Xerox APL

Xerox 560 and Sigma 6/7/9 Computers

Language and Operations Reference Manual

90 19 31C

Xerox Corporation
701 South Aviation Boulevard
El Segundo, California 90245
213 679-4511

XEROX

Xerox APL

Xerox 560 and Sigma 6/7/9 Computers

Language and Operations

Reference Manual

90 19 31C

June 1975

Price: $8.50

© Xerox Corporation 1975

Printed in U.S.A.

REVISION

This publication, 90 19 31C, is a major revision of the Xerox APL LN/OPS Reference Manual, 90 19 31B, dated October 1973. Additions to this edition of the manual include: Appendix D, "Designing and Creating APL Indexed Files"; Appendix E, "APL/EDMS Interface"; and Appendix F, "APL Symbols". Changes to the previous manual are indicated by a vertical line in the margin of the affected page. These changes document the C00 version of the APL system for CP-V.

RELATED PUBLICATIONS

Title	Publication No.
Xerox 560 Computer/Reference Manual	90 30 76
Xerox Sigma 6 Computer/Reference Manual	90 17 13
Xerox Sigma 7 Computer/Reference Manual	90 09 50
Xerox Sigma 9 Computer/Reference Manual	90 17 33
Xerox Control Program-Five (CP-V)/TS Reference Manual	90 09 07
Xerox Control Program-Five (CP-V)/TS User's Guide	90 16 92
Xerox Control Program-Five (CP-V)/BP Reference Manual	90 17 64
Xerox Extended Data Management System (EDMS)/Reference Manual	90 30 12
Xerox Extended Data Management System (EDMS)/User's Guide	90 30 37

Manual Content Codes: BP - batch processing, LN - language, OPS - operations, RP - remote processing, RT - real-time, SM - system management, TS - time-sharing, UT - utilities.

The specifications of the software system described in this publication are subject to change without notice. The availability or performance of some features may depend on a specific configuration of equipment such as additional tape units or larger memory. Customers should consult their Xerox sales representative for details.

CONTENTS

1. INTRODUCTION 1

2. USING APL 4

APL Log On/Log Off Procedures 4
Logging On 4
Logging Off 6
APL Terminal Keyboard 6
General APL Input 7
Character Set 7
Names 7
User Input Versus Computer Output 8
Line Corrections During Input 8
Execution and Definition Modes 9
Prompts 9
Comments 10
Control Keys 11
Statements and System Commands 11
Variables and Operators 11
Defined Functions 12

3. COMMON ELEMENTS IN APL 13

Constants 13
Numeric Constants 13
Text Constants 14
Names 15
Name Format 15
Name Usage 15
Variables 16
Local and Global Variables 17
Arrays and Indexing 19
Operators and Arguments 24
Function References 39
Assignment 40
Simple Assignment 40
Multiple Assignment 41
Indexed Assignment 41
Input/Output 42
Input/Output Devices 42
General Input/Output 42
Types of Input 42
Output 44

4. EXPRESSION EVALUATION 48

Order of Evaluation 48
Right to Left 48
Precedence of Operators 48
Parentheses 48
The Value of a Variable Versus Its Name 48
Syntax Considerations 49
Default Terminal Output 49
Errors and Breaks 50

5. APL OPERATORS 52

Scalar Operators 53
Arithmetic Group 54
Relational Group 63
Logical Group 65
Composite Operators 68
The d/Operator (Reduction) 68
The d\ Operator (Scan) 70
The d.d Operator (Inner Product) 71
The ∘.d Operator (Outer Product) 73
Mixed Operators 74
The ? Operator (Roll and Deal) 74
The ⍳ Operator (Index Generator and Index Of) 74
The , Operator (Ravel and Catenation and Lamination) 75
The ⍴ Operator (Dimension and Reshape) 77
The ⌽ Operator (Reversal and Rotation) 78
The ⍉ Operator (Transposition) 79
The ⍋ Operator (Grade Up) 82
The ⍒ Operator (Grade Down) 82
The ⊥ Operator (Base Value or Decode) 82
The ⊤ Operator (Representation or Encode) 83
The / Operator (Compression) 84
The \ Operator (Expansion) 85
The ↑ Operator (Take) 86
The ↓ Operator (Drop) 86
The ∈ Operator (Membership and Execute) 87
The ⌹ Operator (Matrix Inversion and Matrix Divide) 90
I-Beam Functions 92
T-Bar Functions 93

6. APL STATEMENTS 95

Comment Statements 95
Branch Statements 96
Statement Labels 98
Assignment and Nonassignment Statements 99
Compound Statements 100

7. DEFINED FUNCTIONS 101

User-Defined Functions 101
Creating User-Defined Functions 101
Directives 105
Editing User-Defined Functions 107
Issuing System Commands 115
Function Execution 116
State Indicator 118
Locking Functions 119
Intrinsic Functions 120
ΔFMT, PAGE, NLINES, HEADER, VFCHAR, and ΔXL 120
DELAY 120
DIGITS, ORIGIN, TABS, and WIDTH 120

SETFUZZ 120
SETLINK 121
FIO 121
FIOE 121
ERRN, ERRF, and ERRX 121
ΔGRF 121
ΔTE, ΔCR, and ΔWM 121

8. SYSTEM COMMANDS 122

Workspace Concept 123
Active Workspace 123
Saved Workspace 123
CONTINUE Workspace 123
User Accounts 124
Passwords 124
Commands 124
)CATCH 128
)CLEAR 130
)CONTINUE and)CONTINUE HOLD 130
)COPY 131
)DIGITS 133
)DROP 133
)ERASE 134
)FNS 134
)GROUP 135
)GRP 136
)GRPS 136
)LIB 137
)LOAD 137
)OBSERVE 138
)OFF and)OFF HOLD 139
)OPR and)OPRN 140
)ORIGIN 140
)PCOPY 141
)QLOAD,)QCOPY, and)QPCOPY 141
)SAVE 141
)SEAL 142
)SET 142
)SYMBOLS 147
)TABS 148
)TERMINAL 149
)VARS 150
)WIDTH 150
)WSID 151

9. REPORT FORMATTING 152

ΔFMT 152
Format Specifications 152
Format Specifications Versus Data Types 153
The Format Statement (Left Argument) 153
The Data List (Right Argument) 153
Operation of ΔFMT 153
Scalar Arguments 153
Vector Arguments 154
Matrix Arguments 155
Forms of Output Values 155
Qualifiers and Affixture Codes 156
Use of Result 157
Error Exits 157
Other Output Formatting Aids 157
ΔXL 158

10. EXECUTION STOPS 159

Normal Stop 159
Execution Break 159
Stop for User Input 159
Stop Control Vector 160
Error Stop 161

11. GRAPHICS 163

User Graphic Functions 163
Plotting Functions: DRAW, VS, INT 163
Scaling Functions: SCALE, NOSCALE, CENTER, WHATSCALE 164
Window Functions: WINDOW, WHATWINDOW 165
Auxiliary Plotting Functions: AXES, BOX, DASH, SET, PUT, ERASE, HOME 165
Graphic Input Functions: ⍞Input, WHATCHAR, WHATCOORD 165
Other Functions: SIN, COS, EXP, STRAPIS 166
Direct Control of Graphic I/O 166
⍞Output 166
ΔGRF Intrinsic 167
Unscaled Graphic I/O 169
Unscaled Output 169
Unscaled Input 170

12. WORKSPACE MANAGEMENT FUNCTIONS 171

Namelists and Linelists 171
Canonical Representation 171
Workspace Management 172
Text Editing 175

INDEX 231

APPENDIXES

A. ERROR MESSAGES 179

Sidetracking on Errors and Breaks 184
Setting Sidetracks 186
The Dynamics of Sidetracking 187
Considerations After Gaining a Sidetrack 187
Aids for Sidetrack Users 188

B. NONSTANDARD INPUT/OUTPUT 189

Standard and Nonstandard Devices 189
Using Nonstandard Input/Output 189
Terminal Declaraction 189
Changing Terminal Declaration 189
Input/Output Translation 190
Teletype Usage 192
ESCAPE Key Sequences and APL 192
Operational Differences 193
Batch Operation 193
Intended Usage 193
Input/Output Device Assignments 194
Card Input Format 194
Error Response 195
Operational Differences 195
Tektronix 4013 Usage 195
ASCII Mode 195
APL Mode 195
Logging On 196
Line Editing 196
Strapping Options 197
Data Transmission Rates 197
Non-APL 2741 Terminals 197
Applications 197
Operational Considerations 197
Blind I/O 198
Using Blind I/O 198
Blind I/O on a Device 198
Blind I/O for Files 199
File Input/Output 200
Creating the Set of File I/O Operators 200
Structure of APL Files 201
Opening and Creating Files 202
Closing Files 203
Maintaining Component Range and Current Component Value 204
Key Values Versus Component Values 204
Writing APL Records 205
Writing Non-APL Records 205
Reading APL Records 206
Reading Non-APL Records 207
Deleting Records or Components 207
Sequential Access to Existing APL Files 207
Sequential Access to Non-APL Files 208
Converting Data Types 208
Controlling Access to Shared APL Indexed Files 209
Listing File Names and Numbers 209
Error Reporting 210
Generation of File I/O Subsystems 211

C. INTRINSIC FUNCTIONS 216

D. DESIGNING AND CREATING APL INDEXED FILES 217

Limits and Trade-Offs 217
File Structure 218
Efficiency Considerations 219
Procedure for Creating APL Indexed Files 220

E. APL/EDMS INTERFACE 223

F. APL SYMBOLS 226

FIGURES

1. Example of Logging On and Logging Off APL System 5
2. APL Terminal Keyboard 7
3. Example Showing Effect of Shadowing 19
4. Summary of Formats for Branching 98
B-1. EBCDIC and APL Codes 191

TABLES

1. APL Operators 25
2. I-Beam Functions 93
3. Examples of Defined Functions 102
4. Displaying and Editing Defined Functions 108
5. Summary of System Commands 124
6. ΔGRF Calls 168
A-1. Error Messages 179
A-2. Items Subject to Sidetracking 185
B-1. Xerox Line Printer Graphic Codes 192
B-2. Examples of Problems with False Terminal Declaration 192
B-3. Translation Equivalences for Nonstandard Devices 211
E-1. EDMS Interface Functions 223
F-1. APL Symbols and Names 226

1. INTRODUCTION

This manual describes Xerox's implementation of the APL language – hereafter referred to as Xerox APL, or simply as APL.[t] It defines the language and the general operations of the processor under control of Xerox 560 and Sigma 6/7/9 Control Program-V (CP-V). This manual is intended primarily for use as a reference document by experienced APL programmers. Beginning APL users may find it useful to consult some good APL primers[tt] to augment this manual.

APL is an interpretive, time-sharing, problem-solving language. As an interpretive language, APL does not wait until a program is completed to compile it into object code and execute it; instead APL interprets each line of input as it is entered to produce code that is immediately executed. As a problem-solving language, APL requires minimal computer programming knowledge; a problem is entered into the computer and an answer is received, all in the APL language.

Because APL is powerful, concise, easy to learn, and easy to use, it is widely used by universities, engineers, and statisticians. It also has features that make it attractive for business applications where user interaction and rapid feedback are key issues. One of APL's major strengths is its ability to manipulate vectors and multidimensional arrays as easily as it does scalar values. For example, a matrix addition that might require a number of statements and several loops in other languages can be accomplished as A+B in APL. This type of simplification exemplifies APL's concise power.

Xerox APL is compatible with other APL systems (such as APL/360) and incorporates a broad range of improvements. Many of these improvements are unavailable on other APL systems.

- On-Line and Batch Operation – Complete flexibility of operation is provided. Programs may be developed and executed in either mode. The batch mode is advantageous for either long execution times or voluminous output, while the on-line mode is more advantageous for interactive program development and moderate amounts of execution times and output.

- Operation from Terminals and Teletypes without APL Characters – APL characters may be represented by combinations of alphanumeric and special characters in order to allow programs to be created or modified at any terminal or teletype supported by CP-V.

- Input/Output Assignment Control – An APL system command,)SET, has been added to allow the assignment of normal and 'blind' (see below) I/O to files and devices such as the line printer or magnetic tapes, and to establish format control of printed output.

- Fast Formatted Output – A user-specified format description can be used in lieu of the default automatic formatting. This facilitates the preparation of reports and tables.[ttt]

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

  The APL file I/O system operates with either CP-V keyed files or 'indexed' files (using CP-V random file access). Keyed file access may be with numeric keys (compatible with Xerox EDIT, BASIC, etc.) or text keys (allowing access to any CP-V keyed file). Indexed files may be accessed in shared update mode, using the CP-V Enqueue-Dequeue feature to support shared access control.

---

[t] APL is an acronym for A Programming Language, the language invented by Kenneth Iverson.

[tt] Two such publications are: APL/360 An Interactive Approach by Leonard Gilman and Allen Rose (John Wiley and Sons Inc, New York, 1970); and APL User's Guide by Harry Katzan, Jr. (Van Nostrand Reinhold Company, New York, 1971).

[ttt] This feature was invented and introduced by I.P. Sharp Associates Ltd, Toronto, Canada.

- APL-EDMS Interface – Installations supporting Xerox Extended Data Management System may now access structured shared EDMS data bases via APL. A workspace with a set of 57 user functions supports EDMS operations via APL.

- Compound Statements – More than one statement can be included on the line, using semicolons for separation. Since an element of a compound statement can be a branch, this feature permits conditional execution control within a single statement of a function.

- 'Blind' Input/Output – Blind input/output is a form of terminal input/output that permits untranslated input and output of character data. It is designed to facilitate the use of graphics terminals or other special devices with APL. By the use of the)SET command, blind I/O may also be used to create or access sequential files or routed to devices such as the line printer or magnetic tapes.

- Unequally Spaced Tabs – Tab settings can be unequally spaced as well as equally spaced.

- Easy Function Copying – An entire function can be copied simply by changing the name of an already defined function.

- SI-Damage Protection – Limited protection is provided against SI damage during function definition.

- Expansion of Identity Operator – The identity operator (that is, monadic +) is legal for all domains, not just numeric domains.

- Expansion of Indexing – An indexed argument can itself be indexed.

- Easy Function Line Appendage – In a function edit directive that uses both line number and column number, a column number of zero is taken to mean that the user wishes to append to that line. The line will be displayed with the carrier positioned after the last character, awaiting further characters for that line.

- Enhancements to System Commands –

  - The)SEAL command has been added, to provide protected workspaces. When)SEAL is executed, the current workspace is saved in a special mode with all user functions locked. A sealed workspace cannot be saved or copied by APL and cannot be accessed by processors other than APL except by the originator.

  - The)SET command has been added for on-line control of normal and 'blind' I/O, described earlier.

  - The)TERMINAL command has been expanded to allow independent setting of input and output terminal translation tables.

  - If the following commands are issued without a new value being specified, the current value (i.e., "IS value") is displayed:

)ORIGIN

)WIDTH

)DIGITS

)TABS

)WSID

)TERMINAL

    Similarly, a)SYMBOLS command that does not specify a new value produces a display of the form "x UNUSED OF y", where x is the number of unused entries in the symbol table whose size is y. The)WIDTH command accepts a new width setting in the range from 30 to 254.

- Quiet load and copy commands)QLOAD,)QCOPY, and)QPCOPY have been added to Xerox APL. These commands suppress the "SAVED" message when loading or copying successfully.

- Options have been added to)SI or)SIV commands to allow immediate clearing of the state indicator and also to control function suspension due to errors.

- The)FNS,)VARS, and)GRPS system commands have been extended to allow a range of names to be listed.

- The)COPY and)PCOPY commands have been extended to allow copying a list of objects as well as a single object.

- Autostart. An option on the)SAVE command permits saving workspaces such that execution of specified statement is initiated automatically when the workspace is loaded.

- Availability of Other CP-V Facilities – Most competitive versions of APL are offered primarily as dedicated APL-only time-sharing systems. A subscriber to Xerox APL may use other CP-V processors such as Edit, PCL, and BASIC from the same terminal during the same run.

- The 'execute' Operator – Monadic ϵ allows execution of text strings as though they were evaluated input. The most significant aspect of this feature is that system commands may be formed and executed from within a user-defined function.

- Function Editing in Evaluated Input and 'Execute' Modes – Permits listing or modifying a function while in evaluated input mode or in an 'execute' operation.

- Graphics Capability[†] – Xerox APL supports the Tektronix 4013 terminal for standard APL processing. In addition, the APL processor contains features and a function for the specific purpose of accommodating the graphics input and output capability of that terminal.

- Observation of Intermediate Results – The)OBSERVE command has been added to Xerox APL. This permits the user to view intermediate results while APL interprets a statement.

- Catching Assignments – A debugging aid, the)CATCH command, has been incorporated in Xerox APL. This command permits the user to catch (or intercept momentarily) every assignment to a named variable immediately following each assignment. The assignment is "caught" by means of a function defined by the user according to his debugging requirements.

- Error and Break Control – Xerox APL has a facility to allow the user selective and dynamic control over errors and breaks. Since this facility permits bypassing of standard APL handling of breaks and errors, it is called the "sidetracking" capability.

- Text Editing Functions – A set of five functions has been added to facilitate manipulation of text strings in APL.

- Canonical Representation Functions – Functions have been added to convert user-defined functions into APL text strings and to convert text strings into user-defined functions. This facilitates program-controlled display and editing of user functions.

- Workspace Management Functions – A set of nine functions provides a wide variety of information concerning the user workspace in the form of APL results, which can be used to display and manage workspaces under program control.

[†]This feature was originally developed at the University of California, Irvine, under the direction of Dr. Alfred Bork.

2. USING APL

APL Log On/Log Off Procedures

The procedures described in this chapter are for operation on a standard APL terminal (that is, a terminal with an APL typeball).[†] See Appendix B for procedures on other terminals.

Logging On

Before the user can communicate with APL, he must prepare the APL terminal for use, establish a connection with CP-V, and then establish a connection with APL. He does this as follows:

1. Preparing the APL terminal for use:

   a. Position the keyboard ON/OFF switch at ON. This switch is usually located in the lower right corner of the keyboard (Figure 2).

   b. Position the LOCAL/REMOTE switch at REMOTE. This opens the communications line between the terminal and the computer. When this switch is positioned at LOCAL, the terminal is not connected to the computer and functions as an ordinary typewriter.

   c. Establish communications with the computer via a telephone line connected to the terminal as follows:

      (1) If the telephone has an ON/OFF switch, position it at ON.

      (2) Pick up the telephone receiver and listen for a normal dial tone. (Some telephones may have a button labeled TALK; for these telephones, press the TALK button before picking up the receiver.)

      (3) Dial the telephone number for the computer.

      (4) When a high-pitched tone is heard, place the receiver in the receptacle provided. (Some telephones may have a button labeled DATA; for these telephones, press the DATA button and hang up the receiver.)

   These operating procedures apply to a typical APL terminal. Operating procedures for other terminal models may differ slightly, and so may the names of the switches described above. In addition, some terminal models do not have telephone equipment attached, and require only steps a and b above to establish a direct-line communication between the terminal and the computer. For most terminal models, however, connection with the computer is established via a telephone line as described above.

2. Logging on to CP-V (refer to Figure 1 for an example of each of the following steps):

   a. Type an asterisk and then strike the RETURN key. The computer responds by typing the following message at the terminal:

      ```
      XEROX CP-V AT YOUR SERVICE
      ON AT time and date
      LOGON PLEASE:
      ```

      where "time and date" are the actual time and date, and the last line is a prompt for the user to key in identifying code words.

[†] The APL typeball can be used with several terminal models. Among these are the IBM 2741, the DATEL 20-31, the DURA 1021 and 1051, the NOVAR 5-50, and the TST 707.

* (RET)	The user calls the CP-V system.
XEROX CP-V AT YOUR SERVICE *ON AT 08:52 AUG 15,'72* *LOG ON PLEASE:* 356256,*REJ073453421*(RET)	The CP-V system identifies itself, states the time and date, and requests that the user log on. In response, the user types in his account number (356256) and user identification (REJ073453421). He does not use a password in this example.
8:52 03/31/72 356256 18-36	A page heading is printed by the system; the items of information in the heading are, in order: time, date, account number, and two internal identifiers.
∘*APL*(RET) *APL 07/27/73* *CLEAR WS*	†CP-V TEL types a prompt character, and the user requests the APL processor. APL acknowledges control and prints the current version of APL and the workspace status.
⋮	APL program
)*OFF*(RET)	The user logs off both APL and CP-V. Any information in active workspace is destroyed. Also see the)OFF HOLD,)CONTINUE, and)CONTINUE HOLD commands in Chapter 8.
CPU = .0124 CON = :02 INT = 5 CHG = 0	Summary information for session: the user has used .0124 minutes of central processor time (CPU = .0124); he has been connected to the terminal (from dialing up to the end of summary) two minutes (CON = :02); he has interacted with the system five times (INT = 5); the charge is 0 charge units, an installation-dependent value (CHG = 0).

Note: All characters typed by the system are shown underlined. Everything else has been typed by the user. In addition, the (RET) symbol indicates the RETURN key.

---

†An option is now provided for entering APL with a statement appended to the APL call, for examples:

∘ APL	)QLOAD MYWORKSPACE	Loads the specified workspace (see QLOAD, Chapter 8).
∘ APL	)TERMINAL 13	Identifies user terminal as type 13 (see "TERMINAL", Chapter 8).

Figure 1. Example of Logging On and Logging Off APL System

b. Type identifying code words – account number, user identification, and possibly a password, in that order, separated by commas – and strike the RETURN key. The account is the billing number, the user identification is the personal or group identification, and the password is an account-protection feature assigned by the system manager or the user (see the PASSWORD command in the Xerox CP-V/TS User's Guide, 90 16 92). The account number and password (if present) each consist of from 1 to 8 alphanumeric characters (any combination of the letters A-Z and the numbers 0-9), and the user identification consists of from 1 to 12 alphanumeric characters. (See the log-on example in Figure 1, where 356256 is the account, and REJ073453421 is the user identification.) The code words must be the same as in the computer's list of authorized users. If they are not the same, the computer will request that the user reenter them. If the user's log-on code is accepted, the computer will print a page heading, skip several lines, and then print a ◦ symbol to prompt for input (at the TEL level of CP-V).

3. Calling the APL system:

   Type the command APL and strike the RETURN key to access the APL system. APL acknowledges control by typing the message APL 07/27/73 (the version of APL being used), and a message indicating whether a clear workspace is available or the CONTINUE workspace has been loaded. The user is now in the APL system and can enter any APL assignments, statements, function definitions, system commands, etc. An example of calling APL is shown in Figure 1.

4. Calling from other load modules:

   APL can also be called from other CP-V load modules, in which case, normal exit from APL is a return to the calling module. (See M:LINK in CP-V BP Reference Manual, 90 17 64.)

Logging Off

The user can log off the APL system via any of the following APL system commands:

```
)OFF
)CONTINUE
)OFF HOLD
)CONTINUE HOLD
```

The first two commands,)OFF and)CONTINUE, end the terminal session. That is, they log the user off both APL and CP-V and print a summary-of-accounting message (a summary of the terminal and computer time for the session). After this summary message the user may turn off the terminal and all associated equipment and hang up the telephone receiver.

The last two commands,)OFF HOLD and)CONTINUE HOLD, log the user off the APL system and return control to the CP-V TEL subsystem (which prompts for input with the ◦ symbol). To end the terminal session after returning control to CP-V, enter the TEL command OFF (which produces a summary-of-accounting message and disconnects the terminal from the computer).

Both forms of the)OFF command cause any information in active workspace to be lost, while both forms of the)CONTINUE command cause any information in active workspace to be saved under the name CONTINUE (in the account number used when logging on). These commands are described in more detail, with examples, in Chapter 8, "System Commands".

APL Terminal Keyboard

The APL terminal keyboard arrangement is shown in Figure 2. This keyboard is similar to a standard keyboard in that alphabetic and numeric characters are in the standard positions. (Alphabetic characters print in italic capital letters.) Most of the remaining characters, however, are APL symbols and will not be the same as on a standard keyboard. In addition, some familiar characters are located in different positions on an APL keyboard:

```
_ - + ? ' ( ) ; : * = .
```

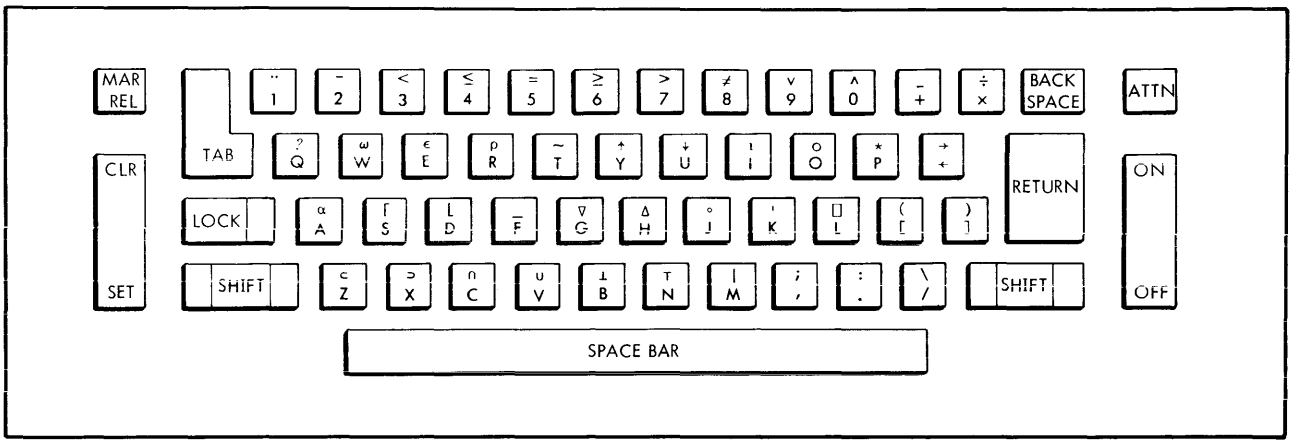


Figure 2. APL Terminal Keyboard

The SHIFT key is used for uppercase characters the same as it is on standard terminals. The RETURN key indicates that the user has entered something and is ready for the APL system to respond. Notice that most of the APL special symbols are included as uppercase characters. Some even appear to have a relationship (often phonetic) to the corresponding numeral or alphabetic letter (to facilitate remembering where they are). For example, α over A, ⊥ (for base) over B, ∊ over E, ⍳ (for index) over I, ' (for quote) over K, | (for magnitude) over M, ○ (for circle symbol) over O, * (for power) over P, ? (for query) over Q, ⍴ (for rho) over R, ⌈ (for ceiling) over S, and ⍵ over W.

General APL Input

Character Set

One of APL's most unique characteristics is the richness of its character set. An APL keyboard (Figure 2) normally has 88 printing graphics. All of these are legal characters. In addition, backspacing may be used to create the following overstrikes, all of which are legal APL characters:

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

⌽ ⍉ ⊖ ⍟ ⍒ ⍋ ! ⌶ ⍞ ⍇ ⍝ ⍱ ⍲ ⍣ ⍠ ⍫ ⍭ $ ⍙ ⍰ ⍱ ⍲

Other legal characters are blank (the space bar), tab (the TAB key, treated as one or more blanks), and carriage return (the RETURN key). Two other characters are also accepted for control purposes: the INDEX key (present on some terminal models) and the ATTN key discussed below under "Line Corrections During Input" and "Control Keys".

Names

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

1. All names except workspace names can contain from 1 to 77 characters. Workspace names can contain from 1 to 11 characters. More characters may be used, but they will be ignored.

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

3. Names cannot begin with a number or SΔ or TΔ.

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

Some examples of names are

```
Δ     PAYROLL     BΔ1     S1234     TEMPERATURE
```

User Input Versus Computer Output

The user can enter input whenever a keyboard is unlocked. (One way to determine if the keyboard is unlocked is to depress the SHIFT key; if it is unlocked, the typeball will rotate slightly.) As soon as the user has typed his input and depressed the RETURN key, the APL system takes control and locks the keyboard. The keyboard remains locked to the user until the APL system has processed the input, printed any result, and prompted for more input, usually by indenting six spaces from the left margin. (A six-space indentation is an APL system prompt for user input. See "Prompts" later in this chapter.)

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

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

Everything at the left margin in this example was typed by the APL system, while everything indented was typed by the user.

Line Corrections During Input

A line can be corrected during input as long as the RETURN key has not been struck. Simply backspace to the error (via the BACKSPACE key)[†] and strike the ATTN key. This produces an inverted caret below the character reached by backspacing, and it signifies that everything from that point on has been deleted from system memory. The user can then type the correction. For example, suppose the user mistakenly types 30÷20 instead of 20+20. He can correct this as follows:

```
      30÷20
      ∨
      20+20
40
```

A variation of this scheme is available for terminals having the INDEX key (line-feed only). The procedure is to backspace to the error and strike the INDEX key. The platen rolls up one line, and the user can immediately type the correction. This is a quicker correction method than the ATTN key scheme. The computer does not need to respond (that is, no inverted caret is displayed). An example follows:

```
      20÷20
        +20
40
```

[†] A word of caution about the BACKSPACE key: it should be struck gently since on some terminals it may repeat if depressed fully. The repeat feature of this key is, of course, useful when desiring a lengthy backup; for instance, when deleting an entire line.

Another correction method can be employed if the user discovers that he has omitted a character and that there is room enough to insert it. As long as the RETURN key has not been struck, the user can simply backspace to where he wants to insert the character and type it. For example, suppose the user types the following line and notices that one left parenthesis is missing:

```
(10H)*2)+(20H)*2
```

He can simply backspace and type the required left parenthesis:

```
((10H)*2)+(20H)*2
```

This illustrates that it is not always necessary to enter characters in order. The user can leave blanks in a line and then backspace and fill them in. As a rule, APL interprets what the user sees at the terminal; this rule is known as visual fidelity.

Execution and Definition Modes

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

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

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

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

Prompts

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

Direct-Line Prompt

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

```
      2+2
4
      4÷2
2
```

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

Function-Line Prompt

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

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

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

Quad Prompt

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

```
      A←⎕÷8
⎕:
      7×2×4
      A
7

      ANSWER←⎕
⎕:
      'YES'
      ANSWER
YES
```

Quote-Quad Prompt

The quote-quad symbol ⍞ (a quad symbol overstruck with a quote) is used to enter literal character data. It is executed similarly to the quad symbol except that no input symbol is printed to signal the user and no six-space indentation takes place. The user enters his character data without enclosing it in quotes. For example:

```
      A1←⍞
YES
      A1
YES
```

Comments

Comments can be written on separate lines or can follow (that is, be tacked onto) statements. They may be included on any line except a system command line or a function edit control line. To enter a comment, type the symbol ⍝

and follow it with the comment. This symbol is produced by typing a ∩ symbol (upper shift C) and overstriking it with a ∘ symbol (upper shift J). Any valid APL characters may appear to the right of the ⍝ symbol. The ⍝ and any characters to the right are ignored in APL expression evaluation but will be printed if the line is displayed. Examples of comments are shown below:

```
      ⍝THIS IS A COMMENT.
      A←B×B   ⍝SET A = B-SQUARED.
[3]   X←Y+5   ⍝COMMENT: X IS SET TO Y+5
```

Control Keys

As mentioned earlier, the APL system accepts two "characters" for control purposes – the ATTN key and the INDEX key. The ATTN key is used to interrupt execution and to initiate editing. The INDEX key (present on some terminal models) is used to initiate editing, as an alternate to the ATTN key.

Statements and System Commands

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

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

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

```
)SAVE NEWJOB
)LOAD OLDJOB
)OFF HOLD
)DIGITS
```

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

Variables and Operators

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

```
A←5
B2←1 2 3
ABC←5+4
B3←A+B2
```

Some character symbols indicate that basic APL operations, such as addition or multiplication, are to be performed. These symbols are called operators. (Some APL programmers refer to these as "primitive functions", but

such terminology is not used in this manual in order to avoid confusion with "defined functions" described in Chapter 7.) Some operators can be monadic (have one argument) or dyadic (have two arguments). Some examples of operators are

× + ⌈ ÷

The domain and range of operator arguments and a list of all the operators are presented in Chapter 3 under "Operators". Chapter 5 is devoted to a detailed discussion of each operator.

Defined Functions

In addition to the mathematical functions included as APL operators, the APL system permits the user to define a function, name it, and store it in his active workspace. The function can then be referenced by name in subsequent statements, either as a program by itself or as a mathematical operation used in a formula. To define a function, the user enters it statement by statement while the APL system is in the definition mode. This mode begins when the user types a del symbol ∇ and ends when he types another ∇. For a detailed description of defined functions, see Chapter 7.

3. COMMON ELEMENTS IN APL

Constants

Numeric Constants

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

```
0 1 2 3 4 5 6 7 8 9 . ¯ E
```

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

Practically any size number can be entered in decimal form (as opposed to exponential form), the only limit being the length of an APL line. (Internal computations are carried out with at most 16 digits precision, however.) The APL system ignores leading and trailing zeros, so that the user need enter only the parts of numbers required for calculations. Thus there is no need for the user to enter data as all integer or all fractional. For example, the user may enter the number one as 1.00, 001.0, 1, etc. Examples of numeric constants entered in decimal form are shown below:

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

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

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

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

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

[†]The negative sign and the minus sign should not be confused with another similar character, the underscore, found over the letter F on the keyboard.

The maximum and minimum representable numbers in Xerox APL, expressed in exponential form, are

7.237005577E75[†]

¯7.237005577E75

A numeric constant can consist of a single number (as already illustrated) or a vector of numbers with blanks separating each number. (One or more blanks may be used on input.) A vector of numbers used as a constant will be treated as a single entity. Examples of numeric vectors are shown below:

```
1  2  3
1  2.5  ¯726E12
2.71828  3.41416  .70714  0
```

It should be noted that noninteger values are handled internally as "double precision floating" numbers. Fractions that are representable exactly in decimal notation, such as .1, are not exactly representable in this internal form. In some instances this will cause results of operations to deviate from expected results, particularly if the anticipated result is displayed to 16 decimal places or is a near-zero value.

Text Constants

Text constants – commonly called literals – are enclosed in quote symbols and can contain any keyboard character including a legal overstrike and the space character. The quote symbols are used to distinguish a text constant from a number, the name of something, or a construct in the language. They are not printed in the display of the literal. For example:

```
      A←'ABCDEF123456'
      A
ABCDEF123456
```

In this example the name A has been assigned the value of the text constant. The next line is a request to display the value of A.

If a quote character is used within a literal, it must be represented by a double quote. Use of the quote character is shown below:

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

One character enclosed in quotes is a scalar. More than one character in quotes is treated as a vector. Text vectors may be generated, compared for equality, indexed, and catenated just like numeric vectors. The characters in a text vector are printed without spaces when the vector is displayed. Some more examples of the use of text vectors are shown below (the operators used below are described in Chapter 5):

```
      A←'ABCDEFGH'
      B←'ABCCEEGH'
      A=B
1  1  1  0  1  0  1  1
      'SYMBOL'='SYMBOL'
1  1  1  1  0  1
      A1←'ABCDEFGHIJKLM'
      4⍴A1
ABCD
      A1[2]
B
```

[†]If a number is created with magnitude greater than 7.23705469E75, comparison operations with that number will give DOMAIN ERR because of the 'fuzz' concept in APL.

A text constant can contain one or more carriage returns. If a carriage return occurs before the closing quote is given, APL "prompts" by accepting the next line of input beginning at the left margin. This occasionally confuses the user who forgets to issue a closing quote; for example:

```
      A←'TEXT CONSTANT
WHY AM I AT THE LEFT MARGIN?
OH, FORGOT THE CLOSING '
      A
TEXT CONSTANT
WHY AM I AT THE LEFT MARGIN?
OH, FORGOT THE CLOSING
```

Names

Name Format

All of the following constituents of the APL language have names (sometimes known as identifiers) so that they may be easily referenced: variables, functions, groups, statement labels, and workspaces. A name can only include letters, digits, and the Δ character, and except for digits these characters can be underscored. A name cannot start with a digit or the combination SΔ or TΔ.

Lengths of names vary, depending on their use. The names of variables, functions, groups, and statement labels can be of any length up to 77 characters.[†] Workspace names – also known as filenames in Xerox APL – can be of any length but only the first 11 characters are retained by APL.

Name Usage

The uses of APL names are described below:

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

2. Function names are treated briefly later in this chapter under "Function References", and in detail in Chapter 7, "Defined Functions".

3. A collection of names can be referenced all at one time by giving the group a name. Included in the group can be the names of variables, functions, and other groups. See the)GROUP command in Chapter 8.

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

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

[†]The high limit on name lengths allows great latitude in naming items. The user should be aware, however, that long names expend a user's workspace and should be avoided if space is a consideration. Names of four characters or less require no space outside the user's symbol table.

6. A password is assigned to a workspace to prevent other users from accessing that workspace. The password must be used in order to access the workspace. Passwords are described in Chapter 8 (also see item 8, below). Passwords need not be names.

7. An account is the identifier of a recognized user's account. The account must be specified when logging on to UTS and when accessing a workspace in another user's account. The use of accounts is described in Chapter 8 (also see item 8, next). Accounts may be, but are not restricted to, names or numbers.

8. In Xerox APL a saved workspace is treated as a logical file. A file identifier (FID) refers to the information needed in a system command to save a workspace or to reference it after it has been saved. A file identifier can take either of the following forms:

[acctnum] workspace [:password]

workspace [.acct | ..password | .acct.password]

where

acctnum is an integer specifying the number of a recognized user's account. It can consist of up to eight digits and must be followed by a blank.

acct is the identifier of a recognized user's account. It can consist of up to eight characters.

workspace is the name assigned to the workspace, or file. It can consist of up to 11 characters.

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

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

Account (acct) and Passwords may include any characters except the period, comma, semicolon, or embedded blanks. It is advisable, however, to avoid use of special APL characters.

Variables

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

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

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

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

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

Another way of respecifying a variable value is to decrease or increase its value by a certain amount. For example, suppose variable A has a value of 5 and the user wants to increase this value by 1. This can be accomplished as follows:

```
      A←A+1
      A
6
```

Notice that the calculation 5+1 is performed first, and then the result 6 is assigned to a variable A. This type of operation is particularly useful for setting up a counter to test the number of occurrences of an event, such as the number of passes through a program loop. Each time through the loop the counter can be increased or decreased by 1 and then tested against a desired value to determine further action.

Local and Global Variables

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

Dummies
Result
Locals
Labels

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

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

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

Local Variables

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

```
∇R←Y F X;A;B;C
```

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

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

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

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

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

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

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

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

```
L←4
```

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

Shadowing

The example in Figure 3 illustrates the effect of shadowing as functions F1 and F2 become active and inactive.

