quote quad (") input for sessions using the COMPOSITE character set. For other character sets, the entered characters returned remain untranslated. What is normally a legal APL overstruck character becomes three characters.

In the following example, the session using the COMPOSITE character set enters an illegal overstrike, which is accepted by APL as one character, squish quad. The legal overstrike character (¤), which is entered as Ctrl/D o \ , is also accepted as one character.

From the session using the TTY character set, APL accepts the illegal overstrike character (.XX) and the legal overstrike character (.TR) as three characters each.

```
[]TT      SESSION USING COMPOSITE CHARACTER SET  
19  
X<[]  
1+[]¤A  
ρX  
5  
X[3 4 5]  
[]¤A  
[]TT*2      SESSION USING TTY CHARACTER SET  
X_.QD  
1+.XX.TRA  
ρX  
9  
X[3 4 5 6 7 8 9]  
.XX.TRA
```

As with quote quad input, if you enter only a Return, or if you enter an attention signal, APL treats the input as an empty character vector. If you enter a single character, the [] input is a character scalar.

## VAX APL Input and Output

### 5.1 Terminal Input and Output

#### 5.1.2 Terminal Output

When APL outputs a character array that fits on one line (of length  $\Box PW$ ), it begins its display in column 1 and outputs the array unchanged. When APL outputs a numeric array that fits on one line, it begins its display in column 1 and separates each element in the array with one blank.

When APL outputs an array that cannot fit on a single line, the remainder of the line prints on succeeding lines, indented six spaces. For example:

```
     $\Box PW$ 
132   A←'THIS LINE IS LONGER THAN 35 CHARACTERS'
      A
      THIS LINE IS LONGER THAN 35 CHARACTERS
       $\Box PW←35$ 
      A
      THIS LINE IS LONGER THAN 35 CHARACTERS
```

When displaying arrays that have three or more axes, APL inserts one blank line between each plane and one additional blank line for each additional axis. For example:

```
      2 2 2 40 140
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
```

Numeric values are broken only between single numbers; character values may be broken between any two character elements. Note that the formatting of numeric arrays is done independently of the current  $\Box PW$  setting. As a result, when a wrap occurs, decimal points in wrapped lines are not necessarily aligned.

In numeric output, APL does not display the following:

- Plus signs
- Trailing zeros after the decimal point
- Trailing decimal points

## VAX APL Input and Output

### 5.1 Terminal Input and Output

- Leading zeros (except for numbers between  $-1$  and  $1$ , which are preceded by one zero)

For example:

```
□←A←+1 9.0 41. .99
1 9 41 0.99
```

In multiline numeric output, APL formats each column independently of the other columns. Within a column, all decimal points line up, and all numbers are right-justified. There is exactly one space between the longest element in a column and the longest element in any adjoining column. In the first column of elements, the element with the most digits to the left of the decimal point begins in column 1. For example:

```
3 3p12.345 1.2
12.345 1.2 12.345
1.2 12.345 1.2
12.345 1.2 12.345
```

APL displays numbers in fixed-point rather than floating-point format, unless one of the following is true:

- The integer part has more than  $\Box PP$  digits.
- The fixed-point representation of the magnitude would require more than  $\Box PP+n+3$  characters (where  $n$  is the number of exponent digits, and the constant 3 allows for a decimal point, an  $E$ , and an exponent sign). In this case, floating-point format is used because it uses less space than fixed-point representation.

For example:

```
□PP
5
□←A←12345678 1234567890
12345678 1234567890
□←A←12345678 12345678901
1.23456E7 1.23456E10
```

The following rules govern the display of floating-point numbers:

- There are  $n+2$  characters allotted for the exponent, where  $n$  is the number of exponent digits, and the constant 2 allows for an  $E$  and an exponent sign.
- Within a column, the  $E$  lines up. As a result, trailing zeros may appear in the fraction field so that the decimal points are aligned.
- The exponent field is left-justified.

## VAX APL Input and Output

### 5.1 Terminal Input and Output

- The decimal point (if any) is positioned to the right of the leftmost digit.
- The exponent of 0 is 0.

For example:

```
3 30 1 253 7E35 2 1.23E9 5 65.3 0 10
1 2.53E2      7E35
2 1.23E9      5E0
65.3 0.00E0    1E1
```

#### 5.1.2.1 Output Catenator

The output catenator, the semicolon ( ; ), prints data from more than one expression on the same line. The expressions can mix both character and numeric simple data, but may not contain heterogeneous or enclosed values.

To use the output catenator, enter a series of expressions, separated by semicolons, in the order in which they are to appear. The expressions in a semicolon list must be separated by exactly one semicolon, and there may not be any leading or trailing semicolons. Incorrect uses of semicolons will result in system messages, as explained in the *VAX APL Reference Manual*.

Although APL evaluates expressions from right to left, it displays values separated by semicolons from left to right. For example:

```
1+1; 'CHAR'; 3+3
2CHAR6
13;15;14
1 2 31 2 3 4 51 2 3 4
```

Note that when you enter two or more expressions separated by semicolons, APL does not add spaces between the results; you must specify a space if you want one:

```
1+1;2+2
24
1+1; ' ' ;2+2
2 4
A←2;B←3;C←4
234
```

In semicolon lists, arrays of rank 2 and greater are displayed on separate lines. For example:

## VAX APL Input and Output

### 5.1 Terminal Input and Output

```
A←2 2⍴14
B←3 3⍴19
A;B
1 2
3 4
1 2 3
4 5 6
7 8 9
16;2 2⍴14
1 2 3 4 5 6
1 2
3 4
2 2⍴14;16
1 2
3 4
1 2 3 4 5 6
```

All expressions delimited by the output catenator must return a value. For example, if a user-defined function *F* does not return an explicit result, the following signals an error:

```
∇F
[1]      'HI THERE'
[2]      ^
1;F;2
HI THERE
11 VALUE ERROR (FUNCTION DOES NOT RETURN A RESULT)
1;F;2
^
```

Note that, in the preceding example, the error occurs after the function *F* is evaluated, but before APL can display the catenation of the three expressions.

```
⎵TRY AGAIN WITH A FUNCTION THAT DOES RETURN A VALUE
∇H+G
[1]      H← 'HI THERE, AGAIN '
[2]      ^
1;G;2
1HI THERE, AGAIN 2
```

Catenated output has no value, even though the individual expressions being catenated do have values. This means you cannot use catenated output in any context that requires a value, such as the argument to a function or an operator. In particular, catenated output may not be used as the argument to the execute function, and may not be surrounded with parentheses.

## VAX APL Input and Output

### 5.1 Terminal Input and Output

Catenated output may be mixed with the statement separator symbol ( $\diamond$ ):

```
A←1 ⋄ B←2
'THES E ARE THE NUMBERS 1 AND 2: ' ⋄ A; ' ' ;B
THESE ARE THE NUMBERS 1 AND 2:
1 2
```

#### 5.1.2.2 Quad Output

The result of quad output ( $\Box\leftarrow$ ) prints on the terminal. For example:

```
A←25
◻←A
25
```

Note that using quad output has the same effect as merely typing the variable name (in this case, *A*). Quad output is helpful when an APL statement contains multiple specification operations. For example:

```
B←3+◻←5×4
20
```

This statement displays the result of the intermediate computation  $5 \times 4$ , then adds 3 and assigns the result to *B*. The use of  $\Box$  output is more efficient than the following:

```
A←5×4
A
20
B←3+A
```

#### 5.1.2.3 Bare Output

Bare output (either  $\Box\leftarrow$  or  $\Box\leftarrow$ ) works the same way as quad output ( $\Box\leftarrow$ ), except that bare output does not print <CR><LF> (not even closing ones) that are not entered by the user. Thus, bare output provides a convenient way to request input on the same line as an output string. Note the difference in the following examples:

```
          AQUAD OUTPUT FOLLOWED BY INPUT
◻←'ENTER YOUR NAME ' ⋄ A←◻
ENTER YOUR NAME  (Quad output inserts <CR><LF>)
```

## VAX APL Input and Output

### 5.1 Terminal Input and Output

```
IRENE
A
IRENE
pA
5
          aBARE OUTPUT WITH INPUT
L←p[]←'ENTER YOUR NAME: '◦ A←[]
ENTER YOUR NAME: IRENE    (APL waits on same line)
A
IRENE      (Bare output became part of A)
pA
22
L+A
IRENE
```

Note also that the input value is preceded by a number of spaces equal to the length of the [] output. If you do not want the spaces, you can use the `BARABOUT` function to reset the bare output buffer (see the *VAX APL Reference Manual* for more information). For example:

```
[]←'ENTER YOUR NAME' ◦ BARABOUT 95 95 32 ◦ A←[]
ENTER YOUR NAME      IRENE
A
IRENE
```

#### 5.1.3 Diverting Input and Output to Another Device

)`INPUT` and )`OUTPUT` allow you to change the source of APL input or the destination of APL output from your terminal to another device. Typically you would select a file (or another terminal) to be the new device by entering the command and the name of the VMS file to be used. For example:

```
)INPUT EMPLOYEE
.
.
.
(User enters the attention signal)
)INPUT/REVERT
```

Optionally, you can specify that the file should be read or written in a characterset other than the terminal's current characterset. In the following example, the output is written to a file using the TTY characterset.

```
)OUTPUT OUTFILE/TTY
```

## VAX APL Input and Output

### 5.1 Terminal Input and Output

Table 5–1 describes the possible values for the */character-set* qualifier. Note how the meaning of the values varies depending on the terminal designator, which you specify when you invoke APL.

**Table 5–1 Character Set for )INPUT and )OUTPUT Files**

Terminal Designator	Qualifier Value				
	/APL	/KEY	/BIT	/TTY	/COMPOSITE
BIT	bit-paired	key-paired	bit-paired	tty	composite
COMPOSITE	composite	key-paired	bit-paired	TTY	composite
VT220	composite	key-paired	bit-paired	TTY	composite
VT240	composite	key-paired	bit-paired	TTY	composite
VT320	composite	key-paired	bit-paired	TTY	composite
VT330	composite	key-paired	bit-paired	TTY	composite
VT340	composite	key-paired	bit-paired	TTY	composite
VS	composite	key-paired	bit-paired	TTY	composite
DECTERM	composite	key-paired	bit-paired	TTY	composite
HDSAVT	key-paired	key-paired	bit-paired	TTY	composite
HDS201	key-paired	key-paired	bit-paired	TTY	composite
HDS221	key-paired	key-paired	bit-paired	TTY	composite
KEY	key-paired	key-paired	bit-paired	TTY	composite
LA	key-paired	key-paired	bit-paired	TTY	composite
APL	key-paired	key-paired	bit-paired	TTY	composite
GIGI	key-paired	key-paired	bit-paired	TTY	composite
4013	key-paired	key-paired	bit-paired	TTY	composite
4015	key-paired	key-paired	bit-paired	TTY	composite
VT102	key-paired	key-paired	bit-paired	TTY	composite
TTY	key-paired	key-paired	bit-paired	TTY	composite
TTY/alternate	alternate	key-paired	bit-paired	TTY	composite

The )INPUT command may be nested to a depth of 10; the )OUTPUT command may not be nested.

If you enter either the weak or strong attention signal while input is being diverted from your terminal, APL stops processing the current )*INPUT* file and puts SYS\$INPUT at the top of the nested input list (even if the list already has 10 input sources). Thus, your terminal becomes the default source of input, and none of the diverted input streams are deleted. The new SYS\$INPUT input stream added to the top of the list will have the same /*character-set* qualifier value as the SYS\$INPUT stream at the bottom of the list.

If you enter either a weak or a strong attention signal while output is being diverted from your terminal, APL responds by displaying output on your terminal as well as in the diverted stream. If output is already being shadowed on your terminal, the attention signal does not affect the output file.

The *VAX APL Reference Manual* has additional information about these commands.

## 5.2 File Input and Output

APL provides an extensive file system that allows you to process external data files with the five types of file organization that are supported by VAX Record Management Services (VAX RMS), the file processing system used by the VMS operating system. APL supports the following types of file organization:

- ASCII sequential organization—standard ASCII files in which each record (except the last) is logically adjacent to the next record.
- Internal sequential organization—files stored in internal APL format. Such files can be accessed faster than ASCII files. Each record (except the last) is logically adjacent to the next record.
- Direct-access organization—shareable, random-access files containing records, called components, that are identified by a unique index called a component number. The VAX RMS name for these files is single-key indexed files; APL uses the component number as the key value.
- Relative organization—shareable, random-access files containing records identified by a relative record number.
- Keyed organization—shareable, random-access files containing records identified by primary and/or secondary keys.

Using the APL file system to process data files is essentially a 3-step process:

1. Associate a file specification and related file information with a channel number. The file can be an existing file or one you want to create.
2. Open the file and read, write, or modify records until there are no more records to be processed.

## VAX APL Input and Output

### 5.2 File Input and Output

3. Close the file and disassociate it from the channel to which it was assigned.

The basic file system functions provide the capabilities needed for typical file-processing applications. For some applications, however, you may need to use some advanced I/O techniques. Thus, APL offers some extensions to the arguments for the basic file system functions as well as some additional file system functions. Advanced file I/O techniques are discussed later in this chapter.

#### 5.2.1 Basic File Concepts

For ASCII sequential files written by APL, a record is a line of APL output; thus, if the value of the output spans more than one line (if it is a matrix, for example), it is written as more than one record. For files written in APL with other than ASCII sequential organization, a record consists of all the data written in a single output operation (for example, all of the elements of a matrix), and the shape information is built into the record.

For all types of file organization, APL writes variable-length records by default. If you want APL to write fixed-length records, you can specify the */RECORDTYPE* switch with the *CLASS* system function (see the *VAX APL Reference Manual*). Direct-access and relative files may seem to be the same to APL users because the syntax used to process them is identical. Both use an integer index to retrieve specific records in the file: for direct-access files, the index is called a component number; for relative files, the index is called a relative record number.

Your choice of which type of file organization to use depends on the records you want the file to contain. Generally, relative files provide faster access than direct-access files because each record in a direct-access file is preceded by VAX RMS retrieval information.

You should not use relative files in two cases:

- When you need to write records that are very long. For direct-access files, components may be segmented; that is, one component may consist internally of more than one VAX RMS record. Relative files, however, cannot contain segmented records; thus, the length of the largest record possible in relative files is smaller than in direct-access files.
- When there is a large difference between the length of the largest and smallest records in the file. The variable-length records in relative files are stored in fixed-length cells, so each record uses the same amount of storage as the largest record in the file.

In a keyed file organization, each record contains one or more fields known as keys of reference. When you create a keyed file, you specify the locations, lengths, and number of keys of reference. Once specified, this structure does not change.

The first key of reference is the primary key, and subsequent keys of reference are alternate keys. The contents of any key of reference field is known as a key value.

When you write data to a keyed file, each record contains the key values that determine the location of the record inside the file. You do not specify a component or record number, since the indexing information is embedded in the record. Also, the length of the components that you write must be shorter than the maximum record length for the file. This behavior is unlike that of /DA files which segment components that are too long.

When you retrieve a record from a keyed file, you can specify a key of reference for sequential access, or a specific key value within a key of reference for random access.

Note that the term *keyed* is synonymous with *indexed*.

For more details on these file organizations and their associated access methods, see the *Introduction to VMS System Routines* and *VMS Record Management Services Manual* documentation and Section 5.2.

#### **5.2.1.1 File Access Methods**

There are two types of file access: sequential and random. Sequential access means that the records are accessed in a serial order; random access means that particular records can be accessed directly at any point in the file. The type of access you can use for a particular file depends on the file's organization.

For files with ASCII sequential or internal sequential organization, only sequential access is possible.

For files with direct-access, relative, or keyed organization, both sequential and random access are possible. When you specify a component number, record number, or key value in the argument to a read or write function, you access the file randomly; when you do not specify a component number, record number, or key value in the argument, you access the file sequentially.

When you use the sequential access method, you access the file's records in a predetermined order. Each record (except the first and last) is said to have a predecessor and a successor. Once you access a record, that record's successor is the only record you can access next.

## VAX APL Input and Output

### 5.2 File Input and Output

For files that have ASCII sequential or internal sequential organization, sequential access means that records are accessed in the order of their insertion into the file. Once you read or write a record, you must reposition the file at its beginning before you can access any earlier records.

APL positions ASCII sequential and internal sequential files at their beginning when the files are opened (unless you specify a star (\*) after the file organization switch). You must then do sequential read operations to get to the particular record you want. If you execute a write function at the beginning of the file, you create a new version of the file. If you execute a read function when you are at the end of a sequentially organized file, APL returns an end-of-file indicator (see the *VAX APL Reference Manual*).

For direct-access, relative, and keyed files, sequential access means that records are accessed in ascending order according to component number (for direct-access), record number (for relative), or key value (for keyed). A sequential write to a direct-access or relative file finds the next record by adding one to the value of the component or record number used in the previous I/O operation. A sequential read from a direct-access, relative, or keyed file retrieves the next available record (it skips empty records) as determined by the value of the file system's internal next-record pointer (see Section 5.2.1.2 for details).

Random access allows you to control the order of record access; you can access records or components in any order at any point in the file; thus, the predecessor-successor relationship is not relevant. To access a particular record, you simply execute an input or output function that specifies an index representing a component number, record number, or key value; APL accesses the component or record identified by the index. If the referenced record does not exist, APL returns an end-of-file indicator (see the *VAX APL Reference Manual*).

#### 5.2.1.2 The Next-Record Pointer

The file system uses an internal mechanism called a next-record pointer to keep track of the next record to be processed by a sequential input function. When sequential files are opened, the next-record pointer points to the beginning of the file. As each record is processed, the next-record pointer is incremented by one.

When direct-access, relative, or keyed files are opened by a sequential read or write function, a random write function, or by a `FLS`, `DVC`, `MBX`, `WAIT`, or `REWIND` function, the next-record pointer points to the beginning of the file. When the same files are opened by a random read function, the next-record pointer is set to the value of the component number, record number, or key value specified in the input function's argument.

While a direct-access, relative, or keyed file is being processed, the value of the next-record pointer changes only when an input function is executed; it never changes when an output function is executed. Thus, if you open a file, write records 1, 2, and 3, and then do a sequential read, APL retrieves record 1. If you then write more records and do another sequential read, APL retrieves record 2. You can always retrieve any record you want in these files by reading it randomly.

### **5.2.1.3 Record Handling and Sequential Operations**

Because direct-access, relative, keyed, and sequential files can be opened for both read and write functions, you can perform a mixture of input and output operations. The following rules apply to these situations:

- A sequential delete deletes the record just read or written (regardless of whether the previous read or write was sequential or random).
- A sequential write rewrites the record just read or deleted. Any number of repetitions of a sequential write followed by a sequential delete (or vice versa) will affect the same record over and over.
- The location of the next-record pointer is affected only by sequential read, random read,  $\square CLS$ , and  $\square REWIND$ . Random write or random delete never affects or modifies the next-record pointer.

Sequential files are generally opened for either read or write operations, but not for both. The four exceptions to this rule are listed below.

- When the file specification you used with  $\square ASS$  represents a terminal device.
- When the file is a mailbox.
- When the file is assigned with the */UPDATE* qualifier.
- When you invoke  $\square REWIND$  on a file initially opened for write operations.

### **5.2.2 Associating Files with Channels**

The APL system functions that read and write records take channel numbers, not file specifications, as arguments. Thus, before you can read or write to a file, you must use the  $\square ASS$  function to associate the file and its related information with a channel number using the following form:

`$\square variable \leftarrow \square ASS' \square [channel] \square filespec \square [/fileorganization] \square [/qualifiers]$`

## VAX APL Input and Output

### 5.2 File Input and Output

#### *variable*

is an optional variable

#### *channel*

is an optional integer scalar whose absolute value represents a channel number in the range 1 through 999. If you do not specify a channel number, APL assigns one for you. APL picks the first available channel number, beginning at 12 and counting down to 1; then APL begins at 13 and counts up to 999.

#### *filespec*

is the VMS file specification associated with the specified channel. If you do not include the file extension, APL uses the default file extension for the file organization qualifier specified. (See Table 5–2.)

#### */fileorganization*

is the qualifier identifying the file organization of the file specified by *filespec*. The possible values of the */fileorganization* qualifier are listed in Table 5–2. The default value is */DA*.

#### */qualifiers*

are the other optional qualifiers listed in the *VAX APL Reference Manual*. These qualifiers can specify the blocksize, buffercount, whether the file can be shared or updated.

**Table 5–2 File Organization Qualifiers**

<i>/fileorganization Qualifier</i>	<i>Default File Extension</i>	<i>Type of File</i>
<i>/AS</i>	<i>.AAS</i>	ASCII sequential; can open for either read or write, or both (when you specify <i>/UPDATE</i> ).
<i>/AS*</i>	<i>.AAS</i>	ASCII sequential; file is positioned at end-of-file to allow appending.
<i>/IS</i>	<i>.AIS</i>	Internal sequential; can open for either read or write, or both (when you specify <i>/UPDATE</i> ).
<i>/IS*</i>	<i>.AIS</i>	Internal sequential; file is positioned at end-of-file to allow appending.

(continued on next page)

**Table 5–2 (Cont.) File Organization Qualifiers**

/fileorganization Qualifier	Default File Extension	Type of File
/DA	.AIX	Direct-access; can do both read and write (this is the default).
/RF	.ARF	Relative; can do both read and write.
/KY	.AKY	Keyed; can do both read and write.

For example:

```

6      ASSIGN THE ASCII SEQUENTIAL FILE TO CHANNEL 6
      DASS '6 FOO/AS'
6      APPEND RECORDS TO TEST.AAS
      DASS '5 TEST.AAS/AS*/OPEN:'
5      APL ASSIGNS THE DIRECT-ACCESS FILE TO CHANNEL 12
      A+DASS 'PROJECTIONS/OPEN: OLD'
      A
12

```

Associating a file with a channel does not open or create the file (unless you specify the /OPEN qualifier on DASS); it merely establishes a connection between a channel number and a file specification. Using the /OPEN qualifier allows you to detect errors related to the opening or creating of a file at the time of assignment instead of at the time of the first I/O operation. This is particularly important when you share files with others, and their current assignments to a file invalidate the assignment you are making.

Some of the related file information you can specify with DASS, like the file's type of organization, is fixed when the file is created; other file information, like the file's shareability, can be changed each time you use DASS. Refer to the *VAX APL Reference Manual* for more information about the optional qualifiers.

The argument to DASS must be specified according to the following rules:

- If you use the /fileorganization qualifier, it must immediately follow the file specification. The other parameters may follow the file specification and organization in any order.
- You may not repeat a qualifier, even with a different value.

## VAX APL Input and Output

### 5.2 File Input and Output

- If you use a qualifier that takes an argument, APL looks for a qualifier delimiter (: or =). If it does not find a delimiter, APL assigns the default value to the qualifier. If APL does find a delimiter, the argument value must follow.
- White space (spaces and tabs) is permitted between any two qualifiers.

When you assign a keyed file to a channel, use the `□ASS` system function with the `/KY` switch. When you are creating the file, you must include a value for at least the primary key of reference (other key of reference values are optional). When the file already exists, the key specifications are optional; if you choose to specify them, APL assumes your specifications are consistent with the file's key structure and allows you to successfully assign the channel. However, if there is an inconsistency, APL signals `IO ERROR (FILE KEY STRUCTURE DOES NOT AGREE WITH USER ASSIGNMENT)` when you attempt any I/O operations.

Each key specification identifies the location of the first byte of the key, the length, in bytes, and the data type of the key, either `INW`, `INL`, `INQ` or `CHARACTER`.

The following expression shows `□ASS` with the `/KY` qualifier. In this example, there are two key specifications: the primary key begins at the first byte of the record, has a length of 10 bytes, and is of type character; the first alternate key begins at the 12th byte of the record, has a length of 4 bytes, and is of type character.

```
□ASS '1 BANGKOK/KY:1:10:CHAR,12:4:CHAR'  
1
```

Like other APL functions, `□ASS` returns a result. The value of the result depends on what channel you assigned and whether that assignment was successful:

- If you specify a channel number *channel*, `□ASS` returns that number as the result.
- If you do not specify a channel number, APL assigns one, and `□ASS` returns that as the result.
- If you assign a channel number that has already been assigned, APL deassigns the channel from its original file and reassigns it to the new file. `□ASS` returns the channel number of the new file as the result.
- If APL encounters an error in the `□ASS` function, it returns a result of 0, indicating that your assignment failed. This is also the case if the argument you specify is empty, and if you do not specify a channel when no unassigned channels are available. After a failed assignment, `□ERROR` contains an error message that describes the reason for failure.

Note that if you are superseding an assigned channel number, and the new assignment fails because of the */OPEN* qualifier, the specified channel number becomes deassigned. For example:

```

    □ASS 'GOMJABBER/DA'
12
    □ASS '12 PLANEX/DA/OPEN=OLD'
0
    □ERROR
33 IO ERROR (FILE NOT FOUND)
    □ASS '12 PLANEX/DA/OPEN=OLD'
           ^
    □ASS 12          (APL outputs a blank line)
    □CHANS          (APL outputs a blank line)

```

Table 5-2 describes the possible values for the */fileorganization* qualifier, the default file extensions implied by the values, and the meaning of the various file organizations. (Section 5.2 has more information on file types and organization.)

### 5.2.2.1 Querying File Assignments

The query form of `□ASS` returns the current value of assignments made previously with the action form. The result of `□ASS` identifies the parameters you associated with the channels specified. For example, the following line displays the assignment made for channel 1:

```

    □ASS '1 PLANS.AAS/AS*/PROT=(S:RWED,O:RWED)'
1
    □ASS 1
1  PLANS.AAS/AS*/PROTECTION:(S:RWED,O:RWED,G:,W:)

```

Note that when the result is a matrix, the shape of the matrix is *n* by *L*, where *n* is the number of channels, and *L* is the length of the longest line in the display.

If the argument is a singleton and the channel you specify is currently unassigned, APL returns a character vector of length 4 with the channel number left-justified with trailing blanks. If the argument is a vector, elements representing an unassigned channel are identified by the number and enough blanks to make the line the appropriate length. If the argument is `10` APL returns a character matrix with the shape `0 0`. In the following example, channels 2 and 4 are not assigned:

## VAX APL Input and Output

### 5.2 File Input and Output

```
1  □ASS '1 PLAN.AAS/AS*/PROT=(S:RWED,O:RWED)'
2
3  □ASS '3 APLREL/RF/DISPOSE:DELETE'
4
5  □ASS '5 APLSEQ/AS/SHARE/READONLY'
6
7  □ASS 2
8
9  □ASS 15
10 PLAN.AAS/AS*/PROTECTION:(S:RWED,O:RWED,G:,W:)
11
12 APLREL/RF/DISPOSE:DELETE
13
14 APLSEQ/AS/SHARE/READONLY
```

#### 5.2.2.2 Returning Channel Numbers

□CHANS displays all of the channel numbers currently associated with file specifications. The result is a vector. In the following example, channels 1, 3, and 5 are each associated with a file:

```
1  □ASS '1 PLAN/AS'
2
3  □ASS '3 BUDGET/AS'
4
5  □ASS '5 ANALYSIS/AS'
6
7  □CHANS
8  1 3 5
9  )CLEAR
CLEAR WS
10 □CHANS
(APL outputs a blank line)
```

To list the parameters associated with all files assigned to channels, use □CHANS as the argument:

```
1  □ASS '1 PLAN/AS'
2
3  □ASS '3 BUDGET/AS'
4
5  □ASS '5 ANALYSIS/AS'
6
7  □ASS □CHANS
1  PLAN/AS
3  BUDGET/AS
5  ANALYSIS/AS
```

If no channels are assigned, □CHANS returns an empty numeric vector.

### 5.2.3 Opening Files and Reading and Writing Records

You use the APL input quad (`⊣`) and output quad (`⊣`) functions to read and write records. They are similar to the terminal I/O system variables (see Section 5.1), except that the input and output is to and from external files rather than to and from your terminal.

If the first reference to a file (by means of a channel) is an output function, APL opens the file—or creates and opens it if it does not exist—and then writes the record. If the first reference is an input function, APL opens the file and reads the record, or returns an error if the file does not exist.

#### 5.2.3.1 Writing and Reading ASCII Sequential Files

You can write or read ASCII sequential file records in any of four character sets: KEY, BIT, TTY, or APL COMPOSITE. The mode parameter indicates the following:

- The character set you want to use to read the file
- Whether to use evaluated (`⊣`), quote quad (`⊣`), or quad del (`⊣`) input to read the next record

The APL character set refers to the character set you specified as your terminal designator (see Section 1.5). The default character set is the one specified by your terminal designator; the default output type is `⊣` mode. Be sure to enclose the mode specification in brackets. Table 5–3 shows the possible values for the mode parameter and the meanings associated with each value.