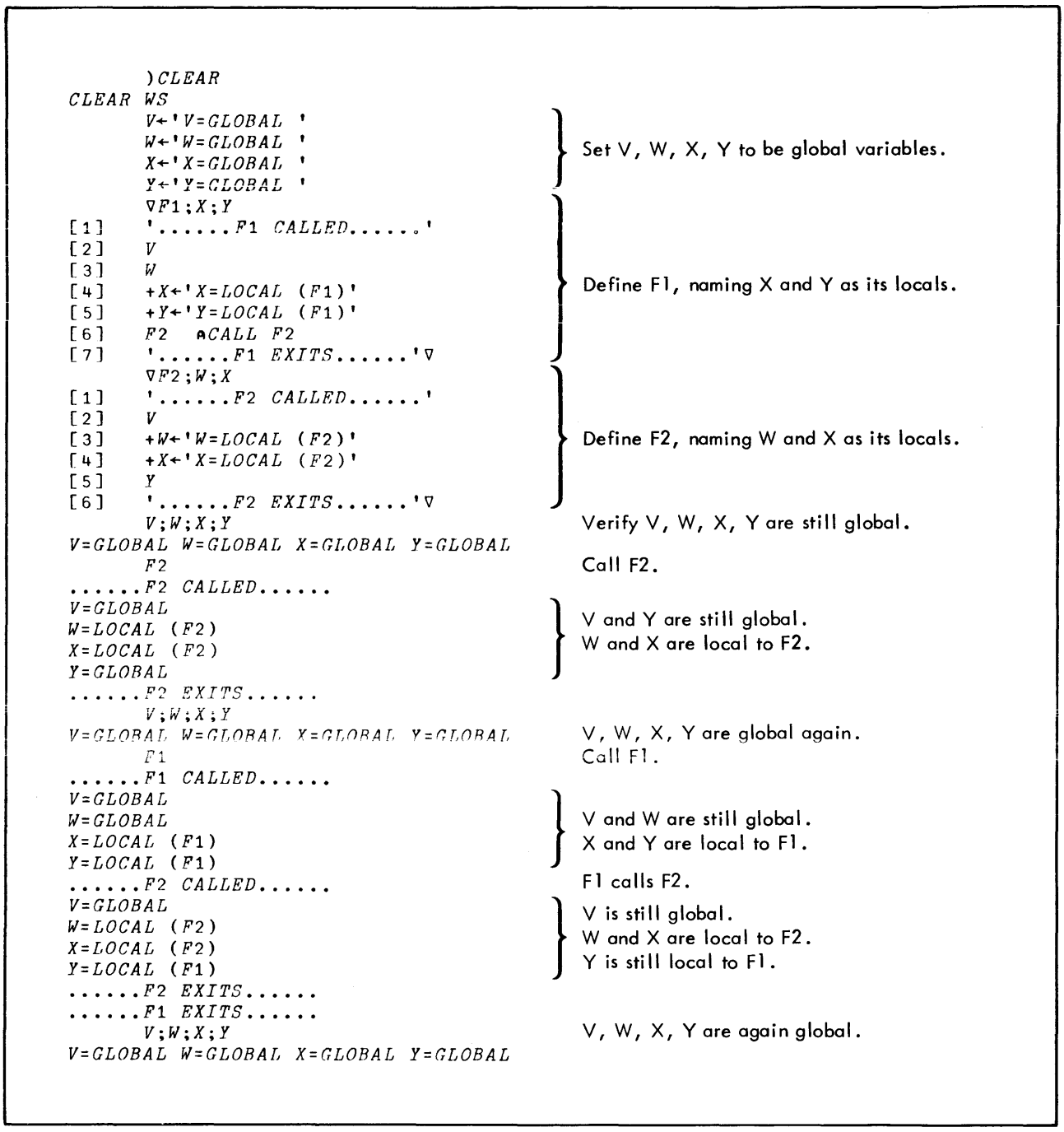


Figure 3. Example Showing Effect of Shadowing

Arrays and Indexing

Description of Arrays

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

```
      SCLR←33
      SCLR
33
```

Although an array is usually made up of more than one element, it can also consist of a single element or even no elements. An array with no elements is called an empty array.

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

```
      VECT
5  7  9  11
```

A matrix is an array with two dimensions[†], and is displayed as a collection of elements arranged in a rectangular pattern. An example of a two-dimensional matrix, named MAT, is shown below:

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

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

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

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

16  17  18  19  20
21  22  23  24  25
26  27  28  29  30
```

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

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

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

A 0 indicates a scalar, a 1 indicates a vector, a 2 indicates a two-dimensional array, a 3 indicates a three-dimensional array, and so on, up to a maximum of 63 dimensions.

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

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

[†]Note that a dimension is sometimes called a coordinate.

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

Indexing of Arrays

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

The indices of all arrays start with 0 or 1, depending on the index origin. When the user first logs on to APL, the index origin is 1 by default. It can be set to 0 with the system command)ORIGIN 0, and reset to 1 with command)ORIGIN 1. An example of the effect of origin setting upon indexing follows.

```
      V←'ABCDE'
      )ORIGIN 1
WAS 1 V[2]
B
      )ORIGIN 0
WAS 1 V[2]
C
      V[1]
B
```

The indices of a two-dimensional matrix also start with 0 or 1, depending on the origin, but two numbers are used in each index. The first number selects the element from a row, and the second number selects the element from a column. The indices are ordered with the rightmost position varying the fastest, then the next rightmost, and so on. For purposes of illustration, consider the matrix named MAT3:

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

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

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

Thus MAT3[1;1] is 3; MAT3[1;2] is 1; MAT3[1;3] is 11; Mat3[1;4] is 2; and so on. Notice that semicolons must be used to separate the numbers of each dimension.

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

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

11  12   6  16
 5   3   9  10
```

To reference the value 8 in this array, one uses the index MAT4[1;2;4]; where 1 denotes the first plane, 2 denotes the second row, and 4 denotes the fourth column. Notice that each additional coordinate always adds a number to the beginning of an index. The rightmost number of an index always refers to a column; the next rightmost to a row; the next rightmost to a plane; the next to a panel of planes; and so on.

Referencing More Than One Element. To reference more than one element of a vector, one simply includes the index of each desired element in brackets after the array name. For example, notice the following vector:

```
      A←5 4 ¯1 3 9 ¯2 7 4
```

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

```
      A[1 3 4]
5 ¯1 3
```

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

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

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

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

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

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

```
      MAT5[1;4 5 2]
8 11 10
      MAT5[1 2;]
 1  10   9   8  11
 2  15   4   5   6
      MAT5[1 2;1 2 3 4 5]
 1  10   9   8  11
 2  15   4   5   6
      MAT5[1 2 3;4]
8 5 13
      MAT5[1 2;4 5]
 8  11
 5   6
      MAT5[;2 4]
10  8
15  5
 3 13
      MAT5[1 2 3;2 4]
10  8
15  5
 3 13
```

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

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

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

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

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

 1   2   4
 6   7   9
 1   2   4

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

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

Example of vector:

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

      WHOOPS←V[ ]←2
      V
2  2  2  2  2  2  2  2  2  2
      WHOOPS
2
```

Example of matrix:

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

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

The value that is assigned to a variable or indexed variable is also the "result" of the assignment. This is illustrated by the example WHOOPS←V[]←2. Since V was a 10-element vector, all 10 index values received the element value 2. But the result, as far as the assignment operation is concerned, is still just the scalar 2. Thus, WHOOPS

becomes a scalar variable having the value 2. In analyzing APL expressions, it is helpful to imagine that assigments are "invisible". For example,

```
3+M[;4]←5
```

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

```
3+      5
```

making the result (8) apparent.

Indexing an Indexed Argument. In Xerox APL an indexed argument may itself be indexed. For example:

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

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

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

```
A[1;][2]←X
```

The user instead is advised in this case to use

```
A[1;2]←X
```

Operators and Arguments

APL expressions are derived from two fundamental entities – functions and arguments. Functions are formed by the user (see Chapter 7, "Defined Functions") or are included as an inherent part of the language. In the latter case they are called operators (some programmers refer to them as primitive functions, but that term is not used in this manual in order to avoid confusion with defined functions). Most operators are represented by a single character. A general treatment of these operators is given in this section; for a detailed treatment, see Chapter 5, "APL Operators".

An argument is a value – the value of an expression. Arguments have certain attributes: domain, rank, and length or shape. The domain of an argument may be text type or numeric type. There are three numeric type domains: logical, integer, or real; however, the user seldom needs to be concerned with this distinction. Logical data represents 1's or 0's and is stored in bit form. Integer data represents positive and negative numbers (using neither decimal point nor exponential form) whose ranges are limited to the size of one computer word. Real data is stored in doubleword form (that is, in floating-point form). Text or character data is stored in byte form. An argument cannot be of mixed domain; it is either all text, all logical, all integer, or all real. If a numeric argument contains numbers that could fit in more than one domain, it is made to uniformly contain numbers in the largest size domain necessary. Thus the following vector argument has integer domain since that is necessary to represent the 2:

```
1 0 1 0 2
```

The rank of an argument is the number of its dimensions (or coordinates). A scalar has a rank of zero, a vector has a rank of one, a matrix has a rank of two, and so forth. The maximum allowed rank is 63.

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

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

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

Certain operators are coordinate-dependent. For example, a matrix rotation can occur about the first coordinate (rotation of rows) or about the second coordinate (rotation of columns). For such operations, the user has the option of specifying this coordinate in the form of a bracketed expression to the right of the operator. The value of this expression must be an integer of appropriate range. These coordinate specifications are considered to be part of the operator – they are not arguments. The following operations may use a coordinate specification:

Reduction	Compression
Reversal	Expansion
Rotation	Catenation

Note: Catenation may also use a real-valued coordinate specification. This form of catenation is called lamination.

Table 1 is a summary treatment of the standard APL operators, indicating their dyadic and monadic operations, if any, and giving simple examples. Notice that the operators in this table are divided into three categories: standard scalar dyadic operators, other standard operators (that is, nondyadic scalar and mixed operators), and composite operators. For a detailed treatment of these operators see Chapter 5, "APL Operators".

Table 1. APL Operators

Type of Operator	Operator	Monadic Operation	Dyadic Operation

Scalar Dyadic Operators	+	Identity: leaves argument unchanged. Examples:
		`+10`
		`10`
		`+'ABC'`
		`ABC`
			Addition: adds two arguments. Example:
			`10+20`
			`30`

	-	Minus: negates the argument that follows it. Example.
		`-(10+5)`
		`¯15`
			Subtraction: subtracts the right argument from the left argument. Example.
			`10-5`
			`5`

	×	Signum: returns a ¯1, 0, or 1, depending on whether its argument is negative, zero, or positive. Example:
		`×¯15`
		`¯1`
			Multiplication: multiplies the left argument by the right argument. Example:
			`10×15`
			`150`

	÷	Reciprocal: divides 1 by the value of its argument. Example:
		`÷1 3 5`
		`1 0.3333333333 0.2`
		Note that this is equivalent to the dyadic use of 1÷1 3 5.
			Division: divides the left argument by the right argument. Example:
			`10÷5 2 1 0.5`
			`2 5 10 20`

	*	Exponential: raises e (i.e., the base of a natural logarithm, having the value 2.71828...) to the power of its argument. Examples:
		`*1`
		`2.718281828`
		`*10`
		`22026.46579`
		`*2.2`
		`9.025013499`
			Exponentiation: raises the left argument to the power indicated by the right argument. Examples:
			`10*2`
			`100`
			`10*10`
			`1E10`
			`2*3`
			`8`

Table 1. APL Operators (cont.)

Type of Operator	Operator	Monadic Operation	Dyadic Operation

Scalar Dyadic Operators (cont.)	⍟	Natural logarithm: computes the natural logarithm of its argument (that is, $\log_e$ of argument). Examples:
		`      ⍟1`
		`0`
		`      ⍟2`
		`0.6931471806`
		`      ⍟3`
		`1.098612289`
		`      ⍟10`
		`2.302585093`
			Logarithm: computes the logarithm of the right argument to the base indicated by the left argument; that is, computes the power to which the left argument must be raised to equal the right argument. Examples:
			`      10⍟100`
			`2`
			`      10⍟1 10 100 1000`
			`0 1 2 3`
			`      2⍟4`
			`2`
			`      2⍟1 2 4 8`
			`0 1 2 3`

	⌊	Floor: returns the greatest integer less than or equal to its argument. Examples:
		`      ⌊10.7`
		`10`
		`      ⌊2 4.1 ¯8.9 ¯2`
		`2 4 ¯9 ¯2`
			Minimum: compares two arguments and returns the value of the smallest argument. Examples:
			`      5⌊2`
			`2`
			`      9⌊3 11 8 ¯2 10`
			`3 9 8 ¯2 9`
			`      5 4 3 2⌊3`
			`3 3 3 2`

	⌈	Ceiling: returns the least integer greater than or equal to its argument. Examples:
		`      ⌈10.7`
		`11`
		`      ⌈2 4.1 ¯8.9 ¯2`
		`2 5 ¯8 ¯2`
			Maximum: compares two arguments and returns the value of the largest argument. Examples:
			`      5⌈2`
			`5`
			`      9⌈3 11 8 ¯2 10`
			`9 11 9 9 10`
			`      5 4 3 2⌈3`
			`5 4 3 3`

	|	Absolute value: returns the absolute value of its argument. Example:
		`      |¯10`
		`10`
			Residue: returns the remainder from dividing the right argument by the left argument. Examples:
			`      2|4`
			`0`
			`      5|15 16 17 18`
			`0 1 2 3`
			`      2 3|7`
			`1 1`

	!	Generalized factorial: for integer arguments, returns the factorial of its argument. The argument may not be a negative integer. (See Chapter 5 for explanation of ! with noninteger arguments.) Examples:
		`      !3`
		`6`
		`      !0 1 2`
		`1 1 2`
			Generalized combination: for positive integer arguments, the right argument represents a population size and the left argument represents a sample size. The result is the number of different samples that can be drawn from the population (see Chapter 5 for explanation of ! with noninteger arguments.) Examples:
			`      13!52`
			`6.350135596E11`
			`      2!10`
			`45`
			`      3!10`
			`120`

Table 1. APL Operators (cont.)

Type of Operator	Operator	Monadic Operation	Dyadic Operation

Scalar Dyadic Operators (cont.)	○	Pi times: multiplies the value of pi (3.14159265353589793) times its argument. Examples:
		`      ○1`
		`3.141592654`
		`      ○2 .1`
		`6.283185307  0.3141592654`
			Circular: returns the result of any of a number of trigonometric functions. The left argument specifies the type of trigonometric function, and must be one of the integers from -7 to 7, as follows:
			-7 arctanh X
			-6 arccosh X
			-5 arcsinh X
			-4 (-1+X*2)*.5
			-3 arctan X
			-2 arccos X
			-1 arcsin X
			0 (1-X*2)*.5
			1 sine X
			2 cosine X
			3 tangent X
			4 (1+X*2)*.5
			5 sinh X
			6 cosh X
			7 tanh X
			where X is the right argument, and may be a scalar, a vector, or an array. For the sine, cosine, and tangent functions, X must be expressed in radians. In addition, the permitted value range of X is limited for various circle functions (for instance, with an arcsine, X must be in the range -1 to 1). Examples:
			`      2○(10×2.5)`
			`0.9912028119`
			`      1○2 4`
			`0.9092974268  ¯0.7568024953`

	<		Less than:[†] tests if the left argument is less than the right argument. Returns a 1 if the test is true, and a 0 if the test is false. Examples:
			`      2<3`
			`1`
			`      3<4 1 2 5`
			`1 0 0 1`

	≤		Less than or equal to:[†] tests if the left argument is less than or equal to the right argument. Returns a 1 if the test is true, and a 0 if the test is false. Examples:
			`      2≤3`
			`1`
			`      2≤1 2 3 4`
			`0 1 1 1`

[†]See Chapter 5 for effect of "fuzz" on relational operators.

Table 1. APL Operators (cont.)

Type of Operator	Operator	Monadic Operation	Dyadic Operation

Scalar Dyadic Operators (cont.)	>		Greater than:† tests if the left argument is greater than the right argument. Returns a 1 if the test is true, and a 0 if the test is false. Examples:
			`2>3`
			`0`
			`2>¯2 0 1 2 3`
			`1 1 1 0 0`

	≥		Greater than or equal to:† tests if the left argument is greater than or equal to the right argument. Returns a 1 if the test is true, and a 0 if the test is false. Examples:
			`2≥3`
			`0`
			`2≥¯2 0 2 3 4`
			`1 1 1 0 0`

	=		Equal to:† tests if the left argument is equal to the right argument. Returns a 1 if the test is true, and a 0 if the test is false. Examples:
			`1=0`
			`0`
			`2=0 1 2 3`
			`0 0 1 0`
			`'A'='CANADA'`
			`0 1 0 1 0 1`

	≠		Not equal to:† tests if the left and right arguments are unequal. Returns a 1 if they are unequal, and a 0 if they are equal. Examples:
			`2≠1`
			`1`
			`3≠¯3 0 3 6`
			`1 1 0 1`
			`'A'≠'CANADA'`
			`1 0 1 0 1 0`

	∧		And: tests if the left argument and the right argument each have a value of 1. (The arguments must be 0 or 1.) Returns a 1 if both arguments are 1, and a 0 for any other combination of arguments, as shown below:
			∧ | 0 1
			0 | 0 0
			1 | 0 1
			Examples:
			`0∧0`
			`0`
			`(1=2)∧(3<4)`
			`0`
			`(1<2)∧(3<1 3)`
			`0 0`
			`(1=1)∧(3<4)`
			`1`

†See Chapter 5 for effect of "fuzz" on relational operators.

Table 1. APL Operators (cont.)

<table>
<tr><th>Type of Operator</th><th>Operator</th><th>Monadic Operation</th><th>Dyadic Operation</th></tr>
<tr><td rowspan="3">Scalar Dyadic Operators (cont.)</td><td>∨</td><td></td><td>Or: tests if either the left argument or the right argument has a value of 1. Returns a 1 if either or both arguments are 1, and a 0 if neither argument is 1, as shown below:

∨ | 0 1
0 | 0 1
1 | 1 1

Examples:

0∨1
1
(1=2)∨(4<3)
0
(3<4)∨(4<5)
1</td></tr>
<tr><td>⍲</td><td></td><td>Nand: tests if both arguments are 1. Returns a 0 if both arguments are 1, and returns a 1 for all other combinations, as shown below:

⍲ | 0 1
0 | 1 1
1 | 1 0

Examples:

0⍲0
1
(2<1)⍲(5<1)
1
(1<2)⍲(1<5)
0</td></tr>
<tr><td>⍱</td><td></td><td>Nor: tests if both arguments are 0. Returns a 1 if both arguments are 0, and returns a 0 for all other combinations, as shown below:

⍱ | 0 1
0 | 1 0
1 | 0 0

Examples:

0⍱0
1
0⍱1
0
(1=2)⍱(2<1)
1
(1=1)⍱(2<3)
0</td></tr>
</table>

Table 1. APL Operators (cont.)

Type of Operator	Operator	Monadic Operation	Dyadic Operation
Nondyadic Scalar and Mixed Operators	~	Not: returns the value 0 if the following argument is 1, and returns a 1 if the argument is 0. Examples: ~0 1 ~1 0 ~(6>4) 0 ~(6<4) 1 ~1 0 1 0 0 1 0 1	
	?	Roll (also known as monadic random): returns an integer pseudorandomly selected† from 1 through the integer specified in the right argument. Examples: ?5 1 ?3 3 3 3 2 2 ?5 8 11 13 2 1 8 9	Deal (also known as dyadic random): returns the number of integers specified in the left argument, each pseudorandomly selected† from the integers specified in the right argument, and with no repetition of numbers in the result. Examples: 4?8 1 4 2 6 4?4 3 2 4 1
		Note that this operator is modified by index origin. If the origin is 0, the pseudo-random selection is from zero to the argument minus 1. If the origin is 1, selection is from 1 to the argument.	
	⍳	Index generator: generates a vector whose length is the value of the argument. If the origin is 1, the vector will contain the positive integers 1 through the value of the argument. If the the origin is 0, the vector will contain the positive integers 0 through the value of the argument minus 1. Examples: ⍳5 1 2 3 4 5)ORIGIN 0 WAS 1 ⍳5 0 1 2 3 4	Index of: returns the position of the right argument in the left argument. If the right argument is not found in the left argument, it is given a value of the length of the left argument plus 1. Examples: 6 4 3 2⍳6 1 6 4 3 2⍳6 2 5 4 3 1 4 5 2 3 6 4 3 2⍳1 5)ORIGIN 0 WAS 1 6 4 3 2⍳6 2 5 4 3 0 3 4 1 2

†Pseudorandomly selected means that numbers are chosen by means of an algorithm which eventually produces repeating results. In other words, a random sequence is, at best, only approximated.

Table 1. APL Operators (cont.)

Type of Operator	Operator	Monadic Operation	Dyadic Operation

Nondyadic Scalar and Mixed Operators (cont.)	,	Ravel: generates a vector from either a scalar or an array of higher dimension. Examples:
		`A`
		`1 2 3 4`
		`5 6 7 8`
		`,A`
		`1 2 3 4 5 6 7 8`
		`B`
		`ABCD`
		`EFGH`
		`,B`
		`ABCDEFGH`
			Catenation: chains together scalars or arrays of conforming dimensions. Examples:
			`A←1 2 3`
			`B←4 5 6 7`
			`A,B`
			`1 2 3 4 5 6 7`
			`C←3`
			`(C÷1),C×3-2`
			`3 3`
			Note: See Chapter 5 for further examples (including "lamination").

	⍴	Shape: returns an empty vector if the argument is a scalar, the length (or number or elements) if the argument is a vector, and a vector containing the length of each dimension if the argument is a higher-order array. Examples:
		`A←2`
		`B←1 5 6 7`
		`C←3 3⍴⍳9`
		`⍴A`
		`⍴B`
		`4`
		`⍴C`
		`3 3`
			Reshape (restructure): generates an array whose dimensions are the left arguments and whose elements are taken from the right argument. Examples:
			`5⍴1`
			`1 1 1 1 1`
			`2 4⍴8`
			`8 8 8 8`
			`8 8 8 8`
			`2 4⍴⍳8`
			`1 2 3 4`
			`5 6 7 8`

	⍋	Grade up: ranks the components of its argument in ascending order, and returns the positions (i.e., indexes of the components). If the origin is 0, the ranking will start with 0. Similarly, if the origin is 1, the ranking will start with 1. The argument must be a vector. Examples:
		`A←1 4 1 2 3 1 5`
		`⍋A`
		`1 3 6 4 5 2 7`
		`)ORIGIN 0`
		`WAS 1`
		`⍋A`
		`0 2 5 3 4 1 6`	

	⍒	Grade down: similar to grade up, except that it returns the indexes in descending order. Example:
		`A←1 4 1 2 3 1 5`
		`⍒A`
		`7 2 5 4 1 3 6`
		`)ORIGIN 0`
		`WAS 1`
		`⍒A`
		`6 1 4 3 0 2 5`	

Table 1. APL Operators (cont.)

Type of Operator	Operator	Monadic Operation	Dyadic Operation

Nondyadic Scalar and Mixed Operators (cont.)	⊥		Base value: allows the user to switch from one number system to another. The right argument contains the numbers to be converted, and the left argument contains the increments needed to convert from one unit to another. The left argument, usually called the radix vector, can be thought of as the base of the number system. Examples:
			`10 10 10⊥5 6 5`
			`565`
			`0 60⊥10 20`
			`620`
			`2 2 2 2⊥1 0 0 1`
			`9`
			`2⊥1 0 0 1`
			`9`

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

	↑		Take: selects the number of components indicated by the left argument, from the right argument. If the left argument is positive, selects the components from the beginning of the right argument. If the left argument is negative, ↑ selects the components from the end of the right argument. Examples:
			`A←2 4 6 8 10`
			`4↑A`
			`2 4 6 8`
			`¯4↑A`
			`4 6 8 10`

	↓		Drop: similar to take except that the indicated elements are dropped instead of selected. Examples:
			`A←2 4 6 8 10`
			`2↓A`
			`6 8 10`
			`¯2↓A`
			`2 4 6`

Table 1. APL Operators (cont.)

Type of Operator	Operator	Monadic Operation	Dyadic Operation

Nondyadic Scalar and Mixed Operators (cont.)	∈	Execute: treats its argument (a text string) as if in response to an evaluated input request. Example:
		`∈'2+3'`
		`5`
		`∈')CLEAR'`
		`CLEAR WS`
		`∈'∇F X∇'`
		`∈')FNS'`
		`F`
		`∈'∇F[1]X+X∇'`
		`F ι4`
		`2 4 6 8`
			Membership: yields a 1 if a given element to the right is an element of a specified vector to the left of the membership symbol, and a 0 if it is not. The result will have the same dimensions as the left argument. Examples:
			`A←1 2 3 4 5 6`
			`B←2 4 6 8`
			`B∈A`
			`1 1 1 0`
			`C←'ABCDEFGHIJK'`
			`D←3 3ρ'HOWAREYOU'`
			`D∈C`
			`1 0 0`
			`1 0 1`
			`0 0 0`

	⌹	Inverse: used to invert matrixes. Example:
		`A`
		`4 2 ¯5`
		`5 ¯4 4`
		`2 2 ¯20`
		`)DIGITS 2`
		`WAS 10`
		`⌹A`
		`0.17 0.072 ¯0.029`
		`0.26 ¯0.17 ¯0.099`
		`0.043 ¯0.0097 ¯0.063`
			Matrix Divide: used for solving systems of linear equations. For example, suppose the user wants to find the values of X, Y, and Z in the following linear equations (using conventional algebraic notation):
			4X + 2Y - 5Z = 22
			5X - 4Y + 4Z = -7
			2X + 2Y - 20Z = 80
			He can set up the coefficients as matrix A and the constants as vector B:
			`A`
			`4 2 ¯5`
			`5 ¯4 4`
			`2 2 ¯20`
			`B`
			`22 ¯7 80`
			and then obtain the solution by taking B matrix divide A:
			`B⌹A`
			`1 ¯1 ¯4`
			Thus in the linear equations given above, X has a value of 1, Y has a value of ¯1, and Z has a value of ¯4.

Table 1. APL Operators (cont.)

Type of Operator	Operator	Monadic Operation	Dyadic Operation

Nondyadic Scalar and Mixed Operators (cont.)	⍉	Monadic transpose: performs row column transposition on its matrix argument. Example:
		`      A`
		`1  2  3  4  5`
		`6  7  8  9 10`
		`11 12 13 14 15`
		`      ⍉A`
		`1  6 11`
		`2  7 12`
		`3  8 13`
		`4  9 14`
		`5 10 15`
			Dyadic transpose: returns an array similar to the right argument except that the coordinates (dimensions) are changed according to the left argument (that is, the left argument specified the new positions of the original coordinates). Example:
			`      B`
			`1  2  3`
			`4  5  6`
			`7  8  9`
			`10 11 12`
			`13 14 15`
			`16 17 18`
			`19 20 21`
			`22 23 24`
			`      2 1 3⍉B`
			`1  2  3`
			`13 14 15`
			`4  5  6`
			`16 17 18`
			`7  8  9`
			`19 20 21`
			`10 11 12`
			`22 23 24`

	⌽	Reversal: reverses the order of the components of a vector, or the components of each row of a matrix. Examples:
		`      A←1 2 4 6`
		`      ⌽A`
		`6 4 2 1`
		`      ⌽⍳5`
		`5 4 3 2 1`
		`      ⎕←MAT←3 4⍴⍳12`
		`1  2  3  4`
		`5  6  7  8`
		`9 10 11 12`
		`      ⌽MAT`
		`4  3  2  1`
		`8  7  6  5`
		`12 11 10  9`
			Rotation: rotates the elements in the right argument as specified by the left argument (i.e., according to the number of places specified in the left argument). Examples:
			`      A←1 2 4 6`
			`      1⌽A`
			`2 4 6 1`
			`      2⌽A`
			`4 6 1 2`
			`      ⎕←MAT←3 4⍴⍳12`
			`1  2  3  4`
			`5  6  7  8`
			`9 10 11 12`
			`      1⌽MAT`
			`2  3  4  1`
			`6  7  8  5`
			`10 11 12  9`

Table 1. APL Operators (cont.)

Type of Operator	Operator	Monadic Operation	Dyadic Operation

Nondyadic Scalar and Mixed Operators (cont.)	⊖	Reversal along the first coordinate: same as above, except along the first coordinate instead of the last. This is equivalent to ⌽[1]. Example:
		`⎕←MAT←3 4⍴⍳12`
		`1 2 3 4`
		`5 6 7 8`
		`9 10 11 12`
		`⊖MAT`
		`9 10 11 12`
		`5 6 7 8`
		`1 2 3 4`
			Rotation along the first coordinate: same as above, except along the first coordinate instead of the last. This is equivalent to ⌽[1]. Examples:
			`⎕←MAT←3 4⍴⍳12`
			`1 2 3 4`
			`5 6 7 8`
			`9 10 11 12`
			`1⊖MAT`
			`5 6 7 8`
			`9 10 11 12`
			`1 2 3 4`
			`2⊖MAT`
			`9 10 11 12`
			`1 2 3 4`
			`5 6 7 8`

	/		Compression: suppresses some elements of a vector and retains others. Elements of the right argument corresponding to a 1 in the left argument are retained, while those corresponding to a 0 are dropped. If either argument contains just one element, it applies to all elements of the other argument. Examples:
			`A←5 7 9 11`
			`B←'ABCD'`
			`1 0 1 1/A`
			`5 9 11`
			`1 0 1 1/B`
			`ACD`
			`⎕←MAT←3 4⍴⍳12`
			`1 2 3 4`
			`5 6 7 8`
			`9 10 11 12`
			`1 0 1 0/MAT`
			`1 3`
			`5 7`
			`9 11`
			Compression can also be used as a logical test and branch situation (like the IF statement in FORTRAN). In this case, if the argument on the left returns a value of 1, APL branches to the statement indicated by the argument on the right. If the left argument returns a 0, program flow falls through to the next statement. Examples:
			`→(2>3)/END` falls through to the next statement.
			`→(2<3)/END` causes a branch to statement labeled END.

Table 1. APL Operators (cont.)

Type of Operator	Operator	Monadic Operation	Dyadic Operation

Nondyadic Scalar and Mixed Operators (cont.)	⌿		Compression along the first coordinate: same as above except that compression is along the first coordinate instead of the last. Equivalent to /[1]. Example:
			`      ⎕←MAT←3 4⍴⍳12`
			`1  2  3  4`
			`5  6  7  8`
			`9 10 11 12`
			`      0 1 0⌿MAT`
			`5 6 7 8`

	\		Expansion: inserts additional elements into an array. For each 0 in the left argument, a blank (for literals) or a zero (for numbers) is inserted in the result, which otherwise is the same as the right argument. Examples:
			`      A←1 2 3 4`
			`      B←'ABCD'`
			`      1 0 1 0 1 0 1\A`
			`1 0 2 0 3 0 4`
			`      1 0 1 0 1 0 1\B`
			`A B C D`
			`      ⎕←M←3 4⍴'ABCDEFGHIJKL'`
			`ABCD`
			`EFGH`
			`IJKL`
			`      1 0 1 0 1 0 1\M`
			`A B C D`
			`E F G H`
			`I J K L`

	⍀		Expansion along the first coordinate: same as above, except expansion occurs along the first coordinate rather than the last. Examples:
			`      ⎕←M←3 4⍴'ABCDEFGHIJKL'`
			`ABCD`
			`EFGH`
			`IJKL`
			`      1 0 1 0 1⍀M`
			`ABCD`
			`EFGH`
			`IJKL`

Table 1. APL Operators (cont.)

Type of Operator	Operator	Monadic Operation	Dyadic Operation

Composite Operators[t]	f/	Reduction: inserts the symbol (APL operator) specified to the left of the / between each element of the right argument, performs the operation from right to left, and returns a single value. Examples:
		`+/1 2 3 4 5`
		`15`
		`-/1 2 3 4 5`
		`3`
		`⎕←N←3 4⍴⍳12`
		`1 2 3 4`
		`5 6 7 8`
		`9 10 11 12`
		`+/N`
		`10 26 42`
		`-/N`
		`¯2 ¯2 ¯2`	

	f⌿	Reduction along the first coordinate: same as above except reduction occurs along the first coordinate rather than the last (equivalent to f/[1]). Examples:
		`⎕←N←3 4⍴⍳12`
		`1 2 3 4`
		`5 6 7 8`
		`9 10 11 12`
		`+⌿N`
		`15 18 21 24`
		`-⌿N`
		`5 6 7 8`	

	f.g		Generalized inner product: This operator is a generalized form of the inner product of matrix multiplication. The particular form that corresponds to traditional matrix multiplication is A+.×B, where the second dimension of matrix A is the same as the first dimension of B. The result has the same first dimension as A and the same second dimension as B. In the conventional matrix inner product, each element of the result is the sum of products of elements from A and B (see Chapter 5 for detailed description). The APL generalized inner product allows different forms such as the sum of equality

[t]The letters f and g stand for any dyadic scalar operator: + - × ÷ * ⍟ ⌈ ⌊ | ! ○ < ≤ ≥ > = ≠ ∧ ∨ ⍲ ⍱

Table 1. APL Operators (cont.)

Type of Operator	Operator	Monadic Operation	Dyadic Operation
Composite Operators (cont.)			tests, the maximum of sums, etc. Examples: A←2 3⍴⍳6 B←3 2⍴-⍳6 A 1 2 3 4 5 6 B ¯1 ¯2 ¯3 ¯4 ¯5 ¯6 A+.×B ¯22 ¯28 ¯49 ¯64 A+.=B 0 0 0 0 A⌈.+B 0 ¯1 3 2 The general form is Af.gB, where f and g represent any dyadic scalar operators. A and B may be vectors, matrices, or higher-order arrays, subject to conformability rules described in Chapter 5.
	∘.f		Generalized outer product: This operator is a generalization of matrix outer product, A∘.×B. The conventional form multiplies each element of A by each element of B. The shape of the result is the catenation of the shapes of A and B. In the generalized form, multiplication may be replaced by any dyadic scalar operation. Examples: A←¯1+⍳6 A∘.+A 0 1 2 3 4 5 1 2 3 4 5 6 2 3 4 5 6 7 3 4 5 6 7 8 4 5 6 7 8 9 5 6 7 8 9 10 A∘.×A 0 0 0 0 0 0 0 1 2 3 4 5 0 2 4 6 8 10 0 3 6 9 12 15 0 4 8 12 16 20 0 5 10 15 20 25 A∘.<A 0 1 1 1 1 1 0 0 1 1 1 1 0 0 0 1 1 1 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0 0 0 0

Table 1. APL Operators (cont.)

Type of Operator	Operator	Monadic Operation	Dyadic Operation

Composite Operators (cont.)	f\	Scan: returns value of same shape as argument. For vectors, i'th result element is formed by taking the first i argument elements, placing f between them, and evaluating right to left. For instance:
		`      +\1 3 5 7 9`
		`1 4 9 16 25`
		`      -\3 1 1 5`
		`3 2 3 ¯2`
		A coordinate specification [K] may be used; if omitted, the last coordinate will be assumed.
		`      +\[1] 2 3 ⍴ ⍳6`
		`1 2 3`
		`5 7 9`	

	f⍀	Scan along the first coordinate: same as f\[1]. Thus, as above,
		`      +⍀2 3 ⍴ ⍳6`
		`1 2 3`
		`5 7 9`	

Function References

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

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

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

```
F  2+⍳3;'POUNDS'
```

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

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

```
2+ (⍳3) D 4
```

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

```
2+result
```

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

```
(⍳3) D 4
```

In addition, any of the classes of defined functions may specify an implicit or explicit result. Thus there are actually six types of defined functions:

	With explicit result	With no result
monadic function	x	x
dyadic function	x	x
niladic function	x	x

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

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

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

Assignment

Simple Assignment

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

`A←5÷2×4`	Assigns the value of the expression 5÷2x4 to variable A.
`B←1 2 3 4 5`	Indicates that B is to be a vector with the values 1, 2, 3, 4, and 5.
`B←⍳5`	Another way of assigning the numbers 1 through 5 to variable B. (The ⍳5 assumes an index origin of 1.)

```
C←2 4⍴⍳8
```

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

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

```
D←2 3⍴5 6 1 2 8 9
```

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

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

```
E←D
```

Indicates that the value of D is assigned to E.

Multiple Assignment

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

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

Indexed Assignment

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

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

Input/Output

Input/Output Devices

The Xerox APL system allows the user a choice of six input/output methods:

- Standard APL terminal input/output: a standard terminal with an APL typeball.[t]
- APL/ASCII terminal input/output: a terminal with either of two APL/ASCII character mappings.
- Nonstandard terminal input/output: a standard terminal with non-APL typeball, or a device equivalent to a Teletype Model 33 to 37.
- Batch input/output.
- File input/output.
- 'Blind' input/output.

The input/output described in this chapter refers to terminals with the APL character set. Nonstandard terminal, batch, blind, and file input/output are discussed in Appendix B.

General Input/Output

After logging on to the APL system, the user is in execution mode and can enter input whenever the keyboard is unlocked. The fundamental item of input to APL is the line. A line is a collection of characters that does not include the carriage return. Striking the RETURN key completes a line, and APL attempts to interpret it and perhaps output data. An incomplete line can be corrected as described in Chapter 2. User input and computer output are easily distinguished at the terminal; computer output usually begins at the left margin while user input is usually indented six spaces from the left margin. An input line is limited to 130 characters, not counting the carriage return (overstrikes count as single characters).

Types of Input

The Xerox APL system acknowledges four kinds of input: direct, evaluated, quote quad, and blind. Direct input results when APL is not executing the user's program, evaluated input results from quad execution, quote-quad input results from quote-quad execution, and blind input results from quad-1 or quad-2 execution. Direct input, evaluated input, and quote-quad input are described below and are considered to exist only after input translation and current-line editing. Blind input is covered in Appendix B. (See also Chapter 11 for discussion of graphics 'quad-zero' input.)

Direct Input

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

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

[t]As noted in Chapter 2, the APL typeball can be used with any terminal model that is operationally equivalent to an IBM 2741.

Evaluated Input

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

```
      8÷□
□:
      2
4


      5×□
□:
      1 2 3 4
5  10  15  20
```

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

```
      ∇CUBE;A
[1]   LOOP: A←□
[2]   A←A×A×A
[3]   A
[4]   →LOOP
[5]   ∇


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

Entering any of the following system commands will terminate an input request:)CLEAR,)LOAD,)OFF,)OFF HOLD,)CONTINUE, or)CONTINUE HOLD. Entering other system commands merely causes the □: to reappear after the command is executed. Entering a)SAVE,)CONTINUE, or)CONTINUE HOLD command causes the workspace to be saved in the □ state; and causes a □: to be printed when that workspace is eventually loaded. This can be useful in attaching a message to a workspace. For example:

```
      'NOTE:  WORKSPACE HAS ORIGIN 0'; 0⍴□
□:
      )CONTINUE
CONTINUE    SAVED  18:07 JUN 27,'72
      .
      .
      .

)APL
APL 07/27/73
CONTINUE    SAVED  18:07 JUN 27,'72
□:
      1
NOTE:  WORKSPACE HAS ORIGIN 0
```

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

- Creating a new function.
- Revising an existing nonpendant† function: inserting a line, deleting a line, replacing a line, and editing characters of a line.
- Displaying one or more lines of the open function.

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

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

Quote-Quad Input

The quote-quad symbol ⍞ (except when to the left of an assignment arrow) denotes literal input. When APL encounters this symbol during statement execution, it skips to the beginning of the next line, unlocks the keyboard, and awaits user input (no symbol is printed to prompt for input). Literal character strings are entered without beginning and ending quote symbols, and a quote within a string is represented by one quote instead of two. Entering no characters produces an empty vector, entering a single character produces a scalar, and entering a string of characters produces a vector. To terminate a request for literal input, enter the following sequence of characters: O backspace U backspace T, which appears at the terminal as ⍬. Examples of quote-quad input are

```
      A←⍞

      ⍴A
0


      B←⍞
QUOTES AREN'T NEEDED
      B
QUOTES AREN'T NEEDED


      X←'CALIFORNIA'∈⍞
ABCDEFGHIJKLM
      X
1  1  1  1  1  0  0  0  1  1
```

Output

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

1. Width of line. Unless the user specifies otherwise, up to 120 characters can be displayed per line. The user can change the number of characters displayed to any number from 30 to 254, via the)WIDTH system command (see Chapter 9) or the WIDTH function (see Chapter 8). Output processing always assumes that the left and right margin stops are placed full left and full right.

†If a function makes an evaluated input request, the function becomes pendant. Therefore, that function cannot be opened during the evaluated input request.

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

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

3. Exponential notation. The APL system usually uses exponential form for printing numbers less than $1E^{-}5$, or greater than 1En (where n is the number of significant digits displayed) or $2^{31}-1$. Decimal form is used for other cases. Numbers printed in exponential form always have a magnitude between one and ten followed by an appropriate exponent.

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

```
1234567.89  1234567890  1.23456789E10
```

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

```
    A
 0.0100003      251.8767434       ¯1.99032
12.3456703       ¯0.09873201       7.76767676

    A*11
1.000330050E¯22  2.588622762E26   ¯1.941565195E3
1.015456727E12   ¯8.690359926E¯12  6.211587288E9
```

4. Significant digits. The APL system carries out all calculations to approximately 16 significant digits (i.e., to 16 digits to the right of a decimal point), and displays the result rounded off to 10 significant digits. Any trailing zeros are suppressed in the display. Examples are shown below:

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

The user can use the)DIGITS system command (see Chapter 8) to change the number of significant digits displayed, with the number ranging from 1 to 16 digits. Examples are shown below:

```
      )DIGITS 4
WAS 10
      4÷3
1.333
      5÷2
2.5
```

If)DIGITS 16 is selected (this is not recommended, however), decimal fractions will tend to be displayed in forms such as .5999999999999999 because of roundoff in internal computations. With)DIGITS 15, this value will print as .6. In performing some calculations (particularly tests of equality), the user should remember that the number of significant digits printed may be less than that retained by the system, and that the significant digits retained by the system may be less than the number originally entered by the user.

5. Fuzz. When comparing noninteger numbers, the question is how close the numbers must be to be considered equal. This tolerance, known as fuzz, is approximately $1.0E^{-}13$. The results of some floor and ceiling operations and comparison operations are affected by this fuzz. Examples of comparison tests illustrating fuzz are shown below:

```
      2.2222222222222222=2.22222222222222229
1
      ⌈11+10*¯12
12
      ⌈11+10*¯13
11
```

6. Numeric and character vectors. Numeric vectors are displayed with two blanks between elements, while character vectors are displayed with no blanks between elements, as shown:

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

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

```
      3 5⍴¯2+⍳15
¯1   0   1   2   3
 4   5   6   7   8
 9  10  11  12  13


      2 3 4⍴4+⍳24
 5   6   7   8
 9  10  11  12
13  14  15  16

17  18  19  20
21  22  23  24
25  26  27  28


      3 4⍴'NOWISTHETIME'
NOWI
STHE
TIME


      2 2 5⍴'ABCDEF GHIJKL MNOPQR'
ABCDE
F GHI

JKL M
NOPQR
```

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

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

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

```
      0\''

      0\⍳0
0
```

Empty vectors are useful in initializing vectors and in branching.

It should be noted that the use of an empty array as the argument of a scalar function will result in an empty array:

```
      3 4+⍴0
      0≠2 0⍴5
```

9. Mixed output. Numeric and literal expressions can be intermixed for output, as long as they are separated with semicolons. The output will be displayed on one line, unless one of the expressions is a nonempty array. Display of a nonempty array value begins on a new line. Examples of mixed output are

```
      'THE SUM OF 20+2+4 IS ';20+2+4
THE SUM OF 20+2+4 IS 26

      'SUM IS ';5+10;'; PRODUCT IS ';5×10;'.'
SUM IS 15; PRODUCT IS 50.

      'THE REPLY IS'; 1 0 0/[1] 3 5⍴'YES   NO    UNSURE'
THE REPLY IS
YES
```

In this context, the semicolons are said to separate the "compound statement" into a series of substatements. Any valid statement can be used as a substatement (even branches); however, a compound statement cannot contain system commands. Consider the following statements:

```
      'AMT = '
AMT =
      A←25×B←100
      ' DOLLARS'
 DOLLARS
```

Notice that the second statement (A←25×B←100) produced no display since it ended on an assignment to A. Statements that end (in an evaluation sense) on assignments are called "assignment statements". Nonassignment statements automatically produce display, while assignment statements do not. The same is true for substatements in Xerox APL (unlike some other versions of APL). Thus if the above statements are combined, the resulting mixed output is as shown:

```
      'AMT = ';A←25×B←100;' DOLLARS'
AMT =  DOLLARS
```

To make the second substatement into a nonassignment substatement is easy; simply insert a + (identity operator) as its first character, as shown here:

```
      'AMT = ';+A←25×B←100;' DOLLARS'
AMT = 2500 DOLLARS
```

The second substatement value (2500) is displayed because it is a nonassignment substatement now. It ends on the identity operation (+), not on an assignment.

10. Blind output. Blind output — treated in Appendix B — is output as one record of text (literal) data.

11. Stopping a display. The user can stop display of output by depressing the ATTN key. This use of the ATTN key is more fully described in Chapter 10, "Execution Stops". A small amount of data may still be output after depressing the ATTN key. This results because CP-V has already prepared to display that data before APL is notified that ATTN was issued.

12. Graphics output. See Chapter 11 for discussion of graphics "quad-zero" output.

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

```
      ⎕←A←2+3
5
```

4. EXPRESSION EVALUATION

Order of Evaluation

Right to Left

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

```
      20×4+5÷2
130
```

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

Precedence of Operators

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

Parentheses

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

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

The Value of a Variable Versus Its Name

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

```
      K←1
      (K←2)+K
3
```

The K to the right of the plus sign was evaluated to the argument having, at that time, value 1. This argument did not change even though K's value changed before the addition was completed. (Some APL systems retain the name rather than the value of an argument. These systems would obtain a result of 4 in the above example.)

Syntax Considerations

Any two variables, constants, or quads cannot appear adjacent to each other. They must be separated by a dyadic operator or a dyadic function reference. Some valid and invalid examples are shown below.

Valid	Invalid
`(3÷4)×2`	`(3÷4)2`
`2×A`	`2A`
`4+¯1`	`4¯1`
`1 2 3`	`(1 2) 3`

Notice in the last example that "1 2 3" is actually only one constant – a numeric vector.

Two dyadic operators cannot appear next to each other, but a dyadic operator and a monadic operator can. Thus the expression

```
5|*⌈X
```

is legal, since the * and ⌈ operators are used as monadic operators, and the | operator is dyadic. The following expression is illegal, however, because ∨ is a dyadic-only operator preceded by an operator:

```
A∧∨B
```

Default Terminal Output

Default terminal output occurs when a nonassignment statement is evaluated. That is, the result defaults to the terminal instead of being stored in memory. For example, 2×4 gives default output:

```
      2×4
8
```

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

```
A←2×4
```

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

When a compound statement (Chapter 6) includes both nonassignment and assignment expressions, the nonassignment expressions produce default output at the terminal while the assignment expressions do not. However, any assignment expression can be transformed into a nonassignment expression by placing a plus sign (that is, identity operator) at its extreme left. Some examples are

```
      4;4
44
      4;'  ';4
4  4
      4+2;A←5+2;4+2
66
      X←⍳5;Y←2+4
      +X←⍳5;+Y←2+4
1  2  3  4  56
```

Notice from these examples that separating spaces must be designated by spaces within quotes (see the 4;' ';4 example), or else the default output from several expressions will run together. Also notice in the following example that the monadic + may be used to force default output even if the result is not numeric:

```
      +X←'TEXT'
TEXT
```

Errors and Breaks

If the user discovers an error in a statement before the RETURN key is depressed, the user can backspace to the error, strike the INDEX key or the ATTN key, and retype the rest of the line as described in Chapter 2. An example (using the ATTN key) is:†

```
      A←5×B←8×
            v
            ÷4
      A
10
```

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

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



      E÷F;E←4÷2+1;F←0;4÷2*.5
DOMAIN ERR
      E÷F;E←4÷2+1;F←0;4÷2*0.5
        ^
      E
1.333333333
      F
0
```

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

---

†On APL/ASCII terminals, the standard CP-V input line editing devices are applicable. See CP-V Time-Sharing Reference Manual, 90 09 07, Chapter 2.

If the user has entered a line and APL detects a simple error before any part of the line is executed, APL returns the following: the message LINE-SCAN ERR, a caret at the error point, and that portion of the line it found acceptable. The user can then complete that portion in the usual manner (as if he were entering it for the first time), correcting the problem. For example:

```
      A←234+(⍳3)⎕*3
LINE-SCAN ERR    ^
      A←234+(⍳3)×⎕*3
⎕:
      4
      A
298   362   426
```

where the underlining indicates the portion of the line typed by APL.

5. APL OPERATORS

An APL operator is a symbol indicating that a basic APL operation, such as addition or division, is to be performed. A symbol denoting an APL operator is either a nonalphanumeric character or a combination of such characters. For example, addition is denoted by the + symbol and division is denoted by the ÷ symbol.

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

- Syntax Conventions

  Syntax conventions used throughout this chapter are as follows:

  R denotes the result of an operation.

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

  A denotes a left argument.

  B denotes a right argument.

  m denotes a monadic operator.

  d denotes a dyadic operator.

  Following are some examples of the use of these conventions:

  $R \leftarrow \mathrm{m}B$ $\qquad\qquad$ $R \leftarrow A\,\mathrm{d}\,B$

- Argument Characteristics

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

  Domain — In general, the type of data element such as integer data, character data, floating-point data. For some operators the domain of an argument may be especially restricted (see the example for the circular operator later in this chapter).

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

  Length — The number of elements in a coordinate of an argument.

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

- Domain Tables

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

  N denotes numeric data.

  C denotes character data.

  L denotes logical data (1 or 0).

I denotes integer data.

F denotes floating-point data.

DE denotes a domain error.

RE denotes a rank error.

Scalar Operators

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

- Monadic Scalar Operators

  The argument used with a monadic scalar operator may have any rank and dimensions. The result has the rank and dimensions of the argument. The domain of the result may be different than that of the argument.

- Dyadic Scalar Operators

  If the rank and dimensions of the arguments used with a dyadic scalar operator are the same, the operation is performed on corresponding components of the two arguments and the result has the same rank and dimensions. If the arguments have different ranks or dimensions and both contain more than single elements, a rank or length error will be reported.

  If one argument has multiple elements and the other is a scalar or single element array, the operation is performed on the single element with each component of the multiple element argument. The result has the rank and dimensions of the multiple element argument. If neither argument has multiple elements but they are not both scalars, the result is given the shape of the higher ranking argument. The shapes of results of scalar operations for various arguments are tabulated below.

Result

Left Argument \ Right Argument	S	V1	M1	H1	V	M	H
S	S	V1	M1	H1	V	M	H
V1	V1	V1	M1	H1	V	M	H
M1	M1	M1	M1	H1	V	M	H
H1	H1	H1	H1	H1	V	M	H
V	V	V	V	V	V[t]	RE	RE
M	M	M	M	M	RE	M[t]	RE
H	H	H	H	H	RE	RE	H[tt]

where

S denotes a scalar.

V denotes a vector.

M denotes a matrix.

H denotes a higher order array.

---

[t] Dimensions of arguments must be identical.

[tt] Rank and dimensions of arguments must be identical.

RE denotes a rank error.

V1 denotes a single element vector.

M1 denotes a single element matrix.

H1 denotes a single element higher order array.

Arithmetic Group

Each operator in the arithmetic group has a monadic and dyadic form. If any argument is in the character domain, a domain error is reported. If an argument is in the logical domain, however, it is considered to be a special case of the integer domain. Results are always in the numeric (integer or floating) domain. (An exception is the monadic + which is valid for any argument, including character data. The result is unchanged in domain.)

The + Operator (Identity and Addition)

- The monadic + is the identity operator. It is essentially "no operator".

  R←+B

  Domain Table:

B	C	L	I	F
R	C	L	I	F

  Examples:

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

- The dyadic + is the addition operator.

  R←A+B

  Domain Table:

A \ B	C	L	I	F
C	DE	DE	DE	DE
L	DE	I	I	F
I	DE	I	I/F[t]	F
F	DE	F	F	F

R

[t] The sum of integers is floating-point if the result exceeds the integer range.

If a floating-point result exceeds the range of floating-point numbers, DOMAIN ERR is reported.

Examples:

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

The - Operator (Negation and Subtraction)

- The monadic - is the negation operator.

R←-B

Domain Table:

B	C	L	I	F
R	DE	I	I	F

Examples:

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

- The dyadic – is the subtraction operator.

R←A-B

Domain Table:

A \ B	C	L	I	F
C	DE	DE	DE	DE
L	DE	I	I	F
I	DE	I	I/F[†]	F
F	DE	F	F	F

} R

Examples:

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

[†]The sum of integers is floating-point if the result exceeds the integer range.

The × Operator (Signum and Multiplication)

- The monadic × is the signum operator.

```
R←×B
```

If B is positive, R is 1. If B is zero, R is 0. If B is negative, R is ¯1.

Domain Table:

B	C	L	I	F
R	DE	I	I	I

Examples:

```
      × ¯2 3.5 0 .0001
¯1  1  0  1
```

- The dyadic × is the multiplication operator.

```
R←A×B
```

Domain Table:

A \ B	C	L	I	F
C	DE	DE	DE	DE
L	DE	I	I	F
I	DE	I	I/F[t]	F
F	DE	F	F	F

} R

Examples:

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

The ÷ Operator (Reciprocal and Division)

- The monadic ÷ is the reciprocal operator.

```
R←÷B
```

[t] The product of integers is floating-point if the result exceeds the integer range.

Domain Table:

B	C	L	I	F
R	DE	F	F	F

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

Examples:

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

- The dyadic ÷ is the division operator.

$R \leftarrow A \div B$

Domain Table:

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

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

Examples:

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

The * Operator (Exponential, Exponentiation)

- The monadic * is called the exponential operator. The monadic * is the equivalent of the dyadic form with e (base of natural logarithms) supplied as a left argument. The value used for e is approximately 2.7182818284590451.

$R \leftarrow *B$

Domain Table:

B	C	L	I	F
R	DE	F	F	F

If B exceeds 174.6730895, DOMAIN ERR is reported. If B is less than ¯179.8716889, R is 0.

Examples:

```
      *1 .5 0 ¯190
2.718281828  1.648721271  1  0
```

- The dyadic * is the exponentiation operator.

  $R \leftarrow A * B$

  Domain Table:

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

R

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

Examples:

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

The ⍟ Operator (Natural Logarithm, Logarithm)

- The monadic ⍟ is the natural logarithm (base e) operator.

  $R \leftarrow ⍟B$

  Domain Table

B	C	L	I	F
R	DE	F	F	F

If B is not a postiive number, a DOMAIN ERR is reported.

Example:

```
      ⍟2.7182818284  1  .04978706837
1  0  ¯3
```

- The dyadic ⍟ is the generalized logarithm (base A) operator.

  $R \leftarrow A ⍟ B$

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

Domain Table:

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

R

Examples:

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

The L and ⌈ Operators (Floor and Ceiling, Minimum and Maximum)

- The monadic L and ⌈ are the floor and ceiling operators.

  $R \leftarrow \lfloor B$

  For L, R is the algebraically greatest integer such that R≤B+FUZZ.

  $R \leftarrow \lceil B$

  For ⌈, R is the algebraically smallest integer such that R≥B-FUZZ.

  FUZZ is approximately 1E⁻13 unless modified by SETFUZZ intrinsic function.

Domain Table:

B	C	L	I	F
R	DE	I	I	I†/F

Examples:

```
      L 2.9  2.99  ¯2.99  2.99999999999999
2  2  ¯3  3
      ⌈ 2.1  2.01  ¯2.01  2.00000000000001
3  3  ¯2  2
```

- Dyadic L and ⌈ are the minimum and maximum operators.

  $R \leftarrow A \lfloor B$ $\qquad$ $R \leftarrow A \lceil B$

  For each element pair of A and B, R is the algebraic minimum or maximum.

---

†Result is floating-point if the value exceeds the range for the integer domain.

Domain Table:

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

R

Examples:

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

The | Operator (Absolute Value and Residue)

- The monadic | is the absolute value operator.

  $R \leftarrow |B$

  Domain Table:

B	C	L	I	F
R	DE	I	I	F

  Examples:

```
      |¯2.15
2.15
      |¯2  ¯4.3  5  7.2
2  4.3  5  7.2
```

- The dyadic | is the residue operator.

  $R \leftarrow A \mid B$

  The result is B modulo A expressed as a nonnegative value; that is, the smallest nonnegative value R such that R=B+n×A for an integer n.

  Domain Table:

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

R

  If A is zero and B is less than zero, DOMAIN ERR is reported. If A is zero and B is greater than or equal to zero, B is assigned to R. DOMAIN ERR is also reported if the ratio of B to A or either value is too large for meaningful modulo operation within the precision range used in computing real values.

Examples:

```
        2 3 ¯4|5 ¯7 6
1   2   2
        3.2|6.5 7.4 17
0.1   1   1
```

The ○ Operator (Pi Times and Circular)

- The monadic ○ is the pi times operator.

  R←○B

  The result is 3.141592653589793 times B.

  Domain Table:

B	C	L	I	F
R	DE	F	F	F

  Examples:

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

- The dyadic ○ is the circular operator.

  R←A○B

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

A	R	Domain of B[t]	Range of R[t]
-7	Arctanh	|B≤1	-18.36840028 to 18.36840028
-6	Arccosh	1≤B≤Max*.5[tt]	0 to 88.02969193
-5	Arcsinh	|B≤Max*.5[tt]	-88.02969193 to 88.02969193
-4	(¯1+B*2)*.5	1≤|B	
-3	Arctan		-π/2 to π/2
-2	Arccos	|B≤1	0 to π
-1	Arcsin	|B≤1	-π/2 to π/2
0	(1-B*2)*.5	|B≤1	0 to 1
1	Sine	|B≤7.074237752E15	-1 to 1
2	Cosine	|B≤7.074237752E15	-1 to 1
3	Tangent	|B≤7.074237752E15[ttt]	-2.86708057E15 to 1.146832228E16
4	(1+B*2)*.5		
5	Sinh	|B≤175.3662366	
6	Cosh	|B≤175.3662366	1 to Max[tt]
7	Tanh		-1 to 1

[t] The domains of B and ranges of R are narrower than those theoretically possible. The limitations reflect the precision with which real number data are represented and with which computations are made in the computer.

[tt] Max = 7.237005577E75.
Max *.5 = 8.507059173E37.

[ttt] For tangent, DOMAIN ERR results if B is indistinguishable from an odd integer multiple of π/2.

The value of B is in radians for the trigonometric functions. For the inverse trigonometric functions, the value of R is in radians. The domain of the result is always floating-point.

Examples:

```
      1○○2
¯1.046360549E¯15
      3○4 5 6
1.157821282  ¯3.380515006  ¯0.2910061914
      ¯7○.5
0.5493061443
```

Notice in the first example that the result (the sine of 2π) should actually be zero. The actual result reflects the effect of computing with approximately 16 decimal-place precision.

The ! Operator (Factorial and Combination)

- The monadic ! is the generalized factorial operator.

```
R←!B
```

The result is B factorial for nonnegative integral values of B. If B is not an integer, the result is the gamma function of B+1.

Domain Table:

B	C	L	I	F
R	DE	I	I	F

Examples:

```
      !7
5○4○
      !.66  ¯.75  0
0.9016683712  3.625609908  1
```

- The dyadic ! is the generalized combination operator.

```
R←A!B
```

If the arguments are positive integers and A is less than or equal to B, the result is the number of combinations of B things taken A at a time:

```
R=(!B)÷(!A)×!B-A
```

The generalized form A!B is related to the Beta function as follows. $\beta(A,B)$ is

```
÷B×(A-1)!A+B-1
```

Domain Table:

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

} R

Examples:

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

Relational Group

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

The < Operator (Less Than)

R←A<B

The result is 1 if (A-B)≤-FUZZ×|B, and is 0 otherwise.

Domain Table:

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

R

Examples:

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

The ≤ Operator (Less Than or Equal To)

R←A≤B

The result is 1 if (A-B)≤FUZZ×|B, and is 0 otherwise.

Domain Table:

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

R

Examples:

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

The = Operator (Equals)

R←A=B

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

Domain Table:

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

R

Examples:

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

The ≥ Operator (Greater Than or Equal to)

R←A≥B

The result is 1 if (A-B)>-FUZZ×|B, and is 0 otherwise.

Domain Table:

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

R

Examples:

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

The > Operator (Greater Than)

R←A>B

The result is 1 if (A-B)>FUZZ×|B, and 0 otherwise.

Domain Table:

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

R

Examples:

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

The ≠ Operator (Not Equal To)

R←A≠B

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

Domain Table:

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

R

Examples:

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

Logical Group

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

The ∧ Operator (And)

R←A∧B

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

Domain Table:

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

R

A domain error results if both A and B are not equal to either 1 or 0 (within FUZZ limits).

Examples:

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

The ∨ Operator (Or)

R←A∨B

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

Domain Table:

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

R

A domain error results if both A and B are not equal to either 1 or 0 (within FUZZ limits).

Examples:

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

The ⍲ Operator (Nand)

$R \leftarrow A \barwedge B$

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

Domain Table:

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

R

A domain error occurs if both A and B are not equal to either 1 or 0 (within FUZZ limits).

Examples:

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

The ⍱ Operator (Nor)

$R \leftarrow A \barvee B$

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

Domain Table:

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

R

A domain error results if both A and B are not equal to either 1 or 0 (within FUZZ limits).

Examples:

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

The ~ Operator (Not)

R←~B

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

Domain Table:

B	C	L	I	F
R	DE	L	L	L

A domain error results if B is not equal to either 1 or 0 (within FUZZ limits).

Examples:

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

Composite Operators

The three composite operators extend dyadic scalar operations to arrays. In the following descriptions of these operations, the bracketed value K represents that coordinate of the argument array along which the specified operator is to act. If K is unspecified, the last coordinate of the array is assumed. The symbol d represents any dyadic scalar operator.

The d/ Operator (Reduction)

R←d/[K]B

The result is an array having a dimension vector equal to that of array B except that the Kth component is not present. If ⌿ is used instead of /, the first coordinate is assumed and [K] may not be specified.

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

For a scalar or an array comprising a single component the result has the same value as B. For an empty array the result has the value of the identity element of operator d, as shown in the table below.

d	Identity Element	Comment	d	Identity Element	Comment
×	1		⍣	None	
+	0		!	1	Left identity only.
÷	1	Right identity only.	⌈	Minimum	-7.237005469E75
-	0	Right identity only.	⌊	Maximum	7.237005469E75
*	1	Right identity only.	>	0	Right identity only.
|	0	Left identity only.	≥	1	Right identity only.
⊛	None		<	0	Left identity only.
○	None		≤	1	Left identity only.
∨	0		=	1	
∧	1		≠	0	
⍲	None				

Domain restrictions for operator d apply. If B includes more than one element, the domain of the result is the same as indicated in domain tables for the dyadic scalar operators.

Examples:

```
      □←R←+/2 4 6 8
20
      □←R←-/2 4 6 8
¯4
      □←R←!/10
10
```

```
      A←2 4⍴⍳8
      A
1 2 3 4
5 6 7 8
      +/A
10 26
      +⌿A
6 8 10 12
      +/+/A
36
      B←2 3 4⍴⍳24
      B
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24
      +/B
10 26 42
58 74 90
      +/[2]B
15 18 21 24
51 54 57 60
      +⌿B
14 16 18 20
22 24 26 28
30 32 34 36
      +/+/B
78 222
      +/+/+/B
300
      +/,B
300

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

The d\ Operator (Scan)

```
R←d\[K]B
```

The result has a dimension vector the same as that of B. If ⍀ is used instead of \, the first coordinate is assumed and [K] may not be specified.

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

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

Thus the last component of the result will equal d/B.

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

Domain restrictions for operator d apply. If B has more than one element, the result domain is that indicated in the domain table for d.

Examples:

```
      +\1 3 5 7
1 4 9 16
      +\¯5 0 7 0 1
¯5 ¯5 2 2 3
      -\3 9 5 1
3 ¯6 ¯1 ¯2
      ×\1 2 3 4 5
1 2 6 24 120
      ÷\1 2 3 4 5
1 0.5 1.5 0.375 1.875
      ≤\7 9 5 ¯4
7 1 0 0
```

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

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

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

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

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

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

The d.d Operator (Inner Product)

$R \leftarrow A d_1 . d_2 B$

The result is an array having a dimension vector equal to all except the last dimension of array A catenated with all except the first dimension of array B. The dimension of the last component of A must be the same as that of the first component of B, or one of those dimensions must be 1. The domain of the result is indicated by the operators d_1 and d_2. Operators d_1 and d_2 may be any dyadic scalar operators. For example, R←A+.×B gives the conventional matrix inner product.

For vector arguments, the result is

$d_1 / A d_2 B$

For example:

```
      4+.×5 6
224
      +/4×5 6
224
      1 2 3+.×4 5 6
32
      +/1 2 3×4 5 6
32
      1 0 1 0+.∧1 1 0 0
1
      1 0 1 0+.∨1 1 0 0
3
```

If A is a vector and B a matrix, the Ith component of the result is

$d_1 / A d_2 B[;I]$

For example:

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

If A is a matrix and B is a vector, the Ith component of the result is

$$d_1/A[I;]d_2B$$

For example:

```
      C←1 2 3 4
      B+.×C
57 56
      B[1;]
3 2 6 8
      B[2;]
5 4 9 4
      +/B[1;]×C
57
      +/B[2;]×C
56
```

For matrix arguments, the I;Jth component of the result is

$$d_1/A[I;]d_2B[;J]$$

For example:

```
      (2 4⍴⍳8)+.×4 2⍴⍳8
 50  60
114 140

      X←3 3⍴'CANDIDATE'
      Y←3 3⍴'DRAMATIZE'
      X∧.=Y
0 0 0
0 0 0
0 0 1
      X
CAN
DID
ATE
      Y
DRA
MAT
IZE
      X∨.=Y
0 1 0
1 0 0
0 0 1
      X∧.≠Y
1 0 1
0 1 1
1 1 0
```

Inner product also applies to higher order arrays. For the example below, the arguments are each three dimensional and the result has four dimensions. The I;J;K;Lth element of the result is +/A[I;J;]×B[;K;L]

```
      A←2 2 3⍴⍳12
      A
 1  2  3
 4  5  6

 7  8  9
10 11 12
      ⎕←B←3 2 2⍴12+⍳12
13 14
15 16

17 18
19 20

21 22
23 24
      A+.×B
110 116
122 128

263 278
293 308


416 440
464 488

569 602
635 668
      +/A[1;1;]×B[;1;1]
110
      +/A[1;1;]×B[;1;2]
116
      +/A[1;2;]×B[;2;2]
308
```

The ∘.d Operator (Outer Product)

```
R←A∘.dB
```

The result is an array having a dimension vector equal to that of A catenated with that of B. The scalar dyadic operation d is performed for each component of A with respect to all components of B. The domain of the result is determined by the rules for the operator d.

For vector arguments, the I;Jth component of the result is

```
A[I]dB[J]
```

For example:

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

Outer product is valid for arguments of higher rank. If, for example, A has rank 3 and B has rank 2, the elements of the result are defined by:

$R[I;J;K;L;M]$ is $A[I;J;K]\,d\,B[L;M]$

Mixed Operators

Operators not categorized previously as monadic or dyadic scalars or composite operators are called mixed operators. Rules for shapes and domains of the arguments and results vary and are described for the individual operators.

The ? Operator (Roll and Deal)

- The monadic ? is the roll operator.

```
R←?B
```

B must be in the integer domain (maximum value of 2,147,483,647). Each element R[I] of the result is an integer selected pseudorandomly from ⍳B. The range of the result depends on the value of the index origin (see the deal operator below). The shape of the result is the same as that of the right argument.

Examples:

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

- The dyadic ? is the deal operator.

```
R←A?B
```

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

A may not exceed B, and both must be single numeric elements.

Examples:

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

The ⍳ Operator (Index Generator and Index Of)

- The monadic ⍳ is the index generator operator.

```
R←⍳B
```

B must be a single numeric element, within FUZZ of an integer. The result is a vector comprising B components, beginning with the index origin and incremented monotonically by 1. Normally the index origin is 1, but the command)ORIGIN 0 can be used to change the origin to 0. If B is 0, the result is the empty vector.

Examples:

```
      □←R←⍳4
1  2  3  4
      )ORIGIN 0
WAS 1
      □←R←⍳4
0  1  2  3
```

- The dyadic ⍳ is the index of operator.

```
R←A⍳B
```

The value of each element of the result is the smallest index I such that A[I] equals the corresponding element in B. The left argument must be a vector. The right argument may have any rank. If no match for an element of B is found in A, then that element of the result is set to 1+⍴A. The shape of the result is the same as the shape of the right argument. The result is in the integer domain.

Note that A may be an empty vector but must not be a scalar, and the value of the result depends on whether the index origin is 1 (the default case) or 0. A and B may be of any domain. Note, however, that if A is character data, for example, and B is numeric, the result will be entirely "no match" values.

Examples:

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

The , Operator (Ravel, Catenation and Lamination)

- The monadic , is the ravel operator.

```
R←,B
```

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

Examples:

```
      B←2 2⍴⍳4
      B
  1  2
  3  4
      ,B
1  2  3  4
      □←C←2 4⍴'SIGMASIX'
SIGM
ASIX
      ,C
SIGMASIX
```

- The dyadic , is the catenation and lamination operator.

```
R←A,B
```

If A and B are vectors or scalars, the result is a vector comprising all components of A followed by all components of B. Both A and B must be either numeric or text.

Examples:

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

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

Examples:

```
      □←M←4 7⍴'M'
MMMMMMM
MMMMMMM
MMMMMMM
MMMMMMM
      X←2 7⍴'X'
      Y←'1234567'
      Z←'1234'
      W←'O'
      M,[1]X
MMMMMMM
MMMMMMM
MMMMMMM
MMMMMMM
XXXXXXX
XXXXXXX
      M,[1]Y
MMMMMMM
MMMMMMM
MMMMMMM
MMMMMMM
1234567
      M,Z
MMMMMM1
MMMMMM2
MMMMMM3
MMMMMM4
      M,[1]W
MMMMMMM
MMMMMMM
MMMMMMM
MMMMMMM
OOOOOOO
      M,W
MMMMMMO
MMMMMMO
MMMMMMO
MMMMMMO
```

Lamination – If a noninteger coordinate value is indicated in catenation, and its value, relative to current origin, is within 1 of a valid coordinate, the operation performed is termed lamination. In this case the variables A and B are joined on a new coordinate. The length of the new coordinate is always 2.

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

Examples:

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

OOOOOOO
OOOOOOO
OOOOOOO
OOOOOOO
      M,[1.25]W
MMMMMMM
OOOOOOO

MMMMMMM
OOOOOOO

MMMMMMM
OOOOOOO

MMMMMMM
OOOOOOO
      M,[2.3]W
MO
MO
MO
MO
MO
MO
MO

MO
MO
MO
MO
MO
MO
MO

MO
MO
MO
MO
MO
MO
MO

MO
MO
MO
MO
MO
MO
MO
      ρM
4 7
      ρM,[.5]W
2 4 7
      ρM,[1.5]W
4 2 7
      ρM,[2.5]W
4 7 2
```

The ρ Operator (Dimension and Reshape)

- The monadic ρ is the dimension operator.

```
R←ρE
```

The result is an integer vector comprising the number of components each index of B contains. That is, R contains the highest index in each coordinate of B. Thus, the expression ρρB, represents the rank of B, assuming an index origin of 1. If B is a scalar, ρB results in the empty vector.

Examples:

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

- The dyadic ρ is the reshape operator.

```
R←AρB
```

The result is an array of the dimensions specified by vector A and the contents of B, if any, in index sequence. Elements of A may be positive integers or zero. If any component of A is zero, R is empty. If A is empty, R is a scalar. If B is empty, it may not be reshaped except to an empty result. If the reshape requires fewer elements than B contains, only the required elements are in the result. If the result requires more elements than B contains, B is reused as required. B may be of any rank or domain.

Examples:

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

The ⌽ Operator (Reversal and Rotation)

- The monadic ⌽ is the reversal operator.

```
R←⌽[K]B
```

The result is a reversal along the Kth coordinate of B. If K is omitted, the last coordinate is assumed. (If ⊖ is used instead of ⌽, the first coordinate is assumed and [K] may not be specified.)

Examples:

```
      ⌽'EMIT'
TIME
      ⌽[1]3 3ρ⍳9
7 8 9
4 5 6
1 2 3
      ⌽3 3ρ⍳9
3 2 1
6 5 4
9 8 7
```

- The dyadic ⌽ is the rotation operator.

```
R←A⌽[K]B
```

The result is a cyclic rotation of B by the number of components determined by A. If A is positive, rotation is to the left; if A is negative, rotation is to the right. Rotation is performed along the Kth coordinate of B. If K is omitted, the last coordinate is assumed. (If ⊖ is used instead of ⌽, the first coordinate is assumed and [K] may not be specified.)

Examples:

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

The ⍉ Operator (Transposition)

- The monadic transposition operation has the following syntax.

```
R←⍉B
```

The result is an array comprising the elements of B with the order of all coordinates reversed. For any B, $(\rho\phi B)=\rho\rho B$. If B is a matrix, for example, the result is a matrix whose rows are the columns of B and whose columns are the rows of B. Monadic transpose of a scalar or vector yields R←B.

Examples:

```
      ⎕←A←3 5⍴'AGENTVIGORAGONY'
AGENT
VIGOR
AGONY
      ⍉A
AVA
GIG
ENO
NOR
TYY
      ⎕←B←2 3 4⍴(⍳12),100+⍳12
  1   2   3   4
  5   6   7   8
  9  10  11  12

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

  2 102
  6 106
 10 110

  3 103
  7 107
 11 111

  4 104
  8 108
 12 112
```

- Dyadic transposition operations have the following syntax.

```
R←A⍉B
```

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

```
(⍴A)=⍴⍴B
```

There are two cases of dyadic transposition:

Case 1: A is a permutation of ⍳⍴⍴B (the coordinates of B). A is then described as the inverse permutation vector. The A[I]th component of ⍴R is the Ith component of ⍴B, and thus the A[I]th coordinate of the result is the Ith coordinate of B. Examples:

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

XA
PO

AT
EN
```

Case 2: A satisfies the relationship (⍳⌈/A)∈A; that is, A is a dense set of the first k coordinates of B, permuted, with some coordinates duplicated. If B is a matrix, the only possible form for A is (1 1), and the result is the principal diagonal of the matrix. Example:

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

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

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

```
      Z
ABCD
EFGH
IJKL

MNOP
QRST
UVWX
      ⍴Z
2 3 4
      A←1 1 1⍉Z
      A
AR
      ⍴A
2
```

Examples: (cont.)

```
      B←1 2 2⍴⍳Z
      B
AFK
MRW
      ⍴B
2 3
      C←2 1 1⍴⍳Z
      C
AM
FR
KW
      ⍴C
3 2
      D←2 1 2⍴⍳Z
      D
AN
ER
IV
      ⍴D
3 2
      E←1 2 1⍴⍳Z
      E
AEI
NRV
      ⍴E
2 3
      F←2 2 1⍴⍳Z
      F
AQ
BR
CS
DT
      ⍴F
4 2
      G←1 1 2⍴⍳Z
      G
ABCD
QRST
      ⍴G
2 4


      X←2 3 4 5⍴⍳120
      1 1 1 1⍴X
1 87
      1 1 1 2⍴X
 1  2  3  4  5
86 87 88 89 90
      2 2 2 1⍴X
 1 86
 2 87
 3 88
 4 89
 5 90
      1 1 2 2⍴X
 1  7 13 19
81 87 93 99
      2 2 1 1⍴X
 1 81
 7 87
13 93
19 99
      1 2 2 3⍴X
 1  2  3  4  5
26 27 28 29 30
51 52 53 54 55

61 62 63 64 65
86 87 88 89 90
111 112 113 114 115
```

Examples: (cont.)

```
      3 2 2 1⍉X
 1    61
26    86
51   111

 2    62
27    87
52   112

 3    63
28    88
53   113

 4    64
29    89
54   114

 5    65
30    90
55   115
```

The ⍋ Operator (Grade Up)

R←⍋*B*

The result of the grade up operation is a vector of indexes (relative to the index origin) of components of B ranked in ascending order of magnitude. B is a numeric vector. Identical components are ranked in index order. Note that the result of B[⍋B] is a "sort" of B in ascending sequence. Thus, "grade up" provides indices for sorting.

Examples:

```
      ⍋5 10 15 20
1 2 3 4
      ⍋5 10 10 15
1 2 3 4
      ⍋3 1 4 1
2 4 1 3
```

The ⍒ Operator (Grade Down)

R←⍒*B*

The result of the grade down operation is a vector of indexes of components of B ranked in descending order of magnitude. B is a numeric vector. Identical components are ranked in index order. The values of the result depends on the index origin. (B[⍒B] is a "sort" of B in descending sequence.)

Examples:

```
      ⍒5 10 15 20
4 3 2 1
      ⍒5 10 10 15
4 2 3 1
      ⍒3 1 4 1
3 1 2 4
```

The ⊥ Operator (Base Value or Decode)

R←*A*⊥*B*

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

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

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

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

Example:

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

The result is formed by W+.xB:

```
                 1x3600     2x60     3x1
WxB  is          3600       120      3
A    is  3723
```

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

Examples:

```
      2⊥1 0 1 1
11
      4⊥3 2 1 0
228
      10⊥9 8 7
987
      1 2 3⊥45 67 89
560
      ⍝ K IS A TABLE OF TIMES REPRESENTED IN DAYS(ROW 1),
      ⍝ HOURS(ROW 2),MINUTES(ROW 3),AND SECONDS.
      K
   0   0   0   0   1  11
   0   0   0   2   3  13
   0   1  16  46  46  46
  10  40  40  40  40  40
      ⍝ EACH COLUMN OF K REPRESENTS A TIME VALUE.
      ⍝ IF K IS OPERATED ON BY THE 'BASE VALUE' OPERATOR,
      ⍝ THE RESULT IS A VECTOR OF TIMES IN SECONDS.
      ⍝ THE RADIX VECTOR IS-- 365 24 60 60
      365 24 60 60⊥K
10  100  1000  10000  100000  1000000
```

The ⊤ Operator (Representation or Encode)

```
R←A⊤B
```

R is a "base A" representation of B. R satisfies the relationship ((×/A)|B-A⊥R)=0. A and B must be numeric, and R is numeric. Note that the ⊤ and ⊥ operators are "opposites". Note also that since Encode carries out a modulus operation, it is subject to the DOMAIN ERR conditions for that operation.