**Table 5–3 /AS Input and Output Modes**

Type	Character Set	Mode
<code>⊣</code>	TTY	1
<code>⊣</code>	TTY	2
<code>⊣</code>	TTY	3
<code>⊣</code>	APL	4
<code>⊣</code>	APL	5
<code>⊣</code>	APL	6
<code>⊣</code>	KEY	7
<code>⊣</code>	KEY	8
<code>⊣</code>	KEY	9

(continued on next page)

## VAX APL Input and Output

### 5.2 File Input and Output

**Table 5-3 (Cont.) /AS Input and Output Modes**

Type	Character Set	Mode
□	BIT	10
□	BIT	11
□	BIT	12
□	COMPOSITE	13
□	COMPOSITE	14
□	COMPOSITE	15

In the following example, the ASCII file *OUTPUT.AAS* is created, and records are written using the key-paired character set. Then the file is closed and the newly created file is read.

```
□ASS '2 OUTPUT/AS'
2
'FIRST RECORD' □ [7]2
'SECOND RECORD' □ [7]2
(2 4p18) □ [7]2
□CLS 2
□ [8]2
FIRST RECORD
□ [8]2
SECOND RECORD
□ [7]2
1 2 3 4
□ [7]2
5 6 7 8
A←□ [7]2
p A
0 75      (EOF encountered)
```

When you use file input (□) with mode 1, 4, 7, 10, or 13, records are processed until APL gets a value to return as the result. Blank lines and comments in the file are ignored, system commands are executed (and their output, if any, is displayed on the terminal), and function editing sequences continue until the operation is closed. Thus, these modes allow you to write files that are fully documented with comments and that can define operations to be used later in expressions within the file. For example:

```

1      DASS '1 TEST/AS'
      (1+1) ⊂ 1      ⋄EXPRESSION GETS EVALUATED BEFORE WRITING
      'V F' ⊂ 1      ⋄NEXT 8 LINES WILL BE PROCESSED BY 1 READ COMMAND
      'A+1' ⊂ 1      ⋄FUNCTION DEFINITION
      'B+2' ⊂ 1
      'V' ⊂ 1      ⋄END FUNCTION DEFINITION
      ')WIDTH' ⊂ 1  ⋄SYSTEM COMMAND
      'F' ⊂ 1      ⋄CALL F, NO RESULT WHEN EXECUTED
      (2+2) ⊂ 1      ⋄FINALLY A VALUE, READ WILL STOP HERE
      DCLS 1        ⋄CLOSE CHANNEL ASSIGNMENT
                  ⋄SHOW NO FUNCTIONS IN WORKSPACE
      )FNS
                  ⋄SHOW NO VARIABLES IN WORKSPACE
      )VARS
      ⊂ 1          ⋄READ FIRST RECORD (1+1)
2
      ⊂ 1          ⋄READ UNTIL NEXT VALUE-RETURNING RECORD (2+2)
WAS 80
50
4
      ⋄SHOW THAT F WAS DEFINED DURING LAST READ
      )FNS
F
      ⋄SHOW THAT F WAS EXECUTED TOO
      )VARS
A      B

```

Note that if the abort signal is read with file input (⊖) in mode 1, 4, 7, 10, or 13, APL cancels the input and signals *INPUT ABORTED*. If ⊖ is executing in one of these modes, and the end of the file is reached while the v editor is being executed, the effect is the same as if ⊖ read the abort signal: the editing is aborted and the value returned by the ⊖ function is 0 75p 0.

### 5.2.3.2 Writing and Reading an Internal Sequential File

When you use ⊕ and ⊖ (see Section 5.2.3) with internal sequential files, you do not have to specify an input mode as you do with ASCII sequential files. Data in internal sequential files is stored in the internal format of APL.

In the following example, three records are written to an internal sequential file and the file is closed; then, the three records are read:

```

1      DASS '1 INT/IS'
      'RECORD 1' ⊂ 1
      'RECORD 2' ⊂ 1
      (2 4p18) ⊂ 1
      DCLS 1

```

## VAX APL Input and Output

### 5.2 File Input and Output

```
      ⊗1  
RECORD 1  
      ⊗1  
RECORD 2  
      ⊗1  
1 2 3 4  
5 6 7 8  
A+⊗1  
A  
          (APL outputs a blank line)  
pA  
0 75      (EOF encountered)
```

As with ASCII sequential files, the end-of-file indicator indicates that you are at the end of the file. Blank records in internal sequential files return blanks.

#### 5.2.3.3 Writing and Reading a Direct-Access or Relative File

The components or records in direct-access or relative files are associated with indexes called component numbers (for direct-access files) or record numbers (for relative files).

To randomly read or write a direct-access or relative file, you specify, in the argument to the file input (`⊣`) or file output (`⊣`) function, an index representing a component or record number in the following form.

`data ⊣ [[record-number]] channel`

When you sequentially access a direct-access file, that is, when you use the `⊣` or `⊣` function and do not specify a component or record number, APL retrieves records as follows:

- For input, APL retrieves the record referenced by the value of the next-record pointer (see Section 5.2.1.2).
- For output, APL writes the record referenced by `⊣FLS[2]+1`, that is, one plus the component or record number used in the previous I/O operation.

If you execute a file output (`⊣`) function that accesses an existing record or component, APL replaces the old value of the record or component with the new value you specified.

There is no end-of-file in direct-access or relative files; however, they may have empty components or records scattered throughout the file. If you try to read a component or record that is empty, APL returns the end-of-file indicator.

You can delete records from direct-access and relative files by using the monadic form of file output (`⊣`).

The following example shows records 20 and 21, and 50 and 51 being written to the file *REL.AR*: Note that when no record number was specified for output, APL used the record number used in the previous I/O operation plus one.

```

DASS '1 REL/RF'
1
'RECORD 20' ⊗ [20]1
'RECORD 21' ⊗ 1
'RECORD 50' ⊗ [50]1
(14) ⊗1
⊗ [20]1
RECORD 20
⊗ [51]1
1 2 3 4
'NEW VALUE FOR 20' ⊗ [20]1
⊗ [20]1
NEW VALUE FOR 20

```

You can access components or records in any order—for instance, you can write component or record 10 before 9—but it is more efficient to access them in ascending order of their component or record numbers. It is also more efficient, when updating a file, to make deletions first and then make replacements and additions.

#### 5.2.3.4 Writing and Reading a Keyed File

When you write records to a keyed file, use the file output (`⊗`) function in the following form:

`data ⊗ channel [[data-type]]`

If you are writing APL objects and plan to read them back into the APL environment, do not specify *data-type*, or specify a value of 0; APL will add header information to the beginning of the record. When you do specify the data type, you imply that you want to write the record in a pure data mode and that you do not want APL to add descriptive information to the beginning of each record. For more information on pure data types, see Section 5.3.4.

In the following example, one record is written to a keyed file.

```

DASS '1 BANKKOK/KY:1:11:CHAR,13:4:CHAR'
1
'TRANFERRED EEUU
505617374 $1050 TO 273924509
EFFECTIVE JULY 8, 1990' ⊗ 1 5

```

To read records randomly from a /KY file, use the `⊗` function in the following form:

`⊗ [[value [[;key-num [[;tech[[;key-type]]]]]] chan [[data-type]]]`

## VAX APL Input and Output

### 5.2 File Input and Output

#### ***value***

Specifies the key value for the record you want to read. It can be in either the near-integer singleton or the character vector domain. With a character vector key, you can specify a key value that is shorter than the field length of the key of reference. In this case, APL interprets the value as a prefix and searches for any keys (within the specified key of reference) that begin with the prefix value. If the defined length of the key is known to the APL environment, *value* is padded to that length with trailing NUL bytes (hex 00). Note that if *value* belongs to the character domain, and the file you want to process is not in APL character set (it was written out in pure data type 6, 11, 12, 13, 14, or 15), you must specify that external data type number (see Table 5–6 in Section 5.3.4) for *data-type* or *key-type* in order to convert the key to the appropriate character set.

#### ***key-num***

Is a near-integer singleton that specifies which key of reference you want to read. Use 0 for the primary key, 1 for the first alternate key, and so on. The default is the primary key.

#### ***tech***

Specifies the search technique that APL uses to retrieve the record you want to read. It belongs to the character vector domain and has three possible values: '*EQL*', '*GTR*', and '*GEQ*'. '*EQL*' is the default value. It means that APL searches for the record with a key that matches exactly the key value that you specify. '*GTR*' means that APL searches for the first record with a key that is greater than the key value that you specify. '*GEQ*' means that APL searches for the first record with a key that is either greater than or equal to the key value that you specify.

#### ***key-type***

Specifies the external data type of the key of the record you want to read. Do not specify this parameter unless you are reading “pure” data. (See Section 5.3.4 for more information on pure data records.) It is not necessary to specify *key-type* when the data type of your key is the same as the rest of the data in the record. The possible values are 0, 1, 5, 6, 11, 12, 13, 14, or 15.

#### ***chan***

Specifies the channel number currently assigned to the /*KY* file.

## VAX APL Input and Output

### 5.2 File Input and Output

#### *data-type*

Specifies the data type of the record you want to read. Do not specify this parameter unless you are reading “pure” data. When you include a value for *data-type*, you imply that the record contains pure data; that is, the beginning of the record contains no header information. If you do not specify *data-type*, or if you specify a value of 0, APL assumes that there is a header at the beginning of the record (see Section 5.3.4). The possible values are 0, 1, 5, 6, 11, 12, 13, 14, or 15.

The following example opens a keyed file, writes a record to the file and then performs a random read of that record by specifying a prefix of the key value of the primary key of reference:

```
QASS '1 BANKGOK/KY:1:11:CHAR,13:4:CHAR'
1
    'TRANSFERRED EEUU
505617374 $1050 TO 273924509
EFFECTIVE JULY 8, 1990' ⌂ 1 5
    ⌂ ['TRAN';0] 1 5
TRANSFERRED EEUU
505617374 $1050 TO 273924509
EFFECTIVE JULY 8, 1990
```

To read records sequentially from a /KY file, use the file input (`⍳`) function in the following form:

`⍳ channel [data-type]`

Keyed files use the next-record pointer mechanism described in Section 5.2.1.2. When you perform a sequential read operation on a /KY file, APL finds the next record in the key of reference specified in the previous I/O operation.

When you create a keyed file with APL, the primary key does not allow duplicate values and alternate keys do allow duplicate values. Duplicate values refers to two records with the same key value in the same key of reference. When you write a record that has a duplicate key to a file that does not allow duplicates, the record you are writing replaces the existing record. For example:

```
QASS '1 MOVASI/KY:1:1:CHAR'
1
    'AEMPIRE' ⌂ 1 5           ⌈PRIMARY KEY INDEX GETS A
    1↓X ← ⌂ ['A']1 5
EMPIRE
    'AFoundation' ⌂ 1 5      ⌈REPLACE EMPIRE WITH FOUNDATION
    1↓X ← ⌂ ['A']1 5
FOUNDATION
```

## VAX APL Input and Output

### 5.2 File Input and Output

When you write a record that has a duplicate key to a file that does allow duplicates, the corresponding index is updated with the duplicate entry. If you randomly specify the duplicate key, APL retrieves the first occurrence of the duplicate. Subsequent sequential reads will retrieve duplicate key values.

The only way to randomly read records that have duplicate keys is by using a different key of reference.

```
□ASS '1 HOGAN/KY:1:2:CHAR,3:5:CHAR'
1
  □PRIMARY KEY INDEX GETS AA, ALTERNATE KEY GETS LUNAR
  'AALUNARVISAR' □ 1 5
  □PRIMARY KEY INDEX GETS ZZ, ALTERNATE KEY GETS
  □ DUPLICATE LUNAR
  'ZZLUNARANYMEDE' □ 1 5
  □RETRIEVE FIRST DUPLICATE-KEY RECORD
  □+X+□ ['LUNAR';1] 1 5
AALUNARVISAR
  □SEQUENTIAL READ RETRIEVES SECOND
  □ DUPLICATE-KEY RECORD
  □+X+□ 1 5
ZZLUNARANYMEDE
  □DEFAULT IS PRIMARY KEY AND EQL TECHNIQUE
  7+X + □ ['ZZ'] 1 5
ANYMEDE
```

If you want a keyed file to have a structure different from those that APL provides, you can create the file with a VMS File Definition Language (FDL) Utility. For example, with the FDL\$CREATE command you can create a key structure that permits duplicate primary keys. Once you create the structure, you can then use APL to write and read records to the existing file. For more information, see the *VMS File Definition Language Facility Manual*.

To delete records from a /KY file, use the file output (□) function in the same form used to read random records:

```
□ [value [;key-num [;tech[;key-type]]]] chan [data-type]
```

#### 5.2.4 Resetting Next-Record Pointer to Start of File

□REWIND allows you to reposition the next record pointer to the first record of a file without closing the file. When you want to return to the beginning of a sequential file you could also use □CLS. (See Section 5.2.5.)

With the monadic form of □REWIND, you can specify a vector of channel numbers in the right argument. This will rewind each of the files associated with the specified channel numbers. If any of the files have a keyed organization, APL performs the rewind on the primary key of reference.

Use the dyadic form for keyed files when you want APL to perform the rewind on a key of reference other than the primary key. The right argument specifies the channel number associated with the keyed file. The left argument specifies the key of reference: 0 indicates the primary key, a 1 indicates the secondary key, and so on. You can specify only one file at a time when you invoke dyadic `□REWIND`.

In the following example, APL rewinds each of the files associated with the channel numbers 10, 9, 8, and 7. Any subsequent sequential read operation on one of these files will select the first record. (If any of the files has a keyed organization, a read operation selects the first record by the primary key of reference.)

```
B← 10 9 8 7    ⎕AB GETS VECTOR OF CHANNEL NUMBERS
    □REWIND B      ⎕AMONADIC FORM
```

The next example shows the dyadic form of `□REWIND`:

```
    ⎕A10 = CHANNEL NUMBER OF A KEYED FILE
3 □REWIND 10 ⎕A3 IS FOURTH KEY OF REFERENCE
```

If an `/AS` or `/IS` file is opened for read operations when you invoke `□REWIND`, it will remain open for read operations. If the file is opened for write operations initially, it will be open for both read and write operations afterward. Because write operations can occur only at the end of a sequential file, you must read through to the end before attempting a write operation. Otherwise, APL signals `IO ERROR ($PUT NOT AT END OF FILE)`.

`□REWIND` does not release locked records on the specified channels.

For sequential files, the second value returned by the `□FLS` system function indicates the number of read and write operations that have taken place since the file was opened or since `□REWIND` was last executed on the file.

### 5.2.5 Closing Files and Disassociating Files from Channels

When you have finished processing a file's records, you can use the `□DAS` function to close the file and end its association with the channel number. If you want to close the file but keep its association with the channel, you can use the `□CLS` function. `□CLS` is useful when you want to return to the beginning of a sequential file. (You can also use the `□REWIND` system function, which does not close files.)

APL automatically closes and deassigns all open files when you type Ctrl/Z or execute a `)LOAD`, `)CLEAR`, `)OFF`, or `)CONTINUE` system command (the `)MON` and `)PUSH` system commands do not have this effect).

## VAX APL Input and Output

### 5.2 File Input and Output

If you access a file after you close a channel, a read function would open the file and read the first record, and a write function would create a new version of the file (except for direct-access, relative and keyed files for which a new file is created only if no version currently exists).

`□CLS` is a quiet function; it does not return a result if it is the leftmost function in a statement. When `□CLS` is not the leftmost function, it returns an empty numeric vector.

Any unassigned channels in the argument are ignored.

The following example closes files:

```
    □ASS □CHANS
1   BANGKOK/KY=1:11:CH,13:4:CH
9   MINTMP/AS
10  NIN/AS
12  TEST/AS
      □CLS 1
      □ASS □CHANS
1   BANGKOK/KY=1:11:CH,13:4:CH
9   MINTMP/AS
10  NIN/AS
12  TEST/AS
      X←□CLS 12
      X
          (APL outputs a blank line)
```

`□DAS` disassociates file specifications from channel numbers. If any files associated with the specified channel numbers have not been closed (by `□CLS`), `□DAS` closes them and then deassigns them.

In general, `□DAS` reverses the actions performed by the `□ASS` system function.

Any unassigned channels in the argument are ignored.

`□DAS` is a quiet function; it does not return a result if it is the leftmost function in a statement. When `□DAS` is not the leftmost function, it returns an empty numeric vector.

The following example deassigns the files associated with channels 1, 3, and 5:

```
    □ASS □CHANS
1   BANGKOK/KY=1:11:CH,13:4:CH
9   MINTMP/AS
10  NIN/AS
12  TEST/AS
      □DAS 1 9 10 12
      □ASS □CHANS
          (APL outputs a blank line)
```

## 5.2.6 Determining Information about Files and Devices

It would be helpful to know something about how a file was written before you read it. The  $\square CHS$  and  $\square FLS$  functions return information about the file.  $\square DVC$  displays the characteristics of where files are stored.

### 5.2.6.1 Returning File Organization and Open Status

Use  $\square CHS$  to determine the file organization and status of the file associated with the channel you specify. If the argument is a singleton,  $\square CHS$  returns a 2-element vector: the first element identifies the file's organization, and the second element identifies the file's open status. If the argument is a vector of  $n$  elements, the result is an array of shape  $n$  by 2.

In the example, the file associated with channel 1 is an ASCII sequential file and is open for input. The second expression returns a 3-by-2 array:

```
□CHS 1
1 3
□←FILS+□CHS 1 3
1 3
7 4
2 2
          □FILS
3 2
```

Table 5-4 gives the meanings of the possible values.

**Table 5-4 Possible  $\square CHS$  Codes**

Code	File Organization	First Element
0	Not applicable	
1	/AS	
2	/IS	
3	Not applicable	
4	/DA	
5	Not applicable	

(continued on next page)

## VAX APL Input and Output

### 5.2 File Input and Output

Table 5-4 (Cont.) Possible  $\square CHS$  Codes

First Element	
Code	File Organization
6	Not applicable
7	/RF
8	/KY

Second Element	
Code	Open Status
0	Channel free
1	Assigned but not open
2	Open for output
3	Open for input
4	Open for input and output

#### 5.2.6.2 Returning File Information

Use  $\square FLS$  to determine information about files. The result contains one row of five values for each channel specified in the argument. The meanings of the values can differ according to each file's organization.

The first value is a 1 if you specified /SHARE in the argument for the associated  $\square ASS$  function; 0 means that you did not.

For sequential files, the second value is the number of records read and written since the file was opened or since  $\square REWIND$  was last executed on the file. For direct-access and relative files, it is the value of the last record or component number used for a successful read or write operation. For keyed files, it is the value of the last key of reference used for a successful read, write, or rewind.

The third value indicates the maximum record length of the file. 0 means there is no user limit on record size.

The fourth value indicates the /BLOCKSIZE setting for the file.

The type of the most recent I/O operation is indicated by the fifth value. You can use this information in determining the location of the next record pointer. There are six possible I/O operations:

Value Returned	I/O Operation
0	None
1	Sequential read
2	Random read
3	Sequential write
4	Random write
5	Sequential delete
6	Random delete

`□FLS` returns a 5-element vector if a single channel number is specified. If the argument specifies more than one channel, the result is an array of shape  $n$  by 5, where  $n$  is the length of the argument.

If its argument is empty, `□FLS` returns a result of `0 5p 0`. If any of the integers in the argument refers to an unassigned channel, `□FLS` returns a row of 5 zeros. For example:

```
□FLS 1
0 1 2044 512 1
```

Note that to return a value for `□FLS`, APL must open files that have been associated with channels but have not yet been opened. (For a list of commands that open files, see Section 5.2.3.) Thus, unopened files associated with channels identified by positive integers in the `□FLS` argument are opened for input; unopened files associated with channels identified in the argument by negative integers are opened for output. Note that when you open a sequential file for output, APL makes a new copy of the file with a version number that is one higher than that of the previous copy.

### 5.2.6.3 Returning Device Characteristics

`□DVC` displays the characteristics of the devices where files are stored.

For each channel specified in the argument, `□DVC` returns one row containing two values: the first value is the VMS device-characteristics longword, and the second value is always 0. For unassigned channels, `□DVC` returns `0 0`.

`□DVC` returns a 2-element vector if a single channel is specified. If more than one channel is specified, the result is a matrix of shape  $n$  by 2, where  $n$  is the length of the argument.

If its argument is empty, `□DVC` returns a result of `0 2p 0`.

## VAX APL Input and Output

### 5.2 File Input and Output

Note that to return a value for  $\square DVC$ , APL must open files that have been associated with channels but have not yet been opened. (For a list of commands that open files, see Section 5.2.3.) Thus, unopened files associated with channels identified by positive integers in the  $\square DVC$  argument are opened for input; unopened files associated with channels identified in the argument by negative integers are opened for output. Note that when you open a sequential file for output, APL makes a new copy of the file with a version number that is one higher than that of the previous copy.

It is usually helpful to convert the device-characteristics longword to binary format before examining it. For example:

```
 $\square PW \leftarrow 50$ 
A  $\leftarrow \square DVC\ 1$ 
A
474824968 0
(32p2)T A[1]
0 0 0 1 1 1 0 0 0 1 0 0 1 1 0 1 0 1 0 1 0 0 0 0 0 1 0
0 0 0 1 0 0 0
```

You can compare the binary value of the longword with the device characteristics in Table 5–5. The first element in the table is associated with the rightmost bit in the longword, the second element is associated with the next rightmost bit, and so forth. Thus, in the previous example, the three rightmost 0's indicate that the device is not record-oriented, is not a carriage-control device, and is not a terminal; the 1 in the fourth position from the right indicates that the device is directory-structured.

**Table 5–5 Device Characteristics Longword**

Bit	Type or Condition of Device
0	Record-oriented
1	Carriage-control
2	Terminal
3	Directory-structured
4	Single directory-structured
5	Sequential, block-oriented
6	Being spooled
7	Open console

(continued on next page)

**Table 5–5 (Cont.) Device Characteristics Longword**

Bit	Type or Condition of Device
8	RA50,RA81,RA82,RH60
9 – 12	(Bits reserved)
13	Network
14	File-oriented
15	(Bit reserved)
16	Shareable
17	Generic
18	Available for use
19	Mounted
20	Mailbox
21	Marked for dismount
22	Error logging enabled
23	Allocated
24	Non-file-structured
25	Software write-locked
26	Capable of providing input
27	Capable of providing output
28	Allows random access
29	Real-time
30	Read-checking enabled
31	Write-checking enabled

### 5.3 Advanced I/O Techniques

The preceding sections explain how to use the file system for typical file-processing applications; however, you may have some applications that are not typical. For those, APL provides facilities such as shared files, event flags, mailboxes, and untranslated data records. These advanced I/O facilities are explained in the following sections.

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

#### 5.3.1 Sharing Files

The APL file system provides a way to make files shareable, that is, to make files available for access by more than one user simultaneously.

##### 5.3.1.1 Sharing Sequential Files

ASCII sequential and internal sequential files are shareable unless the first channel assigned to a file specifies the */NOWRITERS* switch. You do not need to do anything special to share sequential files (the */SHARE* switch is not necessary when you assign the file); APL provides this facility automatically. In all cases, there can be only one writer to a shared sequential file, though there can be more than one reader.

There are four optional switches (on *ASS*) that determine the share characteristics of a file: */NOWRITERS* (no other channels can write to the file), */READONLY* (the assigned channel can only read the file), */UPDATE* (the assigned channel can read and append to the file), and */WRITEONLY* (the assigned channel can only write to the file).

When no switches are specified, the rules for sharing are as follows:

- If the first operation is a write, a new file is created and no other channels can share the file.
- If the first operation is a read, any other channel (that did not specify a switch) can also read the file, but none of these channels can perform writes.

In all situations, any channel can specify *REWIND*, move to the beginning of the file, and then continue reading.

When a channel assigns to a sequential file and uses */UPDATE*, the channel can read and write to the file. The channel can use *REWIND* to move to the beginning of the file, read through the file, and then append new records. Previous assignments can continue reading from the file, and subsequent assignments can also gain read access. If the first operation of the */UPDATE* channel is a write, a new file is created. (If the assignment specified */OPEN:OLD*, a new file is not created. However, the channel can only write to an existing file if the file is empty or if */IS\** was specified for appending.) No other channels can assign to this file with */UPDATE*.

When the first assignment uses */WRITEONLY*, a new file is created when the channel writes to the file. (If the assignment specified */OPEN:OLD*, a new file is not created. However, the channel can only write to an existing file if the file is empty or if */IS\** was specified for appending.) Subsequent assignments can gain read access to the file.

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

When the first assignment uses */READONLY*, other channels can also read the file and one other channel has the opportunity to gain write access (with */UPDATE*).

The following list describes the rules for sharing sequential files. In each case, the list assumes that Chan1 performs the first operation.

Qualifiers	First Operation	Chan1	Other Users
none	read	read	read
none	write	read, write	none
<i>/READONLY</i>	n/a	read	read, 1 writer
<i>/WRITEONLY</i>	n/a	write	read
<i>/UPDATE</i>	n/a	read, write	read

In the following example, the same file is associated with two different channels. Note that when the file is opened for output on channel 1, it cannot be read from channel 2. However, after the file is closed on channel 1, it is opened successfully for input on both channels.

```
CLASS '1 FOO/AS'
1
CLASS '2 FOO/AS'
2
56 ⊗ 1
57 ⊗ 1
58 ⊗ 1
          FILE IS LOCKED BY CHAN1
⊗ 2
32 INVALID SIMULTANEOUS ACCESS (FILE CURRENTLY LOCKED BY ANOTHER
⊗ 2
^
⊗ CLS 1      CHAN1 UNLOCKS FILE
⊗ 2      CHAN2 PERFORMS READ, DOES NOT LOCK FILE
56
⊗ 2
57
⊗ 1      CHAN1 CAN SHARE FILE
56
⊗ 1
57
```

Note that it is possible for two (or more) users to appear to have the same sequential file open for output at the same time.

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

In the following example, the users are creating two new files: each user makes a new copy of *MYFILE.AAS*, and the copy made on channel 2 has a version number that is one higher than the copy made on channel 1.

Note that if the user on channel 1 closes the file and then tries to reopen it with an  $\square$  function, APL signals an error because it tries to open the latest version of *MYFILE.AAS* and that version is locked (still open) by the user on channel 2.

Once the user on channel 2 closes the file, it is no longer locked. When both users reopen the file for input, both get the latest version of the file—the version created on channel 2.

```
          A TWO CHANNELS ASSIGN, FILES ARE NOT OPENED YET
           $\square$ ASS '1 MYFILE/AS'
1           $\square$ ASS '2 MYFILE/AS'
2          1  $\square$ 1      A CHAN1 CREATES NEW VERSION OF MYFILE.AAS
          10  $\square$ 2      A CHAN2 CREATES NEW VERSION OF MYFILE.AAS
           $\square$ CLS 1
           $\square$  1
32 INVALID SIMULTANEOUS ACCESS (FILE CURRENTLY LOCKED BY ANOTHER USER
           $\square$  1
          ^
           $\square$ CLS 2
           $\square$  1
10           $\square$  2
10
```

You may encounter locked records as you perform read operations. APL waits indefinitely for locked records (unless you set  $\square$ WAIT), and you must use the attention signal to regain control. Note that using  $\square$ RELEASE to unlock locked records is one way of avoiding delays while waiting for records. The following example describes such a situation:

```
 $\square$ ASS '1 REX/IS/OPEN:NEW'          A ASSIGN FILE REX
1          A POPULATE FILE AND DEASSIGN
          1  $\diamond$  455 $\square$  1  $\diamond$  355 $\square$  1  $\diamond$   $\square$ DAS 1
1          A OPEN REX FOR READ AND WRITE OPERATIONS
           $\square$ ASS '1 REX/IS/OPEN:OLD/UPDATE/SIGNAL'
1           $\square$  1      A READ AND LOCK RECORD 1
455
```

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

```
    @OPEN REX FOR READ ON CHAN 2
    @ASS '2 REX/IS/OPEN:OLD/SIGNAL'

2      @ 2      @ATTEMPT READ OF LOCKED RECORD
18 ATTENTION SIGNALLED (WAITING FOR RECORD LOCK)
    @ 2      @ATTEMPT READ OF LOCKED RECORD
    ^
2 @WAIT 2      @SET WAIT FOR 2 SECONDS
    @ 2      @ATTEMPT READ AGAIN
33 IO ERROR (TIMEOUT PERIOD EXPIRED)
    @ 2      @ATTEMPT READ AGAIN
    ^
@RELEASE 1      @RELEASE RECORD LOCKED BY CHAN 1
    @ 2      @NOW RECORD IS AVAILABLE TO CHAN 2
455
```

Note that it is possible to use combinations of `@ASS` switches that are contradictory. The contradiction might cause the first I/O operation to fail. For example:

```
    @ASS '1 REX/IS/OPEN:NEW'          @ASSIGN FILE REX
1      @POPULATE FILE AND DEASSIGN
    1 ◊ 455 @ 1 ◊ 355 @ 1 ◊ @DAS 1
1      @OPEN REX FOR READ AND WRITE OPERATIONS
    @ASS '1 REX/IS/OPEN:OLD/UPDATE/SIGNAL'
1      @ 1      @READ AND LOCK RECORD 1
455      @OPEN REX FOR READ ON CHAN 2
    @ASS '2 REX/IS/OPEN:OLD/SIGNAL'

2      @ 2      @ATTEMPT READ OF LOCKED RECORD
18 ATTENTION SIGNALLED (WAITING FOR RECORD LOCK)
    @ 2      @ATTEMPT READ OF LOCKED RECORD
    ^
2 @WAIT 2      @SET WAIT FOR 2 SECONDS
    @ 2      @ATTEMPT READ AGAIN
33 IO ERROR (TIMEOUT PERIOD EXPIRED)
    @ 2      @ATTEMPT READ AGAIN
    ^
@RELEASE 1      @RELEASE RECORD LOCKED BY CHAN 1
    @ 2      @NOW RECORD IS AVAILABLE TO CHAN 2
455
```

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

#### 5.3.1.2 Sharing Direct-Access, Relative, and Keyed Files

APL does not provide any automatic sharing for direct-access and relative files. They are not shareable for input or output unless you specifically make them shareable by using the `□ASS` switch with the `/SHARE` function.

Each user that wants to share the file must use the `/SHARE` switch with the `□ASS` function that associates the file with a channel. If one user opens a file and has not specified `/SHARE`, other users may not share the file, even if they do specify `/SHARE`. If a file is open for sharing, and a user who did not specify `/SHARE` tries to open it, APL signals *FILE CURRENTLY LOCKED BY ANOTHER USER*.

Another `□ASS` switch, `/NOWRITERS`, gives you some control over file sharing. If you specify `/NOWRITERS` with `/SHARE` (`/NOWRITERS` has no meaning without `/SHARE`), only you can modify the file, but all users are permitted to read it (by specifying `/SHARE/READONLY`).