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

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

Examples:

```
      ⍝ BINARY REPRESENTATION
      (8⍴2)⊤75
0 1 0 0 1 0 1 1
      ⍝ OCTAL REPRESENTATION
      (3⍴8)⊤75
1 1 3
      ⍝ DECIMAL REPRESENTATION
      (5⍴10)⊤31415
3 1 4 1 5
      ⍝ VARIED UNIT REPRESENTATION
      24 60 60⊤75432
20 57 12
      ⍝ EXAMPLE OF TRUNCATION
      10 10⊤31415
1 5
      ⍝ THE ARGUMENTS FOR REPRESENTATION NEED NOT BE INTEGERS:
      (8⍴1.5)⊤32.75
1 0.5 1 0 0 0.5 0 1.25
      ⍝ H IS A VECTOR OF TIME VALUES IN SECONDS.
      H
10 100 1000 10000 100000 1000000
      ⍝ H CAN BE ENCODED IN TERMS OF DAYS,HOURS,MINUTES, AND SECONDS.
      365 24 60 60⊤H
  0   0   0   0   1  11
  0   0   0   2   3  13
  0   1  16  46  46  46
 10  40  40  40  40  40
      ⍝ IN THE RESULT, EACH COLUMN REPRESENTS ONE ELEMENT OF H.
      ⍝ ROW 1 IS DAYS, ROW 2 IS HOURS, ROW 3 IS MINUTES, AND ROW 4 IS
      ⍝ SECONDS.
```

The / Operator (Compression)

R←A/[K]B

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

Compression is performed along the Kth coordinate of B. If K is omitted, the last coordinate is assumed. (If ⌿ is used instead of /, the first coordinate is assumed and [K] may not be specified.)

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

Examples:

```
      B←2 2⍴⍳4
      1 0/B
1
3
```

The \ Operator (Expansion)

R←A\[K]B

A must be a vector of 1's and 0's and must include the same number of 1's as the length of the coordinate to be expanded. B may be of any rank and domain. Expansion occurs along the Kth coordinate of B. If K is omitted, the last coordinate is assumed. If ⍀ is used instead of \, the first coordinate is assumed and [K] may not be specified. Thus the difference between \ and ⍀ is

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

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

Expansion consists of extending the length of the affected coordinate of B by insertion of zeros (or blanks if B is text) in positions indicated by zeros in the argument A. The process is best described by example.

```
      1 0 1 0 1\⍳3
1 0 2 0 3
      ⍝ THE FOLLOWING EXAMPLES SHOW EXPANSION ON EACH OF THE
      ⍝ COORDINATES FOR A RANK 3 ARRAY.
      B←2 2 2⍴⍳8
      1 0 1⍀B
 1 2
 3 4

 0 0
 0 0

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

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

 5 0 6
 7 0 8
      A←2 2 2⍴'ABCDEFGH'
      1 0 1⍀A
AB
CD



EF
GH
      1 0 1\[2]A
AB

CD

EF

GH
      1 0 1\A
A B
C D

E F
G H
```

The ↑ Operator (Take)

```
R←A↑B
```

A must be an integer scalar or vector, and the length of A must equal the rank of B[†]. Each element of A controls the "take" from a coordinate of B. R has the same rank as B. The dimension vector of R is ⍴A.

If A[⍳]≥0, then the Ith coordinate of R is the first A[⍳] elements of the Ith coordinate of B. If A[⍳]<0, the last A[⍳] elements are used. If A[⍳] indicates more elements than are present in the coordinate of B, R is padded with 0's (or blanks if B is text).

Examples:

```
      ¯3↑⍳5
3  4  5
      7↑⍳5
1  2  3  4  5  0  0
      B
   1  2
   3  4

   5  6
   7  8
        1 2 3↑B
   1  2  0
   3  4  0
```

The ↓ Operator (Drop)

```
R←A↓B
```

A must be an integer scalar or vector, and the length of A must equal the rank of B[†]. Each element of A controls the "drop" from a coordinate of B. R has the same rank as B. The dimension vector of R is (⍴B)-⍴A. If a dimension in the result thus created would be negative it is set to zero.

If A[⍳]≥0, then the Ith coordinate of R is all but the first A[⍳] elements of the Ith coordinate of B; that is, the first A[⍳] elements are dropped. If A[⍳]<0, the last A[⍳] elements of the Ith coordinate of B are dropped.

Examples:

```
      ¯3↓⍳5
1  2
      3↓⍳5
4  5
      B
   1  2
   3  4

   5  6
   7  8
        1 2 2↓B
        2 2 1↓B
        1 1 1↓B
   8
```

(Note: Result may be an empty array)

[†]If B is a scalar it is treated as though it were a 1 element array whose rank is the length of A.

The ∈ Operator (Membership and Execute)

- The dyadic ∈ is the membership operator.

```
R←A∈B
```

If a given component of A is contained in B, the corresponding component of R is equal to 1; otherwise, it is 0. The result has the same shape as A and is in the logic domain. B may have any rank. If A and B are numeric, FUZZ is used in the equality test.

Examples:

```
      A←'ALPHABET'
      B←'ABCDE'
      C←2 4⍴⍳8
      A∈B
1 0   0 0 1 1 1 0
      1 5 10∈C
1 1   0
      ⍝NOTE THAT MEMBERSHIP MAY BE USED WITH NUMERIC VERSUS
      ⍝  TEXT ARGUMENTS--THE RESULT IS ALWAYS ZEROS.
      A∈C
0 0   0 0 0 0 0 0
      C∈A
0 0 0 0
0 0 0 0
      1 2 3∈'1 2 3'
0 0   0
```

- The monadic ∈ is the execute operator.

```
R←∈B
```

B must be a scalar or vector whose length does not exceed 512. The domain of B must be character type unless B is an empty vector. Ordinarily, the argument B will be a small text vector.

Once the argument has met the above requirements, an execute operator departs from the mold of the other operators. An execute operation becomes a variation of evaluated input. That is, the characters in its argument, if any, are treated much as if they were input after an evaluated input prompt.

Thus, it is possible to execute system commands within a function or even in the midst of an arithmetic expression. Execute operations can be applied so that an APL workspace can define or revise functions, create its own variable names, or compose new formulas and evaluate them.

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

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

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

Note: Since branching is not permitted in evaluated input, the form ∈'→N' is not allowed. ∈'⍎' is permitted.

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

```
      0\∈'⍳0'
0
      0\∈''
0
```

An empty numeric vector also results when executing most system commands.

```
      ÷3,∈')DIGITS 4'
WAS 10
0.3333
```

As far as the execute operation is concerned, a system command either yields an empty result or it yields no result whatsoever. For example, the following commands yield no result:)OFF,)OFF HOLD,)CLEAR, and)LOAD.

Also no result occurs when executing a suspension-clear. This corresponds to the use of a suspension-clear during evaluated input. As illustrated below, the state indicator has the same appearance for an execute operation as for evaluated input.

```
      4,⎕
⎕:
      )SI
⍎
FUNX[2]  *
⎕:
      ⍳0
4
      4,∈')SI'
⍎
FUNX[2]  *
4
      6,⎕
⎕:
      →
      )SI
FUNX[2]  *
      6,∈'→'
      )SI
FUNX[2]  *
```

The execute operator can also be used to access function definition mode, but certain limitations are imposed. A basic limitation exists since only one "statement" (text string) can be the argument of an execute operator. This means that if a function is opened by such a statement it must also be closed by the same statement. In other words, only "single-line" function definition operations are possible with the execute operator. (All or any portion of a function can be created or revised by utilizing multiple execute operations.)

To enter function definition mode via an execute operator, the operator's argument (text vector) must have a del (or locked del) as its first nonblank character. Since only single-line definitions are possible, APL does not require the closing del. However, the argument can include a closing del if the user so desires, except on line deletions. (The line deletion feature is described below). As usual for function definition, the opened function cannot be locked or pendant. This means that a function cannot modify itself, since the function is pendant when it performs an execute operation. It is possible for one function to modify another.

An execute operation produces a result, if successful. The result of executing function definition mode is always an empty integer vector. This is the same result produced when executing system commands.

If an error (for example, DEFN ERROR) is detected during an "executed" function definition, the original definition remains intact with one exception; SI DAMAGE errors perform the revision dictated by the execute operator's argument. In all other error cases, the function is not changed.

Executing function definition mode consumes substantial execution time. When can this capability be used to best advantage? The optimum candidate for an executed function revision is a highly iterative or recursive function. Based on initial conditions (or input supplied by a user), a new function line can be embedded in the heart of a loop. The time consumed in making the change can be inconsequential if this allows the loop to run

faster than would have been possible without the function change capability. The primary characteristic to watch for is a very repetitive loop containing a process that may vary for different runs of that loop.

The ability to enter function definition mode via an execute operation is not limited to revising a function line. Any definition statement that could include an opening and closing del can also be used as the argument of an execute operator. This covers every definition statement except those that allow deleting a line or editing characters of a line. An APL program can effectively edit characters of a line by altering the argument of an executed replacement line. Line deletion is easily accomplished by catenating the INDEX character to the line number designation (see example below).

Simple examples illustrate various features of executed function definition mode. Note that the closing del shown in the examples is optional except for line deletion, where it must be omitted.

Function Creation

```
      ⍎'∇F X∇'
      ⍎'∇F[1]X+X∇'
      ⍎'∇F[2]X-X∇'
```

Line Replacement

```
      ⍎'∇F[2]X÷X∇'
```

Line Insertion

```
      ⍎'∇F[1.5]X×X∇'
```

Display All Lines

```
      ⍎'∇F[⎕]∇'
     ∇ F X
[1]    X+X
[2]    X×X
[3]    X÷X
     ∇
```

Display Starting at Line 2

```
      ⍎'∇F[⎕]2]∇'
[2]    X×X
[3]    X÷X
```

Display Old Line and Replace

```
      ⍎'∇F[2⎕]X*X∇'
[2]    X×X
```

Display Single Line

```
      ⍎'∇F[2⎕]∇'
[2]    X*X
```

Line Deletion

```
      INDEX←2¯32
      ⍎'∇F[2]',INDEX
```

To verify the deletion, the function is displayed below

```
      ∇F[⎕]∇
     ∇ F X
[1]    X+X
[2]    X÷X
     ∇
```

Modification of a Character in a Line

```
      A←'∇F[2]X-X'
      ∈A
      ∇F[⎕]∇
    ∇ F X
[1]   X+X
[2]   X-X
    ∇
      A[6]←'W'
      ∈A
      ∇F[⎕]∇
    ∇ F X
[1]   X+X
[2]   W-X
    ∇
```

The foregoing discussion indicates things that can be done using the execute operator. Things that cannot be done include

- An executed argument cannot contain an unbalanced quote mark (the error message "OPEN QUOTE" is issued in such cases).

- The executed argument cannot translate into a "super long" line (despite the fact that the argument can contain 512 characters). If the translated "input" exceeds 130 columns, the "TRUNCATED" error message is issued.

Error handling is unique in the case of the execute operation. Recall that for evaluated input, the user is prompted to reissue a correct version of the erroneous input. This is not possible for an execute. Furthermore, the argument of one execute operator may contain another, and so on. For the sake of clarity in such cases, after a diagnostic message (such as "DOMAIN ERROR"), the path leading to that diagnostic is displayed until a normal suspension point is reached. The following example illustrates error handling during an involved execute operation.

```
      ∇Z←Y F X
[1]   A←'Y+'
[2]   B←'X'
[3]   C←'∈A,B'
[4]   Z←100+∈C
[5]   ∇
      5 F 4
109
      5 F 'FOUR'
DOMAIN ERR
      Y+X
       ^
      ∈A,B
      ^
F[4]  Z←100+∈C
           ^
```

The ⌹ Operator (Matrix Inversion and Matrix Divide)

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

If B is a nonsingular square matrix, the minimum is (except for computational round-off errors) zero, and R is the solution of a set of simultaneous equations. If, in addition, A is an identity matrix, R is the inverse of B. That is

equivalent to R←⌹B. If A is a vector, R is the solution to one system of simultaneous equations. If A is a matrix, each column of A represents the constants for a linear system with coefficient matrix B, and each column of R is the corresponding solution.

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

If B is singular, that is, has fewer linearly dependent rows than columns, SING ERR is reported.

If B is nonsquare and A is an identity matrix, the result is a left inverse of A and the operation is equivalent to R←⌹B.

Examples:

```
      ⍝INVERSE OF A SQUARE MATRIX

      ⎕←B←3 3⍴3 1 4 1 5 9 2 6 5
  3  1  4
  1  5  9
  2  6  5
      ⎕←R←⌹B
   0.3222222222   ¯0.2111111111    0.1222222222
  ¯0.1444444444   ¯0.07777777778   0.2555555556
   0.04444444444   0.1777777778   ¯0.1555555556
      )DIGITS 5
WAS 10
      ⍝VERIFY THAT THE INNER PRODUCT OF R AND B IS
      ⍝ESSENTIALLY THE IDENTITY MATRIX

      R+.×B
 1.0000E0     6.9389E¯17   6.6613E¯16
 5.5511E¯17   1.0000E0     4.4409E¯16
¯6.9389E¯17  ¯5.4991E¯16   1.0000E0

      ⍝LEFT INVERSE OF A NONSQUARE MATRIX

      ⎕←B←5 3⍴3 1 4 1 5 8 2 7 4 3 5 9 8 7 6
  3  1  4
  1  5  8
  2  7  4
  3  5  9
  8  7  6

      ⎕←R←⌹B
   0.074106   ¯0.082157   ¯0.072245   ¯0.015323    0.13129
  ¯0.10492     0.011612    0.17084    ¯0.048546    0.013386
   0.061261    0.06862    ¯0.073814    0.085902   ¯0.04531

      ⍝AGAIN, VERIFY THAT THE INNER PRODUCT OF R AND B IS
      ⍝THE IDENTITY MATRIX

      R+.×B
 1.0000E0     4.1633E¯17  ¯1.2490E¯16
¯1.2490E¯16   1.0000E0     1.6653E¯16
¯1.5266E¯16  ¯3.8598E¯16   1.0000E0

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

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

      ⎕←R←A⌹B
2.1444  8.2111  5.0889
```

Examples: (cont.)

```
      ⍝VERIFY THAT (B+.×R) APPROXIMATELY=A

      A-B+.×R
1.0658E¯14  2.4869E¯14  7.1054E¯15

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

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

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

      ⍝CHECKING THE RESULT:

      A-B+.×R
 1.0658E¯14  1.0658E¯14
 2.4869E¯14  2.4869E¯14
 7.1054E¯15  7.1054E¯15

      ⍝A LEAST+SQUARES SOLUTION

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

      A←12.03  8.78  6.01  3.75  ¯.31  ¯2.79

      R←A⌹B
      R
14.941  ¯2.9609
      ⍝THE RESULT GIVES THE INTERCEPT AND SLOPE OF THE LINE
      ⍝THAT IS THE LEAST-SQUARES BEST FIT TO THE POINTS OF A.

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

I-Beam Functions

The I-beam functions are a group of system-dependent functions providing information about the workspace or system environment.

```
R←⌶B
```

The ⌶ symbol is formed by the ⍭ overstruck with the ⊥. The argument must be a single integer element in the range 19 through 29, or -1 through -3. The result is as indicated in Table 2.

Table 2. I-Beam Functions

B	Information Given
19	Session connect time, in 60ths of a second.
20	Time of day, in 60ths of a second.
21	APL execution time (CPU time since APL was invoked), in 60ths of a second.
22	Remaining available workspace, in bytes.
23	Numbe of on-line users.
24	User's log-on time, in 60ths of a second.
25	Today's date, mmddyy, in base 10 (where mm is month, dd is day, and yy is year).
26	Current value of line counter. In executing a defined function, this is the number of the line being executed.
27	Vector of line numbers in the state indicator.
28	Terminal type (value set by TERMINAL command.): 1 Standard APL (on-line default) 2 Non-APL 2741 3 Teletype 4 Card reader and/or line printer (off-line default) 13 Tektronix 4013
29	User's account as a character vector.
I-1	Shows overhead time (starting from the point at which APL was invoked) in 60ths of a second. This is primarily an indication of the time taken by the monitor to process input and output.
I-2	Results in an integer scalar whose value is the number of unused entries in the symbol table. This represents the maximum number of new names acceptable for the current workspace.
I-3	Results in an integer scalar whose value is 1 if this is an on-line APL operation, or 0 if this is a batch run.

T-Bar Functions

The T-bar operator, ⍑ (the encode operator, ⍳, overstruck by the negative sign, -) is provided for certain system interfaces. Most of these interface functions are of concern only to installation personnel, and should be avoided by APL users. (Indiscriminate use of the T-bar operator is likely to cause damage to the user's workspace — see the SYS ERR message in Appendix A.)

The monadic use of T-bar results in an integer code for the type of its argument:

code	type of argument
1	Logical
2	Text (i.e., character data)
3	Integer
4	Real
5	Index Sequence
6	List

The first four types are familiar to APL users and were discussed in Chapter 3. An index sequence (code 5) represents integer data; it is a compact representation of a regularly spaced sequence of integers whose fundamental source is the index generator (monadic iota). List type data (code 6) results when parentheses enclose two or more APL expressions that are separated by semicolons. A list has length (the number of such expressions), but instead of values a list has pointers (pointing to the value of each expression in that list). The use of lists is restricted to certain specified intrinsic functions in Xerox APL. Their use in other contexts will result in a SYNTAX ERR or LIST ERR message.

The T-bar operator is also used dyadically. Installation personnel use dyadic T-bar operations when generating workspaces that contain intrinsic functions (such as WSFNS). Essentially, this associates a name, such as DELAY, with a portion of the APL processor (see also Appendix C). Subsequent use of the name DELAY is interpreted as a function reference to that part of the processor. Users need not be concerned with this use of T-bar; usually they will copy or load workspaces having intrinsic functions.

There is one dyadic use of T-bar that is of interest to many APL users; this is the character generator function. It converts integer data into corresponding character data, and thus allows the user to generate special characters, possibly unrecognized by APL. The relationship between an integer (between 0 and 255) and the generated character can be determined by examining Figure B-1 in Appendix B. The integer n corresponds to the nth character in the table of APL Codes. A word of caution, however; the table shows characters in their hexadecimal position; the user must remember to convert this to a decimal position. For example, the INDEX character corresponds to the decimal integer 32 (the hexadecimal value is 20) according to the table.

To generate the nth character, the following form is used.

```
2⍕n
```

The left argument must be the scalar integer 2; this designates that the T-bar operator is to be used for character generation. The right argument may have any shape, but its domain must be integer, with values between 0 and 255. The result has a shape identical to the right argument, but is text data.

For convenience, users may assign generated characters to an easily remembered name. A common example is

```
INDEX←2⍕32
```

Another character that is commonly generated is the backspace. In the following example, this is created and then used to display overstrike characters not recognized by APL:

```
      BKSP←2⍕8
      'A',BKSP,'¨'
Ä
      'ABC',BKSP,BKSP,'¨¨'
AB̈C̈
      'TEST',BKSP,BKSP,BKSP,'¨ ¨'
TÈST̈
```

6. APL STATEMENTS

As mentioned in Chapter 2, each completed line of input is classified as either a statement or a system command. Statements specify the operations to be performed by the APL system, such as calculations, branching, and assignment of values or expressions. System commands – treated in Chapter 8 – are concerned with the mechanical aspects of the system, such as logging on and off, and saving, loading, and deleting workspaces. Statements can be entered when the system is in either execution mode or function definition mode. The user indicates the end of a statement by depressing the RETURN key. In execution mode, the computer then interprets and executes the operations contained in the statement. In definition mode, the computer stores the statement until the entire function is executed. Blanks may appear anywhere in a statement except embedded within a number or a name. In general, an APL statement cannot be continued on another line. A text constant, however, may include one or more carriage returns, thus allowing multiline statements. The user is cautioned that if he opens a text constant with a quote and forgets the closing quote, APL considers all subsequent input to be part of that text constant until an ending quote is reached. For example:

```
      A←'LONG COMMENT, CLOSING QUOTE FORGOTTEN
A
A
)CLEAR
'
      A
LONG COMMENT, CLOSING QUOTE FORGOTTEN
A
A
)CLEAR
      )CLEAR
CLEAR WS
```

In this example the first two requests to display A and the first)CLEAR command were ignored because APL considered them to be part of the text assigned to A. The ending quote allowed resumption of normal input. The display of A now shows the multiline text vector, and the)CLEAR command now works.

For all practical purposes there are four kinds of statements in Xerox APL: comment, branch, assignment and non-assignment, and compound.

Comment Statements

To enter a comment statement, the user types the symbol ⍝ at the beginning of a line and follows it with his comment. The ⍝ symbol is produced by typing a ∩ symbol (upper shift C) and overstriking it with a ∘ symbol (upper shift J). This symbol signals APL that the line is a comment and is not to be executed. Any valid APL characters may be included in a comment; invalid APL characters will produce an error message. If a comment extends over several lines, each line must begin with the ⍝ symbol. Some examples of comments are shown below:

```
⍝ ROOM AREA ROUTINE.
⍝
⍝
⍝ EACH LINE OF A MULTIPLE-LINE
⍝ COMMENT MUST BEGIN WITH A ⍝.
```

A comment statement can be entered as a direct line (during execution mode) or it can be entered as part of a defined function. If a comment statement is entered as a direct line, it is not retained in the user's workspace. If a comment statement is used in a function definition, however, the statement will have a line number, will occupy workspace, and will be displayed like any other function line. A function cannot be closed on a comment line,

because the closing ∇ symbol will be treated as just another symbol in the comment. An example of a comment in a function definition is shown below:

```
      ∇A←H TRIAREA B
[1]   ⍝CALCULATES AREA OF TRIANGLE.
[2]   A←H×B÷2
[3]   ∇
```

In Xerox APL any executable statement may include a comment to its right. Everything to the left of a ⍝ character is considered executable. Everything to the right is considered comment. Some examples are

```
[10]  COST←HOURS×RATE   ⍝COST FOR STRAIGHT-TIME LABOR.
[15]  OCOST←1.5×HOURS×RATE   ⍝ COST FOR OVERTIME LABOR.
```

There is also another way for a user to enter remarks at his terminal. Instead of first typing the ⍝ symbol, the user can type the comment and then backspace to the beginning of the line and strike the ATTN key. This method allows the user to type remarks (including illegal overstruck characters) for his own annotation. In effect, an empty line is produced. An example of this method is shown below:

```
      THIS IS A ATTN REMARK.
      ∨
```

Branch Statements

Branch statements are generally used within function definitions to alter the sequential execution of statements.[†] A branch statement has the general form

```
→exp
```

where exp stands for an integer value. The value determines the statement number of the statement to be executed next, as follows:

1. If the value is a statement number of a statement within the current function, then that statement will be executed next. Thus the statement

   ```
   [5]   →(2>A)×3
   ```

   where A has a value of zero, will cause a branch to statement 3 of the current function. (The value 3 is derived as follows: the expression (2>A) returns a value of 1; and this value is multiplied by 3.)

2. If the value is a statement number outside the function being executed, then execution of that function terminates. For example, the statement

   ```
   [4]   →0
   ```

   indicates a branch to statement 0, which is outside the function. Since functions begin with statement 1, branching to statement 0 is an effective way to terminate a function.

[†]Another form of branch statement is covered later – the branch arrow that is not followed by an expression. A branch arrow by itself can be used to terminate execution of a suspended function and the functions that invoked it, thus effectively clearing the state indicator to the next suspension (if any). This application of the branch arrow is described in Chapter 7.

3. If the value is an empty vector, then no branch occurs and the next sequential line is executed. If there are no more lines, execution of the function is terminated. An empty vector can be created in any of the following ways:

```
0/s
0ρs
s×⍳0
```

where 0 is the result of a comparison expression, and s represents a statement number. (If the result of the comparison statement is 1 instead of 0, the next statement executed will be the one indicated by the statement number.) Substituting the comparison expression A=4, which produces a value of 0 or 1, and the statement number 2 in the above expressions illustrates the simplicity of this type of branching:

```
[5]    →(A=4)/2
[5]    →(A=4)ρ2
[5]    →2×⍳(A=4)
```

In each case if the value of A equals 4 (that is, the comparison expression returns a 1), then line 2 is executed next. If A is any other value, then the comparison expression returns a 0, yielding an empty vector, and statement 6 will be executed next if it exists; otherwise execution of the function terminates.

The expression indicating the statement numbers can be a scalar or a vector. In other words, the user can specify branching to one statement, to one of two statements, or to one of any number of statements. Branching to one statement is described above. Branching to one of two statements can take either of the following forms:

```
→(s1,s2)[1+x op y]
→((x op y),~x op y)/s1,s2
```

where

s1 is the line number to be branched to if the comparison expression yields a 0.

s2 is the line number to be branched to if the comparison expression yields a 1.

x op y is a comparison expression; x and y are the values to be compared, and op is any of the following operators: < ≤ = ≥ > ≠ ∨ ∧ ⍲ ⍱ ∈

Both of these forms cause execution to branch to the first line number if the comparison operation yields a 0, or to the second line number if the comparison operator yields a 1. In illustration, the second form is used in a function definition and then executed with values for x and y:

```
      ∇X F Y
[1]   →((X<Y),~X<Y)/A1,A2
[2]   A1:'STEP A1'
[3]   →0
[4]   A2:'STEP A2'
[5]   →0
[6]   ∇



      1 F 2
STEP A1
      2 F 1
STEP A2
```

Clearly the second form can be expanded to include more statement numbers. Similarly, a branch to one of several statements can also take the form

```
→iϕv
```

where

i is a counter.

ϕ is the rotation function.

v is a vector of statement numbers, the first of which must be a positive integer or zero.

In this case the branch function selects statement iϕv as the next one to be executed. The following illustration shows how this branch function is carried out (see line number 3):

```
      ∇NUMB I
[1]   →0;'LOW';→2×⍳I≥4
[2]   →0;'HIGH';→3×⍳I≤6
[3]   →(I-4)ϕ 4 5 6
[4]   →0;'FOUR'
[5]   →0;'FIVE'
[6]   →0;'SIX'∇

      NUMB 3
LOW
      NUMB 4
FOUR
      NUMB 5
FIVE
      NUMB 6
SIX
      NUMB 7
HIGH
```

See Figure 4 for a summary of some branch function formats that can be used; of course, APL offers many other forms of branching – too numerous to detail in this manual.

Branch to line s or to next line:

```
→(x op y)/s
→(x op y)ρs
→s×⍳x op y
```

Branch to line s1 or line s2:

```
→(s1,s2)[1+x op y]
→((x op y),~x op y)/s1,s2
```

Branch to one of several lines:

```
→((x op y),(x op y),x op y)/s1,s2,s3
→iϕv
```

Figure 4. Summary of Formats for Branching

Statement Labels

Instead of referencing a line number in a branch statement, the user can assign a statement label to the branch point and then reference that label. To assign a label to a statement, precede the statement with a variable name and a colon, as shown:

```
[5]   END: A←B÷2
```

The label END can now be used in a branch statement to transfer execution to this statement. For example, the statement

```
[3]    →(A<1)/END
```

will cause a branch to line 5 if A is less than 1, or a branch to line 4 if A is 1 or more.

The value of a label is the line number with which it is associated at the close of function definition. If new lines are inserted via function editing (see Chapter 7), then the values of the labels are automatically respecified at the closing of the function definition. The value of a label cannot be respecified by an assignment; any attempt to do so will produce a syntax error message.

Like local variables (Chapter 3), the integer values of labels in one function can be accessed in other functions invoked by the function.

Use of a statement label in a branch statement is preferable to use of a line number, since any function editing may change the original line number. If any lines are inserted or deleted during function editing, all lines will be renumbered at the close of a function definition mode. For example, consider the following statement which specifies a branch to statement 5:

```
[3]    →5
```

If two new statements are inserted between lines 3 and 4, the old line 5 will be renumbered as line 7 at the close of function definition. However, the branch statement will still cause a branch to statement 5 instead of line 7 as now desired. This problem can be avoided if labels are used instead of statement numbers as branch points. (See "Changing Suspended Functions" in Chapter 7 for other considerations about labels.)

Assignment and Nonassignment Statements

An assignment statement is one that assigns an expression or value to a variable name. It has the general form

```
name ← expression
```

where name can be any variable name and expression can be any APL expression. Three examples of assignment statements are

```
B←6
A←B÷2
Z←(B<1)+3×5
```

A nonassignment statement is similar to an assignment statement except that it doesn't have the assignment arrow and the variable name to the left of it; however, a nonassignment statement can contain embedded assignments. Examples are

```
      B÷2
3
      (B<1)+3×5
15
      2×4+A←2
12
      +A←1
1
      +C←'BID'
BID
```

Notice the differences between assignment and nonassignment statements: (1) execution of an assignment ends on the assignment, and (2) an assignment statement produces no display, while a nonassignment statement displays the resultant value of the interpreted statement.

Compound Statements

Using semicolons for separation, all of the preceding kinds of statements can be combined in "compound" statements. Compound statements have the following characteristics:

1. A series of nonassignment statements produces the display action described in Chapter 3 (see the mixed output statement). Example:

```
      W←10
      L←20
      'DIMENSIONS ARE ';W;' BY ';L;';  AREA IS ';W×L
DIMENSIONS ARE 10 BY 20;  AREA IS 200
```

2. An assignment statement produces no display. Note, however, that any assignment statement becomes a nonassignment statement by simply placing a plus sign (that is, identity operator) at its extreme left. Example:

```
      5×4÷2;A←4
10
      5×4÷2;+A←4
104
```

   Notice in the last example that the results of a compound statement are printed without intervening spaces unless the spaces are specifically designated. Spaces are designated as shown below:

```
      5×4÷2;'     ';+A←4
10    4
```

3. A comment statement can have no statement to its right. All characters from the comment symbol ⍝ up to the end of the line are considered to be commentary. The semicolon preceding a comment is optional. Example:

```
      3⌈2 3  ⍝SHOWS MAX OPERATOR; THIS IS STILL A COMMENT.
```

4. A branch statement implies no special display. In the no-branch case, statements to the left of the branch will be interpreted; they are ignored if a branch occurs. This provides conditional execution capability. Example:

```
      ∇VERACITY X
[1]   →0;'TRUE';→2×⍳X≠1
[2]   →0;'FALSE';→3×⍳X≠0
[3]   'NEITHER TRUE NOR FALSE'∇

      VERACITY 4=2+2
TRUE
      VERACITY 2+2=4
NEITHER TRUE NOR FALSE
      2+2=4
2
      VERACITY (2+2)=4
TRUE
```

5. Suppose a branch statement has one or more nonassignment statements to its right. If a branch occurs, the appropriate display is produced before control is passed to another line. In the no-branch case, the display remains pending until completion of the compound statement. Example:

```
      ∇LOGIC X
[1]   'NOT ';→0×⍳X=0;'ZERO OR ';→0×⍳X=1;'ONE'∇

      LOGIC 1≠2
ONE
      LOGIC 3>4
ZERO OR ONE
      LOGIC 2+2
NOT ZERO OR ONE
```

7. DEFINED FUNCTIONS

As mentioned in Chapter 3, defined functions are used in the same way as operators, but most defined functions must first be formed by the user instead of being an inherent part of the APL language. In addition, there are some defined functions in Xerox APL that are intrinsic to the processor; that is, they call on code that exists in the processor. Both user-defined functions and intrinsic functions are referenced by name and are treated in the same way, but intrinsic functions are usually faster than functions that the user can define. (Intrinsic functions are always locked; the user can neither modify nor display them.)

User-Defined Functions

The following tasks are handled in function definition mode:

Creating user-defined functions
Displaying user-defined functions
Editing user-defined functions

Once created, most functions can be edited and displayed. Once a locked function is created, however, it cannot be edited or displayed (see "Locking Functions" later in this chapter). Locked function lines cannot even be displayed for error diagnosis. It is possible, however, to erase a locked function.

Function definition mode begins when a function is opened and continues until a function is closed or abandoned. (It is possible to close a different function than was originally opened by revising the name of the function.) A function may be "opened" during direct input or evaluated input (see Chapter 3), and it may be opened briefly during execution (see the "execute" operator, monadic ∈, Chapter 5). A function cannot be opened during any other form of input, such as quote-quad input or blind input; and a different existing function cannot be opened while still in function definition mode. Until a function is closed during function definition mode, APL execution is impossible except for system commands (which are executed and do not become part of the function being defined). Most system commands leave the currently open function intact and return the user to definition mode; however, some system commands cause a function definition to be abandoned (see "Issuing System Commands" later in this chapter).

Creating User-Defined Functions

A del symbol, ∇, followed by a function name specifies a change from the execution mode to the function definition mode. A second ∇ symbol ends function definition mode and declares a change back to execution mode. No execution of statements occurs during function definition, and no errors are reported except for line-scan errors, character errors, and definition errors. Instead, each statement is stored as part of the function.

Each defined function has a header and a body. The function header is the opening line of a function and declares the name (the identifier used to reference the function) and type of a function. The body of a function is the rest of the function. After the user enters a function header, APL responds with a statement number as follows:

```
      ∇CUBE
[1]
```

The line number [1] signifies that the first line of the function program may be entered. Each line thereafter is numbered sequentially until the function is completed. The statements are stored and are not executed until the entire function is called and executed.

Syntax of Defined Functions

A defined function can be niladic, monadic, or dyadic; that is, it can have zero, one, or two arguments. In addition, a defined function may return an explicit result or no result. Thus, there are actually six types of defined functions as illustrated by the following function header syntax possibilities:

Function Header Syntax

	No Explicit Result	Explicit Result
Niladic function	∇name	∇r ← name
Monadic function	∇name y	∇r ← name y
Dyadic function	∇x name y	∇r ← x name y

where

name is the user-assigned function name.

r is a variable to which the result is returned.

x and y are dummy variable names.

The syntax of the function header affects the way a function is referenced in a statement; that is, whether the function requires zero, one, or two arguments for execution. Defined functions with explicit results may appear in compound expressions, much like operators. Defined functions with implicit results must appear alone; they cannot appear in compound expressions except as the last function to be executed. Examples of the creation and use of each function type are shown in Table 3.

Table 3. Examples of Defined Functions

Function Type	Header Syntax	Examples of Definition and Use
Niladic function with explicit result	∇r ← name	∇*RESULT*←*PI* [1] *RESULT*←○1 [2] ∇ *PI* 3.141592654 ∇*RESULT*←*TRIANGLE* [1] *AREA*←0.5×*BASE*×*HEIGHT* [2] *DIAGONAL*←((*HEIGHT**2)+*BASE**2)*0.5 [3] *RESULT*←*AREA*,*DIAGONAL* [4] ∇ *BASE*←5 *HEIGHT*←8 *TRIANGLE* 20 9.433981132

Table 3. Examples of Defined Functions (Cont.)

Function Type	Header Syntax	Examples of Definition and Use
Niladic function with no explicit result	∇name	∇*PI* [1] *X*←○1 [2] *X* [3] ∇ *PI* 3.141592654 ∇*TRIANGLE* [1] *AREA*←0.5×*BASE*×*HEIGHT* [2] *DIAGONAL*←((*HEIGHT**2)+*BASE**2)*0.5 [3] '*AREA IS* ';*AREA* [4] '*DIAGONAL IS* ';*DIAGONAL* [5] ∇ *BASE*←5 *HEIGHT*←8 *TRIANGLE* *AREA IS* 20 *DIAGONAL IS* 9.433981132
Monadic function with explicit result	∇r ← name y	∇*RETURN*←*EXPAND INPUT* [1] *RETURN*←((2×⍴*INPUT*)⍴1 0)*INPUT* [2] ∇ *EXPAND* '*COPY COMMAND*' *C O P Y C O M M A N D* ∇*RETURN*←*DESCENDINGSORT INPUT* [1] *RETURN*←*INPUT*[⍒*INPUT*] [2] ∇ *DESCENDINGSORT* ¯5 ¯3 10 5 6 8 10 8 6 5 ¯3 ¯5
Monadic function with no explicit result	∇name y	∇*EXPAND INPUT* [1] *X*←((2×⍴*INPUT*)⍴1 0)*INPUT* [2] *X* [3] ∇ *EXPAND* '*COPY COMMAND*' *C O P Y C O M M A N D* ∇*DESCENDINGSORT INPUT* [1] *X*←*INPUT*[⍒*INPUT*] [2] *X* [3] ∇ *DESCENDINGSORT* ¯5 ¯3 10 5 6 8 10 8 6 5 ¯3 ¯5

Table 3. Examples of Defined Functions (Cont.)

Function Type	Header Syntax	Examples of Definition and Use
Dyadic function with explicit result	∇r ← x name y	∇*RESULT←BASE TRIANGLE HEIGHT* [1] *AREA←0.5×BASE×HEIGHT* [2] *DIAGONAL←((HEIGHT*2)+BASE*2)*0.5* [3] *RESULT←AREA,DIAGONAL* [4] ∇ *5 TRIANGLE 8* 20 9.433981132
Dyadic function with no explicit result	∇x name y	∇*BASE TRIANGLE HEIGHT* [1] *AREA←0.5×BASE×HEIGHT* [2] *DIAGONAL←((HEIGHT*2)+BASE*2)*0.5* [3] *'AREA IS ';AREA* [4] *'DIAGONAL IS ';DIAGONAL* [5] ∇ *5 TRIANGLE 8* *AREA IS* 20 *DIAGONAL IS* 9.433981132 ∇*X PLUS Y* [1] *ANS←X+Y* [2] *ANS* [3] ∇ *2 PLUS* 5 10 15 20 7 12 17 22

Variables Local to a Defined Function

Three types of variables that can be local to a defined function are

Dummies
Locals
Labels

Dummies and locals are named in the function header, while labels are named in the body of the function.

Dummies. Dummies are used in the header of a defined function to indicate the syntax of a function. For example, notice the header of the following simple function (this function calculates the area of a triangle):

```
     ∇A←H TRIAREA B
[1]  A←H×B÷2∇
```

The dummies A, H, and B in the function header indicate that the function named TRIAREA returns an explicit result and that the function operates on two arguments which must be furnished by the user. For example, suppose the user calls this function with the statement

```
AREA←10 TRIAREA 5
```

The dummy H in the function is assigned the value 10, and the dummy B is assigned the value 5. The result is returned in the dummy A, and is finally assigned to the variable AREA in the calling statement. Dummies possess values only within the function. That is, the use of A, H, and B as dummies does not affect their use as variables outside the function. If variables A, H, and B had values assigned to them before the function was called, they

will have the same values after the function is executed. For example, suppose the variable A (with value 21) had existed in the program before function TRIAREA was called. A display of variable A after the execution of function TRIAREA will demonstrate that A still has the value 21:

```
      A←21
      AREA←10 TRIAREA 5
      AREA
25
      A
21
```

Body of a Function. After the opening statement, in which the user creates the function header, the process of creating a function consists of inputting function statements and, finally, closing function definition. The user is prompted with a function line number each time the system is ready for further input. The process is ended by typing a closing ∇ followed by a RETURN key.

Locals. Locals are variables that retain their values only within the function in which they are defined. While a function is active, its local variables take precedence over any externally defined variables of the same name. A list of a function's local variables are added to the end of the function header, with each variable in the list preceded by a semicolon. For example, the function header

```
∇R←A CIRCLE B;X;Y;Z
```

indicates that the function named CIRCLE has locals X, Y, and Z. The values for these variables are assigned within the function; if these variables are referenced without having a value assigned within the function, an UNDEFINED message will be produced. If variables X, Y, and Z had values assigned to them before the function was called, they will revert to those values after the function is executed.

Labels. Function lines may be labeled to allow symbolically controlled branching (if a function is edited, line numbers may change). A labeled line has the form

```
[n]    name: statement
```

where n is the line number, name is the label, and statement is the content of the line. For example,

```
[4]    ERREXIT: →0;'ERROR EXIT'
```

In this example, the label ERREXIT has the value 4. If an attempt is made to assign a value to ERREXIT during function execution, a syntax error message will be reported. If the function is edited and the line number changes to [5], ERREXIT will then have the value 5.

Changing Suspended Functions. APL permits the user to change a function while it is in one or more suspended states (but never while pendant). This is seldom advisable. It is almost always preferred practice to clear out such suspensions before modifying the function to avoid possible confusion.

At the time a function is suspended, its (current) local variables have been determined by APL, and its labels have already been assigned their values. Changing the suspended function does not alter these determinations. Resuming execution of a suspended function will cause the determined items to take effect again – regardless of how the function was altered.

Directives

During function editing the user issues directives to transfer APL control to a line or to display one or more lines. A directive may take any of the following forms:

[1]	Directs APL to a line – here line 1.
[1□]	Directs APL to display a line and then to stay at that line for further editing – here line 1.
[□2]	Directs APL to display from a line to the end of the function – here beginning at line 2.
[□]	Directs APL to display an entire function.
[1□6]	Directs APL to a line to edit, starting (approximately) at a given column – here line 1 at column 6.

A directive always starts with a left bracket and ends with a right bracket. Only quads, digits, and decimal points are acceptable within the brackets, and no directive can have more than one decimal point and one quad. In addition, blanks are not allowed. The following are examples of illegal directives:

```
[1 2]
[1.21.]
[⍝1⍝]
[1E1]
```

The last directive, which may appear as 10 to someone familiar with constants, is in error; E is not allowed within a directive. Any erroneous directive will cause an error message to be printed. In addition, any characters to the right of the error detection point are disregarded – even a closing del.

Several directives may be used on one line, with the rightmost directive overriding any directives to the left of it. For example, notice the following portion of a function:

```
      ∇FF
[1]   X←Y
[2]   [1] Y←X
[2]   [5] A←B
```

The [1] directive on the second-to-last line overrides the [2] directive to its left and causes the statement on line 1 to be replaced with Y←X; notice that the next line prompt is [2]. (It should be obvious by now that a function line prompt is a form of a directive.) Similarly, the [5] directive on the last line overrides the [2] directive to its left and causes the expression A B to be assigned to line 5; the next line prompt will be [6].

Displaying User-Defined Functions

Once the user has defined a function, he can display it in any of the following ways:

Display all lines of the function.
Display one line.
Display from a specified line to the end of the function.

To display a function, the user opens the function with a del symbol, names the function, and specifies what he wants displayed, all on the same line. He can then either close the function with another del symbol (if no editing is to be done) or leave the function open for further editing.

If the user wants to display all of a function – say function TRIANGLE for example – the procedure is as follows:

```
      ∇TRIANGLE [⍝]∇
    ∇ BASE TRIANGLE HEIGHT
[1]   AREA←0.5×BASE×HEIGHT
[2]   DIAGONAL←((HEIGHT*2)+BASE*2)×0.5
[3]   'AREA IS ';AREA
[4]   'DIAGONAL IS ';DIAGONAL
    ∇
```

If the user wants to display only one line of a function – say line 3 of function TRIANGLE – the procedure is

```
      ∇TRIANGLE [3⍝]∇
[3]   'AREA IS ';AREA
```

Finally, if the user wants to display from one line to the end of a function – say from line 2 on of function TRIANGLE – the procedure is

```
      ∇TRIANGLE [⍝2]∇
[2]   DIAGONAL←((HEIGHT*2)+BASE*2)×0.5
[3]   'AREA IS ';AREA
[4]   'DIAGONAL IS ';DIAGONAL
```

The user can stop the display of lengthy functions at any point by pressing the ATTN key. This is especially useful when the user wants a range of lines displayed. For example, suppose he wants to display lines 10 through 15 of a 20-line function. He can request the display to start at line 10 and then press the ATTN key after line 15 has been displayed. If the display command was closed with a del symbol, APL will be in the execution mode after the interruption; if the closing del was omitted, APL will be in the function definition mode after the interruption.

Notice that the display commands in all of the above examples were closed with a del symbol. This symbol causes control to be returned to the execution mode as soon as the display is complete. If the user wants instead to remain in function definition mode and edit the function, he merely omits the closing del in the display command. See how the above examples appear without a closing del in each display command.