When files are shared, the file system locks any record as it is retrieved by a user; the record remains locked until that user executes another file input (`□`) or file output (`□`) function on the same channel. You can unlock records sooner by using the `□RELEASE` system function (see Section 5.3.1.3).

#### 5.3.1.3 Unlocking Shared Records

`□RELEASE` unlocks any locked records in files associated with the channel numbers specified in the argument.

If you read a record that you do not intend to rewrite, it is a good idea to unlock the record as soon as possible because other users that try to retrieve the record are put in a wait state until the record becomes available.

In the following example, a file is shared on channels 1 and 2. Records 1, 2, and 3 are written from channel 1, and records 4, 5, and 6 are written from channel 2. Then, channel 1 reads record 1, and channel 2 tries to read the same record before it is released by channel 1.

```
1    □ASS '1 COUNT/RF/Sshare'  
2    □ASS '2 COUNT/RF/Sshare'
```

```

'RECORD 1' ⊕ 1
'RECORD 2' ⊕ 1
'RECORD 3' ⊕ 1
'RECORD 4' ⊕ [4]2
'RECORD 5' ⊕ 1
'RECORD 6' ⊕ 1
⊕ 1
RECORD 1
  ⊕ USER WILL ENTER WAIT STATE
  ⊕ 2
                                         (User enters the attention signal)
18 ATTENTION SIGNALLED (WAITING FOR RECORD LOCK)
  ⊕ 2
  ^
  ⊕RELEASE 1
  ⊕ 2
RECORD 1

```

In this example, the user escaped from the wait state on channel 2 by entering the attention signal. The value of the `□WAIT` system function determines APL's response to a locked record. Depending on the current value of `□WAIT`, APL will wait indefinitely for a locked record to become available, wait for a specified amount of time, or escape from the wait state immediately. For more details on `□WAIT`, see Section 5.3.1.4.

#### 5.3.1.4 Limiting Time on Read Functions

Dyadic `□WAIT` specifies the amount of time you want APL to wait when it tries to read a shared record that is locked by another user.

When you set a waiting period, APL will wait even if you specified the `/READONLY: NOLOCKS` switch when you assigned the file to a channel with `□ASS (NOLOCKS)` normally causes a read operation without waiting).

The left argument (*timelimit*) determines the time limit; it has the following meanings:

<b>Value of <i>timelimit</i></b>	<b>Meaning</b>
<code>-1</code>	Don't wait, return immediately.
<code>0</code>	Wait indefinitely (this is the default).
<code>n</code>	Wait for <i>n</i> seconds.

The `□WAIT` function opens the file if it is not already open. (For a list of commands that open files, see Section 5.2.3.) Thus, unopened files associated with channels identified by positive integers in the `□WAIT` argument are opened for input; unopened files associated with channels identified in the argument by negative integers are opened for output. Note that when you open a sequential

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

file for output, APL creates a new copy of the file with a version number that is one higher than that of the previous copy.

`□WAIT` is a quiet function; if your program requires a result, APL returns `1 0`. If you deassign a channel, or if you close a file, the time limit is set to 0, the default value. `□WAIT` affects the reading of shared files or sequential files opened with the `/UPDATE` switch and does not influence output operations. When a time limit has been set on a channel and the record does not become unlocked within the time limit period, APL signals `IO ERROR (TIMEOUT PERIOD EXPIRED)`.

Monadic `□WAIT` queries the system for the current time limits associated with individual channel numbers.

For each channel number in the argument, monadic `□WAIT` returns a value between `-1` and `255` that can have the following meanings:

Value Returned	Current Time Limit
<code>-1</code>	Don't wait
<code>0</code>	Wait indefinitely
<code>n</code>	Wait for <code>n</code> seconds

If the argument is empty, the result is an empty matrix with a shape of `0 1`; if the argument is a singleton, the result is a one element vector; and if the argument is an `n` element vector, the result is a matrix with a shape of `n` by 1. If the channel is unassigned, `□WAIT` returns `0`.

### 5.3.2 Event Flags

An event flag is a switch that is shared by all users who are in the same group (the system manager generally is responsible for assigning users to groups; for details see the *VMS System Manager's Manual*). Event flags are particularly useful for synchronizing access to mailboxes (see Section 5.3.3.4) or shared files.

#### 5.3.2.1 Associating Events Flags with Channels

Event flags, like files, must be associated with a channel number, and, as with files, you use the `□ASS` function to make that association. The `□ASS` switch that sets up event flags has the form:

`/EFN : n`

`n`, the event flag number, is an integer from 64 through 95 inclusive. You must specify a value for `n`.

For example, the following associates event flag number 77 with channel 1:

```
QASS '1 MYFILE/RF/SHARE/EFN:77'
```

The file specification and other information in the `QASS` argument, in this case `MYFILE/RF/SHARE`, are not specifically related to the event flag, except that both are associated with the same channel. However, if you plan to use the event flag to synchronize access to the shared file, it is convenient to associate the file and the event flag with the same channel, thus establishing a logical connection between the two.

If you want to set up an event flag without associating a file with a channel at the same time, use a dummy file name for the file specification in the `QASS` argument. As long as you do not try to open the file, the dummy specification will not generate an error.

Each user in the group who wants access to a specific event flag must associate the event flag number with a channel. Then, any of the users may read, set, or clear the event flag by using the event flag system functions.

VAX APL associates the cluster name `APL$_CHANNL_EFC` using `$ASCEFC` with the common event flag cluster given by the `/EFN=n` switch to `.bxASS`. It does the `$ASCEFC` the first time any one of `QEFC`, `QEFR`, or `QEFS` is used with the event flag. Since APL only allows the event flag on `/EFN` to be in the interval [64..95], APL only allows access to common event flags and only one event flag cluster. The access is allowed inside cooperating processes in the same group so all processes in that group using `/EFN` have to agree on what each event flag means.

### 5.3.2.2 Event Flag System Functions

There are three event flag system functions: `QEFR` to read event flag values, `QEFS` to set event flags (make them equal 1), and `QEFC` to clear event flags (make them equal 0).

The `QEFR` function returns the values of the event flags associated with the channel numbers in its argument. For channels not associated with an event flag, `QEFR` returns -1. For example:

```
QEFR 15  
-1  
1  
-1  
0  
1
```

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

The result is a matrix (or vector, if the argument is a singleton) of shape  $n \times 1$ , where  $n$  is the shape of the argument. In the example, the result indicates that the event flags associated with channels 1, 3, and 5 are set, the event flag associated with channel 2 is clear, and no event flag is associated with channel 4.

The `□EFS` and `□EFC` functions set and clear, respectively, the event flags associated with the channel numbers in their arguments. They return a matrix of shape  $n \times 1$ , where  $n$  is the shape of the argument, and the values are the previous values of the event flags. For channel numbers not associated with event flags, `□EFS` and `□EFC` return -1.

If the argument to `□EFR`, `□EFS`, or `□EFC` is empty, APL returns 0 1p 0 as the result.

#### 5.3.3 Mailboxes

Mailboxes allow you to communicate easily with other users. A mailbox is an area in memory from which you can send and receive messages. In APL, the way you access mailboxes is similar to the way you access files. The difference is that when you send a message to a mailbox, APL prevents you from doing further processing until another user reads the message. In addition, if you read a mailbox that does not contain a message, APL puts you in a suspended state until a message becomes available.

For some applications, you may want to suspend processing when you access a mailbox; for others, you may want to be careful to read from a mailbox only when it contains a message, or to write to a mailbox only when another user is ready to read. Section 5.3.3.4 shows how you can use event flags to synchronize access to mailboxes.

##### 5.3.3.1 Associating Mailboxes with Channels

You use the `□ASS` function with the `/MBX` switch to associate a mailbox with a channel. For example:

```
□ASS '1 MBOX/AS/MBX/SHARE/DISPOSE:KEEP/MAXLEN:80  
/PROTECTION=(S:RWED,O:RWED,G:RWED)'  
1
```

A file specification (in this case `MBOX`) and the `/MBX` switch are required. Mailboxes are sequential devices, so the file organization switch must be `/AS` or `/IS`.

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

The */DISPOSE:KEEP* parameter makes the mailbox a permanent mailbox. If you use */DISPOSE:DELETE*, the mailbox is a temporary mailbox that is deleted when no user has a channel associated with it. Note that */DISPOSE:DELETE* is the default for mailboxes (*/DISPOSE:KEEP* is the default when */MBX* is not specified). A permanent mailbox is deleted whenever the system is rebooted.

If you use */SHARE*, the mailbox is called a public mailbox and its name is included (after the mailbox is opened) in the group logical name table (you can use the command *)PUSH SHOW LOGICAL/GROUP* to see the contents of the group logical name table). All users in the group can find the name in the table and access the mailbox by associating the name with a channel.

If you do not use */SHARE*, the mailbox is called a private mailbox. Other users in the group can still access the mailbox if they know its name, but the name is not included in the group logical name table. Thus, you can restrict access to the mailbox.

When the mailbox is temporary, and you specify */SHARE*, the logical name is associated with a mailbox device and is inserted into the logical name table LNM\$TEMPORARY\_MAILBOX. (Note that logical names of mailboxes are case-sensitive.) By default, LNM\$TEMPORARY\_MAILBOX is associated with the logical name table LNM\$JOB. This logical name table is accessible only to processes within the current job.

When the mailbox is temporary, and you do not specify */SHARE*, the logical name is not defined, and the mailbox can only be accessed by its device name (the physical device number from *\\$MBX*). However, *\\$ASS* does not allow access to mailboxes via device names. For more information about using mailboxes, see the *VMS System Services Reference Manual*.

When the mailbox is permanent, and you specify */SHARE*, the logical name is associated with a mailbox device and is inserted into the logical name table LNM\$PERMANENT\_MAILBOX. (Note that logical names of mailboxes are case-sensitive.) By default, LNM\$PERMANENT\_MAILBOX is associated with the logical name table LNM\$SYSTEM; this logical name table is accessible on a system-wide basis.

When the mailbox is permanent, and you do not specify */SHARE*, the logical name is not defined, and the mailbox can only be accessed by its device name. However, *\\$ASS* does not allow access to mailboxes by means of device names.

The */MAXLEN* parameter establishes the maximum message length for a new mailbox. It is ignored for existing mailboxes.

The */PROTECTION* switch works as described in the *VAX APL Reference Manual*.

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

#### 5.3.3.2 Sending and Receiving Messages

You use the file output (`⊖`) and file input (`⊖`) functions to send messages to and receive messages from mailboxes. You must access mailboxes sequentially. For examples of mailbox I/O, see Section 5.3.3.4.

#### 5.3.3.3 `⊖MBX`—Mailbox System Function

`⊖MBX` returns information on the status of mailboxes. The absolute values of *chans* represent the channel numbers associated with the event flags you want to manipulate. For each channel specified, `⊖MBX` returns a row of three elements denoting the following (from left to right):

- The physical device number assigned to the mailbox (or 0 if the mailbox is remote, and -1 if the channel is not associated with a mailbox).
- The process identification number (PID, returned by `⊖UL`) of the last user to receive a message you sent to the mailbox (or -1 if no messages have been sent).
- The PID of the last user from which you received a message in the mailbox (or -1 if no messages have been received).

The result is a matrix (or a vector if the argument is a singleton) with the shape *n* by 3, where *n* is the length of the argument.

To return a value for `⊖MBX`, APL must open the mailbox if it is not already open. (For a list of commands that open files, see Section 5.2.3.) For channel numbers represented in the argument by positive integers, APL opens the mailbox for input; for channel numbers represented by negative integers, APL opens the mailbox for output. Note that whether a mailbox is opened for input or output is not significant because APL treats mailboxes like terminals; it allows both input and output at the same time, even in sequential modes.

#### 5.3.3.4 Sample Functions That Use Mailboxes

This section describes five groups of functions that use mailboxes. The first three groups include functions that associate a mailbox with a channel, read from a mailbox, and write to a mailbox. Each group is distinguished by the way it synchronizes mailbox processing:

- The first group does not use event flags to synchronize mailbox processing.
- The second group uses one event flag.
- The third group uses two event flags.
- The fourth group uses mailboxes for communication between a parent process and a subprocess.

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

- The fifth group shows the effects of the */SHARE* and */DISPOSE* switches on the logical name definitions of APL mailboxes.

In the first group of mailbox examples, event flags are not used to synchronize access. Comments within each function explain what the function does.

```
    v CHAN←EFOASSIGN
[1]  ⋀THIS FUNCTION RETURNS THE CHANNEL NUMBER THAT
[2]  ⋀THE MAILBOX IS ASSOCIATED WITH.
[3]  CHAN←◻ASS 'MAILBOX/AS/MBX/SHARE'
[4]  v
    v DATA EFOOUT CHAN
[1]  ⋀WHEN YOU USE THIS FUNCTION TO WRITE TO THE
[2]  ⋀MAILBOX, SESSION IS SUSPENDED UNTIL ANOTHER
[3]  ⋀USER TAKES THE MESSAGE OUT OF THE MAILBOX.
[4]  DATA ⓧ[4] CHAN
[5]  v
    v INPUT←EF0IN CHAN
[1]  ⋀WHEN YOU USE THIS FUNCTION TO READ FROM A
[2]  ⋀MAILBOX, SESSION IS SUSPENDED UNTIL A MESSAGE
[3]  ⋀IS AVAILABLE IN THE MAILBOX.
[4]  INPUT←⩧[4]CHAN
[5]  v
```

In the second group of mailbox examples, one event flag is used to synchronize access to a mailbox. These examples use the convention that if the event flag associated with the mailbox's channel is 0, the mailbox is empty; if the event flag is 1, a message is available.

```
    v CHAN←EF1ASSIGN
[1]  ⋀THIS FUNCTION RETURNS THE CHANNEL NUMBER THAT
[2]  ⋀THE MAILBOX IS ASSOCIATED ON.
[3]  CHAN←◻ASS 'MAILBOX/AS/MBX/SHARE/EFN=64'
[4]  v
    v BUSY←DATA EF1OUT CHAN
[1]  ⋀IF THE MAILBOX ALREADY HAS A MESSAGE IN IT,
[2]  ⋀THIS FUNCTION RETURNS 1 AND DOES NOTHING.
[3]  ⋀
[4]  ⋀IF MAILBOX IS EMPY, THIS FUNCITON WRITES
[5]  ⋀DATA INTO THE MAILBOX AND RETURNS 0. NOTE THAT
[6]  ⋀YOUR SESSION IS SUSPENDED UNTIL ANOHER USER
[7]  ⋀READS YOUR MESSAGE.
[8]  → (BUSY←◻EFS CHAN) 0
[9]  DATA ⓧ[4] CHAN
[10] v
```

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

```
[1]   V INPUT←EF1IN CHAN
[2]   ⋀IF THE MAILBOX IS EMPTY, RETURNS 0 75 ⋁ 0,
[3]   ⋀OTHERWISE RETURNS THE CONTENTS OF THE MAILBOX.
[4]   → (0=DEFC CHAN) ⋁ EMPTY
[5]   INPUT←@[4]CHAN ⋁ → 0
[6]   EMPTY: INPUT←0 75 ⋁ 0
[7]   V
```

In the third group of mailbox examples, two event flags are used to synchronize access to a mailbox. These examples use the convention that event flag 81 is set by the receiver to announce that it is searching for a sender and is cleared by the sender when the sender recognizes the receiver. Event flag 82 is set by the sender to announce that it is searching for a receiver and is cleared by the receiver when it recognizes the sender.

```
[1]   V CHANVECTOR←EF2ASSIGN;PRICHAN;SECCHAN
[2]   ⋀THIS FUNCTION RETURNS A 2-ELEMENT VECTOR
[3]   ⋀CONTAINING THE CHANNEL THAT THE MAILBOX AND
[4]   ⋀EVENT FLAG 81 ARE ASSIGNED ON (PRICHAN),
[5]   ⋀FOLLOWED BY THE CHANNEL THAT EVENT FLAG 82
[6]   ⋀IS ASSIGNED ON (SECCHAN).
[7]   CHANVECTOR←0 0
[8]   →(0=PRICHAN←$ASS'MAILBOX/AS/MBX/SHARE/EFN=81') ⋁ 0
[9]   →(0=SECCHAN←$ASS'DUMMY$DEVICE$/AS/EFN=82') ⋁ 0
[10]  CHANVECTOR←PRICHAN,SECCHAN
[11]  V
```

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

▼ NOTSENT←DATA EF2OUT CHANVECTORWAIT  
[1]    @CHANVECTORWAIT IS A 3-ELEMENT VECTOR CONSISTING  
[2]    @OF PRICHAN AND SECCHAN FROM FUNCTION EF2ASSIGN,  
[3]    @FOLLOWED BY THE NUMBER OF SECONDS YOU WANT TO  
[4]    @WAIT BEFORE RETRYING THIS FUNCION IF THE  
[5]    @MESSAGE CANNOT BE SENT.  
[6]    @ EVENT FLAG 81  
[7]    @ SET BY RECEIVER, CLEARED BY SENDER  
[8]    @  
[9]    @ EVENT FLAG 82  
[10]   @ SET BY SENDER, CLEARED BY RECEIVER  
[11]   @THIS FUNCTION CHECKS EVENT FLAG 81 TO SEE IF  
[12]   @ANYONE IS READY TO RECEIVE. IF SO THIS  
[13]   @FUNCTION RECOGNIZES THE RECEIVER (BY CLEARING)  
[14]   @EVENT FLAG 81 AND SENDS THE MESSAGE.  
[15]   @OTHERWISE, THIS FUNCTION EITHER COMPLETES  
[16]   @AND RETURNS 1 INDICATING NO MESSAGE WAS SENT,  
[17]   @OR WAITS THE NUMBER OF SECONDS SPECIFIED BY  
[18]   @THE THIRD ELEMENT IN CHANVECTORWAIT AND TRIES  
[19]   @AGAIN.  
[20]   @THE SENDER LOOKING FOR RECEIVER FLAG (82) IS  
[21]   @NOT CLEARED IF WE RETURN 1, SO A CALLER SHOULD  
[22]   @EITHER CLEAR THE FLAG OR CALL THIS FUNCTION  
[23]   @AGAIN LATER. THE RIGHT THING TO DO IS TO CALL  
[24]   @THE FUNCTION AGAIN BECAUSE CLEARING THE FLAG  
[25]   @COULD CAUSE A RACE CONDITION.  
[26]   @NOTSENT←1 @INITIALIZE RESULT' SET EVENT FLAG 82  
[27]   LOOP: □SINK←□EFS ←1↑2↑CHANVECTORWAIT  
[28]   @IF NO RECEIVERS READY, RETURNS 1 AND EXIT OR  
[29]   @BRANCH TO DELAY, WAIT, AND TRY AGAIN.  
[30]   → (0=□EFC 1↑CHANVECTORWAIT)▷ DELAY  
[31]   @IF THERE IS A RECEIVER, SEND MESSAGE AND EXIT.  
[32]   DATA □[4]1↑CHANVECTORWAIT ◇ → NOTSENT←0  
[33]   @  
[34]   @IF WAIT IS OT 0, RETURN AND TRY AGAIN.  
[35]   DELAY: → (0=←1↑CHANVECTORWAIT)▷ 0  
[36]   □SINK←□DL ←1↑CHANVECTORWAIT   @WAIT  
[37]   → LOOP           @TRY AGAIN  
[38]   ▼

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

```
v INPUT←WAIT EF2IN CHANVECTOR
[1]  a   EVENT FLAG 81
[2]  a      SET BY RECEIVER, CLEARED BY SENDER
[3]  a
[4]  a   EVENT FLAG 82
[5]  a      SET BY SENDER, CLEARED BY RECEIVER
[6]  a
[7]  a THIS FUNCTION CHECKS EVENT FLAG 82 TO SEE IF
[8]  a ANY SENDERS HAVE MESSAGES TO SEND.  IF YES,
[9]  a THE FUNCTION RECOGNIZES THE SENDER AND READS
[10] a THE MESSAGE.  OTHERWISE, IF WAIT=0, RETURNS
[11] a 0 75 p 0; IF WAIT IS NOT 0, WAITS THE NUMBER
[12] a OF SECONDS INDICATED BY WAIT AND TRIES AGAIN.
[13] a
[14] a THE RECEIVER LOOKING FOR SENDER FLAG (81) IS
[15] a NOT CLEARED IF FUNCTION RETURNS 0 75 p 0, SO
[16] a CALLER SHOULD EITHER CLEAR THE FLAG OR CALL
[17] a FUNCTION AGAIN LATER.  THE RIGHT THING TO DO
[18] a IS TO CALL THE FUNCTION LATER, BECAUSE CLEARING
[19] a THE FLAG COULD CAUSE A RACE CONDITION.
[20] a
[21] INPUT←0 75 p 0      aINITIALIZE RESULT
[22] LOOP: ⌂SINK←⌂EFS 1↑CHANVECTOR aANNOUNCE READY
[23] aBRANCH IF NO SENDERS READY
[24] → (0=⌂EFC 1↑CHANVECTOR)p DELAY
[25] INPUT←⌃[4]1↑CHANVECTOR → 0 aGET MESSAGE.
[26] a
[27] aIF WAIT NOT 0, RETURN, ELSE DELAY, TRY AGAIN.
[28] DELAY: →(WAIT=0) p 0 → ⌂SINK←⌂DL WAIT → LOOP
[29] v
```

The fourth group of mailbox examples demonstrates the assignment of a channel to a mailbox; the creation of a function that performs some simple I/O through a parent process that spawns a child subprocess; and the running of the function.

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

```
A PARENT PROCESS COMMUNICATING WITH ITS CHILD PROCESS
A THROUGH A TEMPORARY MAILBOX.

A
A BUILD CHILD FUNCTION
[1] V CHILD MAILBOXNAME ;CH;DATA
[2] A
[3] A ASSIGN CHANNEL FOR A /AS MAILBOX
[4] A
[5] CH ← ⎕ASS MAILBOXNAME,'AS/MBX/SHARE/DISPOSE=DELETE'
[6] A
[7] ADO MAILBOX I/O
[8] A
[9] A READ MAILBOX MESSAGE FROM PARENT
[10] LOOP: DATA ← ⎕[2] CH
[11] A IF EOF RETRY
[12] ↳(ʌ/0 75=2↑p DATA)/'→ LOOP'
[13] A ECHO MESSAGE OVER MAILBOX AND BACK TO PARENT
[14] ('÷÷ RECEIVED ->,'DATA,'<- END') ⎕[2] CH
[15] A CONTINUE UNTIL QUIT RECEIVED
[16] ↳(ʌ/'QUIT'*4↑,DATA)/LOOP
[17] A
[18] A CLEANUP AND QUIT
[19] A
[20] DONE: ⎕DAS ⎕CHANS ⋄ʌ')OFF'
[21] V

A
V ITERATIONS PARENT MAILBOXNAME ;CH
[1] A
[2] A SPAWN SUBPROCESS
[3] A
[4] A
[5] ↳')PUSH/NOWAIT APL/TERM=TTY/SILENT=ALL MAILBOX'
[6] A
[7] A ASSIGN CHANNEL FOR AN /AS MAILBOX
[8] A
[9] CH ← ⎕ASS MAILBOXNAME,'/AS/MBX/SHARE/DISPOSE=DELETE'
[10] A
[11] ADO MAILBOX I/O
[12] A
[13] C ← 0
[14] ADO ITERATIONS
[15] LOOP: →(ITERATIONS ≤ C←C+1)/DONE
[16] A SEND DATA OVER MAILBOX TO CHILD
[17] (i C) ⎕[2] CH
[18] A READ AND ECHO RESPONSE FROM CHILD
[19] ⎕ ← '** MESSAGE ',(⍴ C),',', ⎕[2] CH
[20] → LOOP           A CONTINUE
[21] A
[22] A ALL DONE
[23] A
[24] DONE: 'QUIT' ⎕[2]CH           A SEND QUIT MESSAGE
```

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

```
[25]  aVERIFY QUIT MESSAGE RECEIVED
[26]  ⌂ ← '** MESSAGE ',(¤ C),' ',⌂[2] CH
[27]  ⌂DAS ⌂CHANS          aDEASSIGN
[28]  †')DROP MAILBOX&L;*''
[29]  v
      a
      aSAVE WORKSPACE FOR SUBPROCESS
      a
      ⌂LX ← 'CHILD ''TEMPBOX'''
      )WSID MAILBOX
WAS CLEAR WS
)SAVE
MONDAY 21-JAN-1991 12:01:56.73 17 BLKS MAILBOX
      a
      aSTART MAILBOX I/O
      a
      5 PARENT 'TEMPBOXa
** MESSAGE 1 ÷÷ RECEIVED ->1<- END
** MESSAGE 2 ÷÷ RECEIVED ->1 2<- END
** MESSAGE 3 ÷÷ RECEIVED ->1 2 3<- END
** MESSAGE 4 ÷÷ RECEIVED ->1 2 3 4<- END
** MESSAGE 5 ÷÷ RECEIVED ->QUIT<- END
APLD$:[APL]MAILBOX.APL;1 deleted (36 Blocks)
```

The fifth group of mailbox examples demonstrates the effects of the /SHARE and /DISPOSE switches on the logical name definitions of APL mailboxes.

```
aNOTE: APL DOES NOT ALLOW ASSIGNING CHANNEL NUMBERS TO
      a      MAILBOXES WITH DEVICE NAMES.
      a
      aASSIGN AND OPEN CHANNELS TO MAILBOXES
      a
      ⌂ASS 'TEMPNOSHARE/AS/MBX/DISPOSE=DELETE/OPEN=NEW'
12    ⌂ASS 'TEMPSHARE/AS/MBX/SHARE/DISPOSE=DELETE/OPEN=NEW'
11    ⌂ASS 'PERMNOSHARE/AS/MBX/OPEN=NEW'
10    ⌂ASS 'PERMSHARE/AS/MBX/SHARE/OPEN=NEW'
9
      a
      aVIEW THE MAILBOX DEVICE NUMBERS THAT HAVE BEEN ASSIGNED
      a
      ⌂+ MBX + ⌂MBX ⌂CHANS
179 - 1 - 1
178 - 1 - 1
177 - 1 - 1
176 - 1 - 1
```

```

      A
      EXAMINE THE LOGICAL NAMES THAT HAVE BEEN DEFINED
      A
      )DO SHOW LOGICAL *SHARE

(LNM$PROCESS+TABLE)

(LNM$JOB+80412D10)

      APERMSHAREA = AMBA179:A
      ATEMPSHAREA = AMBA177:A

(LNM$GROUP+0000011)

(LNM$SYSTEM+TABLE)

      ASYS$SHAREA = ASYS$SYSROOT:[SYSLIB]A

(DECW$LOGICAL+NAMES)
      A
      EXAMINE THE DEVICE INFORMATION ON EACH
      A
      A+')DO SHOW DEVICE/FULL MBA'
      A,(# 1↑,1↑[1]MBX),':' ⧺ MBX+1↑[1]MBX

Device MBA179: is online, record-oriented device, shareable, mailbox device.

Error count          0    Operations completed          0
Owner process        AA   Owner UIC           [USERS,APLUSER]
Owner process ID    00000000 Dev Prot     S:RWLP,O:RWLP,G:RWLP,W:RWLP
Reference count      1    Default buffer size    2044

      A,(# 1↑,1↑[1]MBX),':' ⧺ MBX+1↑[1]MBX

Device MBA178: is online, record-oriented device, shareable, mailbox device.

Error count          0    Operations completed          0
Owner process        AA   Owner UIC           [USERS,APLUSER]
Owner process ID    00000000 Dev Prot     S:RWLP,O:RWLP,G:RWLP,W:RWLP
Reference count      1    Default buffer size    2044

      DADS  DCHANS      ADEASSIGN THE OPENED CHANNELS

```

### 5.3.4 Pure Data Records

The data records you write (to other than /AS files) using the **W** function are APL objects. They include more than the data itself: in internal sequential, direct-access, relative, and keyed files, APL also includes other information within each record, such as the object's rank, shape, and data type. APL uses the information to format the data when you read the record using file input (**R**). The format of this additional information is shown in Section 5.3.4.1.

Having APL internal information within records is acceptable if you are working in an APL-only environment. You never see the internal information and when you read a record from an internal sequential, direct-access, relative, or keyed file, APL displays it in APL format.

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

However, if you want to create files for use by programs written in other languages, you may want to write records containing “pure” data, that is, records that are vectors of values with no embedded format information.

Similarly, if you want to read files created by non-APL programs, you must instruct APL not to look for the internal formatting information that would be in records written by APL.

APL interprets a data record as a vector of pure data if you use the *data-type* option with the file input (`⊣`) or file output (`⊣`) function. The forms are as follows:

`data⊣[[index]]⊣channel⊣[data-type]]⊣[[index]]⊣channel⊣[data-type]]⊣`

*data-type* specifies the data type to be used to interpret the data. The values and meanings for the *data-type* parameter are listed in Table 5–6. The *data-type* parameter is invalid with ASCII sequential files.

The other parameters have the same meanings as described in Section 5.2.3. For file input (`⊣`), the *data-type* parameter has the same effect for internal sequential, direct-access, relative, and keyed files. It instructs APL to assume that the record is a vector of pure data, and to assign the indicated data type to it.

For file output (`⊣`), the *data-type* parameter instructs APL to reformat the data in the specified data type and to write it as a single, unsegmented vector of values. If the values cannot be reformatted into the specified data type (for example, floating-point to Boolean), APL signals *DOMAIN ERROR*. If the record cannot fit into a single segment, APL signals *MAXIMUM BLOCK SIZE EXCEEDED*. Data type conversion Tables 5–7 and 5–8, in Section 5.3.4.3), summarize the effects of using the *data-type* parameter with all possible combinations of data types.

The effects of using the *data-type* parameter with `⊣` are the same for all four types of file organization, with one exception: records in direct-access files have headers consisting of two longwords that contain a record number and a segment number. (The count of record and segment numbers begins at 0, not 1. The segment number is always 0, because pure data records cannot be segmented.)