```
      ∇TRIANGLE [□]
    ∇ BASE TRIANGLE HEIGHT
[1]   AREA←0.5×BASE×HEIGHT
[2]   DIAGONAL←((HEIGHT*2)+BASE*2)×0.5
[3]   'AREA IS ';AREA
[4]   'DIAGONAL IS ';DIAGONAL
    ∇
[5]


      ∇TRIANGLE [3□]
[3]   'AREA IS ';AREA
[3]


      ∇TRIANGLE [□2]
[2]   DIAGONAL←((HEIGHT*2)+BASE*2)×0.5
[3]   'AREA IS ';AREA
[4]   'DIAGONAL IS ';DIAGONAL
[5]
```

Notice that after a single-line display, APL reprompts with the same line number; and that after a multiple-line display, APL prompts with the next available line number. The user can then edit the function as described below or he can type another del symbol to close the function. Closing the function definition with a del symbol does not alter the content of that line. For example, the following operation does not change the value of line 3; it will still be 'AREA IS ';AREA:

```
      ∇TRIANGLE [3□]
[3]   'AREA IS ';AREA
[3]   ∇
```

In summary remember that

[□] displays all of a function.

[2□] displays a single line (here 2).

[□2] displays from a line (here 2) to the end of the function.

Editing User–Defined Functions

The editing of user-defined functions is oriented to line-at-a-time editing capabilities:

Deleting a line
Inserting a line
Replacing a line
Modifying a line

The first three capabilities can be performed as shown in Table 4. The last capability – modifying a line – permits character editing (that is, deletion, insertion, and replacement of characters), adding to a line, and overstriking existing characters on a line. All of these capabilities are detailed below.

Table 4. Displaying and Editing Defined Functions[†]

Action to be Taken	Perform Action and Stay in Definition Mode – Function Already Open	Perform Action and Leave Definition Mode – Function Already Open	Open Function, Perform Action, and Stay in Definition Mode	Open Function, Perform Action, and Leave Definition Mode

Display all of a function	`[2] [□]`
	`∇ F`
	`[1] A`
	`[2] B`
	`[3] C`
	`∇`
	`[4]`
		`[2] [□]∇`
		`∇ F`
		`[1] A`
		`[2] B`
		`[3] C`
		`∇`
			`∇F[□]`
			`∇ F`
			`[1] A`
			`[2] B`
			`[3] C`
			`∇`
			`[4]`
				`∇F[□]∇`
				`∇ F`
				`[1] A`
				`[2] B`
				`[3] C`
				`∇`

Display a line	`[4] [2□]`
	`[2] B`
	`[2]`
		`[4] [2□]∇`
		`[2] B`
			`∇F[2□]`
			`[2] B`
			`[2]`
				`∇F[2□]∇`
				`[2] B`

Display a line and change it	`[4] [2□]B←X+Y`
	`[2] B`
	`[3]`
		`[4] [2□]B←X+Y∇`
		`[2] B`
			`∇F[2□]B←X+Y`
			`[2] B`
			`[3]`
				`∇F[2□]B←X+Y∇`
				`[2] B`

Display a function, beginning with a specified line	`[4] [□2]`
	`[2] B←X+Y`
	`[3] C`
	`[4]`
		`[4] [□2]∇`
		`[2] B←X+Y`
		`[3] C`
			`∇F[□2]`
			`[2] B←X+Y`
			`[3] C`
			`[4]`
				`∇F[□2]∇`
				`[2] B←X+Y`
				`[3] C`

Delete a line
 	
Note: The user cannot delete line zero. Also note that the ATTN, INDEX, and RET symbols in the examples stand for the ATTN, INDEX, and RETURN keys respectively.
	`[4] [2]`(INDEX)
	`[3]` (RET)
	
	or
	
	`[4] [2]`(ATTN)
	`∨`
	(RET)
	`[3]`
		`[4] [2]`(INDEX)
		`[3] ∇` (RET)
		
		or
		
		`[4] [2]`(ATTN)
		`∨`
		(RET)
		`[3] ∇`
			`∇F[2]`(INDEX)
			`[3]` (RET)
			
			or
			
			`∇F[2]`(ATTN)
			`∨`
			(RET)
			`[3]`
				`∇F[2]`(INDEX)
				`[3] ∇` (RET)
				
				or
				
				`∇F[2]`(ATTN)
				`∨`
				(RET)
				`[3] ∇`

Insert a line	`[3] [0.5] X` `[0.6]`	`[3] [0.5] X∇`	`∇F[0.5] X` `[0.6]`	`∇F[0.5] X∇`
Replace a line	`[4] [2] Z` `[3]`	`[4] [2] Z∇`	`∇F[2]Z` `[3]`	`∇F[2]Z∇`
Override a line number	`[4] [2]` `[2]`	`[4] [2]` `[2] ∇`	`∇F[4][2]` `[2]`	`∇F[4][2]` `[2] ∇`
Change the function header (this example adds a local variable to the function header)	`[4] [0] F;B` `[1]`	`[4] [0] F;B∇`	`∇F[0] F;B` `[1]`	`∇F[0] F;B∇`
Erase the current function		`[4] )ERASE F`		
Erase another function	`[4] )ERASE G` `[4]`	`[4] )ERASE G` `[4] ∇`		

[†]User input has been underlined throughout this table to distinguish it from APL output. The underlining does not actually appear at the terminal. In addition, a simple three-line function named F has been assumed in the examples in this table (see the first display entry in the table for the original content of function F).

Deleting a Line

A statement in a defined function can be deleted by striking the ATTN key followed by the RETURN key, or by striking the INDEX key followed by the RETURN key immediately to the right of the line number. (See also the notes following Table B-3 concerning terminals other than the standard ones assumed in this section.) Use of the INDEX and RETURN keys, however, is a faster and more convenient delete sequence because the computer does not have to react by printing a caret symbol. As an example, suppose the user wants to delete line 2 of the following function:

```
      ∇BASE TRIANGLE HEIGHT
[1]   ⍝THIS FUNCTION CALCULATES AREA AND HEIGHT OF TRIANGLE
[2]   ⍝BASE AND HEIGHT CANNOT EXCEED 5 AND 15 RESPECTIVELY
[3]   AREA←0.5×BASE×HEIGHT
[4]   DIAGONAL←((HEIGHT*2)+BASE*2)×0.5
[5]   'AREA IS ';AREA
[6]   'DIAGONAL IS ';DIAGONAL
[7]   ∇
```

First the user opens the function and directs the system to line 2

```
      ∇TRIANGLE [2]
```

APL responds with a [2], indicating that control is at line 2. The user strikes the INDEX key and then the RETURN key to delete the line, and APL responds with the next line number as shown:

```
[2]

[3]
```

Notice that the INDEX and RETURN keys do not cause anything to be printed at line 2. If the ATTN and RETURN keys are used instead, the sequence appears at the terminal as

```
[2]
      ^

[3]
```

The user can now either close the definition mode with a del symbol or proceed with further editing (including deleting the next line).[†] A display of the function at this point illustrates that line 2 has been deleted:

```
[3]   [⎕]
    ∇ BASE TRIANGLE HEIGHT
[1]   ⍝THIS FUNCTION CALCULATES AREA AND HEIGHT OF TRIANGLE
[3]   AREA←0.5×BASE×HEIGHT
[4]   DIAGONAL←((HEIGHT*2)+BASE*2)×0.5
[5]   'AREA IS ';AREA
[6]   'DIAGONAL IS ';DIAGONAL
    ∇
[7]
```

The definition mode can now be closed with a del symbol

```
[7]   ∇
```

Once definition mode is closed, APL renumbers the lines in sequential order, as illustrated by another display of the function

```
      ∇TRIANGLE [⎕]∇
    ∇ BASE TRIANGLE HEIGHT
[1]   ⍝THIS FUNCTION CALCULATES AREA AND HEIGHT OF TRIANGLE
[2]   AREA←0.5×BASE×HEIGHT
[3]   DIAGONAL←((HEIGHT*2)+BASE*2)×0.5
[4]   'AREA IS ';AREA
[5]   'DIAGONAL IS ';DIAGONAL
    ∇
```

[†]He can also press the RETURN key if he doesn't want to do anything to the line. APL will simply respond with the line number – in this case [4]. Another RETURN key will cause APL to prompt with [5], and so on.

Inserting a Line

A new line can be inserted in a defined function simply by reopening the function and entering the statement as described below. The user reopens the function by typing a del and the function name, to which APL responds by printing the line number of the next statement to be entered. If the new line is to be inserted at the end of the function, the user can now enter the new statement and close the function as shown:

```
      ∇TRIANGLE
[6]   ⍝THIS FUNCTION IS USED IN ROUTINES 1 AND 2.
[7]   ∇
```

If the new line is to be inserted between any two lines, however, the user must specify a line number between those two lines — any line number as long as it is between the two line numbers (see "Line Numbers" below). APL responds with that line number and the user can enter the new statement. For example, suppose the user had wanted to add a comment as the first line of function TRIANGLE instead of as the last line. He could have done this as follows:

```
      ∇TRIANGLE
[6]   [0.5]
[0.5] ⍝THIS FUNCTION IS USED IN ROUTINES 1 AND 2.
[0.6]
```

Notice the [0.6] prompt in this example. After an insert statement is entered, the APL system adds a 1 to the last place of the number chosen for the insert, and prompts with the new number. (The next prompt after [0.6] will be [0.7]; the next, [0.8]; and so on.) This allows the user to insert several lines.

A display of function TRIANGLE illustrates that line 0.5 has been added

```
[0.6] [□]∇
    ∇ BASE TRIANGLE HEIGHT
[0.5] ⍝THIS FUNCTION IS USED IN ROUTINES 1 AND 2.
[1]   ⍝THIS FUNCTION CALCULATES AREA AND HEIGHT OF TRIANGLE
[2]   AREA←0.5×BASE×HEIGHT
[3]   DIAGONAL←((HEIGHT*2)+BASE*2)×0.5
[4]   'AREA IS ';AREA
[5]   'DIAGONAL IS ';DIAGONAL
    ∇
```

After the function is closed, APL automatically renumbers the lines, as illustrated by the following display:

```
      ∇TRIANGLE [□]∇
    ∇ BASE TRIANGLE HEIGHT
[1]   ⍝THIS FUNCTION IS USED IN ROUTINES 1 AND 2.
[2]   ⍝THIS FUNCTION CALCULATES AREA AND HEIGHT OF TRIANGLE
[3]   AREA←0.5×BASE×HEIGHT
[4]   DIAGONAL←((HEIGHT*2)+BASE*2)×0.5
[5]   'AREA IS ';AREA
[6]   'DIAGONAL IS ';DIAGONAL
    ∇
```

Line Numbers. APL allows the user to type a line number with up to four numbers to the left of the decimal point and up to three numbers to the right. As noted above, after each insert line is entered, APL adds a 1 to the last place of the insert. As illustrated in the following portion of a printout, the next prompt after an .88 insert will be [.89]; the next, [.9]; the next, [1]; and so on:

```
      ∇F
[7]   [.88]
[0.88]
[0.89]
[0.9]
[1]
```

The highest integer line number printed by APL is [9999]; thus the highest possible line number is [9999.999]. If the user is prompted with [9999.999] and he issues a legal statement (say X←Y), APL will prompt him with the same line number since it cannot go any higher.

Replacing a Line

A line in a defined function can be replaced simply by reopening the function, directing control to the statement that is to be replaced, and entering the desired statement. For example, suppose the user wants to replace line 1 of function TRIANGLE with another statement. He reopens the function by typing a del and the function name and directs control to line 1 by typing that line number in brackets. After the RETURN key has been struck, APL responds to this entry by printing the specified line number at the left margin, as shown:

```
      ∇TRIANGLE [1]
[1]
```

Any statement the user enters at this point will replace what previously existed at that line. Suppose he now enters the following comment statement:

```
[1]    ⍝INPUT MUST BE IN FEET
[2]
```

Notice that the next prompt is at line 2. If the user does not want to do any more editing, he can close the function by entering another del

```
[2]    ∇
```

Notice that this action has no effect on line 2; it merely closes the function once more. The following display of function TRIANGLE illustrates the change to line 1:

```
      ∇TRIANGLE [⎕]∇
    ∇ BASE TRIANGLE HEIGHT
[1]   ⍝INPUT MUST BE IN FEET
[2]   ⍝THIS FUNCTION CALCULATES AREA AND HEIGHT OF TRIANGLE
[3]   AREA←0.5×BASE×HEIGHT
[4]   DIAGONAL←((HEIGHT*2)+BASE*2)×0.5
[5]   'AREA IS ';AREA
[6]   'DIAGONAL IS ';DIAGONAL
    ∇
```

Doing Several Things at Once

APL allows the user to open a function, change a line, and close the function all on one line. For example,

```
∇G[1][2]2.2∇
```

In this case the user opens function G, issues a directive to line 1, notices he really wanted to change line 2, changes the directive to line 2, replaces whatever exists on that line with the value 2.2, and then closes the function. This shortcut operation allows the user to change a function without having to interact extensively with the computer. Another example is shown below:

```
      ∇G[1⎕]1.11∇
[1]   1.1

      ∇G[⎕]∇
    ∇ G
[1]   1.11
[2]   2.2
    ∇
```

The first line requests that line 1 of function G be displayed, and the contents of that line changed to the value 1.11. The display of function G shows that line 1 has indeed been changed from 1.1 to 1.11. It should be noted that the user can display one line and change it at the same time, but he cannot display an entire function and change something at the same time.

Modifying a Line

As mentioned earlier, modifying a line involves character editing (that is, deletion, insertion, and replacement of characters), adding to a line, and overstriking existing characters in the line. Modifications to a line can be specified by overriding the present line number with the expression

```
[n⎕c]
```

where n is the number of the line to be edited (0 for a function header), and c is the approximate column at which to begin editing (the column position is the number of spaces from the left margin). APL will normally display the specified line, return to the next line, and space to the designated column.[†] If the typing element is still not in the proper position, the user can backspace or space forward until the desired position is reached.

Deleting Characters. To delete characters, type a slash beneath each character to be deleted

```
      ∇G
[3]   [2⎕8]
[2]   2.2
        /
```

In this case the user is deleting the decimal point. After the RETURN key is depressed, APL will display the corrected line, squeezing out any blanks left by the correction

```
[2]   22
```

Since APL waits at the end of the displayed line, the user can continue editing this line as if he had just typed it in or he can depress the RETURN key to end the line.

Inserting Characters. To insert one or more characters between two adjacent chararacters, type a single digit below the second character, indicating the number of blanks to be inserted (from 1 to 9) between the two characters. For example,

```
[3]   [2⎕8]
[2]   22
        3
```

This example inserts three spaces between the two characters on line 2. After the RETURN key is depressed, APL will display the edited line with the blanks and will backspace to the leftmost blank and wait for the user to insert something. An insertion of 0÷1 will appear at the terminal as

```
[2]   20÷12
```

where 0÷1 has been typed by the user.

[†]Unless the designated line happens to be a line that does not fit on a single line of paper, in which case no character editing can be done. APL will simply display the line and then reprompt the user with the same line. The following is an example of such a line:

```
[2]   'A
      B'
```

If the user wants to insert more than nine characters, he can type any of the letters A through Z instead of a digit to indicate the number of blanks to be inserted. The letter A inserts 5 blanks; the letter B, 10 blanks, and so forth.

The user is cautioned to leave enough room for an insertion. Not leaving enough room may result in illegal overstruck characters as shown below:

```
[3]    [2⎕8]
[2]    22
         2
[2]    20÷2
BAD CHAR
[2]    20÷
```

Any mistakes made in typing an insertion can be corrected with the BACKSPACE-ATTN key sequence ordinarily used for correcting errors. This sequence will erase everything from the point of correction to the end of the line, and the user can then enter the rest of the line as it should be. For example, suppose the user wants to insert the expression o1× in statement 2 and instead types the expression 01× (notice that the first expression means "pi times", while the second expression means "1 times"). He can correct this as shown below:

```
[3]    [2⎕10]
[2]    20÷12
           3
[2]    20÷01×12
           ^
           01×12
[3]
```

A display of line 2 illustrates that the correction has been made

```
[3]    [2⎕]
[2]    20÷01×12
```

Replacing Characters. To replace one character with one or more other characters, type a slash below the character and a digit or letter to the right of the slash indicating the number of characters in the replacement. After the RETURN key is depressed, APL will type the line with the specified number of spaces and will then backspace to the leftmost blank and wait for the user to type the replacement. For example, suppose the user wants to replace the leading character of statement 2 with the expression A←3. This operation will appear at the terminal as

```
[3]    [2⎕7]
[2]    20÷01×12
       /3
[2]    A←30÷01×12
```

where A←3 is the replacement typed by the user.

Adding Characters to End of Line. To add one or more characters to the end of a line, specify a zero as the column at which to begin editing. APL will then display the line unaltered and wait at the end of the line for the user to add something. An example of adding local variables to a function header is shown below:

```
[3]    [0⎕0]
[0]    RETURN←FUNC X;A;B
[1]
```

In this case APL typed the header as RETURN←FUNC X and waited at the end of the line, and the user typed the expression ;A;B.

Overstriking a Character. To edit a line and create a legal overstrike, specify a zero as the column at which to begin editing. APL will display the line and wait at the end of it; the user can then backspace to the character to be overstruck, and type the second character. An example of overstriking a character is shown below:

```
[8]    [5□0]
[5]    A←⍞
```

In this case the first line caused statement 5 — consisting of the expression $A \leftarrow \square$ — to be displayed and APL to wait at the end of the line. The user then backspaced to the quad and typed an apostrophe, thus creating the legal overstrike ⍞.

Another method of creating a legal overstrike is shown below:

```
[8]    [5□9]
[5]    A←□
         0
[5]    A←⍞
```

This method directs APL to just below column 9 instead of to the end of the line. The user then types a zero below the letter to be overstruck, and then depresses the RETURN key. APL displays the line, backspaces to where the zero was typed, and waits for the user to type something. Typing an apostrophe will now create the legal overstrike ⍞.

Editing a Line Number. Line numbers may be edited in the same way that the content of a line is edited. One application of editing line numbers is in repeating a statement at several different lines. For example, the following procedure can be used to repeat the contents of line 2 at line 4.1:

```
       ∇G[2□2]
[2]    A←30÷01×12
 /3
[4.1]    A←30÷01×12
```

In this case the user directs that line 2 is to be displayed and that editing is to begin at column 2. In response to the user entry of /3, APL displays the line, with three spaces replacing the line number, and then backspaces to the leftmost space where the user types the new line number 4.1. Now both lines 2 and 4.1 will contain the same statement.

Quitting Line Editing. To terminate line editing, simply type a period or any other unexpected (but valid) character, as shown below:

```
[1]    ABFG
[2]    [1□7]
[1]    ABFG
       //.
[1]    [□]
     ∇ F
[1]    ABFG
     ∇
[2]
```

Changing a Function Header

There are four changes the user can make to a function header (that is, to line zero).

1. Change the name of the function. Suppose the user reopens an existing function called FF1 and changes only the name of the function to G1 as shown below:

```
       ∇FF1[0]
[0]    RETURN←G1 ARG
[1]
```

This example assumes that G1 does not already exist. (If it did, a DEFN ERR message would occur.)

Changing the function name has no effect on function FF1 – the function still exists as it did before the reopen. Of course, FF1 is no longer the open function – G1 is. G1 is effectively a copy of FF1 with any modifications that the user may have made while function definition mode was in effect. This feature even allows synonymous function names as long as only the header is revised. It is possible for a user to make a locked version of an unlocked function in this manner, retaining the unlocked version only until he is satisfied that the locked version is error-free. Erasing the original function will not affect a synonymous function, nor will subsequent revision of the original. A synonymous function will retain the stop and trace vector supplied with the original function when it was copied.

2. Change the name of the result, change a function with a result to a no-result function, or change a no-result function to a function with a result. The following illustrates the change of function FF1's result from RETURN to R:

```
        ∇FF1[0]
[0]     R←FF1 ARG
[1]
```

3. Change the name of the argument. An example is shown below, where function FF1's argument is changed to X:

```
        ∇FF1[0]
[0]     R←FF1 X
[1]
```

4. Change the names of locals, insert locals, or delete locals.

Xerox APL does not allow the user to delete a function header. Any user attempt to do so will cause APL to print an error message and reprompt the user with line zero. If the user wants to get rid of the function he is working on, he must issue an)ERASE command.

Issuing System Commands

Xerox APL allows the user to enter any system command while he is in function definition mode. Most system commands keep the user in function definition mode, while some system commands (described below) return the user to execution mode or even take the user out of the APL system. After commands that keep the user in definition mode, the APL system will prompt with the same line number at which the command was given. For example, suppose the user is at line 5 of a function and wants to find out the names of variables in the workspace:

```
[5]     )VARS
AAA   BAT   DDD
[5]
```

The system commands that take the user out of function definition mode are)CLEAR,)LOAD,)COPY,)PCOPY,)QLOAD,)QCOPY,)QPCOPY,)CONTINUE,)CONTINUE HOLD,)OFF,)OFF HOLD,)SAVE, and an)ERASE of the current function. All of these commands force a close of the definition mode as though the user had closed it himself, but the resulting disposition of that function depends on the command. The)CLEAR,)LOAD,)QLOAD,)ERASE,)OFF, and)OFF HOLD commands, of course, cause the function to be destroyed; the)SAVE,)COPY,) PCOPY,)QCOPY,)QPCOPY,)CONTINUE, and)CONTINUE HOLD commands cause the function to be automatically reopened by APL after the command action has been taken. In the last situation, as soon as the work on these commands is finished, APL signals the user of the reopening by printing the function name (with an opening del) and prompting him with the next available line number. With the)CONTINUE and)CONTINUE HOLD commands, of course, the function is not opened until the next terminal session. The user should probably display the function before doing any more editing since renumbering may have occurred because of the forced close.

Recursive Function

Xerox APL permits recursive functions — a recursive function being a function that references itself in the body of its definition. As an example, notice the following function which returns the factorial of its argument:

```
      ∇Z←FAC N
[1]   Z←N×FAC N-1 ;→0×⍳N≤1 ;Z←1∇

      FAC 0
1
      FAC 1
1
      FAC 4
24
```

Tracing Execution

Function execution can be traced by displaying the values of statements (some or all) as execution of the function progresses. The user specifies the trace of a function by typing an expression of the form

T∆name ← line

where

name is the name of the function to be traced.

line is an integer or vector of integers that specify the line numbers for which values are to be displayed. Only the integers that correspond to line numbers in the named function are significant.

When any of the specified line numbers is executed, the value of its statement is printed. If the specified line contains a branch statement, the value of the expression to the right of the branch arrow is printed. Specifying a trace vector of 0 or ⍳0 discontinues the trace; for example, T∆FAC←0 or T∆FAC←⍳0 will stop a trace of function FAC.

Shown below is an example of tracing the execution of a function. Notice that all output resulting from a trace is identified by function name and line number.

```
      ∇Z←FAC N
[1]   Z←1
[2]   →0×⍳N≤1
[3]   Z←N×FAC N-1
[4]   ∇

      T∆FAC←1 2 3
      FAC 0
FAC[1]  1
FAC[2]  0
1
      FAC 1
FAC[1]  1
FAC[2]  0
1
      FAC 4
FAC[1]  1
FAC[1]  1
FAC[1]  1
FAC[1]  1
FAC[2]  0
FAC[3]  2
FAC[3]  6
FAC[3]  24
24
      T∆FAC←0
```

The same function written as a compound statement will produce the following trace output:

```
      ∇Z←FAC N
[1]   Z←N×FAC N-1;→0×⍳N≤1;Z←1∇

      TΔFAC←1
      FAC 0
FAC[1]  01
1
      FAC 1
FAC[1]  01
1
      FAC 4
FAC[1]  01
FAC[1]  21
FAC[1]  61
FAC[1]  241
24
      TΔFAC←0
```

A trace vector can also be included as part of a defined function. For example, if the statement TΔFAC←1 is included within the above function, line 1 will also be traced each time the function is invoked. Of course, more complex expressions can be used to produce conditional tracing. In such cases, the condition produces one or more values (line numbers) that are assigned to TΔFAC. This generalization also applied to the stop vector described below.

The)OBSERVE command, described in Chapter 8, extends the tracing facility. It permits the user to see not only the final result of a traced statement, but every intermediate result occurring as APL interprets a traced statement.

Stopping Execution

A planned suspension of function execution – called a function stop – can be established via a stop control vector. This vector is set in the same way that a trace control vector is set for a function trace (see "Tracing Execution" above). The user specifies a function stop by typing an expression of the form

SΔname←line

where

name is the name of the function.

line is an integer or vector of integers that specify the line numbers at which the function is to stop. Of course, only the integers that correspond to line numbers in the named function are significant.

When each specified line number is reached, APL stops function execution, does a line feed, prints the line number, and unlocks the keyboard. Function execution is now in a normal suspended state, and can be terminated or resumed by appropriate branching (see "Suspending Execution" below). Specifying a 0 or a ⍳0 discontinues the stop control vector; for example, SΔFAC←0 or SΔFAC←⍳0 discontinues any function stops on function FAC.

Shown below is an example of stopping execution of a function named CIRCLE:

```
      SΔCIRCLE←2 5
      CIRCLE
CIRCLE[2]
      suspension activities
      →2
13
10
30
CIRCLE[5]
```

Like the trace control vector, the stop control vector can also be used within a defined function – to stop execution after a certain number of loops, for instance. Editing a line that has a trace or stop control set for it removes the control for that line. Deleting a function also deletes trace control and stop control vectors associated with that function.

Execution of a function will be stopped before completion if any of the following occurs: the ATTN key is depressed, an error is encountered (unless sidetracking occurs, see Appendix A), or a user-set stop control is reached (see "Stopping Execution" above). When a suspension occurs, the APL system prints the name of the suspended function and the line number at which the stop applies, and then unlocks the keyboard. At this point, APL is in execution mode. Anything can be done during function suspension that can be done in the execution mode. As long as a function is suspended, its local variables are active and can be examined.

The user can resume execution of a suspended function by specifying a branch: entering a branch arrow followed by a RETURN key will clear that suspension, while specifying a branch to a particular line will resume execution at the beginning of that line (that is, at the right end of that line). Branching to a line outside a functions' range of line numbers will terminate the execution of that function.

As a general rule it is best not to leave a function suspended, because the information about that function occupies space which is precious to the APL user (see "State Indicator" below). In addition, each time the user attempts to execute an already suspended function, even more information about that function is added to computer memory. Thus, if the user has no specific reason to leave a function suspended, he should clear it before proceeding with the rest of his program. (See also the)SI CLEAR command in Chapter 8.)

State Indicator

The APL system contains a "state indicator" that gives a list of all suspended and pendant functions (that is, all "active" functions). A suspended function is one where execution is stopped before completion (see "Suspending Execution" above). A function is pendant unless specifically suspended. Most commonly, this is observed when one (pendant) function has called a suspended function. As a rule, suspended functions are stopped between lines, while pendant functions are stopped in the middle of a line. Note, however, when a function is suspended due to an error, the error marker may indicate the middle of the line; nevertheless, the function is stopped between that line and its predecessor. A display of pendant and suspended functions can be obtained via the system command)SI, with the most recent active function displayed first.

```
      )SI
Z[2]  *
X[4]  *
Y[3]
Z[2]  *
X[2]
W[5]  *
```

An asterisk after an entry indicates a suspended function; absence of an asterisk indicates a pendant function. The bracketed number after a function name is the number of the next line to be executed. If there are no suspended or pendant functions in the state indicator, no report will result from the)SI command. The number of items in the state indicator can be determined by typing the expression ⍴⍳27.

Unlike suspended functions, pendant functions cannot be erased, copied over, or edited. As an example, look at the state indicator list shown above. Functions Z and W can be edited but functions X and Y cannot. Notice that function X is listed as both pendant and suspended; it cannot be edited because it is pendant in one of its states. Also notice that function Z has been suspended twice.

There is one instance in which a pendant function will not be listed in the state indicator. Suppose a dyadic function is about to be executed, pending resolution of its left argument. Assume that argument is obtained as the result of some function, say F, and F is suspended. Then the dyadic function is pendant, because it is ready to execute as soon as F is resumed. But the dyadic function is not listed in the state indicator because it has not yet entered a state of execution. Fortunately, this situation is rare and seldom will confuse the user.

The system command)SIV lists the contents of the state indicator, including a list of variables local to pendant and suspended functions. Using the command)SIV might give the following:

```
      )SIV
Z[2]  *     A     B
X[4]  *     AA
Y[3]
Z[2]  *     A     B
X[2]        AA
W[5]  *
```

As with the)SI command, the most recent active function is displayed first. This example indicates that variables A and B are local to function Z and that variable AA is local to function X. Only the local variables of the most recent active functions can be accessed by the user. Thus, the user can access local variables A and B of the last invocation of function Z or local variable AA of the last invocation of function X, but not local variables A and B of the first invocation of function Z or local variable AA of the earlier invocation of function X (see X[2]).

The user can clear the state indicator by using the branch arrow (that is, →). Each branch arrow clears one suspended function and its associated pendant functions; thus, to clear the entire state indicator, the user enters a branch arrow for each asterisk in the list. For example, the user can clear the previous state indicator like this:

```
      →
      )SIV
X[4]  *     AA
Y[3]
Z[2]  *     A     B
X[2]        AA
W[5]  *
      →
      )SIV
Z[2]  *     A     B
X[2]        AA
W[5]  *
      ,
      →
      )SIV
```

The)SIV commands in this example show what is left in the state indicator after each branch arrow. The user could also have cleared the same state indicator by entering four successive branch arrows.