If you do not use the *data-type* parameter and you try to read a record that is not an APL object (was not written by APL or was written by APL as pure data), APL signals *COMPONENT ERROR (RECORD NOT A COMPONENT)*.

**Table 5–6 Data-Type Parameter Values**

Value	External Data Type
0	No conversion, use type of <i>data</i>
1	32-bit integer
2	1-bit Boolean
3	F_floating single-precision floating point
4	D_floating double-precision floating-point
5	8-bit APL $\square A V$ characters
6	8-bit ASCII characters
7	8-bit numeric bytes
8	G_floating double-precision floating-point
9	H_floating floating-point
10	16-bit integer
11	8-bit Digital multinational characters
12	8-bit APL $\square A V$ characters in TTY mnemonics
13	8-bit APL $\square A V$ characters in KEY-paired APL
14	8-bit APL $\square A V$ characters in BIT-paired APL
15	8-bit APL $\square A V$ characters in COMPOSITE APL

Note that when you specify the *key-type* or *data-type* parameters for /KY files, you can use only the external data type value 0, 1, 5, 6, 11, 12, 13, 14, or 15. These external data types allow APL to compare the length of the key value with the key size defined for the file.

You may want to read /KY records that use different data types for the key values and the actual data. The following example mixes character key values and integer data. Note the form for reading /KY records randomly:

`[[value [[;key-num [[;tech[[;key-type]]]]]] chan [[data-type]]]`

In the example, both the *key-type* and the *data-type* parameters are specified. This is necessary because the data types for the key values and for the actual data are different. If they were the same, APL would use the value for *data-type* as the default for *key-type*. (You could still specify *key-type* explicitly.)

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

```
ASSIGN NEW /KY FILE, PRIMARY KEY = 8 CHARACTERS
CLASS '1 TEST/KY=1:8:CHARACTER/SIGNAL'
1
  CREATE 3 VARIABLES FOR WRITING, 8 CHARS, 10 INT
  X1←('AAAAAAA' ⎕COQ 0 15),110
  X2←('BBBBBBBB' ⎕COQ 0 15),1100+110
  X3←('CCCCCCC' ⎕COQ 0 15),1200+110
  X1 ⎕1 1   AWRITE RECORDS TO CHAN 1 AS INTEGERS
  X2 ⎕1 1
  X3 ⎕1 1
  AKEY-TYPE IS 15; DATA-TYPE IS 1
  Y1←⊖[['AAAAAA';0;'EQL';15]1 1
  X1 ≡ Y1   AFOUND AAAAAA 1 2 ...
1
  Y3←⊖[['CCCCCCC';0;'EQL'15]1 1
  X3 ≡ Y3   AFOUND CCCCCC 210 202 ..
1
```

#### 5.3.4.1 Reading Pure Data Files Sequentially

You can use the `⊖` function with the *data-type* parameter to read VAX RMS disk files sequentially. You can successfully read any such file by assigning any of types 2, 5, 7, or 11 to the records.

If the file was not created by APL, neither the type of file organization the file has nor the type you used when you associated the file with a channel is important; APL reads each record as a vector of values that have the specified data type.

Note that if the language that created the file includes internal formatting information in its records (as APL does), the internal data is returned as part of the record. Thus, if you want to read a file created by another language, you may need to know something about how that language formats records. Figure 5–1 shows the format of an APL record on disk.

Figure 5–1 APL Internal Record Format

component number for /DA: ignore for /IS, /RF, and /KY			
component number for /DA: ignore for /IS, /RF, and /KY			
8 bits ignore	4 bits <i>type</i>	4 bits ignore	16 bits <i>rank</i>
		0	
		0	
<i>x/pDATA</i>			
(pdata)[1]			
(pdata)[2]			
⋮			
(pdata)[ <i>rank</i> ]			
( <i>data</i> [1]			
( <i>data</i> [2]			
⋮			
( <i>data</i> [ <i>length</i> ])			

NU-2231A-RA

***rank***

Is the rank of the data contained in the record.

***type***

Is one of the four APL internal data types described in Table 5–7. The possible values are 0, for floating-point; 1, for integer; 2, for Boolean; and 3, for APL character.

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

#### *length*

Is the number of items in the array.

In the following example, two records with the value `:10` are written to an internal sequential file: the first record is an APL object (the *data-type* parameter is not used); the second record is a record of pure data (the *data-type* parameter is used).

```
DASS '1 TESTIS/IS'
1
  (:10)@1      ⋄RECORD 1 IS AN APL OBJECT
  (:10)@1 1    ⋄RECORD 2 IS INTEGER PURE DATA
  @CLS 1
  C+@1 7      ⋄READ APL OBJECT AS NUMERIC BYTES
  ⍷(((pC)÷4),4)pC ⋄PUT IN INTERNAL RECORD FORMAT
  0 0 0 0
  0 0 0 17
  80 16 0 1
  0 0 0 0
  0 0 0 0
  0 0 0 10
  0 0 0 10
  0 0 0 1
  0 0 0 2
  0 0 0 3
  0 0 0 4
  0 0 0 5
  0 0 0 6
  0 0 0 7
  0 0 0 8
  0 0 0 9
  0 0 0 10
```

#### 5.3.4.2 Reading Pure Data Files Randomly

Relative and keyed files created outside APL can be read randomly. You must know the structure of the records written to a keyed file and the location, length, and data type of each key. Like other languages, APL assigns relative record numbers to `/RF` files, so you can use the [*index*] and *data-type* parameters with file input (`@`) to randomly retrieve a particular record in a relative file and read it as pure data.

You cannot randomly read records in an APL direct-access file as pure data, unless the file was created by APL, or unless the file's records were given a key structure consistent with that used by APL.

#### 5.3.4.3 Data Type Conversion Tables

APL stores all data internally as one of four possible data types:

- APL character—stored as 8-bit values that are indexes (from 0 to 255) of  $\Box A V$ .
- Boolean—stored as 1 bit per value.
- Integer—stored as 32 bits per value.
- Floating-point—stored in VAX D\_floating format (64 bits per value).

Table 5–7 summarizes the effects of converting (using  $\Box$  or  $\Box COQ$ ; see Section 5.2.3 and the *VAX APL Reference Manual*, respectively) an internal APL data type to one of the external data types listed in Table 5–6 in Section 5.3.4.

**Table 5–7 Converting APL Internal Values to External Values**

External Data Type	Boolean (1 bit per value)	Integer (32 bits per values)	Floating-Point (D_floating format; 64 bits per value)	Character (8 bits per value)
type=1 (integer)	Each 1 or 0 is written as a 32-bit integer value	Each integer is written as a 32-bit integer value	If each D_floating value is equal to a near-integer, all are written as 32-bit integer values otherwise, APL signals DOMAIN ERROR	APL signals DOMAIN ERROR
type=2 (Boolean)	Each 1 or 0 is written as a 1-bit value	If each integer is equal to 0 or 1 (near-integer), all are written as 1-bit values; otherwise, APL signals DOMAIN ERROR	If each D_floating value is equal to 0 or 1 (near-integer), all are written as 1-bit values; otherwise, APL signals DOMAIN ERROR	APL signals DOMAIN ERROR
type=3 (F_floating)	Each 1 or 0 is written as a 32-bit F_floating value	Each integer is rounded as necessary and is written as a 32-bit F_floating value	Each D_floating value is written as a 32-bit F_floating value	APL signals DOMAIN ERROR

(continued on next page)

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

**Table 5-7 (Cont.) Converting APL Internal Values to External Values**

External Data Type	Boolean (1 bit per value)	Integer (32 bits per values)	Floating-Point (D_floating format; 64 bits per value)	Character (8 bits per value)
type=4 (D_floating)	Each 1 or 0 is written as a 64-bit D_floating value	Each integer is written as a 64-bit D_floating value	Each D_floating value is written as a 64-bit D_floating value	APL signals <i>DOMAIN ERROR</i>
type=5 (APL characters)	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	Each APL character value is written as an 8-bit APL character value
type=6 (ASCII characters)	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	See Table 5-9
type=7 (numeric byte)	Each 1 or 0 is written as an 8-bit integer value	If each integer is in the range 0 through 255 inclusive, all are written as 8-bit integer values; otherwise, APL signals <i>DOMAIN ERROR</i>	If each D_floating value is equal to a near-integer in the range 0 through 255 inclusive, all are written as 8-bit integer values; otherwise APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>
type=8 (G_floating)	Each 1 or 0 is written as a 64-bit G_floating value	Each integer is written as a 64-bit G_floating value	Each D_floating value is written as a 64-bit G_floating value	APL signals <i>DOMAIN ERROR</i>
type=9 (H_floating)	Each 1 or 0 is written as a 128-bit H_floating value	Each integer is written as a 128-bit H_floating value	Each D_floating value is written as a 128-bit H_floating value	APL signals <i>DOMAIN ERROR</i>

(continued on next page)

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

**Table 5–7 (Cont.) Converting APL Internal Values to External Values**

External Data Type	Boolean (1 bit per value)	Integer (32 bits per values)	Floating-Point (D_floating format; 64 bits per value)	Character (8 bits per value)
type=10 (16-bit integer)	Each 1 or 0 is written as a 16-bit integer value	If each integer is in the range -32768 through 32768 inclusive, all are written as 16-bit integer values; otherwise, APL signals DOMAIN ERROR	If each D_floating value is equal to a near-integer in the range -32768 through 32768 inclusive, all are written as 16-bit integer values' otherwise, APL signals DOMAIN ERROR	APL signals DOMAIN ERROR
type=11 (DEC Multinational characters)	APL signals DOMAIN ERROR	APL signals DOMAIN ERROR	APL signals DOMAIN ERROR	See Table 5–10
type=12 (TTY mnemonic characters)	APL signals DOMAIN ERROR	APL signals DOMAIN ERROR	APL signals DOMAIN ERROR	Each character is translated to the $\square A V$ characters that are its TTY mnemonic (see Table 1–15)
type=13 (KEY-paired characters)	APL signals DOMAIN ERROR	APL signals DOMAIN ERROR	APL signals DOMAIN ERROR	Each character is translated to the $\square A V$ characters that are its KEY-paired APL representation (see Table 1–13)
type=14 (BIT-paired characters)	APL signals DOMAIN ERROR	APL signals DOMAIN ERROR	APL signals DOMAIN ERROR	Each character is translated to the $\square A V$ characters that are its BIT-paired APL representation (see Table 1–14)
type=15 (COMPOSITE characters)	APL signals DOMAIN ERROR	APL signals DOMAIN ERROR	APL signals DOMAIN ERROR	Each character is translated to the $\square A V$ characters that are its APL COMPOSITE representation (see Table 1–16)

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

Table 5–8 summarizes the effects of converting (using `⊣` or `⊣CIQ`; see Section 5.2.3 and the *VAX APL Reference Manual*, respectively) external data types to one of APL internal data types listed in Table 5–6.

**Table 5–8 Converting External Data Types to APL Values**

Data Types	Type Specification APL's Action
type=1 (integer)	Interprets 32 bits at a time; returns the integer value of each 32 bits in the record or variable.
type=2 (Boolean)	Interprets 1 bit at a time; returns a Boolean vector whose length is equal to the length of the record or variable (in bits).
type=3 (F_floating)	Interprets 32 bits at a time; returns the F_floating value of each 32 bits in the record or variable. APL stores F_floating values in D_floating format.
type=4 (D_floating)	Interprets 64 bits at a time; returns the D_floating value of each 64 bits in the record or variable.
type=5 (APL character)	Interprets 8 bits at a time; returns the APL character value (an element of $\Box AV$ ) of each 8 bits in the record or variable.
type=6 (ASCII text)	Interprets 8 bits at a time; returns the APL character that would result from using del quad input with the TTY character set on each 8 bits in the record or variable.
type=7 (numeric byte)	Interprets 8 bits at a time; returns the integer value of each 8 bits in the record or variable. APL stores numeric byte values in 32-bit integer format.
type=8 (G_floating)	Interprets 64 bits at a time; returns the G_floating value of each 64 bits in the record or variable. APL stores G_floating values in D_floating format. If the G_floating magnitude is outside the range (approximately) $0.26E^{-38}$ to $1.7E38$ , APL signals <i>DOMAIN ERROR</i> .
type=9 (H_floating)	Interprets 128 bits at a time; returns the H_floating value of each 128 bits in the record or variable. APL stores H_floating values in D_floating format. If the H_floating magnitude is outside the range (approximately) $0.26E^{-38}$ to $1.7E38$ , APL signals <i>DOMAIN ERROR</i> .
type=10 (16-bit integer)	Interprets 16 bits at a time; returns the integer value of each 16 bits in the record or variable. APL stores 16-bit integer values in 32-bit integer format.

(continued on next page)

**Table 5–8 (Cont.) Converting External Data Types to APL Values**

Data Types	Type Specification APL's Action
type=11 (DEC Multi-national Characters)	Interprets 8 bits at a time; returns the APL character (or characters) that would result from the translation specified in Table 5–12 on each 8 bits in the record or variable.
type=12 (TTY mnemonic character)	Interprets 8 bits at a time; returns the APL character (or characters) that would result from using quote-quaq input with TTY character set on each 8 bits in the record or variable.
type=13 (KEY-paired APL character)	Interprets 8 bits at a time; returns the APL character (or characters) that would result from using quote-quaq input with the KEY character set on each 8 bits in the record or variable.
type=14 (BIT-paired APL character)	Interprets 8 bits at a time; returns the APL character (or characters) that would result from using quote-quaq input with the BIT character set on each 8 bits in the record or variable.
type=15 (APL COMPOSITE character)	Interprets 8 bits at a time; returns the APL character (or characters) that would result from using quote-quaq input with the APL COMPOSITE character set on each 8 bits in the record or variable.

Table 5–9 summarizes the effect of converting APL characters to ASCII.

In addition, the following APL characters translate to ASCII as follows:

- $\square CTRL$  translates into the ASCII characters with hexadecimal codes 00 through 1F, and 7F (for Delete).
- $\square NUM$  translates into the ASCII characters 0123456789.
- $1 \downarrow \square ALPHA$  translates into the ASCII characters A – Z.
- $\square ALPHAL$  translates into the ASCII characters a – z.

All other APL characters will signal *DOMAIN ERROR* if you attempt to convert them to ASCII.

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

**Table 5–9 Converting APL Characters to ASCII ( $\square IO \leftrightarrow 0$ )**

□AV Index	APL Character	Equivalent ASCII	□AV Index	APL Character	Equivalent ASCII
32	space	space	75	'	,
33	"	"	77		
34	)	)	80	*	*
35	<	<	81	?	?
37	=	=	84	~	~
38	>	>	89	↑	^
39	]	]	91	←	—
41	^	&	95	-	-
43	÷	%	123	{	{
44	,	,	125	}	}
45	+	+	126	\$	\$
46	.	.	128	‘	‘
47	/	/	155	@	@
58	(	(	156	"	"
59	[	[	157	#	#
60	;	;	158	%	%
61	×	#	159	&	&
62	:	:	187	!	!
63	\	\	188	¤	"
70	-	-	212	^	^

Note that you cannot translate some APL characters to ASCII, and then back to APL, when using pure data mode 6. The following transformations occur.

APL Character Written Out	ASCII Character External to APL	APL Character When Read Back In
#	#	×
&	&	^
"	"	¤

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

APL Character Written Out	ASCII Character External to APL	APL Character When Read Back In
—	—	↑
%	%	÷
^	^	†

The combination of Table 5–10 and Table 5–11 provide information for the conversion from APL to the Digital Multinational Character Set (MCS) for all 256 characters of the multinational set.

**Table 5–10 Converting from APL to Digital Multinational Characters ( $\square IO \leftrightarrow 0$ )**

□AV Index	APL Character	Equivalent MCS	□AV Index	APL Character	Equivalent MCS
32	space	space	77		
34	)	)	80	*	*
35	<	<	81	?	?
37	=	=	84	~	~
38	>	>	95	-	-
39	]	]	123	{	{
44	,	,	125	}	}
45	+	+	126	\$	\$
46	.	.	128	'	'
47	/	/	155	@	@
58	(	(	156	"	"
59	[	[	157	#	#
60	;	;	158	%	%
62	:	:	159	&	&
63	\	\	187	!	!
70	—	—	212	^	^
75	,	,			

In addition, the following APL characters translate to the Digital Multinational Character Set characters as follows:

- CTRL* translates into the MCS characters with hexadecimal codes 00 through 1F, and 7F (for Delete).

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

- $\square NUM$  translates into the MCS characters 0123456789.
- $1+\square ALPHA$  translates into the MCS characters A – Z.
- $\square ALPHAL$  translates into the MCS characters a – z.

Note that the preceding translation is slightly different from the translation of APL characters to ASCII (data type = 6) in Table 5–9.

For more information about the Digital Multinational Character Set, see Table 5–10.

Certain sequences of APL characters of the form *char1* Backspace *char2* translate into single MCS characters, as described in Table 5–11 ( $\square IO \leftrightarrow 0$ ) and MCS index in decimal where NUL  $\leftrightarrow 0$ .

**Table 5–11 Converting from APL to Digital Multinational Characters  
( $\square IO \leftrightarrow 0$ )**

char1 $\square AV$ Index	char1 APL Char	char2 $\square AV$ Index	char2 APL Char	DEC MCS Index	Equivalent DEC MCS Char
0	NUL	127	DEL	128	unused
1	SOH	127	DEL	129	unused
2	STX	127	DEL	130	unused
3	ETX	127	DEL	131	unused
4	EOT	127	DEL	132	IND
5	ENQ	127	DEL	133	NEL
6	ACK	127	DEL	134	SSA
7	BEL	127	DEL	135	ESA
8	BS	127	DEL	136	HTS
9	HT	127	DEL	137	HTJ
10	LF	127	DEL	138	VTS
11	VT	127	DEL	139	PLD
12	FF	127	DEL	140	PLU
13	CR	127	DEL	141	R1
14	SO	127	DEL	142	SS2
15	SI	127	DEL	143	SS3

(continued on next page)

**Table 5–11 (Cont.) Converting from APL to Digital Multinational Characters  
 $(\square I O \leftrightarrow 0)$**

char1 $\square A V$ Index	char1 APL Char	char2 $\square A V$ Index	char2 APL Char	DEC MCS Index	Equivalent DEC MCS Char	
16	DLE	127	DEL	144	DCS	
17	DC1	127	DEL	145	PU1	
18	DC2	127	DEL	146	PU2	
19	DC3	127	DEL	147	STS	
20	DC4	127	DEL	148	CCH	
21	NAK	127	DEL	149	MW	
22	SYN	127	DEL	150	SPA	
23	ETB	127	DEL	151	EPA	
24	CAN	127	DEL	152	unused	
25	EM	127	DEL	153	unused	
26	SUB	127	DEL	154	unused	
27	ESC	127	DEL	155	CSI	
28	FS	127	DEL	156	ST	
29	GS	127	DEL	157	OSC	
30	RS	127	DEL	158	PM	
31	US	127	DEL	159	APC	
32	space	127	DEL	160	unused	
187	!	187	!	161	¡	inverted !
77	¡	99	¢	162	¢	cent sign
95	-	108	£	163	£	pound sign
126	\$	127	DEL	164	¤	unused
95	-	121	¥	165	¥	yen sign
127	DEL	159	&	166	‘	unused
111	o	115	s	167	§	section sign
111	o	120	x	168	“	currency sign
99	c	111	o	169	©	copyright sign

(continued on next page)

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

**Table 5-11 (Cont.) Converting from APL to Digital Multinational Characters  
( $\square I O \leftrightarrow 0$ )**

char1 $\square A V$ Index	char1 APL Char	char2 $\square A V$ Index	char2 APL Char	DEC MCS Index	Equivalent DEC MCS Char
70	-	97	A	170	ª
35	<	35	<	171	«
44	,	127	DEL	172	¬
95	-	127	DEL	173	-
46	.	127	DEL	174	®
47	/	127	DEL	175	-
48	0	212	^	176	°
45	+	95	-	177	±
50	2	212	^	178	²
51	3	212	^	179	³
52	4	127	DEL	180	›
47	/	117	U	181	µ
112	P	187	!	182	¶
46	.	212	^	183	·
56	8	127	DEL	184	,
49	1	212	^	185	¹
70	-	111	O	186	º
38	>	38	>	187	»
49	1	52	4	188	¼
49	1	50	2	189	½
38	>	127	DEL	190	¾
81	?	81	?	191	¿
97	A	128	'	192	À
75	'	97	A	193	Á

(continued on next page)

**Table 5–11 (Cont.) Converting from APL to Digital Multinational Characters**  
( $\square IO \leftrightarrow 0$ )

char1 $\square AV$ Index	char1 APL Char	char2 $\square AV$ Index	char2 APL Char	DEC MCS Index	Equivalent DEC MCS Char
97	A	212	^	194	Â A circumflex
84	~	97	A	195	Ã A tilde
97	A	156	"	196	Ä A umlaut
80	*	97	A	197	Å A ring
97	A	101	E	198	Æ A E ligature
44	,	99	C	199	Ç C cedilla
101	E	128	'	200	È E grave
75	'	101	E	201	É E acute
101	E	212	^	202	Ê E circumflex
101	E	156	"	203	Ë E umlaut
105	I	128	'	204	Ì I grave
75	'	105	I	205	Í I acute
105	I	212	^	206	Î I circumflex
105	I	156	"	207	Ï I umlaut
112	P	127	DEL	208	Ð unused
84	~	110	N	209	Ñ N tilde
111	O	128	'	210	Ò O grave
75	'	111	O	211	Ó O acute
111	O	212	^	212	Ô O circumflex
84	~	111	O	213	Õ O tilde
111	O	156	"	214	Ö O umlaut
101	E	111	O	215	× O E ligature
47	/	111	O	216	Ø O slash
117	U	128	'	217	Ù U grave
75	'	117	U	218	Ú U acute
117	U	212	^	219	Û U circumflex

(continued on next page)

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

**Table 5–11 (Cont.) Converting from APL to Digital Multinational Characters  
( $\square I O \leftrightarrow 0$ )**

char1 $\square A V$ Index	char1 APL Char	char2 $\square A V$ Index	char2 APL Char	DEC MCS Index	Equivalent DEC MCS Char
117	U	156	"	220	Ü U umlaut
121	Y	156	"	221	Ý Y umlaut
127	DEL	212	^	222	Þ unused
147	S	147	S	223	ß German small sharp s
128	'	129	a	224	à a grave
75	'	129	a	225	á a acute
129	ä	212	^	226	â a circumflex
84	~	129	a	227	ã a tilde
129	ä	156	"	228	ä a umlaut
80	*	129	a	229	å a ring
129	ä	133	e	230	æ a e ligature
44	,	131	c	231	ç c cedilla
128	'	133	e	232	è e grave
75	'	133	e	233	é e acute
133	e	212	^	234	ê e circumflex
133	e	156	"	235	ë e umlaut
128	'	137	i	236	ì i grave
75	'	137	i	237	í i acute
137	i	212	^	238	î i circumflex
137	i	156	"	239	ï i umlaut
127	DEL	144	p	240	ð ð unused
84	~	142	n	241	ñ n tilde
128	'	143	o	242	ò o grave
75	'	143	o	243	ó o acute
143	o	212	^	244	ô o circumflex
84	~	143	o	245	õ o tilde

(continued on next page)

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

**Table 5–11 (Cont.) Converting from APL to Digital Multinational Characters  
( $\square I O \leftrightarrow 0$ )**

char1 $\square A V$ Index	char1 APL Char	char2 $\square A V$ Index	char2 APL Char	DEC MCS Index	Equivalent DEC MCS Char
143	o	156	"	246	ö umlaut
133	e	143	o	247	÷ e ligature
47	/	143	o	248	ø o slash
128	'	149	u	249	ù u grave
75	'	149	u	250	ú u acute
149	u	212	^	251	û u circumflex
149	u	156	"	252	ü u umlaut
153	y	156	"	253	ý y umlaut
84	~	127	DEL	254	ð unused
127	DEL	127	DEL	255	ð unused

The order of *char1* and *char2* is immaterial. For example, A Backspace ' and ' Backspace A both translate to "A acute", MCS character 193.

Every MCS character has a unique translation into APL characters in data type = 11. MCS has 256 characters. The translation for the first 128 characters is shown in Table 5–12. The translation for the second 128 characters is shown in Table 5–11. (Each MCS character whose index is greater than 127 is translated into a 3-character sequence, *char1* Backspace *char2* of APL characters.)

**Table 5–12 Converting from Digital Multinational Character Set to APL Characters  
( $\square I O \leftrightarrow 0$ )**

DEC MCS Index	DEC MCS Char	APL Char	$\square A V$ Index	DEC MCS Index	DEC MCS Char	APL Char	$\square A V$ Index
0	NUL	<i>NUL</i>	0	64	@	@	155
1	SOH	<i>SOH</i>	1	65	A	A	97
2	STX	<i>STX</i>	2	66	B	B	98
3	ETX	<i>ETX</i>	3	67	C	C	99

(continued on next page)

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

**Table 5–12 (Cont.) Converting from Digital Multinational Character Set to APL Characters ( $\square IO \leftrightarrow 0$ )**

DEC MCS Index	DEC MCS Char	APL Char	□AV Index	DEC MCS Index	DEC MCS Char	APL Char	□AV Index
4	EOT	<i>EOT</i>	4	68	D	<i>D</i>	100
5	ENQ	<i>ENQ</i>	5	69	E	<i>E</i>	101
6	ACK	<i>ACK</i>	6	70	F	<i>F</i>	102
7	BEL	<i>BEL</i>	7	71	G	<i>G</i>	103
8	BS	<i>BS</i>	8	72	H	<i>H</i>	104
9	HT	<i>HT</i>	9	73	I	<i>I</i>	105
10	LF	<i>LF</i>	10	74	J	<i>J</i>	106
11	VT	<i>VT</i>	11	75	K	<i>K</i>	107
12	FF	<i>FF</i>	12	76	L	<i>L</i>	108
13	CR	<i>CR</i>	13	77	M	<i>M</i>	109
14	SO	<i>SO</i>	14	78	N	<i>N</i>	110
15	SI	<i>SI</i>	15	79	O	<i>O</i>	111
16	DLE	<i>DLE</i>	16	80	P	<i>P</i>	112
17	DC1	<i>DC1</i>	17	81	Q	<i>Q</i>	113
18	DC2	<i>DC2</i>	18	82	R	<i>R</i>	114
19	DC3	<i>DC3</i>	19	83	S	<i>S</i>	115
20	DC4	<i>DC4</i>	20	84	T	<i>T</i>	116
21	NAK	<i>NAK</i>	21	85	U	<i>U</i>	117
22	SYN	<i>SYN</i>	22	86	V	<i>V</i>	118
23	ETB	<i>ETB</i>	23	87	W	<i>W</i>	119
24	CAN	<i>CAN</i>	24	88	X	<i>X</i>	120
25	EM	<i>EM</i>	25	89	Y	<i>Y</i>	121
26	SUB	<i>SUB</i>	26	90	Z	<i>Z</i>	122
27	ESC	<i>ESC</i>	27	91	[	<i>[</i>	59
28	FS	<i>FS</i>	28	92	\	<i>\</i>	63
29	GS	<i>GS</i>	29	93	]	<i>]</i>	39
30	RS	<i>RS</i>	30	94	^	<i>^</i>	212

(continued on next page)

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

**Table 5–12 (Cont.) Converting from Digital Multinational Character Set to APL Characters ( $\square I O \leftrightarrow 0$ )**

DEC MCS Index	DEC MCS Char	APL Char	□AV Index	DEC MCS Index	DEC MCS Char	APL Char	□AV Index
31	US	<i>us</i>	31	95	-	-	70
32	space	<i>sp</i>	32	96	,	,	128
33	!	!	187	97	a	<i>a</i>	129
34	"	"	156	98	b	<i>b</i>	130
35	#	#	157	99	c	<i>c</i>	131
36	\$	\$	126	100	d	<i>d</i>	132
37	%	÷	158	101	e	<i>e</i>	133
38	&	§	159	102	f	<i>f</i>	134
39	,	,	75	103	g	<i>g</i>	135
40	(	(	58	104	h	<i>h</i>	136
41	)	)	34	105	i	<i>i</i>	137
42	*	*	80	106	j	<i>j</i>	138
43	+	+	45	107	k	<i>k</i>	139
44	,	,	44	108	l	<i>l</i>	140
45	-	-	95	109	m	<i>m</i>	141
46	.	.	46	110	n	<i>n</i>	142
47	/	/	47	111	o	<i>o</i>	143
48	0	0	48	112	p	<i>p</i>	144
49	1	1	49	113	q	<i>q</i>	145
50	2	2	50	114	r	<i>r</i>	146
51	3	3	51	115	s	<i>s</i>	147
52	4	4	52	116	t	<i>t</i>	148
53	5	5	53	117	u	<i>u</i>	149
54	6	6	54	118	v	<i>v</i>	150
55	7	7	55	119	w	<i>w</i>	151
56	8	8	56	120	x	<i>x</i>	152
57	9	9	57	121	y	<i>y</i>	153

(continued on next page)

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

**Table 5–12 (Cont.) Converting from Digital Multinational Character Set to APL Characters ( $\square IO \leftrightarrow 0$ )**

DEC MCS Index	DEC MCS Char	APL Char	$\square AV$ Index	DEC MCS Index	DEC MCS Char	APL Char	$\square AV$ Index
58	:	:	62	122	z	z	154
59	;	;	60	123	{	{	123
60	<	<	35	124			77
61	=	=	37	125	}	}	125
62	>	>	38	126	~	~	84
63	?	?	81	127	DEL	DEL	127

In the following example, the user attempts to write the integer value 60 as each of the external data types. APL signals *DOMAIN ERROR* when the user tries to write 60 as a Boolean or character value.