```
      →
      →
      →
      →
      )SIV
```

In this case, the)SIV command shows that nothing is left in the state indicator. The easiest way to clear the state indicator is to issue an)SI CLEAR command.

Xerox APL provides limited protection against "SI DAMAGE". As an example, suppose the user opens function F and modifies the header, changing the function's type (e.g., monadic to dyadic, result to no-result). He then attempts to close function F. If F is not suspended, the close occurs as usual. If F is suspended, APL issues a warning (to the effect that references in the state indicator will be damaged by the change to the header) and requests a response from the user. The user can either order the close to occur with SI DAMAGE or cancel the close in order to revise the function further, hopefully correcting the header. Only a type change requires this protection. It is perfectly permissible to make other changes to the header, such as adding locals or renaming the result or dummy arguments; however, this is seldom advisable (see "Changing Suspended Functions" above).

Locking Functions

A function can be locked during definition or editing by using an opening or closing ⍒ (∇ overstruck with a ~) instead of a ∇. A locked function can be executed, copied, or erased, but it cannot be displayed or altered. After a function is locked, any associated trace control or stop control cannot be changed. Examples of locking functions are

```
      ⍒HH                 ∇HH                 ⍒HH
[8]   ∇           [8]   ⍒           [8]   ⍒
```

Intrinsic Functions

Xerox APL provides the following set of special intrinsic functions (that is, predefined functions that exist in the APL processor):

ΔFMT	WIDTH	PAGE	ΔWM
DELAY	FIO	NLINES	
DIGITS	FIOE	HEADER	
ORIGIN	ERRN	VFCHAR	
SETFUZZ	ERRF	ΔXL	
SETLINK	ERRX	ΔTE	
TABS	ΔGRF	ΔCR	

These special functions exist in a locked state, normally in workspace WSFNS, (ΔGRF normally exists in workspace GRAF) under account 1. (If the user does not find them under account 1, he should check with the installation manager to determine the correct account). As locked functions, they can be copied, executed, or erased from the user's program, but they cannot be altered or displayed. (See also Appendix C for information about associating a particular name with an intrinsic function.)

ΔFMT, PAGE, NLINES, HEADER, VFCHAR, and ΔXL

These functions are used to format output reports and are described in Chapter 9.

DELAY

Function DELAY delays execution for a designated number of seconds. It has the syntax

```
DELAY  x
```

where x is the approximate number of seconds execution is to be delayed. The DELAY function may be interrupted by a break; this is treated as an ordinary break in execution.

DIGITS, ORIGIN, TABS, and WIDTH

Functions DIGITS, ORIGIN, TABS, and WIDTH are each similar to the system command of the same name, except that each is a function rather than a command and may therefore be used with other functions. The arguments for the functions are subject to the same restrictions as for the corresponding commands; for example, ORIGIN may only be set to 1 or 0. Each function has an explicit result that is the previous value of the relevant system parameter. For instance, consider the following function:

```
      ∇F  X
[1]   X←ORIGIN X
[2]   G
[3]   X←ORIGIN X∇
```

This example will cause APL to execute function G with whatever index origin is specified by the argument of F, and to restore the index origin to the value that it had before the execution of F.

SETFUZZ

Function SETFUZZ specifies the fuzz value to be used in comparisons (that is, the number of low-order bits to be ignored in comparisons). It has the syntax

```
y←SETFUZZ x
```

where

- y is the previous value of fuzz.
- x sets the current value of fuzz and can be an integer between 0 and 31.

SETLINK

Function SETLINK sets the value of the link in the chain of numbers generated in the use of the roll and deal functions. It has the syntax

```
y←SETLINK x
```

where

y is the previous value of the link.

x sets the current value of the link and must be a positive integer (if even, it is also converted to an odd number).

The results produced by the roll and deal functions are not the links themselves, but rather some function of them. The length of the chain (before repetition) is 2^{31}.

FIO

Function FIO is a file input/output primitive; a detailed description is given in Appendix B under "File Input/Output".

FIOE

Function FIOE is an alternate file input/output primitive (see also Appendix B, "File Input/Output"). Unlike FIO, the APL processor deals with errors encountered when using FIOE. This means that certain standard file I/O error messages result or that sidetracking (see Appendix A) is possible.

ERRN, ERRF, and ERRX

ERRN, ERRF, and ERRX are niladic functions. They are discussed in Appendix A under "Sidetracking on Errors or Breaks". Briefly, they return the following results:

ERRN – latest error number and line number of that error (2-element integer vector).

ERRF – name of the function containing the error indicated by ERRN (text vector).

ERRX – hexadecimal value of the latest I/O error or abnormal condition (4-element text vector).

ΔGRF

ΔGRF is the graphic-services primitive used, for instance, to set scaling and window. See the description in Chapter 11 of Xerox APL graphics capability for details.

ΔTE, ΔCR, and ΔWM

These intrinsic functions have varied use in workspace management and are described in detail in Chapter 12.

ΔTE – performs text editing functions.

ΔCR – permits conversion of user functions to text form and the reverse process.

ΔWM – provides varied information about the user's workspace in the form of APL results.

8. SYSTEM COMMANDS

System commands allow the user to control the mechanical aspects of the APL system, and can be divided into three categories:

1. Workspace Control Commands – commands that affect the state of active and saved workspaces.

2. Inquiry Commands – commands that supply information about the active workspace.

3. Communications Commands – commands that send messages to the computer operator and commands that log the user off the APL system.

System commands always begin with a right parenthesis and can be entered when the system is in execution mode or definition mode. By using the Execute operator (see Chapter 5), system commands can be embedded in APL expressions and can be embedded in a function line. Thus, a system command can be placed under the control of such expressions or functions. Only the first four letters of command names are significant. Name characters after the fourth are ignored. Thus ")CLEA" and ")CLEAVAGE" are both interpreted to be the ")CLEAR" command. Note that a blank must separate the command name and any following parameters; for example,)WIDTH 30 is not the same as)WIDTH30. A number of conventions are used in this chapter to describe the command formats.

1. Uppercase letters and special symbols must be typed exactly as they appear (except that only the first four letters of a command are required, as noted above).

2. Lowercase letters are employed to indicate where in a command to substitute a name or numerical value. The meanings of the notations in lowercase letters are as follows:

account	User account.
fname	Name of a function.
grpname	Name of a group.
list	List of names (of functions, variables, groups).
message	Actual message to computer operator.
n	An integer value.
objname	Name of a function, variable, or group.
password	Assigned to a workspace name to restrict user access to the workspace; can consist of from one to eight characters.
string	Any sequence of characters not including a blank or carriage return. If a string includes more than 80 characters, those past the 80th are ignored. Strings are used for range demarcation in certain commands.
vname	Name of a variable.
wsname	A workspace name; can consist of up to 11 characters (letters, underlined letters, numbers, and Δ and $\underline{\Delta}$), as long as the first character is not a number. If longer names are used, the characters after the first 11 are ignored.

The actual system commands are detailed later in this chapter, but first it is necessary to describe the concept of a workspace in order to understand how certain commands are used.

Workspace Concept

Active Workspace

Each terminal hooked up to the APL system is considered to be active. Associated with each active terminal is a storage area in the central computer known as an active workspace. This active workspace contains the following:

1. All control information entered by the user during the terminal session.

2. The variables, functions, and groups entered for calculations during the terminal session.

3. A state indicator that keeps track of the names of suspended and pendant functions and at what point they were interrupted.

4. Parameters that control several features of the APL system, such as indexing origin, seed for random number generation, line width, and number of significant digits (decimal places) printed. These parameters all assume default values when the user first signs on to the APL system, but he can respecify most of them with certain system commands.

When the user first signs on to the APL system, the active workspace is clear (that is, there is nothing in it except the default values of the parameters mentioned above in item 4. An active workspace can also be cleared with the system command)CLEAR.

Saved Workspace

The user can save – or rather, store – his active workspace for future use (via a SAVE command). A workspace can be saved only in the account in which the user logged on under UTS. Once a workspace has been saved, any user who knows its name, account, and password (if present) can load it as an active workspace (via a LOAD command); copy its variables, functions, and groups into active workspace (via a COPY command); or drop it from his own account (via a DROP command). (It should be noted that a user cannot drop a workspace from another user's account.) In addition, the user can list all of the names of saved workspaces in his or other accounts (via a LIB command).

CONTINUE Workspace

An inadvertent line disconnect or any of the following commands causes the user's active workspace to be saved under the name CONTINUE in his account:

```
)SAVE CONTINUE
)CONTINUE
)CONTINUE HOLD
```

The CONTINUE workspace is automatically loaded as an active workspace the next time the user logs on, unless it was created with a)SAVE CONTINUE command. In general, the CONTINUE workspace can be used almost like any other named workspace. It can be saved, copied, loaded, etc. However, it should only be used for temporarily saving a workspace, since another)CONTINUE command or line disconnect will save the then active workspace over what was previously saved. That is, the previous CONTINUE workspace will be wiped out.

Since the CONTINUE workspace is part of the user's account, it is subject to the granule restrictions imposed by an installation. If the user's account is near that limit, the CONTINUE workspace may not be saved, and the information in the active workspace may be lost (see "User Accounts", next).

User Accounts

Each APL user is assigned an account in which workspaces can be saved. The capacity of this user account depends on the granule restrictions imposed by an installation. If near that limit, the user may not be able to save an additional workspace without dropping something else. When a save is disallowed, the APL system will print a diagnostic, if possible. The user specifies the account when logging on (to CP-V) and when accessing information saved in another user's account. He does not have to specify the account when accessing information in his own account. The names of saved workspaces in an account can be listed with the)LIB command.

Passwords

The user account number and a workspace name offers some protection against other users accessing a workspace. To provide even greater protection, a user can assign a password to a workspace when it is saved with the)SAVE command. The password assigned to a workspace must be specified each time the user references that workspace in a system command. A password may contain up to eight characters without blanks, semicolons, periods, or commas.

Commands

The system commands are detailed below in alphabetical order, and are summarized by category in Table 5.

Table 5. Summary of System Commands

Command	Meaning
Workspace Control Commands	
)CATCH	Removes any current catches (i.e., intercepts of assignments to specified variable names).
)CATCH vname VIA fname	Designates that assignments to vname are to be "caught" (intercepted immediately after the assignment) and that the test function fname, a niladic function with no result, is to be entered. This is a debugging aid.
)CLEAR	Clears active workspace and restores default width, digits, origin, fuzz, random number link, etc.
)COPY wsname	Copies all functions, variables, and groups from saved workspace. Any password must be included, and so must the account if different than the user account.
)COPY wsname objname	Copies named object(s) – function(s), variable(s), group(s) – from saved workspace. Any password must be included, and so must the account if different than the user account. If more than one object is named, the object names are separated by blanks.
)DIGITS	Displays current setting for significant digits for output (i.e., number of decimal places).
)DIGITS n	Specifies significant digits for output (i.e., number of decimal places), where n can range from 1 through 16. Also displays the old value of n.
)DROP wsname	Removes a saved workspace from the user's account. Any password must be included after the workspace name.
)ERASE objname	Removes named object – function, variable, or group – from active workspace. More than one object can be specified, with blanks used as separators.

Table 5. Summary of System Commands (cont.)

Command	Meaning
Workspace Control Commands (cont.)	
)GROUP grpname list	Groups objects and names the group.
)GROUP grpname	Disperses the named group.
)LOAD wsname	Moves a replica of saved workspace into active workspace. If the workspace name is protected with a password, that password must be specified. Also, if the saved workspace is in another account, that account must be specified.
)OBSERVE	Specifies that the next (direct input) statement and any traced function statements executed thereby are to be "observed". This displays a number of observations – showing intermediate results as APL interprets those statements.
)ORIGIN	Displays the current index origin (either 0 or 1).
)ORIGIN n	Sets the index origin, where n can be 0 or 1, and displays the previous index origin.
)PCOPY	Same as)COPY, except that an object is not copied if active workspace contains an object with the same name.
)QCOPY	Same as)COPY, except that the "SAVED" message is suppressed, i.e., quiet copy.
)QLOAD	Same as)LOAD, except that the "SAVED" message is suppressed, i.e., quiet load.
)QPCOPY	Same as)PCOPY, except that the "SAVED" message is suppressed, i.e., quiet protected copy.
)SAVE[t]	Saves active workspace that already has an assigned name (see)WSID).
)SAVE wsname[t]	Saves active workspace under the specified name. To save a workspace and protect it with a password, follow the workspace name with two periods and the password name (i.e.,)SAVE wsname..password).
)SEAL wsname[t]	Saves current workspace as a sealed 'execute-only' workspace with the designated name. All functions in the workspace are locked and the workspace is saved on file with the attributes, READ access, NONE, WRITE access, NONE, EXECUTE access, ALL, under APL.
)WIDTH	Displays the current width of a line of output.
)WIDTH n	Changes the width of a line of output, and displays the old width. The width parameter, n, can range from 30 to 254; however, values above 130 are meaningless for normal operations on a standard APL terminal.
)WSID	Displays the name and account (if different than current user account) of active workspace.
)WSID wsname	Assigns a name to active workspace, or changes the name if one already exists and displays the old name.
)WSID wsname..password	A password may also be designated.

[t]May include an AUTOSTART statement.

Table 5. Summary of System Commands (cont.)

Command	Meaning
Inquiry and Communications Commands	
)CONTINUE	Ends terminal session, and saves active workspace under the name CONTINUE.
)CONTINUE HOLD	Returns control to CP-V TEL subsystem, and saves active workspace under the name CONTINUE.
)FNS	Alphabetically lists all defined function names in active workspace.†
)FNS string	Alphabetically lists the defined function names that match or exceed the string.†
)FNS string string	Alphabetically lists the defined function names that match or lie between the strings.†
)GRP name	Lists objects in named group.
)GRPS	Alphabetically lists all group names in active workspace.†
)GRPS string	Alphabetically lists the group names that match or exceed the string.†
)GRPS string string	Alphabetically lists the group names that match or lie between the strings.†
)LIB	Lists names of saved workspaces in current user's account.
)LIB account	Lists names of saved workspaces in another account.
)OFF	Ends the terminal session and clears active workspace (information is lost).
)OFF HOLD	Returns control to CP-V TEL and clears active workspace (information is lost).
)OPR message	Sends message to computer operator, with reply expected; locks keyboard until the user strikes the ATTN key.
)OPRN message	Sends message to computer operator, with no reply expected.
)SET I/O stream {device ID / file ID / pack ID / tape ID} [;option]...	Allows routing of regular output, input, and/or 'blind' I/O channels to files or various devices, and specification of formatting options for device output. Analogous to the SET command in CP-V TEL.
)SI	Lists the contents of the state indicator — a list of suspended and pendant functions.
)SI CLEAR	Clears the entire state indicator.

†Xerox APL uses the following collating sequence in the process of alphabetizing:

blank or end of name
Δ
Δ
underlined alphabetic letters (A through Z)
alphabetic letters without underlines (A through Z)
digits

Table 5. Summary of System Commands (cont.)

Command	Meaning
Inquiry and Communications Commands (cont.)	
)SI OFF	Prevents an error from suspending an active function containing the erroneous statement.
)SI ON	Restores normal state indicator control. If an error occurs in an active function line, APL suspends the function at that line (assuming sidetracking does not occur, see Appendix A).
)SIV	Lists the contents of the state indicator – a list of suspended and pendant functions and the local variables named by those functions.
)SIV CLEAR	Same as)SI CLEAR.
)SIV OFF	Same as)SI OFF.
)SIV ON	Same as)SI ON.
)SYMBOLS	Displays the current length of the symbol table (that is, the number of names allowed in it) and the number of unused entries.
)SYMBOLS n	Sets the length of the symbol table to no less than the number of names indicated by n (the maximum n allowed is 2001). Also displays old value. This command will be executed only if the current workspace is clear.
)TABS	Displays the current tab settings.
)TABS n	Sets tabs. The n parameter indicates column positions and can be a scalar (for equally spaced tab settings) or a vector of up to 16 numbers in increasing order (for unequally spaced tab settings).
)TERMINAL n)TERMINAL INPUT n)TERMINAL OUTPUT n	Identifies to the APL system the input/output devices being used, where n can be any of the following values: 1 for 2741 terminal (or equivalent) with standard APL typeball. 2 indicates 2741 terminal (or equivalent) with non-APL typeball. 3 for Teletype Model 33 or equivalent. 4 for line printer format output or card reader format input. 13 for typewriter-paired APL/ASCII terminals (e.g., Tektronix 4013). 14 for bit-paired APL/ASCII terminals.

Table 5. Summary of System Commands (cont.)

Command	Meaning
)VARS	Alphabetically lists all global variable names in active workspace.†
)VARS string	Alphabetically lists the global variable names that match or exceed the string.†
)VARS string string	Alphabetically lists the global variable names that match or lie between the strings.†

†Xerox APL uses the following collating sequence in the process of alphabetizing:

blank or end of name
Δ
Δ
underlined alphabetic letters (A through Z)
alphabetic letters without underlines (A through Z)
digits

)CATCH Debugging Aid for Intercepting Assignments

The)CATCH command is primarily a debugging tool. It permits the programmer to "catch" (or intercept) each assignment to a specified variable name, immediately after that assignment has been executed. The format of the command invoking a catch is

)CATCH vname VIA fname

where vname is the name of the variable (which may be local or global) and fname is the name of a "test" function. The test function is defined by the user according to his debugging needs. The only restriction is that this function must be niladic with no result. This restriction isolates the test function from the statement or statements assigning values to the specified variable. If the fname is undefined or does not indicate a niladic, no-result function, no error message occurs – the catch is simply ignored (this can be used to advantage – see Example 3).

Suppose the programmer has invoked the following catch,

)CATCH V1 VIA F1

then all assignments to the name V1 cause test function F1 to be called. This includes indexed assignments. F1 is called regardless of whether V1 is a local or global variable. The programmer can modify this catch whenever he wishes to enter a different test function. For example,

)CATCH V1 VIA FTWO

After the above modification, assignments to V1 cause test function FTWO to be called (instead of F1).

The programmer can also invoke a second catch. For instance,

)CATCH VAR2 VIA FOTHER

He could have both catches enter the same test function as in the next example.

)CATCH VAR2 VIA FSAME
)CATCH V1 VIA FSAME

He cannot, however, invoke a third catch; this attempt produces a "BAD COMMAND" error.

The programmer can remove any current catches by issuing the command.

)CATCH

Following this removal, he is free to invoke one or two new catches.

Catches are not saved when a workspace is saved, so loading a workspace does not automatically reinstall catches. The)CLEAR command also removes any current catches.

The simplicity of the catch command may obscure its power as a debugging aid. This power is brought to bear by the test function. A few hypothetical examples are given below to suggest the potential of catch capability.

Example 1. Using a catch to display values assigned to vname.

```
      )CATCH X VIA SHOWX
      ∇SHOWX
[1]   'X IS ';X ∇
```

As long as this catch is in effect, every assignment to X will cause the new value of X to be displayed.

Example 2. Using a catch to stop execution when a particular value is assigned to vname.

(Assume that X is a scalar and 77 is the value of interest.)

```
      )CATCH X VIA CHECK
      ∇CHECK
[1]   →0×⍳X≠77
[2]   SΔCHECK←STOP
[3]   STOP: ∇
```

As long as this catch is in effect, each assignment to X will be tested at line 1 of the CHECK function. If X is not 77, line 1 resumes execution (branching to line 0, causing CHECK to exit). When X receives the value 77, line 2 is executed. Line 2 sets the stop-vector for the CHECK function so that when the line labeled STOP is reached, CHECK will suspend execution.

Example 3. Using a catch to change the value of vname.

(Note that this does not affect the value used by the statement making an assignment to vname; the catch is isolated.)

```
      )CATCH X VIA CHANGE
      ∇CHANGE ;CHANGE
[1]   X←0 ∇
```

As long as this catch is in effect, each assignment to X that occurs "outside" the CHANGE function will cause X to be set to 0. The assignment at line 1 of the CHANGE function will not be "caught" because calling the function temporarily declares the name CHANGE to be a local variable (shadowing the definition of CHANGE as a test function); see the function header line.

Suppose the following statement is executed with the above catch in effect.

```
X + 100 + X←55
```

The answer of 155 results in the following way.

1. The value 55 is obtained.
2. X is assigned the value 55.
3. The catch occurs.
4. X is set to 0 by the CHANGE function.
5. Execution of the original statement resumes, undisturbed (so far, at least) by the catch. This means that the value 55 is the right argument of the next addition.
6. 100 plus that argument yields 155.

7. This value, 155, becomes the right argument of the next addition.

8. The value of X is obtained; it is now 0.

9. 0 plus 155 yields the final result.

)CLEAR Clearing Workspace

This command deletes all groups, functions, variables, and the state indicator from active workspace. Furthermore, it resets:

- Random number link.
- Fuzz setting.
- Origin (1).
- Width (120 for terminals equivalent to an IBM 2741; otherwise, 72).
- Significant digits (10).
- Symbol table size (261).
- Workspace identification (CLEAR WS).
- State indicator control (ON); see also the)SI or)SIV command description.
- Current catches (none), see the)CATCH command description.
- Error number (0); see "Sidetracking on Errors and Breaks" in Appendix A.
- Error location (line number 0 and function name an empty text vector); see also Appendix A.

It does not change the latest tab setting that was specified by the user during the current APL session. (When an APL session begins, an initial tab setting of zero is assumed.) The form of this command is

```
)CLEAR
```

The APL system responds to this command by printing the message CLEAR WS.

Example

```
      )CLEAR
CLEAR WS
```

)CONTINUE and)CONTINUE HOLD Signing Off and Saving Active Workspace in CONTINUE

These commands are similar to the)OFF and)OFF HOLD commands described later in this chapter, except that the active workspace is saved in workspace CONTINUE instead of being deleted. This workspace is automatically loaded the next time the user logs on. The active workspace is also automatically saved in workspace CONTINUE if the phone is accidentally disconnected of if a)SAVE CONTINUE command is given.

The)CONTINUE command saves the user's active workspace in a workspace named CONTINUE, and ends the terminal session. Its form is simply

```
)CONTINUE
```

A successful)CONTINUE command will produce a save report (time and date saved) and the UTS log-off messages. If not enough room remains in the user's account to save the workspace, the system prints an error

message. If this happens, the user will have to delete some workspaces or other files before any APL workspaces may be saved.

The)CONTINUE HOLD command saves the user's active workspace in a workspace named CONTINUE and returns control to the CP-V Terminal Executive Language (TEL). The form of this command is

)CONTINUE HOLD

Note: If a user's workspace is passworded, the password is retained in the saved file. In this case, CONTINUE is not automatically loaded the next time the user logs on.

Caution: If an account already contains a passworded CONTINUE workspace, any subsequent CONTINUE will fail until the passworded version is deleted. Sealed workspaces cannot be saved with CONTINUE.

A successful save results in a save report and in CP-V printing a ◦ prompt character. (This is the same as the ! prompt character shown in CP-V documentation; it's just that the APL typeball prints a ◦ instead of a !.) The user is then free to enter any other CP-V commands at the TEL level. If the save is unsuccessful (because not enough file space is left in the user's account), an error message will be printed and the user will have to delete some saved workspaces or other files before saving any APL workspaces.

If either form of the)CONTINUE command is given during function definition mode, the currently open function is closed by APL. When the CONTINUE workspace is loaded later, APL automatically reopens the function and prompts the user to continue function definition.

The CONTINUE workspace can be used almost like any other named workspace. It can be saved, copied, loaded, etc. However, it should only be used for temporarily saving a workspace since any previous CONTINUE workspace will be wiped out by a new CONTINUE workspace save.

Examples

```
      )CONTINUE
CONTINUE    SAVED  10:57 JUN 30,'72

CPU = .0193 CON= :06 INT = 14 CHG = 0
```

Saves active workspace in CONTINUE and ends terminal session after printing save report and CP-V log-off messages.

```
      )CONTINUE HOLD
CONTINUE    SAVED  10:56 JUN 30,'72

◦
```

Saves active workspace in CONTINUE and returns control to TEL after printing save report. TEL prompts for commands with the ◦ character.

)COPY Copying Information from Saved Workspace to Active Workspace

The copy command enables the user to copy information from a saved workspace to the active workspace. The information can consist of one, several, or all of the functions, global variables, and groups in the saved workspace. This command may take any of the following forms:

```
)COPY wsname
)COPY wsname objname
)COPY wsname.account
)COPY wsname.account objname
)COPY wsname..password
)COPY wsname..password objname
)COPY wsname.account.password
)COPY wsname.account.password objname
```

where

wsname is the name of the saved workspace.

objname is a variable name, function name, or group name. More than one object can also be specified, with blanks used as separators between the objects.

account is the user account under which the workspace was saved. This option is used to access a workspace in another account; it is not necessary when accessing a workspace in the current user's account.

password is a user-assigned password. If a password was used when the workspace was saved, that password must be used to access the workspace.

)COPY cannot be used to access a sealed workspace.

Note that if a workspace is saved with a password, that password must be included in the copy command. Also, if a workspace is being copied from another user's account, the account must be specified in the copy command.

When all of a saved workspace is copied, only functions, global variables, and groups are copied. If copied functions had sidetracks (see Appendix A), stop vectors, or trace vectors set, then these settings also apply to the active workspace. All referents of a copied group are themselves copied into the active workspace. For instance, suppose group G1 is copied, where G1 contains A, B, and G2 with G2 being another group containing X, Y, and Z. Then the following are copied into active workspace: G1, G2, A, B, X, Y, and Z. The digits, line width, origin, random seed, and state indicator are not copied.

A copy attempt may fail if there is not enough room in the active workspace or if there are too many new symbols. After issuing the appropriate error message, APL restores the original active workspace (as it existed prior to the copy command). An infrequent error message of this type is "TOO BIG TO LOAD". This happens when copying from a different account in which two conditions are met. First, the workspace being copied from is large (so large that it cannot even be loaded by the current user). Second, the referenced account was allocated more computer memory than is available to the current user's account (memory allocations are specified by the installation manager). This difficulty can be circumvented with the cooperation of the owner of the larger account. He can copy portions of the large workspace, forming one or more smaller workspaces. After this cooperative activity, the current user can copy required objects out of those smaller workspaces.

Note: Due to extensive input/output processing, a copy command may take a long time to complete. This is particularly true if either the active or copied workspace is relatively large.

If a copy command is issued during function definition mode, the currently open function is temporarily closed. When the copy is completed, the function is automatically reopened. The copy may have replaced the current function. If the copy command names functions that are pendant in the active workspace, they are not replaced. Suspended functions may be replaced and may cause an SI DAMAGE error message to be issued. Use of the)PCOPY command precludes this possibility.

The)PCOPY command, the protected copy command, functions the same as the)COPY command except that an object is not copied if active workspace already contains an object with the same name.

A group of objects can be copied even though the group definition is not. This happens if the group name matches a current pendant function or if the name matches any object in the case of)PCOPY. Alternatively, a group definition may be copied but some of its objects not copied.

Examples

```
      )COPY HENRY.ACCT33.SECRET
HENRY       SAVED  10:56 JUN 30,'72
```

Copies all of a saved workspace named HENRY, which was saved with the password SECRET in another user's account (account ACCT33); and produces a save report giving the time and date HENRY was saved.

```
      )COPY HENRY
HENRY       SAVED  10:56 JUN 30,'72
```

Copies an entire saved workspace named HENRY from the user's own account and produces a save report giving the time and date HENRY was saved.

```
      )COPY HENRY COS MAT
HENRY       SAVED  10:56 JUN 30,'72
```

Copies a function named COS and a group named MAT from a saved workspace named HENRY in the user's own account; and produces a save report giving the time and date HENRY was saved.

```
      )COPY HENRY..SECRET
HENRY       SAVED  10:56 JUN 30,'72
```

Copies all of a saved workspace named HENRY, which was saved with the password SECRET in the current user's account; and produces a save report giving the time and date HENRY was saved.

)DIGITS Specifying Number of Significant Digits

The digits command allows the user to set the number of significant digits in noninteger numerical output to some number between 1 and 16 inclusive. Without this command the APL system will display a maximum of 10 significant noninteger digits. Only displayed output is affected by this command; internal calculations are not affected. The command has two forms

```
)DIGITS
)DIGITS n
```

where n indicates the number of significant noninteger digits to output and can be any number from 1 through 16. The first command form,)DIGITS, simply causes the APL system to print the current setting for singificant digits. The second command form,)DIGITS n, changes the significant digits to be output and causes the APL system to print the previous setting for singificant digits. It should be noted that internal precision of the computer provides a maximum of 16 digits for some simple operations, 15 or less for more complex operations. if)DIGITS 16 is used, at least the last digit is subject to error; for this reason 16 is not a recommended setting.

Examples

```
      )DIGITS
IS  10
```
User requests significant digits be displayed, and APL responds with current setting.

```
      )DIGITS 15
WAS 10
```
User sets significant digits to 15, and APL responds with previous setting.

```
      4÷9
0.4444444444444
```
User requests calculation, and APL responds, showing 15 significant digits.

```
      )DIGITS 5
WAS 15
```
User sets significant digits to 5, and APL responds with previous setting.

```
      )4÷9
0.44444
```
User requests calculation, and APL responds showing 5 significant digits.

The number of significant digits to be output can also be changed with the DIGITS function described in Chapter 7, "Defined Functions".

)DROP Dropping a Saved Workspace

This command allows the user to remove a saved workspace from his account. It has two forms – one for removing unprotected workspaces and another for removing workspaces protected with a password. These two command forms are

```
)DROP wsname
)DROP wsname..password
```

where

wsname is the workspace name to be dropped.

password is the password that was saved along with the workspace.

If the workspace is not found or if a proper password is not provided, APL returns the message WS NOT FOUND. If the workspace is deleted, APL returns a message identifying the workspace and the time it was last saved.

Examples

```
      )DROP PAYROLL..SAVINGS
PAYROLL SAVED  10:37 JAN 30,'75
```
Removes workspace PAYROLL, saved with the password SAVINGS, from the user's account.

```
      )DROP DANEEN
WS NOT FOUND
```
Attempts unsuccessfully to drop workspace named DANEEN. Either the workspace does not exist or a needed password has not been provided.

)DROP may not be used to delete files other than APL workspaces. If an attempt is made to delete an existing file which is not an APL workspace, BAD FILE REF is reported.

)ERASE Removing Objects From Active Workspace

The erase command allows the user to delete one or more named objects – functions, global variables, groups – from the active workspace. This command has the form

ERASE objname

where objname is the name of the object – function, variable, or group – to be erased. Note that it is the object that is erased; its name remains in the symbol table. More than one object name can also be specified in the erase command, with blanks used as separators between the names. Including the names of global variables and functions in an erase command causes the actual variables and functions to be erased. If a group is named in the erase command, that group definition is erased along with any functions or variables named in the group; however, if it also names other groups, they are dispersed (group definitions are deleted, but not their referents). For example, suppose G1 is a group containing variable names A and B plus group name G2. Further suppose that G2 contains variable names X and Y. Then erasing G1 deletes G1, A, B, and G2; but variables X and Y are retained. Pendant functions cannot be erased. During function definition, if the function being defined is erased, its current definition is abandoned (equivalent to closing the function and then erasing it).

Examples

)ERASE MATHFUNCTIONS	Erases a group named MATHFUNCTIONS and the functions and variables it names. It disperses any group named within the group MATHFUNCTIONS.
)ERASE PAYROUTINE GROSS INS	Erases a function named PAYROUTINE and two variables named GROSS and INS.

Note:)ERASE will not remove local variables.

)FNS Listing Function Names

This command allows the user to list alphabetically the names of functions in the active workspace. Three forms are acceptable:

```
)FNS
)FNS string
)FNS string1  string2
```

The first form displays all function names. The second form displays all function names that match or exceed the "string"[†] in an alphabetic ordering. The third form displays all function names that match or exceed "string1" and are also less than or match "string2". Alphabetic ordering is illustrated in the examples below. Note particularly the first)FNS command since it indicates where each name character lies in alphabetic order.

Examples

```
      )FNS
F   FΔ  FΔ  FE  FF  FX  FXY F0  F1  S
      )FNS FF
FF  FX  FXY F0  F1  S
      )FNS F FF
F   FΔ  FΔ  FE  FF
      )FNS FFF FX
FX
      )FNS FXY FXY
FXY
      )FNS Δ S4
F   FΔ  FΔ  FE  FF  FX  FXY F0  F1  S
```

[†]Where "string" is any sequence of characters not including a blank or carriage return. If a string includes more than 80 characters, those past the 80th are ignored. Strings are used for range demarcation.

)GROUP Defining a Group

This command allows the user to reference a group of names – variables, functions, other groups, or just names – collectively. Group definitions can be used in erase and copy commands to facilitate erasing and copying a group of related objects. The command to define a group is

)GROUP grpname list

where

grpname is the name of the group. A group name follows the same formation rules as a variable or function name, except that a group name cannot be the same as a function or global variable in the workspace.

list is a list of the names that make up the group.

Names can be added to an already existing group by merely repeating the group name in any of the command forms

)GROUP grpname grpname list
)GROUP grpname list grpname
)GROUP grpname list grpname list

A group can be dispersed with the command form

)GROUP grpname

This form disperses the group; that is, removes the name references previously associated with grpname. The names and their references are not themselves erased – only the group identity is lost. An erase command could have been used to remove the group, but the erase command removes the group and deletes the group referents (the actual functions or variables) from active workspace.

Examples

```
)GROUP PROB1 COS TAN A B
```
Defines a group named PROB1, consisting of the variable and function names COS, TAN, A, and B.

```
)GROUP PROB1 PROB1 D ST
```
Adds the variables D and ST to the already existing group named PROB1.

```
)GROUP PROB1
```
Deletes the group named PROB1 from the active workspace. The referents of PROB1 are not deleted.

Note that the last example disassociates the function and variable names from the group, but does not delete actual functions and variables from the active workspace. The)GRPS command can be used to verify that the group named PROB1 has been deleted, and the)FNS and)VARS command can be used to verify that the named functions and variables still remain in the active workspace.

```
      )GRPS
      )FNS
COS TAN
      )VARS
A   B   D   ST
```

Also see the)GRP and)GRPS commands below, which list the members of a group and the names of groups in active workspace respectively.

)GRP Listing the Members of a Group

This command allows the user to list object names making up a group. It has the form

```
)GRP name
```

where name is the group name. The APL system responds by printing the names of the objects in the named group.

Examples

```
      )GROUP G1 A B C
      )GRP G1
A   B   C
      )GROUP G1 G1 D
      )GRP G1
A   B   C   D
      )GROUP G1 X Y Z G1 G2
      )GRP G1
X   Y   Z   A   B   C   D   G2
      )GROUP G2 X A F1
      )GRP G2
X   A   F1
      )GRPS
G1  G2
      )GROUP G1
      )GRPS
G2
      )GRP G1
      )GRP G2
X   A   F1
```

)GRPS Listing Group Names

This command allows the user to list alphabetically the names of groups in the active workspace. Three forms are acceptable:

```
)GRPS
)GRPS string
)GRPS string1 string2
```

The first form displays all group names. The second form displays all group names that match or exceed the "string"[t] in alphabetic ordering. The third form displays all group names that match or exceed "string1" and are also less than or match "string2". Alphabetic ordering is illustrated in the examples below. Note particularly the first)GRPS command since it indicates where each name character lies in alphabetic order.

Examples

```
      )GRPS
G   GΔ  GΔ  GG  GG  GH  GHI G0  G1  H
      )GRPS GG
GG  GH  GHI G0  G1  H
      )GRPS G GG
G   GΔ  GΔ  GG  GG
      )GRPS GGG GH
GH
      )GRPS GHI GHI
GHI
      )GRPS Δ H8
G   GΔ  GΔ  GG  GG  GH  GHI G0  G1  H
```

[t] Where "string" is any sequence of characters not including a blank or carriage return. If a string includes more than 80 characters, those past the 80th are ignored. Strings are used for range demarcation.

)LIB Listing Names of Saved Workspace

This command allows the user to list the names of workspaces saved in his own account or another user's account. If a password was saved with a workspace, the workspace name is listed, but not the password. To list names of the current user's saved workspace, use the command

```
)LIB
```

To list names of saved workspaces in another account, use a command of the form

```
)LIB account
```

)LIB will only list workspaces saved by APL release A01 or later. If workspaces are subsequently copied by non-APL processors such as PCL, they may not be listed by)LIB.

Examples

```
      )LIB
APLQUIZ
APLSIDR
PROB1
PROB2
```

Lists names of saved workspaces in the current user's account.

```
      )LIB REI07207
EDITFILE
FACTOR
PAYROLL
```

Lists names of saved workspaces in another account (account REI07207).

)LOAD Retrieving a Saved Workspace

This command allows the user to load a replica of a saved workspace into his active workspace. The saved workspace may be retrieved from a user's own account or another account. This command may take any of the following forms:

```
)LOAD wsname
)LOAD wsname.account
)LOAD wsname..password
)LOAD wsname.account.password
```

where

wsname is the name of the saved workspace.

account is the account under which the workspace was saved. This option is necessary only when accessing a workspace saved in another account; it is not necessary when accessing a saved workspace in the current user's account.

password is a user-assigned password. If a password was saved with a workspace, that password must also be used to access the workspace.

Note that if a saved workspace is being retrieved from another account, the account must be specified in the load command. Also if the workspace was saved with a password, that password must be included in the load command. In response to a successful load, the APL system prints a message giving the time and day that the workspace was saved. If the workspace is not found or if a proper password has not been used, APL prints the message WS NOT FOUND.

If a workspace has been saved using the AUTOSTART option (Pages 130-131), execution will start automatically after the load.

If a workspace was saved during function definition mode, the)LOAD command causes APL to automatically reopen that function and prompt the user to continue function definition or editing. (He may choose to close the function immediately.)

Examples

```
      )LOAD HENRY
HENRY       SAVED  15:28 JUL 01,'72
```

Loads workspace HENRY into active workspace and prints a save report. Workspace HENRY was previously saved in the current user's account.

```
      )LOAD HENRY..SECRET
HENRY       SAVED  15:28 JUN 01,'72
```

Loads workspace HENRY into active workspace and prints a save report. Workspace HENRY was previously saved with password SECRET in the current user's account.

```
      )LOAD HARRY.ACCT33.SECRET
HARRY       SAVED  15:28 JUN 01,'72
```

Loads workspace HARRY into active workspace and prints a save report. Workspace HARRY was previously saved with password SECRET in account ACCT33.

)OBSERVE Observing Intermediate Results as APL Interprets a Statement or Traced Function Statements

The)OBSERVE command allows the user to observe intermediate results developed by APL as it interprets a statement. This could be thought of as a "super-trace" capability. The command format is

)OBSERVE

Following an)OBSERVE command, the succeeding statement is observed along with any traced function lines that are encountered. Subsequent direct statements are not observed unless the user precedes each of them by a new)OBSERVE command. Thus, an)OBSERVE command is short-lived – applicable to only one direct statement. By setting trace-vectors for functions to be encountered during an execution, however, the user can observe arbitrarily selected statements until he issues another direct input line.

While an)OBSERVE command is in effect, Xerox APL displays a series of observations. An observation consists of displaying: the current line being executed, a marker (error caret) beneath some character in that line, and the value resulting at that point in execution (empty results, as usual, cause no value to be displayed). The observation marker often marks the leftmost point reached, so far, during APL interpretation of the line; however, when an operator yields its result, the marker is placed below the operator for clarity. (The only exception is the execute-operator, in which case the "leftmost" rule applies.)

For "observed" lines, observations occur for:

- Each operator result
- Each function result
- Arguments that have not already been observed on this line
- Indexed arguments

Observations are not made for assignments since the assigned value has already been observed prior to the assignment. Observations are also not made for the full variable when it is used as an indexed variable; this eliminates lengthy displays in cases such as the following sample)OBSERVE command.

```
      A←1000⍴'GORP'
      )OBSERVE
      B←A[5]                    This is the observed line.
      B←A[5]
          ^                     Observation of the argument 5.
5
      B←A[5]
        ^                       Observation of the indexed argument A[5].
G
```

Note in the above sample that A was not displayed. This is fortunate since its display would produce 1000 characters, most of which contribute nothing to the observed statement.

The)OBSERVE command has three valuable usages: for debugging, for learning how a calculation is performed, and for developing better APL functions. Its value in debugging is obvious. Suppose a complicated APL statement produces a LENGTH ERR. By using the)OBSERVE command and reissuing that statement, the programmer can view development of values leading up to the error and readily see what caused the problem.

Usage of the)OBSERVE command as a learning tool can be a tremendous timesaver. When presented with a new APL statement or function, the user can spend a great deal of time analyzing how it accomplishes its result. By observing a sample run, the interpretation path and values can be readily inspected, simplifying analysis greatly. The reader might apply this process to the following function.

```
        ∇PRIMESUPTO N;R;I;J
[1]     (∧⌿(0≠(1+R)∘.|J)∨((R←⍳⌊N*0.5)∘.=I))/J←1+I←⍳N-1∇
```

This function produces the prime numbers existing up to the positive integer specified as its argument. To observe this function in a sample case, the user might proceed as follows:

T∆PRIMESUPTO←1	Set to trace line 1 of the function.
)OBSERVE	Request observations.
PRIMESUPTO 15	Call the function.
•	About 30 observations are made; then the)OBSERVE
•	command "disappears".
•	

There are at least two ways in which the)OBSERVE command can be used to develop better APL functions. First, redundant calculations are made obvious and the programmer can then eliminate such redundancies. The following function is an inefficient version of the PRIMESUPTO function. The reader might try observing the function to discover how apparent such redundancies become under the)OBSERVE command.

```
        ∇PRIMESUPTO  N
[1]     (∧⌿(0≠(1+⍳⌊N*0.5)∘.|(1+⍳N-1))∨(((⍳⌊N*0.5)∘.=(⍳N-1)))/1+⍳N-1∇
```

The PRIMESUPTOO function takes considerably more execution time (and produces more observations) than the PRIMESUPTO function shown previously.

The second way in which)OBSERVE is useful for developing better APL functions is not obvious. It depends upon the creativity and imagination of the user. By viewing the manner in which a calculation is carried out, the creative user may recognize patterns that can be more easily produced by other calculations. In other words, observations can suggest alternate approaches to solving a given problem.

One final note about the)OBSERVE command should be presented. Suppose the user suspends execution during an observed run by hitting the break key, for instance. This removes the)OBSERVE command. Subsequent execution will not be observed unless the user issues a fresh)OBSERVE. As stated earlier, this command is short-lived. At times its short life can be inconvenient, but considering the voluminous output possible with the)OBSERVE command this is more often a convenience.

)OFF and **)OFF HOLD** Signing Off

These commands provide two ways for the user to sign off the APL system. The)OFF command deletes the active workspace and logs off both A:L and CP-V (producing the CP-V log-off messages). Its form is simply

)OFF

The)OFF HOLD command also deletes the active workspace, but logs the user off APL and returns control to the Terminal Executive Language (TEL) subsystem. The form of this command is simply

)OFF HOLD

In response to the)OFF HOLD command, the TEL subsystem prints a ∘ prompt character to signal that it has control. (This is the same as the ! prompt character shown in CP-V documentation; it's just that an APL typeball prints a ∘ instead of a !.) The user is then free to enter any other TEL commands.

Note: If APL has been called from a load module, not directly from TEL,)OFF HOLD returns to the calling load module.

Examples

```
      )OFF
CPU = .0095 CON= :01 INT = 4 CHG = 0
```

Logs the user and produces the CP-V log-off messages. (See Figure 1 in Chapter 2 for a description of the log-off messages.)