Reading the records back in as Boolean values indicates how they were stored (note that all records must be at least 8 bits, so type 2 may have to be padded with 0s).

Then, the example shows what happens when the records are using each of the external data types.

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

```
QASS '1 TEST/RF'
1
 60@1 1
 60@1 2
15 DOMAIN ERROR
 60@1 2
  ^
 60@1 3
 60@1 4
 60@1 5
15 DOMAIN ERROR (ILLEGAL DATA TYPE CONVERSION)
 60@1 5
  ^
 60@1 6
15 DOMAIN ERROR (ILLEGAL DATA TYPE CONVERSION)
 60@1 6
  ^
 60@1 7
 60@1 8
 60@1 9
 60@1 10
 60@1 11
15 DOMAIN ERROR (ILLEGAL DATA TYPE CONVERSION)
 60@1 11
  ^
 60@1 12
15 DOMAIN ERROR (ILLEGAL DATA TYPE CONVERSION)
 60@1 12
  ^
 60@1 13
15 DOMAIN ERROR (ILLEGAL DATA TYPE CONVERSION)
 60@1 13
  ^
 60@1 14
15 DOMAIN ERROR (ILLEGAL DATA TYPE CONVERSION)
 60@1 14
  ^
 60@1 15
15 DOMAIN ERROR (ILLEGAL DATA TYPE CONVERSION)
 60@1 15
  ^
 60@1 16
15 DOMAIN ERROR (INVALID EXTERNAL DATA TYPE)
 60@1 16
  ^
```

## **VAX APL Input and Output**

### **5.3 Advanced I/O Techniques**

## VAX APL Input and Output 5.3 Advanced I/O Techniques

## VAX APL Input and Output

### 5.3 Advanced I/O Techniques

Numeric byte data is stored as 8-bit values, so you can read it back as characters. For example:

```
□ASS '1 TEST/RF'
1
  65 66 67 68 □[11]1 7
  □[11]1 7
65 66 67 68
  □[11]1 5      □AS APL VALUES
a1n1
  □ [11]1 6      □AS ASCII VALUES
ABCD
  □[11]1 2      □BOOLEAN VECTOR EQUIVALENT IS:
1 0 0 0 0 0 1 0 0 1 0 0 0 0 1 0 1 1 0 0 0 0 1 0 0 0 1 0 0 0 1 0
  □[11]1 1      □INTEGER INTERPRETS AS ONE VALUE
1145258561
```

When you read ASCII values (type 6) as APL characters (type 5), you get the character produced in the equivalent position on the ASCII and APL (key-paired) keyboards. For example, when the keyboard is shifted and you type 3, the ASCII keyboard produces the number sign (#), and the APL (key-paired) keyboard produces the less-than symbol (<).

```
□ASS '1 TEST/RF'
1
  !x $ ÷ ↑ ^ *! □ [21]1 6
  □[21]1 6      □ASCII
× $ ÷ ↑ ^ *
  □[21]1 5      □APL EQUIVALENTS
< ≤ = ≥ > ≠
```

# 6

## Calling External Routines

VAX APL allows you to call external routines (not written in APL) from within the APL environment. You can call library routines and routines written in FORTRAN, C, BLISS, PL/I, PASCAL, and other languages that support the VAX Procedure Calling and Condition Handling Standard. You cannot call VMS system service routines directly since they do not reside in a shared image.

You cannot call external routines that have extensive environments other than VMS. Routines written in APL, LISP, and interpreted BASIC are examples of such routines. To communicate with routines of this type, you can use a VMS subprocess and the mailbox facility.

Note that you cannot call VAX APL routines from programs written in other VMS software languages.

Calling external routines from APL requires three steps:

1. Write a routine and link it into a VMS shared image (see Section 6.1).
2. Define the external routine to APL (within APL) with dyadic  $\Box MAP$  (see Section 6.2.1). After using dyadic  $\Box MAP$ , you can use monadic  $\Box MAP$  to query for a summary of the definitions that have been associated between the external routine and APL (see Section 6.2.2).
3. From APL, call the external routine (see Section 6.3).

The *VAX MACRO and Instruction Set Reference Manual*, the *Introduction to VMS System Routines*, and the *VMS Run-Time Library Routines Volume* contain details about calling external routines and passing parameters. You should be familiar with these subjects before you use the VAX APL call-out facility. Note that the term procedure is synonymous with the phrase external routine.

## Calling External Routines

### 6.1 Linking a Routine into a VMS Shared Image

#### 6.1 Linking a Routine into a VMS Shared Image

Write an external routine and compile the routine with debugging information so the symbol names and source lines will be available. Note that the /DEBUG and /NOOPTIMIZE qualifiers used only during development.

In the following example, the file named F.FOR contains FORTRAN source code. The function F in F.FOR takes an integer vector and its length as arguments. It adds 1 to each element of the vector that is less than 100. The value returned by F is the number of elements that were not incremented.

```
$ type f.for
      FUNCTION F (LEN, ARRAY)
      INTEGER F, LEN, ARRAY (LEN), I
      F = 0
      DO 100 I = 1, LEN
          IF (ARRAY (I) .LT. 100) THEN
              ARRAY (I) = ARRAY (I) + 1
          ELSE
              F = F + 1
          ENDIF
100      CONTINUE
      RETURN
      END)
```

The FORTRAN function is compiled with debugging information so the symbol names and source lines will be available during debugging. It is compiled without optimizations since FORTRAN optimizations invalidate certain debugging information.

```
$ fortran/debug/nooptimize f
```

The preceding compilation creates the file named F.OBJ.

Link the object module into a shareable image. Use the UNIVERSAL linker option to specify the entry points that will be available to the APL call-out facility.

Continuing the example, the FORTRAN function is linked into a shared image with debug support.

```
$ link/sharable=fshr/debug f,sys$input:/options universal=f
```

The preceding LINK command creates the file named FSHR.EXE.

Define a logical name that refers to the shared image.

```
$ define f_image user$:[user]fshr
```

The default file extension for external routine images is .EXE.

## 6.2 Mapping the Routine into APL

Use the dyadic  $\square MAP$  system function to define an external routine to APL. Once a routine is defined in a workspace, the workspace can be saved, loaded, or copied, and the definition for the routine remains intact. Each time you  $\square LOAD$  or  $\square COPY$  a workspace and then invoke an external routine, the shared image that contains the external routine is loaded (providing the shared image exists), arguments (if any) are passed, and the routine is executed. After the first call to an external routine, subsequent calls do not require a reloading of the shared image (thus reducing the amount of time required to invoke the external routine).

The monadic  $\square MAP$  system function returns an operation header that provides information on the current definition associated with an external routine.

### 6.2.1 Dyadic Map

Use the following form:

*func-res* $\leftarrow \square [ext-rout-res/attrib] func-name [arg/attrib] \square MAP image-def$

#### ***func-res***

is the result of dyadic  $\square MAP$ , and specifies the name of the function that has just been defined.

#### ***ext-rout-res/attrib***

if included, specifies that the external function returns a result. Note that the result must be a scalar. The result attributes specify the type of the result in the from of  $/TYPE:vms-data-type$  and must be one of the external data types listed in Table 6–1 (excluding  $/TYPE:Z$ ).

Do not specify the  $/MECHANISM$  attribute for the result of an external routine. APL determines the mechanism by the value specified for  $/TYPE$ .

#### ***func-name***

specifies the name you want APL to associate with the shared image entry point. The function-name, used to call the external routine as if it were a user-defined operation has a name class value of 3. Dyadic  $\square MAP$  signals *DOMAIN ERROR (NAME IN USE)* if the function-name is the same name as an existing label, variable, or group, or if it is the same name as an existing operation that is pendent or suspended. If an operation already exists in your workspace with the same name, and it is not pendent or suspended,  $\square MAP$  replaces it.

## Calling External Routines

### 6.2 Mapping the Routine into APL

#### *arg/attrib*

specifies the names of the function's formal parameters. These names are similar to the dummy arguments of a user-defined operation; they are placeholders only, and you specify the actual values for these parameters when you invoke the function.

The attributes for each of the arguments specify the kind of access that the external routine has to the parameter (either read, write or both), the data type of the parameter, and the passing mechanism used to send the parameter between APL and the external routine.

The possible forms for the attributes are as follows:

```
/ACCESS: [IN | INOUT | OUT]
/TYPE: vms-data-type
/MECHANISM: [IMMEDIATE | REFERENCE | DESCRIPTOR]
```

#### *image-definition*

specifies the name of the VMS shared image. Use either the VMS logical name, or specify a file name, not the complete file specification.

Optional qualifiers include the following:

- /ENTRY to specify the name of the starting address of the executable code in the shared image. The default entry point is the same as *function-name*.
- /VALUE to specify the name of a global constant, a 32-bit signed longword) in the shared image. When you specify /VALUE, then the *function-header* must specify a niladic function that returns a value with a return type of *L* (for example, 'Z/TYP:L←F'). If /VALUE is specified without a value, APL assumes that the name of the global constant is the same as *function-name*.

In the following example, the external routine result is Z, and the function-name, F has two arguments for passing the routine parameters, A and B.

```
$apl/silent/t=d
      X←'Z/TYP:L←F A/TYP:L/MECH:REF'
      X←X,'B/TYP:L/MECH:REF/ACC:INOUT'
      X ⌊MAP 'F_IMAGE'
F
```

## Calling External Routines

### 6.2 Mapping the Routine into APL

#### 6.2.2 Monadic Map

The monadic form of  $\Box MAP$  returns information on the current definition associated with an external routine when used in the following form.

$\Box MAP func-name$

If *func-name* is empty, the result is an empty character vector. If the value of *func-name* does not name an external routine, APL signals *DOMAIN ERROR (NOT AN EXTERNAL FUNCTION)*.

APL returns an operation header (*ext-rout-def*). This is the same header that dyadic  $\Box MAP$  uses when you successfully define the external routine to APL.

**Table 6-1 Characteristics of External Data Types**

External Type	Type Name	DEFAULT result /MECHANISM	Length in Bytes
Z	Unspecified	N/S	
BU	Byte Logical	IMM	1
WU	Word Logical	IMM	2
LU	Longword Logical	IMM	4
QU	Quadword Logical	N/S	
OU	Octaword Logical	N/S	
B	Byte Integer	IMM	1
W	Word Integer	IMM	2
L	Longword Integer	IMM	4
Q	Quadword Integer	N/S	
O	Octaword Integer	N/S	
F	F_floating	IMM	4
D	D_floating	IMM	8
G	G_floating	IMM	8
H	H_floating	REF	16

---

**Key to Default result /MECHANISM**

N/S—not supported  
IMM—by value  
REF—by reference  
DES—by description

(continued on next page)

## Calling External Routines

### 6.2 Mapping the Routine into APL

**Table 6-1 (Cont.) Characteristics of External Data Types**

External Type	Type Name	DEFAULT result /MECHANISM	Length in Bytes
FC	F complex	IMM	8
DC	D complex	REF	16
GC	G complex	REF	16
HC	H complex	REF	32
CIT	COBOL Temp	N/S	
T	8-bit Text	DES	1
VT	Varying Text	REF	1
NU	Numeric String	DES	1
NL	Left Sign String	DES	1
NLO	Left Overpunch String	DES	1
NR	Right Sign String	DES	1
NRO	Right Overpunch String	DES	1
NZ	Zoned Sign String	DES	1
P	Packed Decimal	N/S	
V	Bit	IMM	1
VU	Bit Unaligned	N/S	
ZI	Instructions	N/S	
ZEM	Entry Mask	N/S	
DSC	Descriptor	N/S	
BPV	Bound Procedure	N/S	
BLV	Bound Label	N/S	
ADT	Date/Time	N/S	
other	DEC or user reserved	N/S	

**Key to Default result /MECHANISM**

N/S—not supported

IMM—by value

REF—by reference

DES—by description

## Calling External Routines

### 6.2 Mapping the Routine into APL

**Table 6–2 Converting Internal Data to External Data Types**

External Type	Boolean	Integer	D_Floating	Character
Z	No conversion	No conversion	No conversion	No conversion
BU	Each 1 or 0 is passed as an 8-bit unsigned byte	If each integer is in the range 0 through 255 inclusive, all are passed as 8-bit unsigned bytes; otherwise, APL signals <i>DOMAIN ERROR</i>	If each D_floating value is equal to a near-integer in the range 0 through 255 inclusive, all are passed as unsigned bytes; otherwise, APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>
WU	Each 1 or 0 is passed as a 16-bit unsigned word	If each integer is in the range 0 through 65535 inclusive, all are passed as 16-bit unsigned words; otherwise, APL signals <i>DOMAIN ERROR</i>	If each D_floating value is equal to a near-integer in the range 0 through 65535 inclusive, all are passed as 16-bit unsigned words; otherwise APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>
LU	Each 1 or 0 is passed as a 32-bit unsigned longword	If each integer is $\geq$ , all are passed as 32-bit unsigned longwords; otherwise, APL signals <i>DOMAIN ERROR</i>	If each D_floating value is equal to a near interger $\geq$ , all are passed as 32-bit unsigned longwords; otherwise, APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>
QU	Not supported	Not supported	Not supported	Not supported
OU	Not supported	Not supported	Not supported	Not supported

(continued on next page)

## Calling External Routines

### 6.2 Mapping the Routine into APL

**Table 6–2 (Cont.) Converting Internal Data to External Data Types**

External Type		Boolean	Integer	D_Floating	Character
B	Each 1 or 0 is passed as an 8-bit signed byte	If each integer is in the range –128 through 127 inclusive, all are passed as 8-bit signed bytes; otherwise, APL signals <i>DOMAIN ERROR</i>	If each D_floating value is equal to a near-integer in the range –128 through 127 inclusive, all are passed as 8-bit signed bytes; otherwise, APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	
W	Each 1 or 0 is passed as a 16-bit signed word	If each integer is in the range –32767 through 32767 inclusive, all are passed as 16-bit signed words; otherwise, APL signals <i>DOMAIN ERROR</i>	If each D_floating value is equal to a near-integer in the range –32767 through 32767 inclusive, all are passed as 16-bit signed words; otherwise, APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	
L	Each 1 or 0 is passed as a 32-bit signed longword	Each integer is passed as a 32-bit signed longword	If each D_floating value is equal to a near-integer, all are passed as 32-bit signed longwords; otherwise, APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	
Q	Not supported	Not supported	Not supported	Not supported	
O	Not supported	Not supported	Not supported	Not supported	
F	Each 1 or 0 is passed as a 32-bit F_floating value	Each integer is rounded as necessary and passed as a 32-bit F_floating value	Each D_floating value is passed as a 32-bit F_floating value	APL signals <i>DOMAIN ERROR</i>	

(continued on next page)

## Calling External Routines

### 6.2 Mapping the Routine into APL

**Table 6–2 (Cont.) Converting Internal Data to External Data Types**

External Type	Boolean	Integer	D_Floating	Character
D	Each 1 or 0 is passed as a 64-bit D_floating value	Each integer is passed as a 64-bit D_floating value	Each D_floating value is passed as a 64-bit D_floating value	APL signals DOMAIN ERROR
G	Each 1 or 0 is passed as a 64-bit G_floating value	Each integer is passed as a 64-bit G_floating value	Each D_floating value is passed as a 64-bit G_floating value	APL signals DOMAIN ERROR
H	Each 1 or 0 is passed as a 128-bit H_floating value	Each integer is passed as a 128-bit H_floating value	Each D_floating value is passed as a 128-bit H_floating value	APL signals DOMAIN ERROR
FC	Each pair of values is treated as the real and the imaginary part of a complex number; each 1 or 0 is passed as a 32-bit F_floating value	Each pair of values is treated as the real and the imaginary part of a complex number; each integer is passed as a 32-bit F_floating value	Each pair of values is treated as the real and the imaginary part of a complex number; each D_floating value is passed as a 32-bit F_floating value	APL signals DOMAIN ERROR
DC	Each pair of values is treated as the real and the imaginary part of a complex number; each 1 or 0 is passed as a 64-bit D_floating value	Each pair of values is treated as the real and the imaginary part of a complex number; each integer is passed as a 64-bit D_floating value	Each pair of values is treated as the real and the imaginary part of a complex number; each D_floating value is passed as a 64-bit D_floating value	APL signals DOMAIN ERROR

(continued on next page)

## Calling External Routines

### 6.2 Mapping the Routine into APL

**Table 6–2 (Cont.) Converting Internal Data to External Data Types**

External Type	Boolean	Integer	D_Floating	Character
GC	Each pair of values is treated as the real and the imaginary part of a complex number; each 1 or 0 is passed as a 64-bit G_floating value	Each pair of values is treated as the real and the imaginary part of a complex number; each integer is passed as a 64-bit G_floating value	Each pair of values is treated as the real and the imaginary part of a complex number; each G_floating value is passed as a 64-bit G_floating value	APL signals <i>DOMAIN ERROR</i>
HC	Each pair of values is treated as the real and the imaginary part of a complex number; each 1 or 0 is passed as a 128-bit H_floating value	Each pair of values is treated as the real and the imaginary part of a complex number; each integer is passed as a 128-bit H_floating value	Each pair of values is treated as the real and the imaginary part of a complex number; each D_floating value is passed as a 128-bit H_floating value	APL signals <i>DOMAIN ERROR</i>
CIT	Not supported	Not supported	Not supported	Not supported
T	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	Each APL character is translated to its ASCII equivalent if possible (see Table 5–9); otherwise, APL signals <i>DOMAIN ERROR</i>

(continued on next page)

## Calling External Routines

### 6.2 Mapping the Routine into APL

**Table 6–2 (Cont.) Converting Internal Data to External Data Types**

External Type	Boolean	Integer	D_Floating	Character
VT	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	Each APL character is translated to its ASCII equivalent if possible (see Table 5–9) with a 16-bit length field preceding the string; otherwise, APL signals <i>DOMAIN ERROR</i>
NU	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	No conversion
NL	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	No conversion
NLO	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	No conversion
NR	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	No conversion
NRO	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	No conversion
NZ	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>	No conversion
P	Not supported	Not supported	Not supported	Not supported
V	Each 1 or 0 is passed as a 1-bit value	If each integer is equal to 0 or 1 (near-integer), all are passed as 1-bit values; otherwise APL signals <i>DOMAIN ERROR</i>	If each D_floating value is equal to 0 or 1 (near-integer), all are passed as 1-bit values; otherwise APL signals <i>DOMAIN ERROR</i>	APL signals <i>DOMAIN ERROR</i>
VU	Not supported	Not supported	Not supported	Not supported
ZI	Not supported	Not supported	Not supported	Not supported
ZEM	Not supported	Not supported	Not supported	Not supported
DSC	Not supported	Not supported	Not supported	Not supported
BPV	Not supported	Not supported	Not supported	Not supported

(continued on next page)

## Calling External Routines

### 6.2 Mapping the Routine into APL

**Table 6–2 (Cont.) Converting Internal Data to External Data Types**

External Type	Boolean	Integer	D_Floating	Character
BLV	Not supported	Not supported	Not supported	Not supported
ADT	Not supported	Not supported	Not supported	Not supported
any other	Not supported	Not supported	Not supported	Not supported

Note that external data types passed to external routines are the same as pure data types used for input and output:

External Type	Pure Data Type
BU	7
LU or L	1
W	10
F	3
D	4
G	8
H	9
T	6
V	2

**Table 6–3 Converting External Data Types to Internal Data**

External Type	Internal Type	Entering Workspace
Z	Not supported	
BU	Integer	Converts each 8-bit unsigned byte to 32-bit integer format
WU	Integer	Converts each 16-bit unsigned word to 32-bit integer format
LU	Interger	Treats each 32-bit longword as 32-bit integer format (signed)

(continued on next page)

## Calling External Routines

### 6.2 Mapping the Routine into APL

**Table 6–3 (Cont.) Converting External Data Types to Internal Data**

External Type	Internal Type	Entering Workspace
QU	Not supported	
OU	Not supported	
B	Integer	Converts each 8-bit signed byte to 32-bit integer format
W	Integer	Converts each 16-bit signed word to 32-bit integer format
L	Integer	Treats each 32-bit signed longword as 32-bit integer format (signed)
Q	Not supported	
O	Not supported	
F	D_floating	Converts each 32-bit F_floating value to 64-bit D_floating format
D	D_floating	Treats each 64-bit quadword as 64-bit D_floating format (reserved operand checking is performed to detect illegal floating point format)
G	D_floating	Converts each 64-bit G_floating value to 64-bit D_floating format (overflow may occur)
H	D_floating	Converts each 128-bit H_floating value to 64-bit D_floating format (overflow may occur)
FC	D_floating	Treats each complex number as 2 values; converts each 32-bit F_floating value to 64-bit D_floating format
DC	D_floating	Treats each complex number as 2 values; treats each 64-bit quadword as 64-bit D_floating format (reserved operand checking is performed to detect illegal floating-point format)
GC	D_floating	Treats each complex number as 2 values; converts each 64-bit G_floating value to 64-bit D_floating format (overflow may occur)
HC	D_floating	Treats each complex number as 2 values; converts each 128-bit H_floating value to 64-bit D_floating format (overflow may occur)
CIT	Not supported	

(continued on next page)

## Calling External Routines

### 6.2 Mapping the Routine into APL

**Table 6–3 (Cont.) Converting External Data Types to Internal Data**

External Type	Internal Type	Entering Workspace
T	APL characters	Converts each 8-bit character to the APL character that would result from using quad del input with the TTY character set
VT	APL characters	Converts each 8-bit character to the APL character that would result from using quad del input with the TTY character set, skipping the 16-bit length field that precedes the string
NU	APL characters	No conversion
NL	APL characters	No conversion
NLO	APL characters	No conversion
NR	APL characters	No conversion
NRO	APL characters	No conversion
NZ	APL characters	No conversion
P	Not supported	
V	Boolean	Treats each bit as 1-bit Boolean format
VU	Not supported	
ZI	Not supported	
ZEM	Not supported	
DSC	Not supported	
BPV	Not supported	
BLV	Not supported	
ADT	Not supported	
any other	Not supported	

The VMS documentation set introduces VMS data usages. Table 6–4 shows the relationship between data usages and the APL /TYPE attribute. (Note that NA indicates Not Applicable.)

## Calling External Routines

### 6.2 Mapping the Routine into APL

**Table 6–4 VMS Data Structures**

Data Usage	APL Attribute	Data Usage	APL Attribute
access_bit_names	NA	logical_name	T
access_mode	BU	longword_signed	L
address	NA	longword_unsigned	LU
address_range	NA	mask_byte	BU
arg_list	NA	mask_longword	LU
ast_procedure	NA	mask_quadword	NA
Boolean	V	mask_word	WU
byte_signed	B	null_arg	LU
byte_unsigned	BU	octaword_signed	NA
channel	WU	octaword_unsigned	NA
char_string	T	page_protection	LU
complex_number	FC	procedure	NA
	DC	process_id	LU
	GC	process_name	T
cond_value	HC	quadword_signed	NA
	LU	quadword_unsigned	NA
	NA	rights_holder	NA
context	NA	rights_id	LU
date_time	NA	rab	NA
device_name	T	section_id	NA
ef_cluster_name	T	section_name	T
ef_number	LU	system_access_id	NA
exit_handler_block	NA	time_name	T
fab	NA	uic	LU
floating_point	WU	user_arg	LU
	F	varying_arg	NA
	D	vector_byte_signed	B
function_code	G	vector_byte_unsigned	BU
	H	vector_longword_signed	L
	NA		

(continued on next page)

## Calling External Routines

### 6.2 Mapping the Routine into APL

**Table 6-4 (Cont.) VMS Data Structures**

Data Usage	APL Attribute	Data Usage	APL Attribute
io_status_block	NA	vector_longword_unsigned	LU
item_list_1	NA	vector_quadword_signed	NA
item_list_2	NA	vector_quadword_unsigned	NA
item_list_3	NA	vector_word_signed	W
item_quota_list	NA	vector_word_unsigned	WU
lock_id	LU	word_signed	W
lock_status_block	NA	word_unsigned	WU
lock_value_block	NA		

### 6.3 Invoking External Routines

Once an external routine is defined to APL using dyadic  $\Box MAP$ , you can invoke the routine as if it were a user-defined operation: specify the function name (as defined in the left argument of dyadic  $\Box MAP$ ) and any argument in the following form:

*function-name* **[[arg]]**

Unlike a user-defined function, the right argument of the external routine may take an argument list. In this case, use the following form:

*function-name* **(arg1; arg2; . . . ; argn)**

The argument list is delimited by semicolons and must be surrounded by parentheses. Each element of the list must be a simple, homogeneous array.

Alternatively, the right argument can be specified as a strand of values. In this case, each item of the strand must be a simple, homogeneous array.

*function-name* **arg1 arg2 . . . argn**

When you specify a semicolon list, you do not have to specify an argument for each of the formal parameters. However, you must delimit the locations of any missing arguments. (Because it is not possible to delimit the locations of missing arguments with a strand right argument, the strand form cannot be used unless all arguments are specified.) For example, if there are three arguments, and you want to leave the second argument empty, use the following form:

*function-name* **(arg1 ; ; arg3)**

Empty lists are also allowed:

*function-name () function-name ()*

The argument list can contain only missing arguments when the access defined for the corresponding formal parameters is *IN*; you must supply a value for each parameter that is defined as *OUT* or *INOUT*. In all cases where you supply a value, the value must already be defined to APL; you cannot supply an undefined variable.

If, for parameters defined as *OUT* or *INOUT*, you specify a variable that does not currently have a value, APL assumes that it will have a scalar value after it is modified by the external routine; APL then determines the appropriate passing mechanism based on the value you specified for */TYPE*. If you specify a variable that currently has a value, APL passes the address of the value.

When the argument list contains missing arguments, the passing mechanism defined for the corresponding formal parameters must be either *REFERENCE* or *DESCRIPTOR*. This is necessary because APL passes a value of 0 when it encounters an empty argument; if you specify */MECHANISM:IMMEDIATE*, the external routine cannot determine whether 0 is the value of the argument or the indicator of a missing argument.

When the argument list contains a nonscalar value whose corresponding formal parameter is defined as */MECHANISM:REFERENCE*, APL does not have control of the length of the value because it is passing the address and not the value itself. In this case, the external routine must expect the length of the value contained in the address.

Note that for any arguments to be passed from an external routine to APL, the access defined for the corresponding formal parameters must be either *OUT* or *INOUT*.

APL treats external routines as locked operations. However, you can erase an external function with the *)ERASE* system command, and you can replace an external routine definition with an APL function definition by using the *FX* system function.

Note that any changes made to your terminal characteristics by the external function remain in effect when the function completes execution. (For example, these terminal characteristics include the print width, broadcast, or line editing.)

When an external routine signals an error to APL, APL signals *SIGNAL FROM EXTERNAL ROUTINE xxx*, where *xxx* is the message sent from the external routine.

## Calling External Routines

### 6.4 Debugging External Routines

## 6.4 Debugging External Routines

VAX APL features include support for the VMS Debugger.

Also, you can use `□TRACE`, `□STOP`, and `□MONITOR` with external routines (with line numbers 0, 1, and 0, respectively).

The following is an example of debugging an external routine called from APL. The example uses the following system functions:

- `□STOP` to invoke the VMS Debugger
- `□TRACE` to trace the value returned by the external routine
- `□MONITOR` to time the execution of the external routine

For more information on `□TRACE`, `□STOP`, and `□MONITOR`, see the *VAX APL Reference Manual*.

Continuing the example from the previous sections, debug F from inside APL.

```
1 □STOP 'F'          ;SET BREAKPOINT ON FIRST LINE OF F
0
      X←5
      F (5;'X')
11 VALUE ERROR
      F (5;'X')
      ^
```

Setting module F

break at routine F

```
1:           FUNCTION F (LEN, ARRAY)
DBG> set language fortran
DBG> step
stepped to F\%LINE 2
2:           INTEGER F, LEN, ARRAY (LEN), I
DBG> step
stepped to F\%LINE 4
4:           F = 0
DBG> examine len
F\LEN: 5
DBG> examine array
F\ARRAY
(1):      1
(2):      2
(3):      3
(4):      4
(5):      5
DBG> go
```

## Calling External Routines

### 6.4 Debugging External Routines

Trace the result of F:

```
V FF N ; X
[1] L: X←N
[2]   X[0]M 2|X]←200
[3]   ⌊SINK←F (N; 'X')
[4]   +(0<N←N-1)/L
[5] V
  0 ⌈TRACE 'F'
1
  FF 4
F[0] 2
F[0] 2
F[0] 1
F[0] 1
  ' ' ⌈TRACE 'F'
```

Time the execution of F:

```
0 ⌈MONITOR 'F'
1
  FF 1000
  ⌈MONITOR 'F'
0 1000 840
```

## 6.5 Examples of Calls to External Routines

The following subsections describe possible uses of calls to external routines.