```
      )OFF HOLD
```

Ends APL system communication and returns control to TEL.

∘

)OPR and **)OPRN** Communicating With Computer Center Operator

These commands allow the user to communicate with the operator in the computer center. The)OPR command allows the user to send messages to the operator and request a reply; it has the form

)OPR message

where message is the actual message to the operator; it cannot exceed 120 characters. The APL system responds by printing the word SENT and by locking the keyboard until the user depresses the ATTN key. Note that the operator's console will not include special APL characters, so messages should be limited to ordinary alphanumeric characters.

The)OPRN command is similar to the)OPR command except that no reply is expected from the computer operator. This command has the form

)OPRN message

where message is the same as described above. The APL system responds to this command with message SENT and then unlocks the keyboard.

Examples

```
      )OPR  CP-V UP SUNDAY?
SENT

YES,FOR AWHILE∘
```

Illustrates sending message to operator and receiving reply.

```
      )OPRN  TRIAL MESSAGE.  DON'T REPLY.
SENT
```

Illustrates sending message to operator, with no reply expected.

)ORIGIN Setting Index Origin

This command allows the user to set or display the origin for indexing on arrays and for operators related to indexing. There are two origins available — 0 and 1. The operators affected are index of and index generator (⍳), indexing ([ ]), grade up (⍋), grade down (⍒), and random (?). To set the origin to 1 or 0, the command form is

)ORIGIN n

where n is either 0 or 1. This command causes the APL system to reset the index origin and to print a message indicating the previous index origin. If the user does not supply parameter n when issuing this command, the current index origin will be displayed. Note that the)ORIGIN command affects the active workspace and is saved along with a workspace. The index origin can also be changed with the ORIGIN function described in Chapter 7, "Defined Functions".

Examples

```
      )ORIGIN
IS  1
      )ORIGIN 0
WAS 1
      )ORIGIN 1
WAS 0
```

)PCOPY Copying Information from Saved Workspace to Active Workspace

This command, the protected copy command, functions the same as the copy command except that an object is not copied if the active workspace already contains an object with the same name. See the)COPY writeup earlier in this chapter.

)QLOAD,)QCOPY, and)QPCOPY Quiet Load and Copy Commands

)QLOAD,)QCOPY, and)QPCOPY are slight variants of)LOAD,)COPY, and)PCOPY. The "Q" stands for quiet – the "SAVED" message normally shown at the conclusion of a load or copy is suppressed on a quiet load or copy. No other message (i.e., error diagnostic) is suppressed by the quiet commands.

Certain APL application programs benefit from the quiet commands – programs that use execute-operations to load or copy without user intervention. Since the user is unaware that such load or copy commands are executed, he would be puzzled by "SAVED" messages.

The quiet commands should be inserted in programs only after the application is well checked out. In the event of an error subsequent to a quiet load or copy, it may be difficult to isolate the problem for lack of knowledge about the workspace environment.

)SAVE Saving Active Workspace

This command saves a copy of the active workspace in the current user's account. It does not allow the user to save the active workspace in another account. This command may take any of the following forms:

```
)SAVE
)SAVE wsname
)SAVE wsname..password
)SAVE ;expression
)SAVE wsname ; expression
)SAVE wsname..password ; expression
```

The first form saves a workspace that already has a name; that is, one that was named with the)WSID command or that was previously saved with a name and then loaded into the active workspace.

The second form saves a workspace and assigns a name to it (where wsname is the name of the workspace). Like other APL names, the workspace name can consist of one or a combination of letters (A to Z, or A to Z), numbers, and Δ or Δ, with the restriction that the first character cannot be a number or the combination TΔ or SΔ. Unlike other APL names, a workspace name is limited to 11 characters; more characters can be used but the APL system will ignore them. It is strongly recommended that only letters and digits be used in workspace names. The other characters can lead to confusion if those names are presented to subsystems of UTS other than APL.

The third form saves a workspace and assigns a name and a password to it (where wsname is as described above). A word of caution is necessary about using passwords in the save command. If a saved workspace already exists with a given name and password, specifying the same name with a new password in the save command will not change the password. Instead it results in the error message BAD FILE REF. The previous passworded workspace must be deleted before a new version can be saved. To delete the prior workspace, use)DROP with the name and password.

A sealed workspace may not be saved. If the current workspace was loaded as a sealed workspace or the)SEAL command has been issued, the workspace can no longer be saved by)SAVE or)CONTINUE. Attempts to do so will result in BAD FILE REF error.

AUTOSTART. The fourth, fifth, and sixth forms are similar to the first three except that the expression to the right of the semicolon is saved with the workspace, and execution of that expression occurs automatically each time the workspace is loaded. The expression must fit on the same line as the)SAVE command and may be any executable APL expression. Typically it will be a call to a user-defined function which initiates processing and requests user interaction. The 'autostart' form of)SAVE may be issued only in direct input mode (not in evaluated input, function definition, or execute mode).

When a workspace is successfully saved, the APL system prints a save report giving the name of the workspace and the time and date of the save. This is printed on the line following the save command. The)SAVE command also

updates the current workspace identification, i.e., WSID. The name of the saved workspace along with its password (if any) becomes the WSID for the active workspace. If the workspace cannot be saved – because it exceeds the available space in the user account – the system prints an error message. In this case the user will have to delete some workspaces or other files from his account before he can save any APL workspaces.

If a save command is issued during function definition mode, the currently open function is temporarily closed. The saved workspace carries an indication that a function should be reopened on)LOAD. After the save command, APL reopens the function and prompts the user to continue function definition or editing.

Examples

```
      )SAVE
CONTINUE    SAVED  15:28 JUL 01,'72
```

Saves copy of an active workspace that already has a workspace name (CONTINUE), and produces a save report.

```
      )SAVE KAWA
KAWA        SAVED  15:28 JUL 01,'72
```

Saves copy of active workspace with the name KAWA, and produces a save report.

```
      )SAVE SANDY..PAT
SANDY       SAVED  15:28 JUL 01,'72
```

Saves copy of active workspace with name SANDY and password PAT, and produces a save report.

```
      )SAVE FILEOPS; FIODESCRIBE
```

FIODESCRIBE, in this case, may be a function which identifies the nature of the workspace and permits the user to request detailed descriptions of file I/O functions. When FILEOPS is loaded, FIODESCRIBE will be called automatically.

)SEAL Saving a 'Sealed' Execute-Only Workspace

This command may take any of the following forms:

```
)SEAL wsname
)SEAL wsname..password
)SEAL wsname; expression
)SEAL wsname..password; expression
```

SEAL first locks all functions in the current workspace and identifies the workspace as sealed. The workspace is then saved with the attributes: READ access, NONE, WRITE access, NONE, EXECUTE access, ALL, via APL only. The saved workspace thus may not be accessed by any processor other than the current official version of APL. Any APL user may execute the workspace, but may not display user functions or resave the workspace.

A candidate for a sealed workspace should be a thoroughly tested, self-sufficient, application package in APL. The originator should maintain a backup unsealed version, passworded, to allow for unforeseen error corrections or other changes.

A sealed workspace is protected from any form of inspection which will divulge the contents of functions in that workspace. Displays of other workspace information, such as names of variables, are also prohibited. Normal operation can only be resumed after the sealed information is removed by a command such as)CLEAR or)LOAD.

If the save is not successful, BAD FILE REF, FILE SPACE TOO LOW, or other relevant error messages may be issued. The active workspace will not be sealed in this case.

)SET Changing Assignments of Input/Output Streams

The)SET command applies to four input/output streams as follows:

- INPUT – Source input for APL. Normally assigned to user's console for on-line users; card reader for off-line users.
- OUTPUT – Normal terminal or printer output stream. Normally assigned to user's console for on-line users; line printer for batch users.
- [1] or [2] 'blind input/output streams (Appendix B).

Syntax of)SET

The following options are available:

```
                 { UC                  }
                 { logical device      }
                 { DC/fid              }
)SET INPUT       { DP[packid]/fid      } [;dopt[;dopt]...]
                 { 9T[tapeid][/fid]    }
                 { BT[tapeid][/fid]    }
                 { MT[tapeid][/fid]    }

                 { UC                  }
                 { logical device      }
                 { LP                  }
)SET OUTPUT      { DC/fid              }
                 { DP[packid]/fid      } [;dopt[;dopt]...]
                 { 9T[tapeid][/fid]    }
                 { BT[tapeid][/fid]    }
                 { MT[tapeid][/fid]    }

                 BCD
                 BIN
                 NODRC
                 DRC
                 { IN    }
)SET {[1]}       { INOUT }
     {[2]}       { OUTIN }
                 { OUT   }
                 { UC                  }
                 { logical device      }
                 { LP                  }
                 { DC/fid              }
                 { DP[packid]/fid      }  [;dopt[;dopt]...] [;SIZE=N]
                 { 9T[tapeid][/fid]    }
                 { BT[tapeid][/fid]    }
                 { MT[tapeid][/fid]    }
```

where

UC is users console.

DC is disk.

DP is disk pack.

9T is 9 track tape, 800 BPI.

BT is 9 track tape, 1600 BPI.

MT is 9 track tape, system default density.

If 9T, BT, or MT is not followed by Tapeid and input mode is selected, an I/O error will result. If output mode is selected without Tapeid, a scratch tape is generated.

Logical Devices are installation-dependent name designations for devices such as the Remote Batch Terminal, card reader, etc.

Packid is private pack serial no., of form # serial no.

Tapeid if not followed by /fid, refers to external serial number of a free-form tape. In this case the tape is considered a device.

Tapeid/fid specifies a labeled tape with SN equal to the designated number.

Tapeid has the form # serial no.

Serial Numbers are 1 to 4 alphanumeric characters.

Fid is the standard CP-V file identification format.

'dopt' includes

COUNT = C	(C = 1 to 140)	
DATA = C	(C = 1 to 144)	
LINES = N	(N = 1 to 32,767)	If LP (N = 1 to 255) If UC
SPACE = N	(N = 1 to 255)	
VFC		
NOVFC		

name [. account | . account . password | .. password]

where

SIZE = N sets the byte count for reads through [1] or [2] . N = 1 to 32,767. Default is 512.

COUNT = C turns on page control and initializes the page count, which will appear at column C at the top of each page.

DATA = C sets the leftmost column at which output will appear (line printer only).

LINES = N sets the number of lines per page. If a large value is used, this means no header output.

SPACE = N sets line spacing if mode is NOVFC.

VFC sets Vertical Format Control On.

NOVFC sets Vertical Format Control Off.

BCD BCD device read-write mode.

BIN Binary device read-write mode.

NODRC Monitor special formatting On.

DRC no special formatting of read-write records.

)SET functions analogously to !SET in TEL (CP-V TS/Reference Manual, 90 09 07). There are, however, some restrictions and differences.

Syntactic Differences between)SET and !SET

- Names and APL symbols are used instead of DCB names.
- For [1] and [2] , IN, INOUT, OUTIN, or OUT must precede the file/device designation. It is not an option.
- For output operations, the mode is automatically output and SAVE. There is no provision to create temporary files.
- No explicit file mode options are permitted. Access is always sequential. Output files are consecutive.

Operation of)SET

There are operational differences between !SET and)SET. !SET operates only between job steps with the relevant DCBs closed.)SET may be executed while a DCB is open and active.

If)SET is executed and the relevant DCB is open, the DCB is closed with SAVE prior to executing the changes in the DCB. It is then reopened.

The close is executed with a 'remove' option; thus, whenever magnetic tape has been specified and a new)SET is issued, the tape reel currently in use will be rewound and a dismount message issued to the operator.

The availability of input/output devices, particularly magnetic tape and private disc volumes, is subject to installation control. APL users who access such devices should be aware of such controls and local conventions on operator requests for tape mounts.

Default Device Settings

The following are considered default device settings. Any time an I/O stream is given a device assignment which differs from its prior assignment, the defaults are reestablished unless other options are specified in the)SET command.

DATA = 1
LINES = 32767
SPACE = 1
NOVFC
BCD
NODRC

Meanings of Device Options

Additional information concerning the device options for SET and their use in formatting output is contained in the CP-V TS/Reference Manual, 90 09 07, particularly Tables 5 and 40.

User Prompts

If either output or input is diverted from the terminal, the prompts normally issued to the user will be omitted. On 2741 equivalent terminals, the user can determine readiness for input by the terminal lock-unlock status. On teletype and ASCII terminals with full duplex, the echoing of characters indicates input is being accepted.

Break and Error Response

Errors on [1] and [2]. There is no essential change in the handling of errors in [1] and [2] input/output. The inclusion of)SET within APL permits error action by the user or, if error control is used, in APL functions.

Errors on INPUT or OUTPUT. If normal input or output has been reassigned from the 'home' device by)SET and an I/O error occurs, control is returned to the home device(s). This is user's console for on-line users, card reader and line printer for off-line. If error control is not in effect, an I/O error message is then output. If control is in effect for I/O errors, no error message is output. The user's error control function should note that input and output have been restored 'home' from their)SET assignments.

In order to avoid I/O translation problems, APL will also revert to 'home')TERMINAL type if the input or output terminal type has been individually modified. The CP-V Time-Sharing Reference Manual, Appendix B, includes tables of CP-V error codes for input/output errors.

Break Response. If normal input or output has been reassigned from the home device by)SET, it is restored to the home device by a Break. Translate tables will be restored to 'home')TERMINAL type. If the user has taken break control, the function which manages break control should note that input and output have been restored to the 'home' device and terminal.

)SET and VFC

In addition to modifying the DCB to indicate Vertical Format Control, VFC modifies APL output routines to automatically prefix a vertical format character to each output line. Provision, described in Chapter 9, is added to APL to permit user control of output spacing with format control characters.

BCD/BIN and NODRC/DRC

These device options affect device read-write formatting control exercised by CP-V.

Default is BCD;NODRC.

BIN;DRC provides 'transparent' input/output, with no translation or special treatment of characters by CP-V.

SIZE Option for [1] and [2] Input

The SIZE option allows relief of the restriction on the size of records input via [1] or [2]. Two benefits are thus derived:

- Large records can be read.
- In 'transparent' mode via user's console, the SIZE can be reduced as desired to avoid 'waiting for input' conditions. (In this mode an input record is not processed by CP-V until the specified byte count has been reached.)

)SI and **)SIV** Listing or Controlling the State Indicator

The following commands allow the user to find out what functions have been suspended during execution and where. They have the forms

```
)SI
)SIV
```

The)SI command displays the contents of the APL system state indicator, which is a list of suspended and pendant functions. For example,

```
      )SI
A[2]  *
XY[5]
B[3]  *
```

The most recent suspended function is listed first. An asterisk after an entry indicates a suspended function; no asterisk indicates a function that is pendant. In the above example, function A has been suspended just before line number 2, and function B just before line number 3. Function XY is pendant because it referenced function A at line number 5. If)SI is issued when an input quad is pending, the input request will also be displayed, using the □ character. If)SI is issued when an 'execute' is pending, the execute call will be displayed, using the ∈ character.

The)SIV command lists the same information as the)SI command except that it also lists the local variable names appearing in the suspended and pendant functions. For example,

```
      )SIV
A[2]  *        BB    CC    DD
XY[5]
B[3]  *        PAY   VAL
```

where BB, CC, and DD are local variables appearing in function A; and PAY and VAL are local variables appearing in function B. Errors causing suspended functions should be corrected as soon as possible. Suspended functions can be cleared from the state indicator with the branch arrow (→). (Remember that the state indicator with its list of suspended and pendant functions and local variables may take up a lot of workspace. Each branch arrow clears the most recent suspended function and all pendant functions associated with

it. This can be repeated until all suspended and pendant functions have been cleared; that is, until the)SI command returns a blank line. Applied to the above example, this would give

```
      →
      )SI
B[3]  *          PAY    VAL
      →
      )SI
```

A more convenient method for clearing the state indicator is to issue an)SI CLEAR command, see below.

For a discussion of the state indicators and suspended and pendant functions, see "State Indicators" and "Function Execution" in Chapter 7.

Three options are available with an)SI command: CLEAR, OFF, and ON. These may also be used with an)SIV command, with identical results. The options provide control over the state indicator.

The CLEAR option simply removes every entry in the state indicator. This often frees a substantial amount of workspace and is a valuable tool for recovering from WS FULL errors. The usual syntax is

```
)SI CLEAR
```

As with other command syntax, there must be at least one blank between the command and the option. This also applies to the ON and OFF options described below. Only the first four letters, CLEA, are necessary, but others may be supplied (as described for system command names).

The ON and OFF options set state indicator control for errors that may occur during subsequent execution of functions in the active workspace. APL normally (the ON setting) suspends the executing function if an error occurs. This is useful when debugging the workspace since it allows access to local variables for the suspended function. Suspending the function, however, expends a certain amount of the active workspace, and this can be a disadvantage. Use of the command

```
)SI OFF
```

sets state indicator control to avoid suspending a function when an error occurs. Note that the OFF setting applies only to errors. It has no influence over execution breaks or stop vectors; these may still cause function suspension.

To restore normal state indicator control, the command

```
)SI ON
```

may be issued. This setting also occurs automatically if a)CLEAR command is issued.

The ON or OFF state indicator control is saved when the active workspace is saved and loaded when the workspace is loaded. Copying does not alter the control of the active workspace.

)SYMBOLS Altering Length of Symbol Table

This command allows the user to display or change the current length of the internal symbol table. (The default is 261 possible names.) It has two forms

```
)SYMBOLS
)SYMBOLS n
```

The first form of this command,)SYMBOLS, causes the APL system to print the current allocation and the number of unused entries at present. The second form,)SYMBOLS n, causes APL to change the symbol table allocation and to print the previous allocation. The n parameter is the number of names the user wants to be allowed to use. If this number is reasonable, he gets at least the requested allocation, generally a little more; however, n may not

exceed 2001. The)SYMBOLS n command is allowed only when the active workspace is empty. Usually the user will issue a)SAVE,)CLEAR, and)SYMBOLS sequence, and then issue a)COPY command – not a)LOAD – to make use of the new allocation.

Examples

```
      )SYMBOLS
250  UNUSED OF 261
```

Indicates that 250 more names can be entered in a workspace whose symbol table allows 261 names; therefore the workspace presently contains 11 names.

```
      )CLEAR
CLEAR WS
      )SYMBOLS 300
WAS 261
```

Changes the symbol table allocation to 300 or more, and prints the previous allocation (i.e., 261).

)TABS Setting Tabs

This command can be used to display the current tab settings or as part of the procedure in setting new tabs for input/output. To display the current tab settings, the command is simply

```
)TABS
```

to which APL responds with a message giving the current tab settings. To set new tabs, the user must follow these steps:

1. Clear existing tabs, using the CLR portion of the CLR/SET key at the left side of the keyboard.

2. Set tabs at desired positions, using the SET portion of the CLR/SET key.

3. Type a command of the form

   ```
   )TABS n
   ```

   where n indicates the tab settings, and can consist of a scalar (one integer) or a vector (a series of integers) as follows:

   scalar indicates that tab settings are to be equally spaced or (if zero) that the APL tab feature is to be turned off. Specifying an integer in the range 3 to 127 indicates that tabs are to be equally spaced that size (integer) apart. For example, the command)TABS 15 sets tabs at columns 15, 30, 45, 60, etc.

   vector indicates unequally spaced tab settings. The numbers must be in increasing order and must start at a value of 3 or more. The highest number permitted is 127. For example, the command)TABS 5 25 35 40 sets tabs at columns 5, 25, 35, and 40. Up to 16 tab settings can be specified in this way.

   The APL system responds to this command with a message giving the previous tab settings. If this command is not specified, the APL system assumes a tab setting of zero by default.

Note that setting tabs is advantageous for speeding output on a standard APL terminal with true physical tabbing capability. The use of the tabs command is discouraged on terminals without physical tabs (such as a teletype) because tab simulation uses computer time without providing any real advantage. Tabs can also be set with the TABS function described in Chapter 7, "Defined Functions".

Examples

```
      )TABS
IS  5 25 37
```
Indicates that current tab settings are at columns 5, 25, and 37.

```
      )TABS 10
WAS 0
```
Sets tabs at columns 10, 20, 30, 40, etc. The previous setting was 0, i.e., no tabs were recognized.

```
      )TABS 5 25 37
WAS 10
```
Sets tabs at columns 5, 25, and 37. The previous tab settings were 10, 20, 30, 40, etc.

)TERMINAL Specifying Input/Output Device

This command is used to identify to the APL system the input/output device being used. It has the forms

)TERMINAL n
)TERMINAL INPUT n
)TERMINAL OUTPUT n

where n indicates the device to be assumed by the APL system and can be any of the following values:

1 indicates 2741 terminal (or equivalent) with APL typeball – this is the standard APL terminal.

2 indicates 2741 terminal (or equivalent) with non-APL typeball.

3 indicates Teletype Model 33.

4 indicates line printer format output or card reader format input.

13 indicates a typewriter-paired APL/ASCII terminal, such as Tektronix 4013; the user must not falsely declare this setting!

14 indicates a bit-paired APL/ASCII terminal.

This command is not required for users operating on a standard APL terminal or submitting batch runs for card input and line-printer output (type 1 and type 4, respectively, are assumed by default). New terminal declarations are acceptable at any time during an APL run, but the user should be aware of the consequences (such as error message discrepancies and input/output translation problems). See Appendix B for more about this command and the sign-on protocol on nonstandard terminals. I-beam 28 also results in the integer n; this may be useful for APL programs that are sensitive to terminal type.

The form)TERMINAL INPUT or)TERMINAL OUTPUT modifies only the specified (input or output) translation table. This form is useful when normal input or output has been diverted to an alternate device by the)SET command.

Examples

```
      )TERM 2
WAS 1
```
Indicates that a 2741 terminal (or equivalent) with a non-APL typeball is being used.

```
      )TERM
IS  2
```
Shows that the non-APL 2741 terminal was most recently declared.

```
      )TERM OUTPUT 4
WAS 1
```
Sets output translation for line printer. Does not change input translation.

As extended, APL now recognizes three separate parameters for input/output character translation.

'Home' terminal type is the sign-on default. It may be changed by the)TERM n command, for example, a Tektronix 4013 user starts with home terminal type 3 and will normally issue)TERM 13 to use the APL character set.

Home terminal type is the one returned to on break or input/output error. Input terminal type is changed by either)TERM INPUT n or)TERM n. Output terminal type is changed by either)TERM OUTPUT n or)TERM n.

Use of)TERM with)SET

The combination of)TERM and)SET permits a variety of I/O operations with devices and files. The user should be warned that some choices, particularly changes to 'home' terminal, can result in inability to carry on further terminal communication. In general,)TERM OUTPUT 4 should be used for line printer output, but not for filed output which may be later used for input (such as function displays). Output which will be filed and reread by the same user should preferably use home terminal type. If several users with different terminals will want to access the file, a common type, probably 1, should be agreed on.

)VARS Listing Global Variable Names

This command allows the user to list alphabetically the names of global variables in the active workspace. Three forms are acceptable:

```
)VARS
)VARS string
)VARS string1 string2
```

The first form displays all global variable names. The second form displays all global variable names that match or exceed the "string"[†] in alphabetic ordering. The third form displays all global variable names that match or exceed "string1" and are also less than or match "string2". Alphabetic ordering is illustrated in the examples below. Note particularly the first)VARS command since it indicates where each name character lies in alphabetic order.

Examples

```
      )VARS
A   AΔ  AΔ  AΔ  AA  AB  ABC A0  A1  B
      )VARS AA
AA  AB  ABC A0  A1  B
      )VARS A AA
A   AΔ  AΔ  AΔ  AA
      )VARS AAA AB
AB
      )VARS ABC ABC
ABC
      )VARS Δ B3
A   AΔ  AΔ  AΔ  AA  AB  ABC A0  A1  B
```

)WIDTH Setting Line Width

In a clear workspace the width of a line of output is set at 120 characters (or 72 characters if a non-2741 type terminal was initially recongized by APL). The)WIDTH command allows the user to change this width or to display the current width, and has two forms

```
)WIDTH
)WIDTH n
```

where n is the number of spaces in a line of output and can be any number ranging from 30 to 254. The first command form,)WIDTH causes the APL system to print the current line width setting. The second command form,)WIDTH n, causes the APL system to change the line width setting and to print the previous line width setting. This command is saved along with a workspace. The width of a line of output can also be changed with the WIDTH function described in Chapter 7, "Defined Functions".

It should be noted that a width greater than 130 is improper for normal operations on a standard APL terminal. Also note that if the PLATEN command, at the CP-V level, has been used to set a shorter width than declared by)WIDTH, then the PLATEN command overrides WIDTH. In this case lines may be split in the middle of individual values.

[†]Where "string" is any sequence of characters not including a blank or carriage return. If a string includes more than 80 characters, those past the 80th are ignored. Strings are used for range demarcation.

Examples

```
      )WIDTH
IS  120
```

Displays the current width of a line of output (i.e., 120 spaces).

```
      )WIDTH 50
WAS 120
```

Changes the width of an output line to 50 spaces. The previous line width setting was 120.

)WSID Identifying Active Workspace or Changing Its Name

This command allows the user to identify the active workspace or to change its name. To identify the active workspace, use the command

)WSID

and APL will return the name and account (if different than the current user account) of the active workspace. To change the name of the active workspace, use the command form

)WSID wsname
)WSID wsname..password

where wsname is the new name of active workspace. The APL system responds with a message showing the previous workspace name. This name can be from 1 to 11 characters. A password may be specified, but a previous password is never displayed.

)WSID cannot be used to change the name of a sealed workspace.

Examples

```
      )WSID
IS          CONTINUE
```

Lists the name (CONTINUE) of the active workspace.

```
      )WSID
IS  RE107207 JONES
```

Lists the name (JONES) and account (RE107207) of active workspace.

```
      )WSID SMITH
WAS         JONES
```

Changes name of active workspace from JONES to SMITH.

9. REPORT FORMATTING

Xerox APL provides a formatted output capability by means of a set of fast intrinsic functions. They reside in a designated "public" workspace (WSFNS), may be copied by any user, and occupy a negligible amount of the user's workspace since the functions are actually implemented as part of the APL processor. (See Appendix C.)

ΔFMT[†]

The formatted output function, called ΔFMT, utilizes a set of format control phrases that are applied to a list of APL expressions. The content of the APL expressions may evaluate to numeric or character scalars, vectors, or matrices. The format control phrases, called format specifications, are described below.

Format Specifications

ΔFMT recognizes six data format codes:

A	Alphanumeric specification.
E	Floating-point with exponent (scientific format).
F	Floating-point to fixed decimal position.
I	Decimal integer.
X	Blank insertion.
⍞ *TEXT* ⍞	Text insertion.

Format specifications may be in any of the following forms:

[r] Aw

[r] Ew.s

[r][q] Fw.d

[r][q] Iw

[r] Xw

[r] ⍞ *TEXT* ⍞

where

- r is an optional unsigned integer constant indicating the specification is to be repeated r times. When r is omitted it is taken as 1.
- w is an integer constant indicating the total field width, or number of print positions occupied by the formatted value (or blanks, for X type).
- s is an integer constant indicating the number of significant digits to be printed in E formats; s must be less than w-5.
- q is an optional "qualifier" or "affixture" code used to position and affix characters to the results of I and F output forms. The available codes and their uses are described later in this chapter.
- d is an integer constant indicating the number of digits to the right of the decimal point in F formats; d must be less than w.

[†]This feature was invented and introduced by I.P. Sharp Associates Ltd., Toronto, Canada.

Format Specifications Versus Data Types

Format A may be applied to character data only. Formats E, F, and I may be applied to numeric data and logical values only.

Arrays with rank above 2 (matrix) cannot be processed. If a value cannot meaningfully be expressed in the format and field width specified, the field is filled with asterisks.

The Format Statement (Left Argument)

A format statement is the left argument of ΔFMT, operating on data values in the right argument. The format statement consists of a character vector made up of one or more format specifications separated by commas. The left argument of ΔFMT must always be a valid format statement. For example,

```
'3I5,2E8.2,X12,I3' ΔFMT...
```

The Data List (Right Argument)

The right argument of ΔFMT must be a list of APL variables or expressions, separated by semicolons or a variable formed by assigning such a list to a name. The expressions may represent scalars, vectors, and arrays. For example,

```
... ΔFMT (VARIABLE1;VARIABLE1+VARIABLE2;'SUM')
```

If the list contains only one expression, the parentheses may be omitted.

Operation of ΔFMT

ΔFMT uses the format specifications in its left argument (the format statement) to control printing of its data list (right argument) in one or more columns. The syntax is

format stmt ΔFMT expr

or

format stmt ΔFMT list

The result of executing ΔFMT is one or more "lines" of formatted character data. A line may be as long as workspace allows. In printing, long lines are broken up according to the)WIDTH setting. If more than one line is produced (as will be the case if the data list includes vectors or arrays with more than one row) all lines are of the same length. The result, then, is a character matrix.

If ΔFMT is not used within a larger expression, the amount of temporary workspace required is only the length of one line. Thus, formatted output may be used to process output that would overflow available workspace if assigned or used in its entirety. If ΔFMT is used within a larger expression, the result is always a matrix, even if only one line, and space for the full matrix is required.

The operation of ΔFMT on various right arguments is described below.

Scalar Arguments

If the data list consists exclusively of scalars, a single line is created. Format specifications are used in turn to process elements of the data list in left to right order. Blank insertion and text insertion specifications do not "use up" elements of the data list, however. A repetition indicator will cause a particular specification to be applied the designated number of times to successive elements of the data list. If there are fewer format specifications (counting repetition indicators) than values to be formatted, the list of format specifications is reused as necessary until the data list is exhausted.

Examples:

```
      'I3,A5,X5' ΔFMT (100;'A';200;'B')
100   A    200  B

      '5F5.2' ΔFMT (1;10;100;¯10;¯1)
 1.0010.00**********¯1.00
```

This last example illustrates the use of the repetition indicator. Also note the asterisks indicating that the values 100 and ¯10 would not fit in the specified format.

Vector Arguments

If the data list includes vector and scalar arguments (or vectors only), the number of lines generated equals the length of the vector with the most elements. Each vector creates a "column" in the resulting character array. The columns of shorter vectors or scalars are extended by blanks.

Examples:

```
      'E10.4' ΔFMT 3.1 .123 ¯1.234 5678
 3.100E0
 1.230E1
¯1.234E0
 5.678E3

     '2I5,A2' ΔFMT (1 2 3 4;10.4 10.6;'ABCDEF')
1    10 A
2    11 B
3       C
4       D
        E
        F
```

In the last example, note the rounding off of values as required for I format specifications, and also note the different column lengths.

Formatting a Vector on One Line

The normal result of ΔFMT on vector arguments is columnar formatting, but it is often desirable to create row formats for vectors. There are two ways this can be done:

- Ravel the result of ΔFMT. This method is appropriate if the result would contain a single column.

  Examples:

```
      ,'A2' ΔFMT 'DOUBLE SPACE'
D O U B L E   S P A C E

      ,'I5' ΔFMT .14 1.4 14 140 1400
0    1   14  140 1400
```

- Reshape the vector as a 1 by N matrix. (This method uses a property of the operation of ΔFMT on matrices, as discussed below.)

```
      V←'TRIPLE+SPACE'
     'A3' ΔFMT (1,ρV)ρV
T  R  I  P  L  E  +  S  P  A  C  E
```

Matrix Arguments

If the data list includes a matrix argument, each column of the matrix will create a "column" in the formatted output. Each row of the matrix will create an entry on a "line" of output. Note that a 1 by N matrix creates a row of a column, and an N by 1 matrix creates the same output form as an N element vector.

In essence, ΔFMT outputs matrices in the same shape as unformatted output would, but permits control of decimal placement, column positioning, etc.

Examples:

```
      IOTA←3 5ρι15
     'F5.1' ΔFMT IOTA
 1.0  2.0  3.0  4.0  5.0
 6.0  7.0  8.0  9.0 10.0
11.0 12.0 13.0 14.0 15.0
```

```
      JKL←'JKL'
      PI←3.14
      VECT←1 2 3 4
      MAT←2 2ρ .1 2.0 30 ¯4
      'A1,F6.3,I5,2F6.1' ΔFMT (JKL;PI;VECT;MAT)
J  3.140    1   0.1   2.0
K           2  30.0  ¯4.0
L           3
            4
```

Forms of Output Values

The following rules determine spacing and content of output fields for various format specifications.

- Right justification. For A, I, and F specifications, the value is right justified in the field and preceded by blanks where appropriate to fill out the field.

- E format. The letter E always occupies the fourth space from the right in the field. Three spaces are reserved for the exponent value and exponent sign. If less than three spaces are needed, the rightmost space or spaces are blank. In this format there will be columnar alignment of the decimal points and letter E.

- ⍞ *TEXT* ⍞ format. Characters between the quote-quads are inserted directly into the output line. There will be as many insertions as there are lines of output. No data list elements are expended by text insertion.

- Significance of results. The)DIGITS setting is ignored in ΔFMT output; a maximum of 16 significant positions will be displayed, however. If a format specification requests more than 16 significant digits, digits beyond the sixteenth and to the left of the decimal point are replaced by underscores. Excess digits to the right of the decimal point are replaced by blanks.

- Field width. If field widths are too small to hold formatted values according to the specification, the fields are filled with asterisks.

Qualifiers and Affixture Codes

I and F format specifications may be immediately preceded by one or more qualifier or affixture codes.

- Qualifier codes

  B leaves the field blank if the result would otherwise be zero.

  C inserts commas between triads of digits in the integer part of the result.

  L left justifies the value in the result field.

  Z fills unused leading positions in the result with zeros (and commas if C is also used) instead of blanks.

- Affixture codes

  M␣ TEXT ␣ prefixes negative results with the text instead of the negative sign.

  N␣ TEXT ␣ postfixes negative results with the text.

  P␣ TEXT ␣ prefixes positive results with the text.

  Q␣ TEXT ␣ postfixes positive results with the text.

  R␣ TEXT ␣ presets the field to the text, which is used as many times as necessary to fill the field. The text will be replaced in parts of the field filled by the result.

  Note: If B and R are both specified, R overrides B.

Qualifier and affixture code do not extend field widths. The modified result must fit in the field width specified or asterisks will be substituted.

N and Q affixtures, since they postfix the text, shift results to the left by the number of characters to be postfixed.

Examples:

```
      V←128 0 ¯.25 ¯64 ¯12345.67
      'BF10.1,X2,BI8,X2,CI10,X2,LI9' ΔFMT (V;V;V;V)
    128.0        128          128  128
                                0  0
     ¯0.3                       0  0
    ¯64.0        ¯64          ¯64  ¯64
¯12345.7     ¯12346      ¯12,346  ¯12346


      'ZF10.2,X2,M␣**␣F10.1,X2,P␣+␣I8' ΔFMT (V;V;V)
0000128.00       128.0       +128
0000000.00         0.0         +0
¯000000.25       **0.3         ¯0
¯000064.00      **64.0        ¯64
¯012345.67   **12345.7     ¯12346


     'Q␣+++␣I9,X2,R␣*␣I8' ΔFMT (V;V)
128+++   *****128
  0+++   *******0
  0+++   *******0
   ¯64   *****¯64
¯12346   **¯12346
```

Combinations of qualifiers and affixtures may be used together to provide various output forms as shown below.

```
      'M⍞-$⍞P⍞$⍞CF12.2' ΔFMT (12345.67;¯9.98)
$12,345.67       -$9.98
```

Use of Result

The principal use of ΔFMT is to provide lines of formatted output to the user's console. However, if ΔFMT is used as part of a larger APL expression, the result of executing ΔFMT is a character matrix which may be manipulated and used just as any other character matrix.

Error Exits

If the right argument includes an array of higher order than matrix, or the left argument is not a vector, a RANK ERR results.

If the left argument is numeric rather than character data, or contains no format specifications, or contains a format specification with inconsistent parameters (such as d greater than w, or w = 0), a DOMAIN ERR results.

If there is incorrect syntax in the right argument, a SYNTAX ERR results. If there is incorrect syntax in the left argument, a FORMAT SYNTAX ERR results.

If the line length of the result is too big for the remaining workspace, or in the case that ΔFMT is included in a larger expression and the total result exceeds the workspace, WS FULL results.

Other Output Formatting Aids

In addition to ΔFMT, the following intrinsic functions may be used to aid in output formatting. These functions are also kept in WSFNS or may be created as described in Appendix C. The)SET and)TERMINAL commands, described in Chapter 8, may also be used in the overall process of output report generation.

PAGE

PAGE is a niladic function with an empty vector result. When executed, if output is to a printing device, the current page will be ejected. If output is to the user's console and LINES has been established by)SET, a standard header line will also be produced, unless the option DRC was indicated.

NLINES

NLINES is a niladic function with an integer result. If output is to a device, with line count applicable, the result is the number of lines remaining. If not, the result is zero.

HEADER

HEADER is a monadic function with empty vector result. The right argument must be a text vector length 127 or less. This function establishes the output header line which will be issued at the start of each page if output is via a printing device. This intrinsic function uses a CP-V facility which is not cognizant of special APL characters. If special characters or overstrikes are included, HEADER will not produce correct headings.

VFCHAR

VFCHAR is a monadic function with empty vector result. Right argument must be a single text character. When VFCHAR is executed, the character in the right argument becomes the vertical format control character for the next print line. After that line is printed, the default blank character is restored as the vertical format control character.

The following characters have meaning as right arguments to VFCHAR:

'_'	Inhibit space after printing (not possible on some terminals)	
'A'	Space 1 additional line before printing	
'B'	2	lines
:	:	:
.	.	
'|'	9	
2 ¯ 202	10	
2 ¯ 203	11	
:	:	
.	.	
2 ¯ 207	15	
'0'	Skip to bottom of page before printing	
'1'	Skip to top of page before printing	

VFCHAR is effective only when VFC mode is on (see SET command, Chapter 8).

Example:

Line 5 of function FORM is to print a title which is spaced 3 lines below the preceding output line. VFC mode has been set.