### 6.5.1 Example 1: Calling RTL MTH\$DACOSD

The VMS Run-time Library (RTL) routine MTH\$DACOSD takes a D\_FLOATING point number as the cosine of an angle and returns the angle, in degrees, as a D\_FLOATING point value. (See the *VMS Run-Time Library Routines Volume* for more information.) This routine can be called from VAX APL with the following statements:

```
A← 'Z/TYP:D ← DACOSD A/TYP:D/MECH:REF'
B← 'MTHRTL/ENTRY:MTH$DACOSD'
A ⌈MAP B
DACOSD
  ⌈MAP 'DACOSD'
Z/TYPE:D/MECHANISM:IMMEDIATE←
DACOSD/IMAGE:MTHRTL/ENTRY:MTH$DACOSD
A/ACCESS:IN/TYPE:D/MECHANISM:REFERENCE
  DACOSD 0.33333
70.52898194
```

## Calling External Routines

### 6.5 Examples of Calls to External Routines

#### 6.5.2 Example 2: Calling RTL LIB\$ERASE\_PAGE

The VMS RTL routine LIB\$ERASE\_PAGE erases a video screen from the current cursor position to the end of the screen. It takes two optional 16-bit integer parameters that specify the line number and the column number at which to position the cursor before doing the erase. By positioning the cursor at line number 1 and column number 1, the entire screen can be erased. The routine returns a status value as an integer result. This routine can be called from VAX APL with the following statements:

```
A← 'Z/TYP:L ←ERASE_PAGE L/TYP:W/MECH:REF'
B← 'SCRSHR/ENTRY:LIB$ERASE_PAGE'
A ⓥMAP B
ERASE_PAGE
  ⓥMAP 'ERASE_PAGE'
Z/TYPE:L/MECHANISM:IMMEDIATE←
ERASE_PAGE/IMAGE:SCRSHR/ENTRY:LIB$ERASE_PAGE
L/ACCESS:IN/TYPE:W/MECHANISM:REFERENCE
  ⓧCLEAR THE ENTIRE SCREEN
STATUS← ERASE_PAGE 1
```

#### 6.5.3 Example 3: Calling LIB\$PUT\_SCREEN

The VMS RTL routine LIB\$PUT\_SCREEN displays text at a specified cursor location on the video screen. The routine takes up to 4 arguments:

- TEXT—a read-only character string, passed by descriptor, that is the string to display. No carriage return or line feed control characters are inserted.
- LINE-NO—an optional read-only 16-bit integer (word), passed by reference, that specifies the video screen line number at which to display the text. If omitted, the default is the current line number.
- COL-NO—an optional read-only 16-bit integer (word), passed by reference, that specifies the video screen column number at which to display the text. If omitted, the default is the current column number.
- FLAGS—a read-only longword, passed by reference, that specifies terminal characteristics as bits:
  - Bit 0 on means bold
  - Bit 1 on means reverse video
  - Bit 2 on means blinking
  - Bit 3 on means underscored

## 6.5 Examples of Calls to External Routines

The routine returns a status value as an integer. The following example shows various calls to LIB\$PUT\_SCREEN from VAX APL:

```

A← 'Z/TYP:L ← PUT_SCREEN C/TYP:T/MECH:DESC '
A← A, 'L/TYP:W/MECH:REF C/TYP:W/MECH:REF '
A← A, 'F/TYP:W/MECH:REF'
B← 'SCRSHR/ENTRY:LIB$PUT_SCREEN'
A ⊞MAP B
PUT_SCREEN
    ⊞MAP 'PUT_SCREEN'
Z/TYPE:L/MECHANISM:IMMEDIATE-
PUT_SCREEN/IMAGE:SCRSHR/ENTRY:LIB$PUT_SCREEN
C/ACCESS:IN/TYPE:T/MECHANISM:DESCRIPTOR
L/ACCESS:IN/TYPE:W/MECHANISM:REFERENCE
C/ACCESS:IN/TYPE:W/MECHANISM:REFERENCE
F/ACCESS:IN/TYPE:W/MECHANISM:REFERENCE
    STRING ← 'SOME TEXT' , ⊞CTRL [14 11]
    ⊞PUT STRING AT CURRENT CURSOR POSITION
    STATUS← PUT_SCREEN STRING
    ⊞PUT STRING AT LINE 5 COLUMN 10
    STATUS← PUT_SCREEN (STRING ; 5 ; 10)
    ⊞PUT STRING AT LINE 6 COLUMN 10
    ⊞IN REVERSE VIDEO
    STATUS← PUT_SCREEN (STRING ; 6 ; 10 ; 2)

```

### 6.5.4 Example 4: Calling RTL LIB\$GET\_SCREEN

The VMS RTL routine LIB\$GET\_SCREEN reads an input string from the terminal. The routine takes up to 3 arguments:

- INPUT-TEXT—a write-only character string, passed by descriptor, that contains the text read from the terminal.
- PROMPT-STR—an optional read-only character string, passed by descriptor, that contains a prompt to display on the terminal before doing the read.
- OUT-LEN—an optional read-write 16-bit integer (word), passed by reference, that contains the length of the string put into the INPUT-TEXT argument.

The following example shows various calls to LIB\$PUT\_SCREEN from VAX APL:

## Calling External Routines

### 6.5 Examples of Calls to External Routines

```
A← 'Z/TYP:L ← GET SCREEN I/TYP:T/MECH:DESC/ACC:INOUT '
A← A , 'P/TYP:T/MECH:DESC L/TYP:WU/MECH:REF/ACC:INOUT'
A ⊞MAP 'SCRSHR/ENTRY:LIB$GET_SCREEN'

GET_SCREEN
    ⊞MAP 'GET_SCREEN'
Z/TYPE:L/MECHANISM:IMMEDIATE+
GET_SCREEN/IMAGE:SCRSHR/ENTRY:LIB$GET_SCREEN
I/ACCESS:INOUT/TYPE:T/MECHANISM:DESCRIPTOR
P/ACCESS:IN/TYPE:T/MECHANISM:DESCRIPTOR
L/ACCESS:INOUT/TYPE:WU/MECHANISM:REFERENCE
    TEXTLEN ← 80
    INTEXT ← 80 Ⓛ ' '
    PROMPT ← 'ENTER YOU DATA: '
    ⓁTEXT WILL BE READ INTO VARIABLE INTEXT
    ⓁPROMPT WILL APPEAR
    ⓁTHE LENGTH OF THE TEXT READ INTO
    Ⓛ INTEXT WILL BE PUT INTO TEXTLEN
    STATUS ← GET_SCREEN ('INTEXT' ; PROMPT; 'TEXTLEN')
ENTER YOUR DATA: HI THERE
    Ⓛ ← INTEXT
HI THERE
8
TEXTLEN
8
```

#### 6.5.5 Example 5: Calling VMS SORT

The following example shows VAX APL calling the VMS SORT Utility. For details on this utility, see the *VMS Sort/Merge Utility Manual* or the *VMS Utility Routines Manual*.

```
Z← 'X/TYP:L ← FILES '
Z← Z, 'A/TYP:T/MECH:DESC'           ⓁINPUT FILE
Z← Z, 'B/TYP:T/MECH:DESC'           ⓁOUTPUT FILE
Z ⊞MAP 'SORTSHR/ENTRY:SOR$PASS+FILES'

FILES
Z← 'X/TYP:L← INIT '
Z← Z, 'A/TYP:W/MECH:REF'           ⓁKEY COUNT, TYPE, ASC, START, LEN
Z← Z, 'B/TYP:W/MECH:REF'           ⓁLONGEST REC LEN
Z← Z, 'C/TYP:W/MECH:REF'           ⓁOPTION FLAGS
Z← Z, 'D/TYP:W/MECH:REF'           ⓁNUMBER OF WORK FILES
Z← Z, 'E/TYP:W/MECH:REF'           ⓁSORT TYPE
Z ⊞MAP 'SORTSHR/ENTRY:SOR$INIT+SORT'

INIT
Z← 'X/TYPE:L ← VMSSORT'
Z ⊞MAP 'SORTSHR/ENTRY:SOR$SORT+MERGE'

VMSSORT
Z← 'X/TYPE:L ← END'
Z ⊞MAP 'SORTSHR/ENTRY:SOR$END+SORT'

END
```

## Calling External Routines

### 6.5 Examples of Calls to External Routines

```
    □MAP 'FILES'
X/TYPE:L/MECHANISM:IMMEDIATE-
FILES/IMAGE:SORSHR/ENTRY:SOR$PASS_FILES
A/ACCESS:IN/TYPE:T/MECHANISM:DESCRIPTOR
B/ACCESS:IN/TYPE:T/MECHANISM:DESCRIPTOR
    □MAP 'INIT'
X/TYPE:L/MECHANISM:IMMEDIATE-
INIT/IMAGE:SORSHR/ENTRY:SOR$INIT_SORT
A/ACCESS:IN/TYPE:W/MECHANISM:REFERENCE
B/ACCESS:IN/TYPE:W/MECHANISM:REFERENCE
C/ACCESS:IN/TYPE:W/MECHANISM:REFERENCE
D/ACCESS:IN/TYPE:W/MECHANISM:REFERENCE
E/ACCESS:IN/TYPE:W/MECHANISM:REFERENCE
    □MAP 'VMSSORT'
X/TYPE:L/MECHANISM:IMMEDIATE-
VMSSORT/IMAGE:SORSHR/ENTRY:SOR$SORT_MERGE
    □MAP 'END'
X/TYPE:L/MECHANISM:IMMEDIATE-
END/IMAGE:SORSHR/ENTRY:SOR$END_SORT
    ▽ CREATE FILE ; I
[1]  □SINK ← □ASS '1 ',FILE,'/AS/OPEN:NEW'
[2]  I+27
[3]  L:(80p□ALPHA[I]) ⊕ [2]1
[4]  → (2 ≤ I←I-1)/L
[5]  □DAS 1
[6]  ▽
    ▽ DISPLAY FILE
[1]  □TRAP ← 'EOF'
[2]  □SINK←□ASS '1 ',FILE,'/AS/SIGNAL/OPEN:OLD'
[3]  L:@[2]1
[4]  → L
[5]  EOF:□ERROR
[6]  □DAS 1
[7]  ▽
CREATE 'DESCEND.AAS'
DISPLAY 'DESCEND.AAS'
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ . .
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY . .
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX . .
.
.
.
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD . .
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC . .
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB . .
AAAAAAAAAAAAAAABBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB . .
68 END OF FILE ENCOUNTERED
DISPLAY[3] L:@[2]1
^
FILES ('DESCEND.AAS' ; 'ASCEND.AAS')
```

## Calling External Routines

## **6.5 Examples of Calls to External Routines**

```
ONE SORT KEY: CHARACTER ASCENDING
STARTING AT CHAR 1 FOR 80 CHARS
NO MAXIMUM RECORD LENGTH
NO OPTIONS
USE 3 WORK FILES
STABLE TYPE SORT
INIT (5p1 1 0 1 80 ; ; ; 3 ; 1)
1
VMSSORT
1
END
1
DISPLAY 'ASCEND.AAS'
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA . .
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB . .
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC . .
.
.
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX . .
YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY . .
ZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZZ . .
68 END OF FILE ENCOUNTERED
DISPLAY[3] L:@[2]1
^
```

### **6.5.6 Example 6: Calling VAX FORTRAN**

The following example shows VAX APL calling a VAX FORTRAN function. The FORTRAN function RHYME expects a character argument, which it modifies. Then, the function returns the length of the new character value as the function value. Assume the function is contained in a file named T.FOR:

```

$type t.for
FUNCTION RHYME (C)
INTEGER RHYME
CHARACTER*(*) C

INTEGER L

L = LEN (C)
IF (L .GE. 22 .AND. C(1:9) .EQ. '.RO.RO.RO') THEN
  C = 'GENTLY DOWN THE STREAM'
  RHYME = 22
ELSEIF (L .GE. 19 .AND. C(1:4) .EQ. '4.RO') THEN
  C = 'LIFE IS BUT A DREAM'
  RHYME = 19
ELSE
  C = 'WHAT ?'
  RHYME = 6
ENDIF

```

## Calling External Routines

### 6.5 Examples of Calls to External Routines

```
RETURN  
END
```

The following commands create a shared image containing the FORTRAN function RHYME:

```
$ fortran t.for/object=t.obj/optimize  
$ link/shar=shrt t.obj,sys$input:/options universal=rhyme
```

The APL function *F* sets up and calls the FORTRAN function RHYME. Note that a job-wide logical name must be used to point to the shared image DEVDIR:SHRT containing RHYME.

```
∇ F ;L1;L2;X;Z  
[1] ⎕SINK ← ⎕XQ ')DO DEFINE/JOB XX DEVDIR:SHRT'  
[2] ⎕XQ 'DO SHOW LOGICAL/JOB XX'  
[3] 'Z/TYP:L ← RHYME A/TYP:T/MECH:DESC/ACC:INOUT' ⎕MAP 'XX'  
[4] L1← ('opp YOUR BOAT' ⎕COQ 0 12) ⎕CIQ 0 11  
[5] L1 ⎕OM 34≠⎕CTRL ⌊ L1] ← ''  
[6] L2← ('3p'MERRILY,''' ⎕COQ 0 12) ⎕CIQ 0 11  
[7] L2← ⎕OM 34≠⎕CTRL ⌊ L2] ← ''  
[8] L1  
[9] X←22 ⎨ L1 ⋄ Z←RHYME 'X'  
[10] )ZSX  
[11] )L2  
[12] )X ← 10$L2 ⋄ Z←RHYME 'X'  
[13] )ZSX  
[14] )V  
F  
AXXA = ⎕DEVDIR:SHRTA (LNMSJOB+80E14CB0)  
  
RHYME  
.RO.RO.RO YOUR BOAT  
GENTLY DOWN THE STREAM  
3.RO'MERRILY,  
LIFE IS BUT A DREAM
```

#### 6.5.7 Example 7: Calling VAX DATATRIEVE

The following examples establish definitions in VAX DATATRIEVE, use ⎕MAP to define the DATATRIEVE external routines to APL, create user-defined functions that interact with the DATATRIEVE routines, and then show APL calling DATATRIEVE.

Prior to executing APL, the following must be established in DATATRIEVE:

1. The following DATATRIEVE commands define the record structure that is used for both the PARTS files and the communications port:

## Calling External Routines

### 6.5 Examples of Calls to External Routines

```
DTR> define record parts_rec using
DFN> 01 parts_rec.
DFN>      03 partno  pic      9(5)    comp.
DFN>      03 desc    pic      x(20).
DFN>      03 value   pic      9(6)_v99      comp-2.
DFN> ;
```

2. The following commands define the domain for the PARTS file and establish an empty ISAM file with a single index key, the part number, represented by PARTNO:

```
DTR> define domain parts using parts_rec on parts.dat;
DTR> define file for parts key = partno;
```

3. The following command defines the communications port that is used to pass data records between APL and DATATRIEVE:

```
DTR> define port tport using parts_rec;
```

4. Before executing APL, you should define the logical name CDD\$DEFAULT to be the DATATRIEVE directory that contains the domains and record definitions that you have just created. This can be done as follows:

```
$ define cdd$default cdd$top.subdirectorypath
```

Where *subdirectorypath* represents the subdirectory path names that identify where the definitions reside. For example:

```
$ define cdd$default cdd$top.apl
```

5. You can now execute APL and use the callable interface detailed below to read and write records using DATATRIEVE.

The following example describes the external functions that must be defined to APL in order to use DATATRIEVE:

1. DTR\$INIT—initializes the interface to callable DATATRIEVE:

```
MAP 'DTR$INIT'
STATUS/TYPE:L/MECHANISM:IMMEDIATE+
DTR$INIT/IMAGE:DTR$SHR/ENTRY:DTR$INIT
DAB/ACCESS:INOUT/TYPE:WU/MECHANISM:REFERENCE
SIZE/ACCESS:IN/TYPE:L/MECHANISM:REFERENCE
MSGBUF/ACCESS:IN/TYPE:T/MECHANISM:DESCRIPTOR
AUXBUF/ACCESS:IN/TYPE:T/MECHANISM:DESCRIPTOR
OPTION/ACCESS:IN/TYPE:L/MECHANISM:REFERENCE
```

## Calling External Routines

### 6.5 Examples of Calls to External Routines

2. DTR\$COMMAND—passes a command to DATATRIEVE:

```
MAP 'DTRCOMMAND'  
STATUS/TYPE:L/MECHANISM:IMMEDIATE<  
    DTRCOMMAND/IMAGE:DTRSHR/ENTRY:DTR$COMMAND  
    DAB/ACCESS:INOUT/TYPE:WU/MECHANISM:REFERENCE  
    CMD/ACCESS:IN/TYPE:WU/MECHANISM:_DESCRIPTOR
```

3. DTR\$GET\_PORT—reads a record from the communications port:

```
MAP 'DTRGET_PORT'  
STATUS/TYPE:L/MECHANISM:IMMEDIATE<  
    DTRGET_PORT/IMAGE:DTRSHR/ENTRY:DTR$GET_PORT  
    DAB/ACCESS:INOUT/TYPE:WU/MECHANISM:REFERENCE  
    RECORD/ACCESS:IN/TYPE:WU/MECHANISM:REFERENCE
```

4. DTR\$PUT\_PORT—writes a record to the communications port:

```
MAP 'DTRPUT_PORT'  
STATUS/TYPE:L/MECHANISM:IMMEDIATE<  
    DTRPUT_PORT/IMAGE:DTRSHR/ENTRY:DTR$PUT_PORT  
    DAB/ACCESS:INOUT/TYPE:WU/MECHANISM:REFERENCE  
    RECORD/ACCESS:IN/TYPE:WU/MECHANISM:REFERENCE
```

5. DTR\$PORT\_EOF—terminates a sequence of records written to the communications port:

```
MAP 'DTRREPORT_EOF'  
STATUS/TYPE:L/MECHANISM:IMMEDIATE<  
    DTRREPORT_EOF/IMAGE:DTRSHR/ENTRY:DTR$PORT_EOF  
    DAB/ACCESS:INOUT/TYPE:WU/MECHANISM:REFERENCE
```

6. DTR\$CONTINUE—scans error codes returned by DATATRIEVE:

```
MAP 'DTRACONTINUE'  
STATUS/TYPE:L/MECHANISM:IMMEDIATE<  
    DTRACONTINUE/IMAGE:DTRSHR/ENTRY:DTR$CONTINUE  
    DAB/ACCESS:INOUT/TYPE:WU/MECHANISM:REFERENCE
```

7. DTR\$FINISH—terminates callable DATATRIEVE:

```
MAP 'DTRAFINISH'  
STATUS/TYPE:L/MECHANISM:IMMEDIATE<  
    DTRAFINISH/IMAGE:DTRSHR/ENTRY:DTR$FINISH  
    DAB/ACCESS:INOUT/TYPE:WU/MECHANISM:REFERENCE
```

## Calling External Routines

### 6.5 Examples of Calls to External Routines

8. DTR\$UNWIND—cancels unused commands:

```
MAP 'DTRAUNWIND'
STATUS/TYPE:L/MECHANISM:IMMEDIATE+
DTRAUNWIND/IMAGE:DTRSHR/ENTRY:DTR$UNWIND
DAB/ACCESS:INOUT/TYPE:WU/MECHANISM:REFERENCE
```

9. LIB\$SYS\_GETMSG—gets a system error message:

```
MAP 'ERRORMESSAGE'
STATUS/TYPE:L/MECHANISM:IMMEDIATE+
ERRORMESSAGE/IMAGE:LIBRTL/ENTRY:LIB$SYS_GETMSG
ERRNO/ACCESS:INOUT/TYPE:L/MECHANISM:REFERENCE
LENGTH/ACCESS:INOUT/TYPE:WU/MECHANISM:REFERENCE
ERRORTEXT/ACCESS:INOUT/TYPE:T/MECHANISM:_DESCRIPTOR
FLAGS/ACCESS:INOUT/TYPE:L/MECHANISM:REFERENCE
OUTBUF/ACCESS:INOUT/TYPE:T/MECHANISM:REFERENCE
```

The following examples describe the APL functions used to invoke the external DATATRIEVE functions:

1. INIT—establishes communications with DATATRIEVE and establishes the domains to be used. In the example that follows, PARTS is a file domain that contains the data to be manipulated. TPORt is a port domain used for passing records between APL and DATATRIEVE.

DAB is a control block used to pass status and other control information between APL and callable DATATRIEVE. It is used as a vector of UNSIGNED WORD values because this allows easy access to the two fields that are important. DAB[2 3] holds the two halves of the condition value and DAB[13] holds the current state of the call interface. The size of the DAB control block is 100 bytes; thus, DAB is defined to be a vector of 50 integers that will be mapped into 50 UNSIGNED WORDS.

The third and subsequent arguments to DTR\$INIT are optional and are not used in this example. The size (in pages) to be allocated for the DATATRIEVE stack is 100.

```
VINIT ;Z
[1] DAB ← 50 ♂ 0
[2] Z← DTRAINIT ('DAB';100;;)
[3] Z← DTRACOMMAND ('DAB';'READY PARTS WRITE; ')
READY TPORt WRITE;!
[4] CONT
[5] V
```

## Calling External Routines

### 6.5 Examples of Calls to External Routines

2. *COPYVIN*—instructs DATATRIEVE to send all records of PARTS to the port.

```
VCOPYVIN 'Z
Z←DTRACOMMAND ('DAB';'FOR PARTS STORE TPORT USING PARTS_REC )
= PARTS_REC;
V
```

3. *READVALL*—reads the records sequentially from the port:

The DATATRIEVE record definition for the PARTS file specifies that the VALUE field has 2 decimal places. DATATRIEVE will scale the data stored into VALUE by 2 decimal places when the record is put into the file by DATATRIEVE. The data is not scaled down again when the record is read back through the callable interface, so this function divides the VALUE field passed by DATATRIEVE by 100.

```
VREADAALL;Z;RECORD;PARTNO;DESC;VALUE
[1] RECORD+8p0
[2] L0:Z ← DTRAGET_PORT ('DAB';'RECORD')
[3] → (1≠ Z) / L1
[4] PARTNO+RECORD[1]
[5] DESC ← 1 20 p RECORD [1+15] 0CIQ 0 6
[6] VALUE ← (RECORD [7 8] 0CIQ 0 4) ÷ 10
[7] 'I5, 4X, 20A1, 4X, F8.2' 0FMT (PARTNO;DESC;VALUE)
[8] → L0
[9] L1:CONT
[10] V
```

4. *COPYVOUT*—instructs DATATRIEVE to store records from the port into the PARTS file:

```
VWRITE A;Z
[1] Z← DTRAPUT_PORT ('DAB';A)
[2] V
VWRITEΔEOF ;Z
[1] Z←DTRAPORT_EOF ('DAB')
[2] CONT
[3] Z←DTRACOMMAND ('DAB';';')
[4] CONT
[5] V
```

5. *WRITE*—writes a record to the port:

```
VWRITE A;Z
[1] Z← DTRAPUT_PORT ('DAB';A)
[2] V
```

6. *WRITEΔEOF*—terminates a sequence of records written to the port:

## Calling External Routines

### 6.5 Examples of Calls to External Routines

The semicolon (;) is required to terminate the command sequence that writes the records to the port.

```
VWRITE$EOF ;Z  
[1] Z←DTR$PORT_EOF ('DAB')  
[2] CONT  
[3] Z←DTR$COMMAND ('DAB';';')  
[4] CONT  
[5] V
```

7. *FINISH*—terminates communications with DATATRIEVE:

```
VFINISH ;Z  
[1] Z←DTR$VFINISH ('DAB')  
[2] V
```

DAB[2 3] hold the two halves (in reverse order) of the condition value. This function calculates the condition value from the two DAB elements.

```
V Z ← ERROR  
[1] Z←DAB[2] + DAB[3] × 65536  
[2] V
```

8. *CONT*—reads error conditions returned from DATATRIEVE.

After a command has been sent to DATATRIEVE, the function returns the value of 1, with the state being returned in the DAB. The state field in the DAB is element 13 (DAB[13]). If this has the value 4, the success or failure of the function is indicated in the condition value in the DAB, elements 2 and 3. We must successively execute DTR\$CONTINUE until the state field becomes 1, which indicates that DATATRIEVE is waiting for another command. The condition value of 9274723 is DTR\$\_SUCCESS, that is, successful command execution.

```
VCONT ;Z  
[1] → ( 4 ≠ DAB[13]) /L1  
[2] L0: Z ← DTR$CONTINUE ('DAB') ◊ → (1 = DAB[13])/ 0  
[3] → (9274723 = Z ← ERROR) /L0  
[4] DISPLAY$ERROR Z ◊ → L0  
[5] L1: → (1 = DAB[13]) /0  
[6] DAB[13]  
[7] V
```

9. *DISPLAY\$ERROR*—displays the error message corresponding to the error condition returned from DATATRIEVE:

## Calling External Routines

### 6.5 Examples of Calls to External Routines

```
VDISPLAY@ERROR ERRNO; ERROR@TEXT;LENGTH;Z  
[1] LENGTH ← 0 ⋄ ERROR@TEXT ← 256p ''  
[2] Z+ERROR@MESSAGE (ERRNO; 'LENGTH';'ERROR@TEXT';;)  
[3] ERROR@TEXT [↑LENGTH]  
[4] V
```

10. *FORMATV RECORD*—formats a record from its constituent fields.

The **VALUE** field must be scaled up by 2 decimal places before being written to DATATRIEVE because DATATRIEVE stores **VALUE** in the file in a scaled format.

```
VA FORMAT@RECORD X;TEMP  
[1] TEMP← 8 p 0  
[2] TEMP [1]←‡,X[1;]  
[3] TEMP [1+↑5] ← (‡,X[2;]) ⌠COQ 0  
[4] TEMP [7 8] ← (100×‡,X[3;1]) ⌠COQ 0  
[5] ‡ A, '←TEMP'  
[6] V
```

11. *FIELDS*—the fields that make up a record:

```
FIELDS  
PARTNO  
DESC  
VALUE
```

The following series of APL functions initialize the communications channel to DATATRIEVE and display all of the records therein.

Notice that record number 3 is missing; we will add it in the second half of the example. The individual fields are set to the appropriate values and formatted into the record, which is then written to the PARTS file. The example then rereads all of the records from the PARTS file to check that the insertion worked. Finally, the communications channel is closed.

## Calling External Routines

### 6.5 Examples of Calls to External Routines

```
INIT
COPYAIN
READALL
1      PART NUMBER 1          12.34
2      PART NUMBER 2          34.56
4      PART NUMBER 4          32.69
5      PART NUMBER 5          1234.56
PARTNO← 3
DESC ← 20 S 'PART NUMBER 3'
VALUE← 987.65
'RECORD' FORMATARECORD FIELDS
COPYAOUT
WRITE RECORD
WRITEAEEOF
COPYAIN
READALL
1      PART NUMBER 1          12.34
2      PART NUMBER 2          34.56
3      PART NUMBER           987.69
4      PART NUMBER 4          32.69
5      PART NUMBER 5          1234.56
FINISH
```

#### 6.5.8 Example 8: Using □MAP with /VALUE

The following example shows the use of the / VALUE switch with □MAP.

The DATATRIEVE error message "%DTR-E-ERROR, Statement abandoned due to error" is associated with the DATATRIEVE symbol DTR\$\_ERROR. This symbol is defined in SYS\$SHARE:DTRSHR.EXE, the DATATRIEVE shared image, as a global constant. The value of this global constant can be brought into an APL workspace using the / VALUE switch to □MAP:

```
A← 'Z/TYPE:L← DTRA_ERROR'
B← 'DTRSHR/VALUE:DTR$_ERROR'
A □MAP B
DTRA_ERROR
□MAP 'DTRA_ERROR'
Z/TYPE:L/MECHANISM:IMMEDIATE←
DTRA_ERROR/IMAGE:DTRSHR/VALUE:DTR$_ERROR
    DTRA_ERROR
9273530
```

You can use the APL identifier *DTRV\_ERROR* to check the status returned by various DATATRIEVE functions to see if "statement abandoned due to error" has occurred.

## 6.5 Examples of Calls to External Routines

## 6.5.9 Example 9: Calling a VMS System Service

**MAP** cannot be used to call VMS system services directly because the VMS system services do not reside in a shared image. To call a system service, write a routine in a compiled VMS programming language that calls the system service, link that routine into a shared image, and invoke that routine from inside APL.

The following BLISS program in the file TERMINAL.BLI contains the routine TOGGLE\_ECHO, which toggles the NOECHO flag on the terminal assigned to SYS\$INPUT:.

```

Stype terminal.bli
MODULE TERMINAL =
BEGIN

LIBRARY 'SYS$LIBRARY:STARLET';

!+
!
!   The following .BXMAP expression is required to define
!   TOGGLE_ECHO as an external function:
!
!   'STATUS/TYP:L _ T.USECHO' .BXMAP 'TERMSHR/ENTRY:TOGGLE.USECHO'
!
!-
GLOBAL ROUTINE TOGGLE_ECHO =
!+
!
!   This routine toggles the NOECHO bit in the terminal
!   assigned to SYS$INPUT:. When NOECHO is set, input is
!   not echoed to the terminal but output is displayed.
!-

BEGIN
LOCAL
      STATUS,
      TERMINAL_CHAN,
      TERMINAL_DSC : VECTOR [2],
      DEVICE_BLOCK : BLOCK [12, BYTE] ;

BIND
      TT1 = DEVICE_BLOCK [4,0,0,0] : BLOCK [,BYTE];

TERMINAL_DSC [0] = %CHARCOUNT ('SYS$INPUT:');
TERMINAL_DSC [1] = UPLIT ('SYS$INPUT:');
$ASSIGN (DEVNAM=TERMINAL_DSC, CHAN=TERMINAL_CHAN);

```

## Calling External Routines

### 6.5 Examples of Calls to External Routines

```
STATUS = $QIOW (CHAN = .TERMINAL_CHAN,
                 FUNC = (IO$_SENSEMODE),
                 P1   = DEVICE_BLOCK,
                 P2   = 12;)

TT1 [TT$V_NOECHO] = NOT .TT1 [TT$V_NOECHO];

STATUS = $QIOW (CHAN = .TERMINAL_CHAN,
                 FUNC = (IO$_SETMODE),
                 P1   = DEVICE_BLOCK,
                 P2   = 12;)

RETURN .STATUS;

END;           ! End of TOGGLE_ECHO routine

END           ! End of TERMINAL module
ELUDOM
```

1. Compile the BLISS program called TERMINAL.BLI to create TERMINAL.OBJ:

```
$ bliss terminal.bli
```

2. Create a shared image called TERMINAL.EXE from TERMINAL.BLI:

```
$ link/shareable=terminal terminal,sys$input:/options
universal=toggle_echo
```

3. Define the logical name TERMSHR to point to the shared image on the device and directory where TERMINAL.EXE resides:

```
$ define termshr device:[directory]terminal.exe
```

4. Invoke APL and use TOGGLE\_ECHO:

```
$ apl/terminal=key/silent
      A← 'STATUS/TYPE:L ← T_ECHO'
      B← 'TERMSHR/ENTRY:TOGGLE_ECHO'
      A ⊞MAP B
      TOGGLE_ECHO
      ↗NOECHO IS CURRENTLY OFF SINCE INPUT IS BEING ECHOED
      ↗TOGGLE_ECHO
      2      (1 1 is the input but only the answer is displayed)
      1      (TOGGLE_ECHO is the input; only its result is displayed)
```

## Calling External Routines

### 6.5 Examples of Calls to External Routines

#### 6.5.10 Example 10: Calling SMG\$ Routines

The following example invokes four external SMG\$ routines. The example creates a pasteboard and a virtual display, and writes some text on the board.

##### SMG\$CREATE\_PASTEBOARD

```
R 'STATUS/TYP:L CREATEPB '
R_R,'PBID/TYP:L/ACCESS:OUT/MECH:REFERENCE '
R_R,'OUTPUT/TYP:T/ACCESS:IN/MECH:DESC '
R_R,'ROWS/TYP:L/ACCESS:OUT/MECH:REFERENCE '
R_R,'COLS/TYP:L/ACCESS:OUT/MECH:REFERENCE '
R_R,'FLAG/TYP:L/ACCESS:IN/MECH:REFERENCE'
R .BXMAP 'SMGSHR/ENTRY:SMG$CREATE.PASTEBOARD'
```

##### SMG\$CREATE\_VIRTUAL\_DISPLAY

```
R 'STATUS/TYP:L CREATEVD '
R_R,'ROWS/TYP:L/ACCESS:IN/MECH:REFERENCE '
R_R,'COLS/TYP:L/ACCESS:IN/MECH:REFERENCE '
R_R,'DISPID/TYP:L/ACCESS:OUT/MECH:REFERENCE '
R_R,'DISATT/TYP:L/ACCESS:IN/MECH:REFERENCE '
R_R,'VIDATT/TYP:L/ACCESS:IN/MECH:REFERENCE'
R .BXMAP 'SMGSHR/ENTRY:SMG$CREATE.USVIRTUAL.USDISPLAY'
```

##### SMG\$PASTE\_VIRTUAL\_DISPLAY

```
R 'STATUS/TYP:L PASTEVD '
R_R,'DISPID/TYP:L/ACCESS:IN/MECH:REFERENCE '
R_R,'PBID/TYP:L/ACCESS:IN/MECH:REFERENCE '
R_R,'PBROW/TYP:L/ACCESS:IN/MECH:REFERENCE '
R_R,'PBCOL/TYP:L/ACCESS:IN/MECH:REFERENCE '
R .BXMAP 'SMGSHR/ENTRY:SMG$PASTE.USVIRTUAL.USDISPLAY'
```

##### SMG\$PUT\_CHARS

```
R 'STATUS/TYP:L PUTCH '
R_R,'DISPID/TYP:L/ACCESS:IN/MECH:REFERENCE '
R_R,'TEXT/TYP:T/ACCESS:IN/MECH:DESC '
R_R,'STARTROW/TYP:L/ACCESS:IN/MECH:REFERENCE '
R_R,'STARTCOL/TYP:L/ACCESS:IN/MECH:REFERENCE '
R_R,'ERASEFG/TYP:L/ACCESS:IN/MECH:REFERENCE '
R_R,'RENDSET/TYP:L/ACCESS:IN/MECH:REFERENCE '
R_R,'RENDCMP/TYP:L/ACCESS:IN/MECH:REFERENCE'
R .BXMAP 'SMGSHR/ENTRY:SMG$PUT_CHARS'
```

The four SMG\$ routines are now defined in the APL workspace. The example continues by defining the text that will be written to the pasteboard, setting the attributes for the virtual display, and calling the external routines.

## Calling External Routines

### 6.5 Examples of Calls to External Routines

```
TEXT←'THIS IS THE DEMO FOR USING SM$ ROUTINES'  
TEXT1←'TO SHOW HOW TO CREATE A WINDOW ON THE'  
TEXT2←'TERMINAL SCREEN, SMG$PUTCHAR PUT DATA HERE.'  
□SINK←CREATEPB('PBID';;;;  
ROWS←7 ◊ COLUMNS←50           ASIZE OF VIRTUAL DISPLAY  
BORDER←1 ◊ BOLD←1            ASPECIFY THE DEFAULT RENDITION FOR  
◊ DISPLAY AND VIDEO ATTRIBUTES  
□SINK←CREATEVD(ROWS;COLUMNS;'DISPID1';BORDER;BOLD)  
□SINK←PUTCH(DISPID1;TEXT;2;1;;;  
□SINK←PUTCH(DISPID1;TEXT1;4;1;;;  
□SINK←PUTCH(DISPID1;TEXT2;6;1;;;  
□SINK←PASTEVD(DISPID1;PDID;4;15)  
  
+-----+  
| THIS IS THE DEMO FOR USING SMG$ ROUTINES |  
| TO SHOW HOW TO CREATE A WINDOW ON THE |  
| TERMINAL SCREEN, SMG$PUTCHAR PUT DATA HERE. |  
+-----+
```

# A

---

## VAX APL Workspace Interchange Standard

The VAX APL Workspace Interchange Standard (WSIS) describes a method for transferring workspaces from one APL implementation to another. The WSIS allows a workspace to be transferred regardless of its internal APL format

or the size and content of the particular implementation. (Note that you cannot transfer nested arrays.)

The WSIS has been agreed to by implementors of APL and documented in the article "Workspace Interchange Convention," *APL Quote-Quad*, Vol. 9, No. 3, March 1979.

A workspace to be transferred is converted into a standard format and written to a magnetic tape (or, optionally, to a disk file). Then, the workspace can be read from the tape and converted from the standard format to a particular implementation's format.

If you want to use the WSIS, you must install the optional WSIS software when you install VAX APL (for details, see the *VAX APL Installation Guide*.) The optional WSIS software consists of the following:

- APLTAP.EXE A VMS program that copies WSIS-formatted files from disk to tape and from tape to disk.
- WSOUT.APL A VAX APL workspace that contains the function `QQWSOUT`, which converts VAX APL workspaces to WSIS-formatted workspaces.
- WSIN.APL A VAX APL workspace that contains the function `QQWSIN`, which converts WSIS-formatted workspaces to VAX APL workspaces.

### A.1 Converting VAX APL Workspaces to WSIS-Formatted Workspaces

To create a tape file containing VAX APL workspaces that are to be transferred to a different APL implementation, follow these steps:

## VAX APL Workspace Interchange Standard

### A.1 Converting VAX APL Workspaces to WSIS-Formatted Workspaces

1. Invoke VAX APL, load the workspace that is to be transferred, copy the VAX APL workspace WSOUT from SYS\$LIBRARY, and execute the APL function QQWSOUT. For example:

```
$ apl/term=dec/silent  
 )LOAD WSNAM  
 )COPY SYS$LIBRARY:WSOUT  
 QQWSOUT 'FILENAME'
```

This writes the workspace identified by *wsname* as a disk file with the name *file-name*. (The default file type of *file-name* is *.AIS*.) The disk file includes the workspace's functions, operators, and variables, except for those whose names begin with QQ. Certain VAX APL system variables are also copied. The WSIS software does not provide a way to copy the state indicator stack, groups, or channel assignments.

2. Repeat step 1 for each workspace to be transferred. Use a different file name for each workspace written to disk.
3. Execute APLTAP.EXE (from SYS\$LIBRARY) to write the disk files to a tape (note that you can put multiple workspaces on a single tape). You will need to use the following APLTAP commands:

INITIALIZE	Opens a tape file and writes initial interchange information, which prepares the tape to receive the workspace named by the WRITE command. You will be prompted for the name of the tape file. The default file type of the tape file is <i>.AXF</i> .
WRITE	Copies a disk file to tape (the tape must have been initialized). You will be prompted for the name of the disk file that contains the WSIS-formatted workspace. Records in the disk file may contain a maximum of 512 bytes. The default file type of the disk file is <i>.AIS</i> .
TERMINATE	Closes the tape file.
EXIT	Exits from the APLTAP program. Closes any tape files that were initialized but not terminated.
Ctrl/Z>	Closes the tape file and exits from the APLTAP program (just as if you had executed the TERMINATE and EXIT commands).

## **A.1 Converting VAX APL Workspaces to WSIS-Formatted Workspaces**

For example:

```
$ run sys$library:apltap
APLTAP!initialize
Enter tape file specification:tape
APLTAP!write
Enter file name:file-name-1
APLTAP!write
Enter file name:file-name-2
.
.
.
APLTAP!terminate
APLTAP!exit
$
```

Note that APLTAP prompts for commands with APLTAP!. You may enter APLTAP commands in either uppercase or lowercase, and you may abbreviate them to the shortest unique spelling.

APLTAP requires that the tape have a standard ANSI label. APLTAP writes fixed-length 1892-byte (8-bit bytes) records (it pads the last record with spaces). Other characteristics of the tape, such as density and parity, are not specified by the WSIS; APLTAP will execute successfully only if the sender and receiver have agreed on these characteristics.

You can use APLTAP to copy a WSIS-formatted file to a device other than tape (such as a disk file). If you respond to the tape file specification prompt with a disk file specification, APLTAP prints a warning but continues processing. Thus, although APLTAP will not write to an unlabeled tape, you could copy the WSIS-formatted files to an unlabeled tape by first using APLTAP to create a disk file, and then using some other mechanism to write the disk file to an unlabeled tape.

## **A.2 Converting WSIS-Formatted Workspaces to VAX APL Workspaces**

To convert workspaces from WSIS format to VAX APL format, follow these steps:

1. Execute APLTAP.EXE (from SYS\$LIBRARY) to copy the WSIS-formatted tape files to disk and to create a command file that will be used to convert the disk files to workspaces.

## VAX APL Workspace Interchange Standard

### A.2 Converting WSIS-Formatted Workspaces to VAX APL Workspaces

You will need to use the following APLTAP commands:

READ       Reads one or more tape files and creates disk files for input to the APL function *QQWSIN*. The default file type for these tape files is .AXF. Also creates a command file that contains the APL statements needed to execute *QQWSIN*. The default file type for the command file is .AAS. If you choose a different file type, then you will have to specify it when you use the *)INPUT* command (see step 2).

EXIT or Ctrl/Z   Exits from the APLTAP program.

For example:

```
$ run sys$library:apltap
APLTAP!read
Enter name of command file: command-file
Enter next tape file name (DONE to exit): tape-file-1
Total number of errors =
Enter next tape file name (DONE to exit): tape-file-2
Total number of errors =
.
.
.
Enter next tape file name (DONE to exit): done
APLTAP!exit
$
```

APLTAP first prompts for the name of the command file to be created, and then it successively prompts for tape files to process until you enter DONE.

Note that APLTAP prompts for commands with APLTAP!. You may enter APLTAP commands in either uppercase or lowercase, and you may abbreviate them to the shortest unique spelling.

APLTAP creates a disk file named WSINnnnn.AIS for each tape file entered (*nnnn* is a 4-digit decimal number; the first file is assigned 0000). The names will be used by the command file created for step 2.

A tape record may not exceed 4096 8-bit bytes in length. If the specification you supply as the tape file is not actually a tape device, APLTAP prints a warning but continues processing. Thus, although APLTAP will read only labeled tapes, you can copy a WSIS-formatted workspace from an unlabeled tape by first using some other mechanism to create a disk file from the unlabeled tape, and by then using APLTAP to process the disk file.

2. Invoke VAX APL and use the *)INPUT* command to execute the command file created in step 1. For example:

```
$ apl/term=dec/silent
      )INPUT command-file/TTY
```

Note that the command file is created in TTY character set.

## A.2 Converting WSIS Formatted Workspaces to VAX APL Workspaces

This procedure creates workspaces with file names taken from the WSIS tape; each workspace has a file type of .APL. If the name of any of the new workspaces is already in use in your default directory, WSIN changes the file type of the new workspace to .Wnn, where nn is a 2-digit decimal number.

WSIN lists the function, operator, and variable names on the terminal as it copies them to the new workspace. When WSIN has processed the entire file, it deletes the WSINnnnn.AIS file produced by APLTAP, but does not delete the command file.

## A.3 Sample WSIS Session

In the following sample session, a VAX APL workspace (TEST.APL) is written to tape in WSIS format. The file is read and the WSIS-formatted workspaces are recreated as VAX APL workspaces.

```
$ apl/term=dec/silent
  )LOAD TEST
SAVED WEDNESDAY 17-APR-1991 09:40:01.77 12 BLKS
  )COPY SYS$LIBRARY:WSOUT
SAVED WEDNESDAY 13-MAR-1991 13:19:08.56 27 BLKS
  QQWSOUT 'TEST'
CREATING OUTPUT FILE: 1    TEST/IS
OPERATIONS:
  FN
  FN1
VARIABLES:
  B
  A
  □VPC
  □TT
  □TLE
  □TIMEOUT
  □TIMELIMIT
  □TERSE
  □SINK
  □SF
  □RL
  □R
  □PW
  □NG
  □LX
  □L
  □IO
  □GAG
  □ERROR
  □DML
```

## VAX APL Workspace Interchange Standard

### A.3 Sample WSIS Session

```
VARIABLE'S VALUE IS NESTED OR HETEROGENEOUS: )DC
 )DC
 )CT
 )AUS
 )TRAP
 )PP
 )QWSOUT IS DONE
 )OFF
```

Run APLTAP.EXE to copy the WSIS-formatted file (TEST.AIS) to tape.

```
$ run sys$library:apltap
APLTAP!initialize
Enter tape file specification: mka500:trnx
APLTAP!write
enter file name: test
APLTAP!terminate
APLTAP!exit
```

The WSIS-formatted file has been copied to the tape. Now APLTAP.EXE is used to read the file and restore the workspace.

```
$ run sys$library:apltap
APLTAP!read
Enter name of command file: conv
Enter next tape file name (DONE to exit): mka500:trnx
Total number of errors =
Enter next tape file name (DONE to exit): done
APLTAP!exit
```

APLTAP.EXE copied the file to the default disk area and named the file WSIN0000.AIS. APLTAP.EXE also created a command file, CONV.AAS, to be executed inside of APL using the )INPUT command. The contents of that command file are as follows:

```
$ type conv.aas
)CLEAR
)COPY SYSSLIBRARY:WSIN
.ZQ.ZQWSIN '$USERS:[APLUSER]WSIN0000'
```

Invoke APL and use )INPUT to execute the command file, CONV.AAS.

## VAX APL Workspace Interchange Standard

### A.3 Sample WSIS Session

```
)INPUT CONV/TTY
)CLEAR
CLEAR WS
)COPY SYS$LIBRARY:WSIN
SAVED WEDNESDAY 13-MAR-1991 13:19:07.48 41 BLKS
QQWSIN      '$USERS:[APLUSER]WSIN0000'
READING FROM $USERS:[APLUSER]WSIN0000
OLD WSID WAS TEST
SUPERSENDING    TEST.W00
SUPERSENDING    TEST.W01
NEW WSID IS     TEST.W01
NOTE: CREATED ON WEDNESDAY 17-APR-1991 09:40:01.77 BY
[USERS,APLUSER] AT TWA4: WITH T4.0-875
CREATED OPERATION: FN
CREATED OPERATION: FN1
CREATED NUMERIC VARIABLE: B
CREATED NUMERIC VARIABLE: A
CREATED NUMERIC VARIABLE: □VPC
CREATED NUMERIC VARIABLE: □TT
CREATED NUMERIC VARIABLE: □TLE
CREATED NUMERIC VARIABLE: □TIMEOUT
CREATED NUMERIC VARIABLE: □TIMELIMIT
CREATED NUMERIC VARIABLE: □TERSE
CREATED NUMERIC VARIABLE: □SINK
CREATED CHARACTER VARIABLE: □SF
CREATED NUMERIC VARIABLE: □RL
CREATED NUMERIC VARIABLE: □R
CREATED NUMERIC VARIABLE: □PW
CREATED NUMERIC VARIABLE: □NG
CREATED CHARACTER VARIABLE: □LX
CREATED NUMERIC VARIABLE: □L
CREATED NUMERIC VARIABLE: □IO
CREATED NUMERIC VARIABLE: □GAG
CREATED CHARACTER VARIABLE: □ERROR
CREATED NUMERIC VARIABLE: □DML
CREATED NUMERIC VARIABLE: □CT
CREATED NUMERIC VARIABLE: □AUS
CREATED CHARACTER VARIABLE: □TRAP
CREATED NUMERIC VARIABLE: □PP
DONE WITH INPUT FILE $USERS:[APLUSER]WSIN0000
÷DELETE-I-FILDEL, $USERS:[APLUSER]WSIN0000.AIS;1 deleted (6 blocks)

)LOAD      TEST.W01
)ERASE QQWSIN
)SAVE
WEDNESDAY 17-APR-1991 10:01:31.38 22 BLKS TEST.W01
)DROP $USERS:[APLUSER]WSIN0000.AAS;0
÷DELETE-I-FILDEL, $USERS:[APLUSER]WSIN0000.AAS;1 deleted (3 blocks)
```

## VAX APL Workspace Interchange Standard

### A.3 Sample WSIS Session

The *QQWSOUT* and *QQIN* functions can also be used to transfer files. To convert a file to WSIS output form, load the WSOUT workspace from SYS\$LIBRARY: and execute the *QQWSOUT* function with a left argument which is the file specification of the file to be converted. (The default file type is .AIX; for other file types include the appropriate switch; For instance, /AS for ASCII sequential, .AAS types.) The right argument is the output tape specification, the same as when transferring workspaces. The following example illustrates transferring an ASCII file called FOO.AAS. Because the file is transferred to a disk instead of tape, APLTAP outputs a warning message.

```
$ apl/term=dec/silent
    )LOAD SYS$LIBRARY:WSOUT
SAVED WEDNESDAY 13-MAR-1991 13:19:08.56 25 BLKS
'FOO/AS' QQWSOUT 'OUTFILE'
CREATING OUTPUT FILE: 1    OUTFILE/IS
    REC0000
    REC0000
QQWSOUT IS DONE
)OFF

$ run sys$library:apltap
APLTAP!initialize
Enter tape file specification: $users:[apluser]trnx
WARNING: Target device is not tape.
APLTAP!WRITE
Enter file name: outfile
APLTAP!terminate
APLTAP!exit
```

Now input the transferred file.

```
$ run sys$library:apltap
APLTAP!read
Enter name of command file: doit
Enter next tape file name (DONE to exit): $users:[apluser]trnx
WARNING: Target device is not Tape.
Total number of errors =
Enter next tape file name (DONE to exit): done
APLTAP!exit
$ apl/term=dec/silent
    )INPUT DOIT/TTY
    )CLEAR
CLEAR WS
    )COPY SYS$LIBRARY:WSIN
SAVED WEDNESDAY 13-MAR-1991 13:19:07.48 41 BLKS
    QQWSIN      '$USERS:[APLUSER]WSIN0002'
READING FROM $USERS:[APLUSER]WSIN0002
OLD FILEID WAS FOO/AS
NEW FILEID IS FOO.X00
DONE WITH INPUT FILE $USERS:[APLUSER]WSIN0002
+DELETE-I-FILDEL, $USERS:[APLUSER]WSIN0002.AIS;1 deleted (6 blocks)
```

```
)DROP $USERS:[APLUSER]WSIN0002.AAS;0
÷DELETE-I-FILDEL, $USERS:[APLUSER]WSIN0002.AAS;1 deleted (3 blocks)
)OFF
```

## A.4 Error Messages and Warnings Generated by WSIS Software

When you use the WSIS software, some error and warning messages may be displayed. This section identifies those messages and explains (if necessary) what they mean. In the messages, *xx* is a hexadecimal number, typically the error code from VMS Record Management Services.

### A.4.1 WSOUT Messages

*QQWSOUT IS DONE*

**Explanation:** ÓÓWSOUT has completed processing.

*UNABLE TO ASSIGN THE OUTPUT FILE*

**Explanation:** *QQQQWSOUT* was unable to assign the output file to channel 1.

*CREATING OUTPUT FILE: filespec*

**Explanation:** Informational.

*OPERATION LOCKED: name*

**Explanation:** *WSOUT* cannot transfer locked function or operator.

*UNABLE TO ASSIGN INPUT FILE*

**Explanation:** It is not possible to open the file to be transferred.

*INVALID WORKSPACE IDENTIFIER*

**Explanation:** The workspace to be transferred has an invalid identifier.

### A.4.2 WSIN Messages

*INPUT FILE file NOT FOUND*

**Explanation:** *QQWSIN* was unable to find the specified input file.

*FILE IN INCORRECT FORMAT*

**Explanation:** The specified input file is not in the expected format.

*UNKNOWN PSEUDOVARIABLE name IGNORED*

*RANK: r SHAPE: s*

**Explanation:** The named pseudovariable is unknown.

## VAX APL Workspace Interchange Standard

### A.4 Error Messages and Warnings Generated by WSIS Software

*UNEXPECTED END OF FILE*

*WS FULL. YOU MUST START OVER.*

*UNEXPECTED ERROR NUMBER n*

*WARNING, IDENTIFIER: xxxx BEING TRUNCATED TO: xxx*

**Explanation:** If an incoming identifier has more than 31 characters, it is truncated to the first 31 characters.

*WARNING, IDENTIFIER SAME AS PREVIOUS ONE*

**Explanation:** A truncated identifier matches some previously converted identifier.

*FIX OF OPERATION name FAILED AT LINE n*

**Explanation:** The function or operator could not be created for some reason. The operation is left as an operation with all its lines turned into comments.

*UNABLE TO FIX OPERATION name AT LINE n*

**Explanation:** A function or operator with all its lines commented out cannot be fixed. The operation is left as a character array.

*UNABLE TO ASSIGN THE OUTPUT FILE*

**Explanation:** A file that is being transferred cannot be created.

*EXECUTABLE EXPRESSION: expression*

*SIGNALS THE FOLLOWING ERROR.*

**Explanation:** An executable expression received an error.

*SCALARS OF TYPE type ARE NOT ALLOWED. IGNORED.*

**Explanation:** An invalid pseudovariable was found.

*TOO MANY VERSIONS OF THE SAME NAME--RAN OUT OF SUFFIXES*

**Explanation:** There are more than 99 files with the same name.

*\*\*\*\*\*ERROR, BAD RANK IN name*

**Explanation:** The rank information for a transferred object is invalid.

*\*\*\*\*\*ERROR, BAD SHAPE IN name*

**Explanation:** The shape information for a transferred object is invalid or inconsistent with the rank information.

## A.4 Error Messages and Warnings Generated by WSIS Software

\*\*\*\*\**ERROR CREATING NUMERIC VARIABLE name*

**Explanation:** An attempt was made to create an invalid numeric array. The variable is left in character form for possible repair. Too large an exponent is a possible cause of this message.

\*\*\*\*\**ERROR CREATING NUMERIC COMPONENT: name*

**Explanation:** An invalid numeric array was found when transferring a file.

*CREATED CHARACTER VARIABLE: name*

**Explanation:** Successful transfer of a variable.

*CREATED NUMERIC VARIABLE: name*

**Explanation:** Successful transfer of a variable.

*CREATED OPERATION: name*

**Explanation:** Successful transfer of a function or operator.

*EXECUTED EXECUTABLE EXPRESSION: expression*

**Explanation:** Performed the execution of the expression passed in the executable expression pseudovariable.

*DONE WITH INPUT FILE filespec*

**Explanation:** A file has been successfully transferred and created.

### A.4.3 APLTAP Messages

#### Command Syntax Errors

Illegal command: Not one of READ, WRITE, INIT, TERM, or EXIT.

Tape has already been initialized.

**Explanation:** An INITIALIZE command was entered for a tape that was initialized for writing.

Tape initialized for writing.

**Explanation:** A READ command was entered for a tape that was initialized for writing.

## VAX APL Workspace Interchange Standard

### A.4 Error Messages and Warnings Generated by WSIS Software

Tape not initialized.

**Explanation:** A WRITE or TERMINATE command was entered for a tape that has not been initialized.

#### I/O Errors

Unable to open <SYS\$INPUT | *tape-file* | *source-file*> (xx).

**Explanation:** An error occurred when APLTAP tried to open the indicated file.

Unable to create <*tape-file* | *wzin-file* | *log-file*> (xx).

**Explanation:** An error occurred when APLTAP tried to create the indicated file (*log-file* refers to the command file).

Unable to connect to <SYS\$INPUT | *log-file* | *tape-file* | *wzin-file* | *source-file*> (xx).

**Explanation:** An error occurred when APLTAP tried to connect to the indicated file (*log-file* refers to the command file).

Error closing <*input-file* | *tape-file*>.

**Explanation:** An error occurred when APLTAP tried to close the indicated file.

Unable to write out prologue (xx).

**Explanation:** An error occurred when APLTAP tried to write the WSIS prologue to the tape. The initialization is aborted.

Unable to write END pseudovariable (xx).

**Explanation:** An error occurred when APLTAP tried to write the END pseudovariable to the tape. The tape file is closed.

Unable to write to log file (xx).

**Explanation:** An error occurred while APLTAP was writing to the command file. READ processing is terminated.

Error writing to tape file (xx).

**Explanation:** An error occurred while APLTAP was writing a data block to the tape. Writing of this file is terminated.

VAX APL Workspace Interchange Standard

## A.4 Error Messages and Warnings Generated by WSIS Software

Cannot write to WSIN file (xx).

**Explanation:** An error occurred while APLTAP was writing to the WSINnnnn file. READING of that tape file is stopped, but READ processing continues.

Error reading from <input-file | SYS\$INPUT | tape-file> (xx).

**Explanation:** An error occurred while APLTAP was reading from the specified file. Processing of that file is stopped.

Unexpected end of file reading from tape.

**Explanation:** End of file occurred while APLTAP was reading the WSIS or TRANSLATE pseudovariables.

Unexpected error reading from tape (xx).

**Explanation:** An error occurred while APLTAP was reading from the tape file. READING of that file is stopped, but READ processing continues.

### Tape Format Errors

Number in tape input too large.

**Explanation:** The WSIS or TRANSLATE pseudovariable contained a numeric string that was too long to translate to a 32-bit integer.

No WSINnnnn name available for use (nnnn is a 4-digit decimal number).

**Explanation:** All names of the form WSINnnnn are in use.

Second vector is not TRANSLATE pseudovariable.

**Explanation:** TRANSLATE pseudovariable is not a matrix.

TRANSLATE contains too many rows.

TRANSLATE contains too few columns.

First vector is not WSIS pseudovariable.

Format on tape is not convention 0.

**Explanation:** The WSIS software supports version 0 of the workspace convention.

### Warnings from APLTAP

## VAX APL Workspace Interchange Standard

### A.4 Error Messages and Warnings Generated by WSIS Software

Target device is not tape.

**Explanation:** The file specification given for the tape file does not correspond to a magnetic tape device.

Incorrect length for WSIS pseudovariable.

Nonblank padding at end of WSIS.

Nonblank padding at end of TRANSLATE.

TRANSLATE contains a character with more than two overstrikes at entry (xx).

TRANSLATE contains character not found in  $\square AV$  at entry (xx).

Illegal reference to undefined character (xxx).

Total number of errors = (nn).

**Explanation:** Note that if no errors occur, nn in this message is blank.

---

# Index

## A

---

Abort input signal, 3–24, 3–36  
  definition of, 1–33  
Absolute tab format phrase, 4–15  
Access methods  
  file, 5–15  
  random, 5–16, 5–26  
  sequential, 5–15  
Active workspace  
  definition of, 1–30  
Actual arguments, 3–2  
Alternate character set, TTY, 5–23  
Ambivalent derived functions, 3–3  
Ambivalent functions, 3–2, 3–39e  
APL character set, 1–2 to 1–5, 5–23  
APL command line, 1–11  
APL interpreter  
  reentrant, 1–1  
  shareable, 1–1  
APL interpreter output, 1–27  
APL keyboards, 1–2  
APL names, 1–5t  
APL operating modes, 1–27, 3–1  
APL operations and programs, 3–1  
APL session  
  exiting from, 1–35  
  interrupting, 1–33  
  starting, 1–11  
APL terminals, 1–2, 1–37  
Arguments  
  actual, 3–2  
  dummy, 3–2  
  extending singletons, 2–10

Arguments (cont'd)  
  shape of function, 2–9  
Arrays  
  axes of, 2–3  
  character, 2–2  
  coordinates of, 2–3  
  definition of, 2–1  
  depth of, 2–8  
  dimensions of, 2–3  
  displaying, 2–8  
  empty, 2–12  
  format for displaying, 5–6  
  formation rules, 2–25  
  indexing, 2–34, 2–35  
  maximum axes, 2–3  
  numeric, 2–2  
  output precision of, 2–2  
  rank of, 2–3, 2–7  
  reshaping, 2–6  
  shape of, 2–4  
  spaces in, 2–3  
  structure of, 2–2  
  type if empty, 2–13  
  types of, 2–1  
ASCII  
  character set, 1–2, 1–45t  
  control characters, 1–5, 1–28  
  graphics, 1–5, 1–8t  
  keyword mnemonics, 1–4, 1–5  
Attention signal, 1–28, 1–29t, 1–33  
Axis argument in functions, 3–39e

## B

---

Background format phrase decorator, 4–25  
Backspace, 1–29t  
Banner line, 1–20  
Bare branch, 3–44  
Bare output, 5–10  
    resetting buffer, 5–11  
Bit-paired character set, 1–2, 1–24t, 1–37, 5–23  
Blank when zero format phrase qualifier, 4–21  
Boolean number, 2–2  
Branch function, 3–10, 3–13e  
Break system function, 3–49  
Buffer  
    resetting bare output, 5–11  
Byte data format phrase, 4–13

## C

---

Calls to external routines, 6–1  
Carriage return  
    as quote quad input, 5–4  
    suppressing, 5–10  
Changing selected items in arrays, 2–42  
Channels  
    assigning files to, 5–17  
    listing active, 5–22  
    numbers, 5–33  
    status of, 5–33  
Channels system function, 5–22  
Character  
    arrays, 2–2  
    constants, 2–19  
    editing, 1–5, 1–36  
    errors, 1–39  
Character data  
    formatting, 4–28  
    mixing with numeric, 4–10  
Character editing, 3–32  
    escaping from, 3–36  
Character format phrase, 4–5

Character set, 1–36  
    alternate, 5–23  
    APL, 1–2 to 1–5, 5–23  
    ASCII, 1–2, 1–45t  
    atomic vector, 1–46t  
    bit-paired, 1–2, 1–24t, 1–37, 5–23  
    composite, 1–2, 1–42t, 5–23  
    key-paired, 1–2, 1–24t, 1–37, 5–23  
    multinational, 1–44t, 5–67t  
    overstruck, 1–7t  
    TTY, 1–4, 1–38, 1–39t, 5–23  
    typewriter-paired, 1–2

Character-Cell interface  
    buffer  
        definition of, 1–17  
        editing operations, 3–20  
        initialization stream, 1–17  
        starting, 1–17  
        windows, 1–18

Characters  
    allowable, in names, 2–17  
    arrays of, 2–2  
    fill, 2–13  
    keyboard editing, 1–28  
    overstruck, 1–4, 1–24t, 1–25t  
    single-strike, 1–5

Character\_Cell Interface  
    editing variables, 2–32

Clear event flag system function, 5–45

Clear workspace  
    definition of, 1–30

Closing files system function, 5–31

Command level, returning to, 1–29t

Command line, APL, 1–11

Comments, 2–25, 3–14

Composite character set, 1–2, 1–42t, 5–23

Conditional branching, 3–11

Constants  
    character, 2–19  
    indexing, 2–39  
    numeric, 2–18

CONTINUE workspace, 1–30, 1–32

Control characters, 1–9t  
    ASCII, 1–5, 1–28  
    Ctrl/C, 1–22, 1–29t, 1–33

**Control characters (cont'd)**  
Ctrl/D, 1-4, 1-33, 1-35  
Ctrl/O, 1-29t  
Ctrl/R, 1-29t  
Ctrl/T, 1-29t  
Ctrl/U, 1-29t  
Ctrl/X, 1-29t  
Ctrl/Y, 1-29t, 1-33  
Ctrl/Z, 1-22, 1-36  
editing lines containing, 3-36  
Ctrl/Z, 1-22

## D

---

**Data**  
converting, 5-56, 5-61  
external types of, 5-64  
internal representation, outputting, 4-13  
internal types of, 5-61  
outputting literal, 4-18  
reformatting, 4-1, 5-56  
**DATATRIEVE**  
calling from APL, 6-25e  
**Debugging**  
error trapping, 3-49  
external routines, 6-18  
state indicator, 3-44  
stop vector, 3-48  
suspended operations, 3-42  
trace vector, 3-46  
**Decimal point**  
in pattern data with TTY, 4-11  
**Decorators**  
background, 4-25  
negative, 4-24  
positive, 4-24  
zero, 4-24  
**Decorators, format phrase**, 4-19  
**DECwindows interface**  
command line, 1-16  
initialization stream, 1-16  
starting, 1-16  
transcript area, 1-16  
window, 1-16  
**DECwindows Interface**  
editing operations, 3-18

**DECwindows Interface (cont'd)**  
editing variables, 2-30  
**Defaults**  
of index origin, 2-35  
workspace name, 1-31t  
**Del Quad input**  
**See Quad Del input**  
**Del, protected**, 3-14  
Deleting input character, 1-29t  
Delimiters in operation header, 3-4  
Depth of arrays, 2-8  
**Derived functions**  
types of  
ambivalent, 3-3  
**Designators**  
terminal, 1-21  
**Device**  
changing for input, 5-11  
displaying characteristics of, 5-35  
mailbox number of, 5-48  
**Diamond character**, 2-24  
**Direct-access files**  
deleting records from, 5-26  
end-of-file, 5-26  
reading and writing, 5-26  
**Display format**  
output, 5-6  
**Dollar sign representation**, 1-4  
**Domains of arguments**, 2-9  
**Dummy arguments**, 3-2  
**Dyadic functions**, 3-2, 3-38e  
definition of, 2-10  
**Dyadic operators**, 3-3, 3-41e

## E

---

**Echoing input lines**, 5-2  
**/EDIT Qualifier**, 1-15  
**)EDIT system command**, 3-22  
**Editing**  
**See DECwindows interface**  
character, 1-5, 1-36, 3-32  
commands for, 3-24t  
deleting lines, 3-27  
displaying lines, 3-28

**Editing (cont'd)**

- immediate mode, 1-28, 3-36
- inserting lines, 3-26
- keyboard, 1-28
- locking operations, 3-14
- operation headers, 3-31
- operations, 3-23
- search and replace, 3-29
  - See Character-Cell interface**, 3-20
- Editing Operations**, 3-17
- )EDITsystem command, 2-34

**Empty arguments**

- to report formatter, 4-4

**Empty arrays**, 2-12

**Empty components**

- in direct-access files, 5-26
- in relative files, 5-26

**Empty user-defined operation**, 3-24

**End of line**

- typing beyond, 2-25

**End-of-file**

- direct access, 5-26
- internal sequential, 5-26
- relative, 5-26

**Error handling**, 2-42

**Error messages**

- primary, 2-43
- secondary, 2-43

**Error system variable**, 3-49

**Errors**

- character, 1-39
- in user-defined operations, 2-44
- order of checking, 2-43
- signaling, 3-49
- trapping, 3-49
- typing, 1-28

**Evaluated input**

- See Quad input**

**Event flags**, 5-44, 5-49e

**Execution**

- changing order of, 3-10
- interrupting, 1-28
- order of statement, 2-24
- stopping, 3-11

**Execution functions**

- pending, 3-45
- suspended, 3-45

**Execution mode**, 1-27

**Exit from APL session**, 1-29t, 1-33, 1-35

**Exponent digits format phrase qualifier**, 4-23

**Expressions**

- definition of, 2-16
- indexing, 2-39
- interrupting evaluation of, 1-28
- order of evaluating, 2-22

**Extension of singletons**, 2-10

**External data types**, 5-64

**External routines**, 6-19e

- calling, 6-1
- data types, 6-6t
- defining to APL, 6-3
- invoking, 6-16
- linking, 6-2
- querying APL definition, 6-5
- writing, 6-2

---

**F**

**File access**

- methods for, 5-15
- synchronizing, 5-44

**File input system function**, 5-23

**File organization qualifiers**, 5-18

**File output system function**, 5-23

**File specification**

- VMS, 1-31

**File status**, 5-33

**Files**

- assigning to channels, 5-17
- closing, 5-31, 5-32
- creating, 5-23
- deassigning, 5-32
- HI, 1-15
- initialization, 1-12
- keyed, 5-15
- locked, 5-40
- opening, 5-23, 5-35, 5-48
- organization of, 5-33

**Files (cont'd)**

- reading, 5-23
  - direct-access, 5-26
  - internal sequential, 5-25
  - keyed, 5-27
  - non-APL, 5-56, 5-58
  - relative, 5-26
- returning to the beginning of, 5-31
- sharing, 5-34, 5-38, 5-42
- writing, 5-23

**Fill character**, 2-13

**Fill element**

- definition of, 2-15

**Fill item**

- definition of, 2-15

**Fixed-point format phrase**, 4-8

**Floating-point format phrase**, 4-6

**Floating-point numbers**, 2-2

- display of, 5-7

**Font files**, 1-26

**Format**

- of internal records, 5-60
- of terminal output, 5-6

**Format phrases**, 4-2

- decorators, **See** Decorators, format phrase
- qualifiers, **See** Qualifiers, format phrase
  - absolute tab, 4-15
  - byte data, 4-13
  - character, 4-5
  - digits parameter, 4-3
  - fixed-point, 4-8
  - floating-point, 4-6
  - integer, 4-12
  - literal, 4-18
  - matching with target columns, 4-3
  - parameters to, 4-18
  - pattern data, 4-9
  - qualifiers parameter, 4-3
  - relative tab, 4-16
  - repetition parameter, 4-3
  - syntax, 4-2
  - syntax summary, 4-5
  - type parameter, 4-3
  - types of, 4-5
  - width parameter, 4-3

**Forming arrays**, 2-25

**FORTRAN**

- calling from APL, 6-24e

**Function**

- characteristics, 2-21
- kinds of, 2-21
  - parts
    - arguments, shape of, 2-9

**Function names**

- localizing, 3-6

**Function-definition mode**, 1-27

**Functions**

- APL specification, 2-42
- defining, 3-1
- results of, 3-3
- types of
  - ambivalent, 3-2
  - dyadic, 3-2
  - monadic, 3-2
  - niladic, 3-2

---

**G**

**Global symbols**, 3-6

- naming, 3-6

**Graphics**

- ASCII, 1-5, 1-8t

**Group logical name table**, 5-47

**Group names**, 1-32, 2-16

---

**H**

**Headers**

- of an operation, 3-2
- of functions, 3-1
- of records, 5-56

**Hexadecimal output**

- of internal data representation, 4-13

**HI file**, 1-15

**/HI Qualifier**, 1-15

**High minus sign**, 2-18

# I

---

## I/O

- file, 5-23
- terminal, 5-1
- variables, 5-1

## Identifier

- definition of, 2-16
- Illegal overstrike, 1-5, 2-19
  - in quote quad input, 5-4
  - in TTY mode, 1-39

- Immediate mode, 1-27
  - editing in, 1-28, 3-36

- Inactive workspace, 1-30

- Index origin, 2-35

## Indexed files

- See** Keyed files

## Indexing arrays

- See** Arrays

## Initialization

- file, 1-11, 1-12
- order of processing, 1-20
- parameters, 1-12
- streams
  - definition of, 1-11

## Initialization file

- creation, 1-12

## Input

- abort input signal, 1-33
- diverting, 5-11
- escaping from quad, 5-2
- prompt, 1-27
- quad, 5-2
- quad del, 5-5
- quote quad  $\text{\texttt{}}\text{\texttt{}}$ , 5-4
- requesting inside an operation, 5-3
- untranslated, 5-5

## Input device

- changing default, 5-11

## Input lines

- canceling, 5-2
- correcting, 1-29t
- deleting, 1-29t
- echoing, 5-2

## Input lines (cont'd)

- entering, 2-25
- length of, 2-25
- wrapping, 5-2

- Input prompt system variable, 5-2

- /INPUT qualifier, 1-16

- ) INPUT system command, 5-11

- Insert commas format phrase qualifier, 4-21

- Integer format phrase, 4-12

- /INTERFACE qualifier, 1-16

- Intermediate results of operation execution, 3-46

- Internal data types, 5-61

- Internal record format, 5-60

- Internal sequential files

- reading and writing, 5-25

## Items

- definition of, 2-1

# K

---

- Key-paired character set, 1-2, 1-24t, 1-37, 5-23

- Keyboard editing, 1-28

- Keyboard editing characters, 1-28t

- Keyboards, APL, 1-2

- Keyed files, 5-15

- assigning channels, 5-20

- reading, 5-27

- writing, 5-27

- /KY file organization switch, 5-20

# L

---

## Labels

- names of, 2-16

- operation-line, 3-6

- Lamp character, 3-14

- Latent expression system variable, 1-21

- Left-justify format phrase qualifier, 4-22

## Length

- of input lines, 2-25

- of names, 2-17

- of operations, 3-5

- LIB\$ERASE\_PAGE

- calling from APL, 6-20e

- LIB\$GET\_SCREEN**  
calling from APL, 6–21e  
**LIB\$PUT\_SCREEN**  
calling from APL, 6–20e  
**Limit**  
of axes in arrays, 2–3  
**Line**  
banner, 1–20  
canceling input, 5–2  
correcting input, 1–29t  
definition of, 2–16, 2–25  
deleting input, 1–29t  
echoing input, 5–2  
edit characteristics, 1–25  
entering input, 2–25  
length of input, 2–25  
typing beyond end of, 2–25  
wrapping input, 5–2  
**Line counter system function**, 3–44  
**Linking external routines**, 6–2  
**Literal format phrase**, 4–18  
**Local functions**, 3–6  
**Local operators**, 3–6  
**Local symbols**, 3–4, 3–6  
naming, 3–6, 3–7  
**Local variables**, value of, 3–8  
**Locked files**, 5–40  
**Locked operations**, 3–14  
**Locked records**, 5–42  
**Logical names**  
table of group, 5–47
- 
- M**
- Mailbox system function**, 5–48  
**Mailboxes**, 5–46  
**Map system function**, 6–3, 6–5  
**Matrix**  
definition of, 2–4  
domain definition, 2–9  
**MCS**  
translation, 5–73t  
**Messages**  
length of mailbox, 5–47  
primary, 2–42
- Messages (cont'd)**  
secondary, 2–42  
to and from other users, 5–46, 5–48  
**Minus sign**, 2–18  
**Mnemonics**  
ASCII keyword, 1–4, 1–5  
in quote quad input, 5–5  
**Mode**  
superedit, 1–36  
**Mode parameter**  
output, 5–23  
**Modes**  
APL operating, 3–1  
execution, 1–27  
function-definition, 1–27, 3–1  
immediate, 1–27, 3–1  
operator-definition, 3–1  
superedit, 3–32  
**Monadic functions**, 3–2, 3–38e  
**Monadic operators**, 3–3  
**MTH\$DACOSD**  
calling from APL, 6–19e  
**Multikey**  
See Keyed files  
**Multinational Character Set**, 1–44t, 5–67t  
**Multistatement line**  
error in, 2–24
- 
- N**
- Names**  
APL, 1–5t  
characters allowed in, 2–17  
groups, 1–32, 2–16  
labels, 2–16  
length of, 2–17  
localizing, 3–6  
rules for forming, 2–17  
symbolic, 3–5  
user-defined operations, 2–17  
variables, 2–16  
workspaces, 1–31  
**Near-integer**, 2–2  
**Negative format phrase decorators**, 4–24

- Negative numbers  
     representing, 2-18
- Negative sign, 2-18  
     replacing, 4-24
- Nested input list, 5-12
- Next-record pointer, 5-16, 5-26, 5-30
- Niladic functions, 3-2, 3-37e
- /NOEDIT Qualifier, 1-15
- /NOHI Qualifier, 1-15
- /NOINPUT qualifier, 1-16
- /NOINTERFACE qualifier, 1-16
- Non-APL files  
     reading, 5-56, 5-58
- Non-APL terminals, 1-2, 1-38  
     definition of, 1-4
- /NOSILENT qualifier, 1-19
- /NOTERMINAL qualifier, 1-19
- /NOVECTOR qualifier, 1-20
- Null password, 1-32
- Numbers  
     Boolean, 2-2  
     channel, 5-33  
     floating-point, 2-2  
     output precision of, 2-2  
     representing negative, 2-18
- Numeric arrays, 2-2
- Numeric constants, 2-18
- Numeric data  
     mixing with character, 4-10
- 
- O**
- Operating modes  
     APL, 1-27
- Operating system  
     returning to command level, 1-29t
- Operation execution  
     changing order of, 3-10  
     intermediate results of, 3-46  
     stopping, 3-11, 3-42
- Operations  
     editing, 3-17  
         See Character-Cell interface  
         See DECwindows interface  
         See VAXTPU editor
- Operations (cont'd)  
     pendent, 1-32  
     suspended, 1-32  
     types of  
         empty, 3-24  
         locked, 3-14  
         nested, 3-4, 3-5  
         stub, 3-24
- user-defined  
     debugging, 3-42  
     deleting lines, 3-27  
     displaying lines, 3-28  
     displaying operands, 3-42  
     editing, 3-23  
     editing the header, 3-31  
     errors in, 2-44  
     inserting lines, 3-26  
     labels for lines, 3-6  
     listing lines, 3-28  
     localizing names, 3-6  
     maximum lines, 3-5  
     naming, 3-2  
     naming rules, 2-17  
     parts  
         body, 3-1  
         delimiters for header, 3-4  
         header, 3-1, 3-2  
         result, 3-3  
         requesting input inside, 5-3  
         using system commands, 3-5
- Operators, 2-21  
     defining, 3-1  
     parts  
         adding lines, 3-25  
         suspended, 3-42  
     types of  
         dyadic, 3-3  
         local, 3-6  
         monadic, 3-3
- Operators, types of  
     user-defined, 3-1
- Optional character set with TTY, 5-23
- Origin  
     default index, 2-35

- Output**
- APL interpreter, 1–27
  - bare, 5–10
  - displaying, 5–6
  - file, 5–23
  - quad, 5–10
  - quotation marks, 2–19
  - suppressing terminal, 1–29t
- Output catenator, 2–25, 5–3, 5–8
- Output mode parameter, 5–23
- Overstruck characters, 1–4, 1–7t, 1–24t, 1–25t
- illegal, 1–5, 2–19
  - in quote quad input, 5–4
  - in TTY mode, 1–39
  - with tab character, 1–5
- 
- P**
- Panic exit, 1–29t, 1–33
- Parameters
- definition of, 1–11
  - initialization, 1–12
- Parameters, format phrases, 4–18
- Parentheses
- use in expressions, 2–22
- Password
- null, 1–32
  - workspace, 1–31
- Pattern data format phrase, 4–9
- Pendent operations, 1–32, 3–45
- editing, 3–43
- Permanent mailbox, 5–47
- Pervasive functions
- definition of, 2–10
- Physical device number mailbox, 5–48
- Pointer
- next-record, 5–16, 5–26, 5–30
- Positive format phrase decorators, 4–24
- Precedence of local symbols, 3–8
- Primitive functions, optimized, 3–54
- Print column pointer
- altering, 4–15, 4–16
- Printing Operations, 3–16
- Private mailbox, 5–47
- Process identification number, 5–48
- Process, suspended, 5–46
- Processing
- order of, 1–20
- Programming considerations, 3–54
- Prompts
- input, 1–27
- Protected del, 3–14
- Prototype
- definition of, 2–14
- Public mailbox, 5–47
- Pure data records, 5–55
- 
- Q**
- Quad Del input, 5–5
- Quad input, 5–2
- escaping from, 5–2
  - pending, 3–45
- Quad output, 5–10
- Qualifiers
- APL, 1–11
  - blank when zero, 4–21
  - definition of, 1–11
  - exponent digits, 4–23
  - file organization, 5–18
  - insert commas, 4–21
  - left-justify, 4–22
  - priority, 1–13
  - scale factor, 4–22
  - standard symbol substitution, 4–22
  - values, 1–14
  - zero fill, 4–23
- Qualifiers, APL
- /EDIT, 1–15
  - /HI, 1–15
  - /INPUT, 1–16
  - /INTERFACE, 1–16
  - /NOEDIT, 1–15
  - /NOHI, 1–15
  - /NOINPUT, 1–16
  - /NOINTERFACE, 1–16
  - /NOSILENT, 1–19
  - /NOTERMINAL, 1–19

Qualifiers, APL (cont'd)  
/NOVECTOR, 1-20  
/SILENT, 1-19  
/TERMINAL, 1-19  
/VECTOR, 1-20  
Qualifiers, format phrase, 4-19  
Question mark, 1-22  
Quiet functions, 5-1  
Quotation marks, 2-2  
    output, 2-19  
Quote quad input, 5-4

## R

Random access method, 5-16, 5-26  
Rank of arrays, 2-3, 2-7  
Read event flag system function, 5-45  
Records  
    headers for, 5-56  
    index for, 5-26  
    internal format of, 5-58  
    locked, 5-42  
    pure data, 5-55  
    releasing locked, 5-42  
    segmented, 5-14  
Reentrant APL interpreter, 1-1  
Reformatting data, 5-56  
Relative files  
    deleting records from, 5-26  
    end-of-file for, 5-26  
    reading and writing, 5-26  
Relative tab format phrase, 4-16  
Release system function, 5-42  
Replacing selected items in arrays, 2-42  
Report formatter  
    see Format phrases  
Report formatter system function, 4-1  
    result array, 4-27  
Representation  
    of dollar sign, 1-4  
Reset system function, 3-46  
Results of functions, 3-3  
Rewind system function, 5-30  
RMS, 5-15

## S

---

Saved workspace, 1-30  
Scalar  
    definition of, 2-3  
    domain definition, 2-9  
    extension, definition of, 2-10  
    product, definition of, 2-10  
Scale factor format phrase qualifier, 4-22  
Segmented records, 5-14  
Selective assignment, 2-42  
Semicolon, 2-25  
    use of, 5-8  
Sequential access method, 5-15  
Set event flag system function, 5-45  
Shadowing symbols, 3-8  
Shape  
    array, 2-2, 2-4  
    function argument, 2-9  
    indexing result, 2-40  
Shareable APL interpreter, 1-1  
Shared images, 6-2  
Signal system function, 3-49  
Signals  
    abort input, 1-33  
    attention, 1-28, 1-29t, 1-33  
Signals, abort input, 3-24, 3-36  
/SILENT qualifier, 1-19  
Single-strike characters, 1-5  
Singleton, 2-9  
    definition of, 2-3  
    extending arguments, 2-10  
    extension definition of, 2-10  
    shape, 2-5  
SMG\$ routines  
    calling from APL, 6-35e  
SORT, calling from APL, 6-22  
Spaces  
    as quote quad input, 5-4  
    in arrays, 2-3  
    use in APL, 2-22  
    whitespace, 2-22  
Specification function, 2-42

Standard symbol substitution format phrase  
    qualifier, 4-22

State indicator, 1-32, 3-44  
    clearing, 3-46

Statement  
    definition of, 2-24  
    order of executing, 2-24

Status  
    channel, 5-33  
    file, 5-33

Stop system function, 3-48

Stop vector, 3-48

STOPSET error, 3-48

Strand notation  
    definition of, 2-20

Strong attention signal, 1-33

Structure of arrays, 2-2

Stub operations, 3-24

Substituting selected items in arrays, 2-42

Superedit mode, 1-36, 3-32

Suspended operations, 1-32, 3-42, 3-45  
    editing, 3-43  
    restarting, 3-43  
    terminating, 3-44

Suspended process, 5-46

Symbol table, 3-5

Symbols  
    global, 3-6  
    local, 3-4, 3-6

System commands  
    in operations, 3-5

System commands in APL  
    edit, 2-34, 3-22  
    )INPUT, 5-11

System functions in APL  
    [], 5-23  
    [], 5-23  
    break, 3-49  
    channel status, 5-33  
    channels, 5-22  
    )CHANS, 5-22  
    )CHS, 5-33  
    clear event flag, 5-45  
    closing files, 5-31  
    )CLS, 5-31

System functions in APL (cont'd)

    )DAS, 5-32  
    deassigning files, 5-32  
    device characteristics, 5-35  
    )DVC, 5-35  
    )EFC, 5-45  
    )EFFR, 5-45  
    )EFS, 5-45  
    file I/O, 5-23  
    file sharing, 5-34  
    )FLS, 5-34  
    line counter, 3-44  
    mailbox, 5-48  
    )MAP, 6-3, 6-5  
    map external routine, 6-3, 6-5  
    )MBX, 5-48  
    read event flag, 5-45  
    release, 5-42  
    )RELEASE, 5-42  
    reset, 3-46  
    rewind, 5-30  
    )REWIND, 5-30  
    set event flag, 5-45  
    stop, 3-48  
    trace, 3-46  
    wait, 5-43  
    )WAIT, 5-43

System functions, APL  
    report formatter, 4-1

System manager, 5-44

System services  
    calling from APL, 6-33e

System variables in APL  
    bare output [] or [], 5-10  
    error message, 3-49  
    input prompt [], 5-2  
    quad del input [], 5-5  
    quad output [], 5-10  
    trap, 3-49

System variables, APL  
    latent expression, 1-21

## T

Tab character, 1-29t

  overstriking, 1-5

  use in APL, 2-22

Tab position

  absolute, 4-15

  relative, 4-16

Telephone

  disconnecting, 1-36

Temporary mailbox, 5-47

Terminal

  designators, 1-21

  font files, 1-26

  I/O, 5-1

  output

    displaying, 5-6

    suppressing, 1-29t

  overstruck characters, 1-4, 1-25t

  type, 1-19

/TERMINAL qualifier, 1-19

Terminals

  APL, 1-2, 1-22t, 1-37

  BIT, 1-22t

  COMPOSITE, 1-22t

  DECTERM, 1-23t

  GIGI, 1-22t

  HDS201, 1-22t

  HDSAVT, 1-22t

  HSD221, 1-22t

  KEY, 1-22t

  LA, 1-22t

  non-APL, 1-2, 1-4, 1-23t, 1-38

  Tektronix 4013, 1-22t

  Tektronix 4015, 1-22t

  VS, 1-23t

  VT102, 1-22t

  VT220, 1-22t

  VT240, 1-22t

  VT320, 1-23t

  VT330, 1-23t

  VT340, 1-23t

TPU

  See VAXTPU editor

TPU editor, 2-34

Trace system function, 3-46

Trace vector, 3-46

Trap system variable, 3-49

TTY

  character set, 1-4, 1-38, 1-39t, 5-23

  decimal point, in patterns, 4-11

  mnemonics

    in quote quad input, 5-5

  optional character set with, 5-23

  overstruck characters, 1-39

Type parameter

  with input function, 5-56

  with output function, 5-56

Typewriter-paired character set, 1-2

## U

Unconditional branching, 3-10

Untranslated input, 5-5

User-defined functions, 3-1

User-defined operations

  errors in, 2-44

  names, 2-17

User-defined operators, 3-1

## V

Value display, 5-6

Variables

  editing with the Character-Cell interface, 2-32

  editing with the DECwindows interface, 2-30

  I/O, 5-1

  names of, 2-16

VAXTPU

  APL interface, 1-17

  buffer, 1-17

VAXTPU editor

  editing variables, 2-27

  syntax form, 2-34, 3-22

Vector

  definition of, 2-3

  domain definition, 2-9

Vector (cont'd)  
  stop, 3-48  
  trace, 3-46  
Vector notation  
  definition of, 2-20  
/VECTOR qualifier, 1-20  
VMS  
  file specification, 1-31  
  subprocess, 3-58  
VMS SORT  
  calling from APL, 6-22e

## W

---

Wait system function, 5-43  
Weak attention signal, 1-33  
White space, 2-22  
Wildcards  
  in identifiers, 2-17  
Workspace

clearing, 1-30  
CONTINUE, 1-30  
definition of, 1-29  
inactive, 1-30  
names, 1-31  
  defaults for, 1-31t  
  format of, 1-31  
password, 1-31  
saved, 1-30  
size, 1-33  
space considerations, 3-57  
types, 1-30  
Writing external routines, 6-2  
WSPRINT.APL, 3-16  
*WSAPRINT*, 3-16

## Z

---

Zero fill format phrase qualifier, 4-23  
Zero format phrase decorator, 4-24



# How to Order Additional Documentation

---

## Technical Support

If you need help deciding which documentation best meets your needs, call 800-343-4040 before placing your electronic, telephone, or direct mail order.

## Electronic Orders

To place an order at the Electronic Store, dial 800-DEC-DEMO (800-332-3366) using a 1200- or 2400-baud modem. If you need assistance using the Electronic Store, call 800-DIGITAL (800-344-4825).

## Telephone and Direct Mail Orders

Your Location	Call	Contact
Continental USA, Alaska, or Hawaii	800-DIGITAL	Digital Equipment Corporation P.O. Box CS2008 Nashua, New Hampshire 03061
Puerto Rico	809-754-7575	Local Digital subsidiary
Canada	800-267-6215	Digital Equipment of Canada Attn: DECdirect Operations KAO2/2 P.O. Box 13000 100 Herzberg Road Kanata, Ontario, Canada K2K 2A6
International	_____	Local Digital subsidiary or approved distributor
Internal <sup>1</sup>	_____	USASSB Order Processing - WMO/E15 <i>or</i> U.S. Area Software Supply Business Digital Equipment Corporation Westminster, Massachusetts 01473

---

<sup>1</sup>For internal orders, you must submit an Internal Software Order Form (EN-01740-07).



# Reader's Comments

VAX APL  
User's Guide

AA-P142E-TE

---

Please use this postage-paid form to comment on this manual. If you require a written reply to a software problem and are eligible to receive one under Software Performance Report (SPR) service, submit your comments on an SPR form.

Thank you for your assistance.

I rate this manual's:	Excellent	Good	Fair	Poor
Accuracy (software works as manual says)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Completeness (enough information)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Clarity (easy to understand)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Organization (structure of subject matter)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Figures (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Examples (useful)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Index (ability to find topic)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Page layout (easy to find information)	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

I would like to see more/less \_\_\_\_\_

---

What I like best about this manual is \_\_\_\_\_

---

What I like least about this manual is \_\_\_\_\_

---

I found the following errors in this manual:

Page      Description

---

---

---

Additional comments or suggestions to improve this manual:

---

---

---

I am using Version \_\_\_\_\_ of the software this manual describes.

Name/Title \_\_\_\_\_ Dept. \_\_\_\_\_

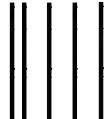
Company \_\_\_\_\_ Date \_\_\_\_\_

Mailing Address \_\_\_\_\_

Phone \_\_\_\_\_

Do Not Tear - Fold Here and Tape

**digital**™



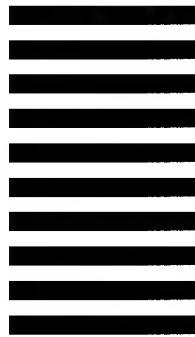
No Postage  
Necessary  
If Mailed  
in the  
United States

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL EQUIPMENT CORPORATION  
Corporate User Information Products  
PK03-1/D30  
129 PARKER STREET  
MAYNARD, MA 01754-9975



Do Not Tear - Fold Here