```
FORM[5]  'PART TWO: COSTS VS PRODUCT'; VFCHAR 'C'
```

ΔXL

The translate operator, ΔXL, facilitates special character set translations within APL. The form is

```
R ← A ΔXL B
```

The left argument must be a character vector of length 256. The right argument may be any text entity. The result has the same shape as the right argument and consists of a translation of the right argument based on the left argument, as follows: The value of a right argument element determines the offset into the left argument. The result element is the character at that offset in the left argument. Figure B-1, in Appendix B, shows the collating order of the full EBCDIC and APL character sets.

This feature is designed for sophisticated users, to overcome problems encountered in character set differences between various devices and processors. It allows any character mapping, including mapping several characters to the same result character. An example of this use might be to map all 'illegal' characters to some unique character. Another example is as follows:

```
L←¯1+⍳256
L[128+⍳9]←192+⍳9
L[144+⍳9]←208+⍳9
L[161+⍳8]←225+⍳8
```

This value of L, used as left argument of ΔXL, will convert underscored letters in the right argument to similar letters, not underscored, in the result.

10. EXECUTION STOPS

Execution is stopped if any of the following conditions occurs:

1. Execution is completed (a normal stop).
2. Execution break occurs (ATTN key is struck), and sidetracking (see Appendix A) does not occur.
3. User input is required (quad or quote-quad input).
4. Stop control vector is encountered.
5. Error is encountered, and sidetracking (see Appendix A) does not occur.

Normal Stop

Execution comes to a normal stop after any action indicated by direct input has been completed. It should be noted, however, that a direct input prompt does not necessarily mean that all pending execution is completed. The user can determine whether any execution is pending via the)SI or)SIV commands described in Chapter 8.

Execution Break

An execution break (that is, the ATTN key) can be issued by the user at any time except during terminal input (when ATTN becomes an editing signal). There may be a variable delay – sometimes several seconds for a fully loaded system – until output stops. Sidetracking (see Appendix A) can be used to gain break control within an APL function; in this case execution does not stop, but is "diverted".

APL's reaction to ATTN also depends on whether the key is struck during execution mode or definition mode. If ATTN is used during (nonfunction) execution, APL will stop any output and will skip to the next line and indent six spaces to prompt for new input. If ATTN is used during execution of a defined function, APL prompts the user with the function name and the line number being executed; the line being executed may have been partially completed.

If ATTN is used during display of a function, APL will exit from function definition mode if a closing del was included in the display command; if the display command did not have a closing del, APL will remain in function definition mode and will prompt with the next line number to be assigned to a statement.

Execution breaks are usually not allowed to interrupt the execution of a system command. However, those that produce lengthy display can be stopped:)FNS,)GRPS,)LIB,)SI,)SIV, and)VARS. ATTN is also used to break out of the)OPR system command.

Stop for User Input

Execution may be stopped by a pending input request in a line. The normal response to a quad or quote-quad input request is a line of input. While input is pending, ATTN is not considered an execution break and thus

does not cause an exit from the input request. If the user's program contains a loop such that he is repeatedly prompted for input, he may escape as follows:

1. For quad input, type a branch arrow (→) followed by a RETURN key. An example is shown below:

```
        ∇PITIMES[□]∇
     ∇ PITIMES
[1]    ○□
[2]    →1
     ∇
        PITIMES
□:
        1
3.141592654
□:
        ¯1
¯3.141592654
□:
        )SI
□
PITIMES[1]
□:
        →
        )SI
```

   In this example the user has defined a function, PITIMES, that repeatedly requests input and provides a result. The first)SI command shows that an input request and line 1 of PITIMES are pendant. After the →, the input request is no longer repeated. The second)SI command shows that the loop has been broken and PITIMES is no longer in use.

2. For quote-quad input, type the following sequence of characters and then strike the RETURN key: O backspace U backspace T. This sequence appears at the terminal as ⍬.

Stop Control Vector

As described in Chapter 7 (under "Stopping Execution"), a stop control vector can be used to specify the exact place a function suspension is to occur. The user can set a stop control vector by typing the symbols S∆ followed by the function name, an assignment arrow, and the line numbers at which function execution is to be suspended. For example, suppose the user wants to suspend execution of function HH at lines 2 and 4; he will type the expression

```
S∆HH←2 4
```

APL will then suspend function execution just before each specified line number is reached, print the function name and line number at which suspension occurred, and skip to the next line and indent six spaces to prompt for user input.

```
      HH
HH[2]
      ↑
      └──── carrier is here
```

The user may then operate as desired, in direct mode, with the function suspended. He can resume or terminate function execution at any time. Function execution can be resumed by appropriate branching; for example, an entry of →3 will resume execution of the suspended function at statement 3. Termination can be accomplished by a branch to a nonexistent line number (→0 is a convenient choice). The function suspension can also be abandoned by a suspension clear statement, which is a branch arrow without any line number.

A stop control vector can be specified during execution mode, or during function execution as one of the statements of a defined function. To discontinue an active stop control vector, assign an empty vector or a nonexistent line number (such as 0) to that stop control vector; for example, $S\Delta HH\leftarrow 0$ will turn off the stop control for function HH.

Error Stop

As soon as APL detects an error in a statement, execution of that statement is terminated and any partial result is lost – unless an assignment was completed before the error was detected, in which case the assignment is effective. Unless sidetracking occurs (see Appendix A), APL prints a message indicating the type of error, types the erroneous statement with a caret below the place the error was detected[†], and prompts for user input. The user can then re-enter the statement correctly. An example of error detection is shown here:

```
      X1←4÷0
DOMAIN ERR
      X1←4÷0
          ^
```

If a statement contains more than one error, only the first (rightmost) one detected by APL will result in an error report. The next error will not be detected until the user has corrected the first error, as illustrated here:

```
      X1←(4÷0)×(2÷0)
DOMAIN ERR
      X1←(4÷0)×(2÷0)
                ^
      X1←(4÷0)×(2÷1)
DOMAIN ERR
      X1←(4÷0)×(2÷1)
            ^
```

If an error is detected in a compound statement or in a statement with multiple specifications, any assignments or statements to the right of the error will be completed, as illustrated here:

```
      4÷C←B×0 ⋄ B←5
DOMAIN ERR
      4÷C←B×0 ⋄ B←5
        ^
      B
5
      C
0
```

During function definition some types of errors are detected immediately while other types are not detected until later when the function is executed. Definition errors, and character errors are detected immediately and must be corrected as soon as an error report is printed.

```
      ∇R←B TRI H
[1]   AREA←0.5×B×H
[2]   DIAGONAL←((H*2)B*2)*0.5
[3]   R←AREA⋄DIAGONAL
[4]   [0.5  TRI CALCULATES AREA AND DIAGONAL OF TRIANGLE
 DEFN ERR    ^
[0.5] ⍝TRI CALCULATES AREA AND DIAGONAL OF TRIANGLE.
[0.6] ∇
```

Line-scan errors are detected immediately and may be corrected immediately by function editing or its correction may be deferred. All other errors in a defined function are detected when the function is executed. When APL encounters each error during function execution, it suspends execution and prints an error report containing the following information: the type of error and the function name and offending line and statement (with a caret marking

[†]See also the discussion of error messages for the Execute operator in Chapter 5.

the place the error was detected). For example, the following error message is produced because a multiplication sign has been omitted after the right parenthesis:

```
      5 TRI 8
SYNTAX ERR
TRI[3]  DIAGONAL←((H*2)B*2)*0.5
                          ^
```

An error that causes suspended execution can be corrected during the suspension or after termination of execution.

1. To correct an error during suspended execution, the user can follow normal function editing procedures (see Chapter 7). For example,

```
SYNTAX ERR
TRI[3]  DIAGONAL←((H*2)B*2)0.5
                           ^
      ∇TRI[3] DIAGONAL←((H*2)×B*2)*0.5∇
```

   After correcting an error, the user can resume execution at the point it was suspended by specifying a branch to that line number. Thus, the expression →3 will resume execution at line 3 (starting at the right, as usual for APL).

2. To correct an error with termination of execution, the user enters a branch arrow to terminate function execution, edits the function as necessary, and then reexecutes the function. For example,

```
SYNTAX ERR
TRI[3]  DIAGONAL←((H*2)B*2)*0.5
                           ^
      →
      ∇TRI[3] DIAGONAL←((H*2)×B*2)*0.5∇
      5 TRI 8
```

   Each branch arrow removes the most recent suspension from the state indicator list. Thus if several suspensions have occurred since the last suspension clear, more than one branch arrow (suspension clear) will be required to clear the state indicator. A convenient method for clearing the entire state indicator is to issue an)SI CLEAR command.

11. GRAPHICS

When the user is on-line with a Tektronix 4013 graphics terminal, and has declared)TERMINAL 13, he may make use of the graphics capabilities described below. These facilities are implemented in terms of a graphic I/O primitive ⍞ and an intrinsic function ∆GRF, described later in this chapter. However, most users will find it more convenient to employ the graphic functions defined in a workspace named GRAF described below. These features were originally developed at University of California Irvine under the direction of Dr. Alfred Bork.

The graphics capability may also be used on a Tektronix 4015, which APL accesses in 4013 mode, or on a 4010. The 4010 does not support the APL character set (see "Teletype Usage", Appendix B).

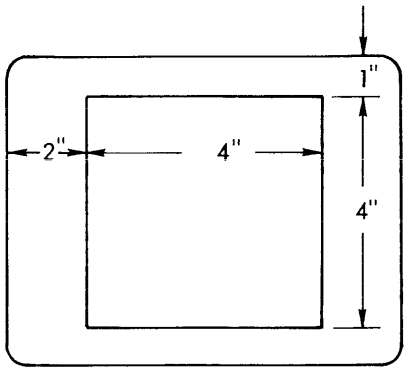
User Graphic Functions

The following functions are available in workspace GRAF:

DRAW, INT, VS	Plotting Functions
SCALE, NOSCALE, CENTER, WHATSCALE	Scaling Functions
WINDOW, WHATWINDOW	Window Functions
AXES, BOX, DASH, SET, PUT, ERASE, HOME	Auxiliary Plotting Functions
WHATCOORD, WHATCHAR	Graphic Input Functions
SIN, COS, EXP, STRAPIS	Other Functions

Plotting Functions: DRAW, VS, INT

DRAW produces a curve on the screen, working from one-dimensional, two-dimensional, or three-dimensional data, and determines where the curve is to appear. The left argument of DRAW specifies a "window", a rectangular portion of the screen in which the curve is to appear. The window argument is a four-element vector giving, in inches, the x (horizontal) and y (vertical) window limits. Thus, 2 6 1 5 would specify the 4-inch square window positioned as shown:



If the left argument of DRAW is a scalar, the current window remains in effect; the value of the scalar is ignored.

The data to be plotted is given by the right argument of DRAW, which is taken as a collection of one-dimensional, two-dimensional, or three-dimensional points. For an argument composed of N two-dimensional (2D) points (i.e., the argument is 2-by-N or N-by-2), the plotted curve consists of N-1 straight line segments, each joining one given (x, y) point to the next. For one-dimensional data (i.e., the argument is a vector, or N-by-1, or 1-by-N), the data is taken as y values and plotted against X = N (which depends on the index origin). Three-dimensional (3D) data (N-by-3 or 3-by-N) are projected on the two-dimensional (2D) window and plotted as (x, y, z) points.

After a DRAW, the cursor is returned to the next writable line. DRAW does not erase the screen, so it can be used to overplot curves.

The right argument for DRAW is often set up by use of the VS and INT (interval) functions. INT A, B, N produces a vector of N equally spaced values between A and B. Thus, if

```
X←INT¯6  6  100
```

X contains 100 values, equally spaced from ¯6 to 6.

VS is used to "glue together" sets of values to form two-dimensional or three-dimensional groups suitable for use as DRAW's right argument. Thus

```
0  DRAW  Y  VS  X
```

uses VS to form a 2-by-N matrix from the two N-vectors X and Y. For three-dimensional (3D) plotting, the corresponding call is

```
0  DRAW  Z  VS  Y  VS  X
```

The following statement produces a curve of the function Y = COS X for 0≤x≤6.28 (in radians):

```
0  DRAW  (COS X) VS (X←INT 0,6.28,40)
```

Scaling Functions: SCALE, NOSCALE, CENTER, WHATSCALE

SCALE, NOSCALE, and CENTER determine the placement of the picture within the window; i.e., the limits of x, y, (and z) values represented by the edges of the window. To set these limits to particular values, use SCALE as follows:

```
SCALE xmin, xmax, ymin, ymax
```

for 2D scaling, or

```
SCALE xmin, xmax, ymin, ymax, zmin, zmax
```

for 3D scaling.

In the absence of a fixed scaling specification, "automatic scaling" is in effect: In this mode, each DRAW finds the minima and maxima of its data along each coordinate, and scales the data to occupy the entire window. NOSCALE (with no arguments) may be used to restore automatic scaling once an explicit scaling has been set by SCALE. CENTER (with no arguments) also establishes automatic scaling, but in such a way that the origin is placed at the center of the window, and the data is scaled to fit the window. Center-scaling is the default scaling rule in effect before any scaling functions are called.

To find out what scaling is currently in effect, or what scaling was established by the last DRAW if automatic scaling is in effect, use WHATSCALE:

```
S←WHATSCALE
```

sets S to either a 4-vector or a 6-vector scale:

```
S = xmin, xmax, ymin, ymax
```

or

```
S = xmin, xmax, ymin, ymax, zmin, zmax
```

Scaling and window data is saved as part of the workspace, so a DRAW command just after a)LOAD will draw the same curve it did just before the workspace was saved.

Window Functions: WINDOW, WHATWINDOW

The window limits may be set by the left argument of a DRAW command, as seen above, or by a WINDOW command. Thus, either

```
2 6 1 5 DRAW Y VS X
```

or

```
WINDOW 2 6 1 5
```

will establish the window as shown in the diagram above. Window limits are always set by a four-vector (lx, ux, ly, uy) where the l's and u's are lower and upper limits, respectively, in inches from the lower left corner of the screen.

The function WHATWINDOW returns this four-vector as its value.

Window and scaling data is saved as part of a workspace and restored by)LOAD.

Auxiliary Plotting Functions: AXES, BOX, DASH, SET, PUT, ERASE, HOME

AXES. Draws axes according to the current scaling and window conventions, outputs the end-values, and returns the cursor to the next writable line.

BOX. Draws a box displaying the current window limits and returns the cursor to the next writable line.

DASH. Causes the next curve plotted with DRAW to be dashed. Applies only to the next curve.

SET x,y or SET x,y,z. Places the cursor at the given point in the window. The original cursor position is remembered; it may be subsequently restored by a set with an empty right argument, e.g., SET ".

(x,y) PUT value or (x,y,z) PUT value. Places the cursor at the specified point in the window, then outputs "value", whether it is character or numeric. Thus,

```
0.1  3.14  PUT  'CURVE A'
```

will put the words CURVE A on the screen starting at the point x=0.1, y=3.14. PUT returns the cursor to the next writable line.

ERASE. Erases the screen and automatically puts the cursor at its "home" position near the upper left corner of the screen.

HOME. Places the cursor at its "home" position near the upper left corner of the screen.

Graphic Input Functions: ⍞Input, WHATCHAR, WHATCOORD

The graphics input facility allows the user to locate a position on a screen and to communicate the coordinates of that position to his program. The expression `A ← ⍞` causes the 4013 to display the "crosshair cursor", a pair of lines – one horizontal, one vertical. The user can then move these lines, via two knobs on the terminal, to locate any point on the screen. When he then types a character, the crosshairs disappear and the coordinates of the intersection of the crosshairs become the result of ⍞. In the above expression, A would be set to the value of the coordinates. The result is a two-element numeric vector. The first element is the X, and the second element the Y coordinate of the intersection, relative to current scaling. After the input, the alphanumeric cursor returns to the next line.

The input character and the coordinates are saved internally and may be accessed by the functions WHATCHAR, which gives the character as a result, and WHATCOORD, which gives the coordinates as a result.

If scaling is in effect (which is normally the case) and a crosshair intersection is selected that is not in the currently defined WINDOW, the crosshairs are turned back on and the user must retry, selecting some point inside the window, or "break" to escape graphic input mode.

If scaling is not in effect (see "unscaled graphic I/O later in this chapter) coordinates are returned in raster units and current window and scale parameters are ignored.

Other Functions: SIN, COS, EXP, STRAPIS

For the user's convenience, the GRAF workspace also contains the following functions:

SIN X	equivalent to 1oX
COS X	equivalent to 2oX
EXP X	equivalent to *X

STRAPIS is a function to allow changing the response of APL graphics software to accommodate different "strapping" options for particular 4013 terminals. APL graphics assumes that the user's 4013 is set to deliver 7 characters in response to graphics input requests. Some terminals are set to deliver 5 or 6 characters. In this case, if the user executes a function such as BOX, the terminal will not respond until one or two characters are input. If this occurs,

```
STRAPIS 6
```

or

```
STRAPIS 5
```

can be used so that APL will only try to read 6 or 5 characters.

Note: If STRAPIS 5 or 6 is used when the actual strapping is for 7 characters, functions such as WHATCOORD will provide incorrect results. The STRAPIS function should be used with caution.

Direct Control of Graphic I/O

The more sophisticated user may wish to exercise more control over the graphics I/O facility than is possible with the functions described above. He may do so by making direct use of the graphic I/O symbol ⍞ and the intrinsic function ΔGRF, as described below.

⍞ Output

The expression ⍞ ← *A* (where A is not a scalar) performs the same function as 0 DRAW A; that is, it takes A as an ordered set of points which it scales according to the current scaling rule, and plots in the window. If A is a scalar, ⍞ ← *A* sends one ASCII character to the terminal; in this case, A is the integer value of the character.

Caution: Scalar output should only be attempted with thorough knowledge of the terminal operating characteristics.

The following table shows the legal possibilities for the shape of A, with the resulting interpretations:

⍴A	Meaning
(empty)	A is scalar: send as ASCII character
N or 1 N or M 1	data is 1D; plot X=⍳N, Y=A
2 N	data is 2D; plot X=A[1;], Y=A[2;]
M 2	data is 2D; plot X=A[;1], Y=A[;2]
3 N	data is 3D; plot X=A[1;], Y=A[2;], Z=A[3;]
M 3	data is 3D; plot X=A[;1], Y=A[;2], Z=A[;3]
0 or 0 K or K 0	data is empty; plot nothing

where $N \geq 1$, $M \geq 4$, and $K \geq 0$.

The data is processed as follows:

- Data is scaled according to the current rule:
  - automatic scaling (NOSCALE)
  - centered automatic scaling (CENTER)
  - fixed scaling (SCALE x_1 x_2,y_1,y_2 [,z_1,z_2])
  - "transparent" scaling (SCALE x_1,x_2,y_1,y_2 [,z_1,z_2])
- Three-dimensional (3D) data is reduced to two-dimensional (2D) data by planar projection.
- Data that falls outside the window (possible only under the fixed scaling rule) is suppressed; where the data crosses the window boundary, the curve is extended to the boundary by linear interpolation. If the "reentrance" mode is <u>on</u> (via 10 ΔGRF 2), curve plotting resumes at the point where the data reenters the window; if the "reentrance" mode is <u>off</u> (via 10 ΔGRF 1), curve plotting stops when the first out-of-range point is encountered.
- If a preceding DASH command has been given, the curve is broken up into short pieces, one per point; otherwise, it is drawn continuously. DASH applies only to the plotting of this one curve.
- The resulting picture is drawn on the screen in the position specified by the current window parameters.
- After drawing the curve, the cursor is restored to its original position at the beginning of the next APL input line.

Δ GRF Intrinsic

ΔGRF is a dyadic intrinsic function used to control graphic I/O. It is declared in the GRAF workspace as follows:

```
ΔGRF ← 14 ⍤ 3
```

The name for this function could be any name assigned to 14 ⍑ 3, but is assumed here to be ΔGRF. The form of the function is

f ΔGRF p

The left argument, f, is an integer specifying which action is to be performed: $0 \le f \le 11$. The right, p, consists of any parameters the selected action may require; p is often empty, in which case any type of empty vector will do, such as '' or ⍳0. ΔGRF returns a result: it is usually the empty integer vector ⍳0, but (11 ΔGRF k) returns a numeric vector of length 2, 4, or 6 (see Table 6 below), or a character scalar.

Table 6 shows all of the actions that ΔGRF can perform, together with the corresponding user functions where the latter have been provided.

Table 6. ΔGRF Calls

GRF Call	Corresponding Function Call	Action Taken
0 ΔGRF 1		Turns off ⎕input scaling
0 ΔGRF 2		Turns on ⎕input scaling (default)
1 ΔGRF 3		Sets terminal device type = 4013
2 ΔGRF ''	WINDOW ''	Sets default window (default)
2 ΔGRF x_1, x_2, y_1, y_2	WINDOW x_1, x_2, y_1, y_2	Sets up given window params
3 ΔGRF ''	SCALE ''	Sets 'transparent' ⎕output scaling and full-screen window
3 ΔGRF x_1, x_2, y_1, y_2	SCALE x_1, x_2, y_1, y_2	Sets x and y limits for fixed 2D scaling
3 ΔGRF $x_1, x_2, y_1, y_2, z_1, z_2$	SCALE $x_1, x_2, y_1, y_2, z_1, z_2$	Sets x, y, and z limits for fixed 3D scaling
4 ΔGRF ''	SET ''	Restores cursor loc saved by SET x, ...
4 ΔGRF x, y	SET x, y	Saves cursor loc; moves it to (x, y)
4 ΔGRF x, y, z	SET x, y, z	Saves cursor loc; moves it to (x, y, z)
5 ΔGRF ''	AXES	Draws axes according to current scaling
6 ΔGRF 1	NOSCALE	Sets noncentered automatic scaling
6 ΔGRF 2	CENTER	Sets centered automatic scaling (default)
6 ΔGRF 3		Fixes current scaling
7 ΔGRF ''	DASH	Specifies next curve to be dashed
8 ΔGRF ''		Erases screen and homes cursor
8 ΔGRF 1	ERASE	Erases screen (cursor homes automatically)
8 ΔGRF 2	HOME	Homes cursor
9 ΔGRF ''	BOX	Draws box according to current window
10 ΔGRF 1		Turns off curve reentrance mode

Table 6. ΔGRF Calls (cont.)

GRF Call	Corresponding Function Call	Action Taken
10 ΔGRF 2		Turns on reentrance mode (default)
11 ΔGRF 1	WHATSCALE	Returns scaling params as result; result is 4-vector (x_1, x_2, y_1, y_2) or 6-vector $(x_1, x_2, y_1, y_2, z_1, z_2)$
11 ΔGRF 2	WHATWINDOW	Returns the current window params (in inches) as a 4-vector (x_1, x_2, y_1, y_2)
11 ΔGRF 3	WHATCOORDS	Returns the most recent ⍁ input coordinates as a 2-vector (x, y)
11 ΔGRF 4	WHATCHAR	Returns the most recent ⍁input character as a scalar
11 ΔGRF 5,6,07	STRAPIS	Sets program to read 5, 6, or 7 characters on graphic input requests. Used to accommodate different "strappings" at the 4013. Default is 7.

Unscaled Graphic I/O

Ordinarily, the x, y (and possibly z) coordinates of the points given as arguments of SET and DRAW are interpreted in conjunction with current scaling and window parameters: Scaling determines where in the window a given point will lie, and the window parameters locate the window on the screen. If desired, however, one may circumvent these actions and deal with the screen as a "directly addressable" entity; in this mode no scaling or windowing is done (the "window" is the entire screen), and point locations are specified by x and y coordinates in raster-point units. The ranges of such coordinates are

$0 \le x \le 1023,$

$0 \le y \le 789.$

Unscaled Output

Output scaling may be killed by

SCALE ' '

This command establishes the "transparent scaling" (i.e., no scaling) rule, and sets the window to full-screen. Thereafter, point data supplied as right arguments of DRAW or SET commands should have coordinates in the ranges indicated above. Three-dimensional points will be projected on the xy plane, as usual; for 3D data, the x and y ranges are as given above, and the z-range is the same as the x range.

Ordinary output scaling may be resumed via

NOSCALE

or

CENTER

or

SCALE x , x , ...

A smaller window may be reestablished by

WINDOW x_1, x_2, y_1, y_2

or

(x_1, x_2, y_1, y_2) DRAW ...

the default window may be restored by

WINDOW ' '

Unscaled Input

Input scaling is turned off by

```
0 ΔGRF 1
```

This command does not alter the existing scaling and window parameters, but it does eliminate dependence on them by ⍞input. While in this mode, a character struck anywhere on the screen will be accepted, and its raster-point coordinates (in the ranges given above) will be those returned by WHATCOORD.

To resume ordinary input scaling (using the existing scaling and window information), input

```
0 ΔGRF 2
```

12. WORKSPACE MANAGEMENT FUNCTIONS

Xerox APL provides a set of intrinsic functions, ΔCR, ΔWM, and ΔTE to aid in user workspace management and display. The functions may be copied from workspace WSFNS or created as described in Appendix C.

Namelists and Linelists

The introduction of two concepts is useful to allow abbreviated descriptions of arguments and results associated with the following intrinsic functions. Namelists and linelists are particular forms of character arrays, useful for storing text data with minimal waste for blank padding.

A namelist is any text vector or matrix for which each row consists of one or more valid APL names separated by blanks, embedded carriage returns, or embedded line feed characters.

When namelists are created by the intrinsic functions, they will always be vectors. Individual names will be separated by embedded line feed characters. The purpose of interspersing line feeds is to avoid the arbitrary splitting of names in terminal displays of namelists. For this reason, it is advisable for users to create namelists while using the smallest width setting which will be encountered during their use.

Namelists which are character matrixes may be used as arguments of functions which call for namelists. In this case, each row of the matrix must satisfy the rules for a vector namelist. Matrix namelists have been allowed to satisfy circumstances in which users find it advantageous to build rectangular arrays of names.

A linelist is a text vector consisting of character strings separated by embedded carriage returns. In use, the first 'line' must consist of appropriate text for a user function header, and all subsequent 'lines' must consist of text which would be valid function lines. This includes line length limitations.

Line Feed and Carriage Return

The line feed character is hexadecimal 15 (or decimal, 21). Carriage return is hexadecimal 0D (or decimal, 13). On terminal output, either character generates the line feed-carriage return combination of physical action.

When APL input lines with open quotes are extended, the line separator used internally by APL is the line feed character. The carriage return was selected as the separator for linelists to distinguish this separator from the line feed character which occurs in open quote line extensions.

Error Reporting for Namelists and Linelists

If a function requires a namelist or linelist as an argument, the argument will first be checked for rank of 1 or 2 and character data type. DOMAIN ERR will be reported if these checks fail. Namelists are checked for the presence of forms other than valid APL names. If this test fails, DOMAIN ERR is reported.

Linelists may generate any of the errors which would be encountered in regular input of the individual 'lines' including LENGTH ERR for lines of excessive length.

Canonical Representation

The ΔCR intrinsic function is used for conversion of user functions to text form, for program-controlled function definition, and to lock existing functions. The intrinsic is created by ΔCR←14¯5.

Function to Text

```
R←1  ΔCR A
```

If A is not a text vector representing a valid name in APL, DOMAIN ERR is reported.

If A contains a name which does not represent a user-defined function in the dynamic environment, DEFN ERR is reported. This may occur, for example, if the name represents a function at global level but is currently shadowed by a local use as a variable. Functions which will use ΔCR should avoid the use of common names for dummy or local variables.

If no error is indicated, R is a linelist, that is, a text vector consisting of lines of the defined function with embedded carriage returns as separators. The text vector does not include opening or closing dels or line numbers (it is not the display form). If A is the name of a locked function, R is the empty text vector. The form of R is the internal APL character representation, in which valid overstrikes are mapped as single characters. This fact is noted for users of non-APL keyboards. Mnemonics are not created internally, so indexing by visual position will be misleading. Note that the separator for function lines is the carriage return (hexadecimal OD), which can be distinguished in editing operations from line feed.

Text to Function

```
R←2  ΔCR  LL
```

If LL is not a linelist, DOMAIN ERR results.

If any 'line' of LL exceeds the maximum length for APL input lines, LENGTH ERR is reported. If the linelist does not contain at least one line in addition to the function header line, DOMAIN ERR is reported. DEFN ERR is reported if the 'header' line is not in the proper format for a function definition or if the function name has an active referent which is not a user function or is a locked user function.

DEFN ERR also occurs if the body of the linelist includes constructs including the colon character, not in a quote string, other than valid line labels.

Errors which would give BAD CHAR or LINESCAN ERR in normal input are accepted and detected later as SYNTAX ERR during function execution.

If no errors occur, a user function, with the name specified by A, is created. If the new function name is currently a local symbol, the function will exist as a local entity.

R is a text vector indicating the name of the function created.

Locking Functions

```
R←3  ΔCR  NL
```

NL must be a namelist. For each name in NL, if the current referent is a function, it is locked. If not, the name is included in R.

R is a namelist consisting of any names in NL which were not current function names. If no such names were in NL, R is empty.

Workspace Management

The workspace management function, ΔWM←14¯T6, is a dyadic intrinsic function providing a variety of operations described below.

Expunge, Local (Active)

```
R←1  ΔWM  NL
```

NL must be a namelist. The active referents of names found in NL are erased. R is a namelist of any names for which referents were found but not erased (e.g., pendant functions). Note that a current active referent may be (and often is) the global referent.

Expunge, Global

R←2 ΔWM NL

Same as 1 ΔWM NL except that only global referents of names are erased.

List Workspace Named Items

R←3 ΔWM I

The value of I must be an integer from 1 to 6. R is a namelist. The entities named depend on I.

I	Category Listed
1	Labels.
2	Active variables.
3	Active functions.
4	Groups.
5	Global variables.
6	Global functions.

List Elements of a Group

R←4 ΔWM A

A must be a text vector containing one name. If A is not the name of a group, DOMAIN ERR is reported.

R is a namelist with names of the elements grouped by A.

List Workspace Parameters

R←5 ΔWM I

The value of I must be an integer from 1 to 8. R depends on the value of I.

I	R
1	WSID as text vector.
2	State indicator as text vector with embedded line feeds.
3	Origin as integer.
4	Width as integer.
5	Digits as integer.
6	Tabs as integer scalar or vector.
7	Symbol table size.
8	Number of symbols still available.

Identify Local Use of Names

R←6 ΔWM NL

The namelist, NL, is scanned for current use of the names. R is a numeric vector. Values are as indicated.

0	No current referent.
1	Logical variable.
2	Character variable.
3	Integer variable.
4	Real variable.
5	Index sequence. (This is an integer vector with equally spaced elements. It is functionally an integer variable; but is compressed in storage.)
6	List (used by ΔFMT, ΔTE).
7	Label.
8	User-defined function, niladic, no result.
9	User-defined function, niladic, with result.
10	User-defined function, monadic, no result.
11	User-defined function, monadic, with result.
12	User-defined function, dyadic, no result.
13	User-defined function, dyadic, with result.
14	Intrinsic function, dyadic.
15	Intrinsic function, monadic.
16	Intrinsic function, niladic.
17	Group.

Note: Intrinsic functions are analogous to locked user functions in that they cannot be displayed.

Identify Global Use of Names

R←7 ΔWM NL

Similar to 6 ΔWM except that global use of names is indicated.

List Storage Requirements for Named Active Items

R←8 ΔWM NL

NL is a namelist. R is a numeric vector. NL is scanned for active referents of names. If there is no active referent for a given name, that element of R will be 0. Each element of R is the number of bytes of workspace occupied by the active referent.

List Storage Requirements for Named Global Items

R←9 ΔWM NL

NL is a namelist. R is a numeric vector. NL is scanned for global referents of names. If there is no global referent, that element of R will be 0. If a name is a group, only the group overhead is listed (to get space requirement for the members of a group, use 9 ΔWM 4 ΔWM G, where G is the group name). Each element of R is the number of bytes of workspace occupied by the global referent of the corresponding name.

Note: Storage requirements cited by use of 8 ΔWM or 9 ΔWM do not include the storage required by long names. Such names may be shared by local and global referents and are thus not unambiguously accountable.

Text Editing

The text editing function, ΔTE←14$\bar{1}$7, provides five capabilities, described below, to facilitate the examination and modification of text variables in APL.

APL has suffered a limitation, in handling text, in that most primitive functions work on single characters and that extensive text variables must be managed as rectangular arrays. This poses problems in wasted space and, much worse, a very awkward form for modification. In a rectangular array, an edit operation cannot alter the length of any row without altering the lengths of all rows by the same amount.

The use of embedded carriage returns in text vectors is a solution allowing better packing of text variables and making text substitutions of unequal length a more straightforward operation. ΔTE facilitates use of such text vectors.

Text Index Function

R←1 ΔTE L

L is a 'list' with two elements.

L ← (TV;DV)

TV may be any text vector.

DV is a text scalar or vector of 'delimiters'. Typically, delimiters will be blanks, carriage returns, line feeds, commas, or combinations thereof. Any character may be used as a delimiter.

R is an n by 2 mumeric matrix. Each row contains the index and length of a string of non-delimiter characters in TV. The values of column 1 of R are ORIGIN dependent. R is null if TV is a null text vector or includes only delimiter characters.

Example:

TV is a namelist, NL. DV is DV←' ', LF, where LF is the line feed character and the quotes enclose a blank.

R←1 ΔTE (NL;DV) provides the indices and lengths of the names in NL.

DOMAIN ERR is reported if L is not a two-element list, with text elements. LENGTH ERR is reported if DV is an empty text vector. RANK ERR is reported if TV is a text scalar.

Substring Search

R←2 ΔTE L

L must be a list with 2, 3, or 4 elements

R←(TV;SS)

TV may be any text vector.

SS may be any text scalar or vector not longer than TV.

L←(TV;SS;FCOL)

FCOL may be any integer scalar value such that FCOL + (number of elements in SS) -1 is less than or equal to the highest index value of TV. FCOL may also be null. FCOL indicates the first column in TV at which search is to start or

L←(TV;SS;FCOL;LCOL)

LCOL may be any integer scalar value less than or equal to the highest index value of TV and greater than or equal to FCOL-1 + (number of elements in SS). LCOL is the last column of TV involved in the search.

R is a numeric vector with the beginning indices of occurrences of SS in TV, starting at position FCOL and ending at LCOL. If there are no occurrences, R is empty. 'Occurrences' may not overlap; that is, successive values in R are always separated by the length of SS.

Example:

TV is a namelist, NL, whose length is 100. SS is SS←'BOB'.

R←2 ΔTE (NL;SS) provides the starting indices of all occurrences of 'BOB' in NL.

R←2 ΔTE (NL;SS;30) provides the starting indices of all occurrences of 'BOB' in NL from NL[30] to the end of NL.

R←2 ΔTE (NL;SS;;30) provides the starting indices of all occurrences of 'BOB' lying within the subset of NL from the beginning through NL[30].

R←2 ΔTE (NL;SS;30;60) provides the starting indices of all occurrences of 'BOB' lying within the subset of NL from NL[30] through NL[60].

DOMAIN ERR is reported if L is not a 2, 3, or 4 element list with TV, a text vector and SS text scalar or vector, or FCOL and LCOL not integer values.

LENGTH ERR is reported if the length of SS is greater than the length of TV, or FCOL and LCOL are not scalars.

INDEX ERR is reported if the length of SS is greater than the length of TV with FCOL or LCOL specified, or FCOL and LCOL specify indices outside the range of TV.

RANK ERR is reported if TV is a scalar.

Since R is a vector of indices, it is ORIGIN dependent. FCOL and LCOL are index positions of TV and, therefore, are ORIGIN dependent.

Substring Search and Replacement

R←3 ΔTE L

L must be a list of 3, 4, or 5 elements.

L←(TV;SS;RS)

TV may be a text scalar or vector.

SS may be a text scalar or vector not longer than TV.

RS may be any text scalar or vector not longer than 255 elements, including the empty vector.

R is a text vector formed by replaced occurrences of SS, in TV, by RS. Replacement is on a non-overlap basis.

or

```
L←(TV;SS;RS;FCOL)
```

FCOL may be any integer scalar value such that FCOL + (number of elements in SS) -1 is less than or equal to the highest index value of TV. FCOL may also be null.

or

```
L←(TV;SS;RS;FCOL;LCOL)
```

LCOL may be any integer scalar value less than or equal to the highest index value of TV and greater than or equal to FCOL-1 + (number of elements in SS).

Example:

NL is a namelist with the blanks and embedded carriage returns. Several names may print on one line. BLANK has been assigned the blank character, and CR the carriage return.

```
VNL←3  ΔTE (NL;BLANK;CR)
```

VNL has all blanks replaced by CR, and prints one name per line.

```
VNL←3  ΔTE (NL;BLANK;CR;20)
```

VNL has all blanks from NL[20] to the end of NL replaced by CR.

```
VNL←3  ΔTE (NL;BLANK;CR;20;30)
```

VNL has all blanks from NL[20] through NL[30] replaced by CR.

```
VNL←3  ΔTE (NL;BLANK;CR;;30)
```

VNL has all blanks from the beginning of NL through NL[30] replaced by CR.

```
VNL←3  ΔTE (NL;BLANK;NULL)
```

VNL has all the blanks in NL removed. (NULL is the empty vector.)

DOMAIN ERR is reported if L is not a 3, 4, or 5 element list with TV, SS, and BS text elements, or FCOL and LCOL not integer values, or the length of RS is greater than 255.

LENGTH ERR is reported if the length of SS is greater than the length of TV, or FCOL and LCOL are not scalars.

INDEX ERR is reported if the length of SS is greater than the length of TV with FCOL or LCOL specified, or FCOL and LCOL specify indices outside the range of TV.

Substring Replacement (Without Search)

```
R←4  ΔTE  L
```

L is a list with 4 elements.

```
L←(TV;RS;FCOL;LCOL)
```

TV must be a non-empty text vector.

RS must be a text vector or scalar. It may be empty.

FCOL must be an integer scalar representing a valid index of TV.

LCOL must be an integer scalar representing a valid index of TV. LCOL must be greater than or equal to FCOL.

R is formed by replacing that portion of TV bounded by FCOL and LCOL by the string RS. If RS is empty, this constitutes deletion of a specified subset of TV.

FCOL and LCOL are index positions of TV, thus ORIGIN dependent.

DOMAIN ERR is reported if L is not a 4-element list, if TV is not a non-empty text vector, if RS is not a text vector or a scalar, or if FCOL and LCOL are not integer scalars.

INDEX ERR is reported if FCOL and LCOL are not value indices of TV, with current ORIGIN setting, or if LCOL is less than FCOL.

Example:

```
      ORIGIN is 1
      TV←'THE PRICE OF PRODUCT-NAME SHOULD BE'
      RS←'WHEATIES'
      R←4 ΔTE (TV;RS;14;25)
      R
THE PRICE OF WHEATIES SHOULD BE
```

String Comparison

R←5 ΔTE L .

L must be a list with two elements:

L←(A;B)

A and B must be text vectors of length 1 to 255 characters or text scalars. (Text scalars are treated as one-element vectors.)

R is a two-element numeric vector describing the comparison of A and B. Comparison is based on the collating sequence of Figure B-1, Appendix B, which is the EBCDIC collating sequence as modified to support the full APL character set.

The first element of R indicates which element of L should be first in left to right sorted order.

0 means the text vectors are identical.

1 means A should sort first.

2 means B should sort first.

The second element of R indicates the lowest index position i at which A[i] and B[i] differ.

If A and B are identical, the second element of R is -1. Thus R is 0 ¯1.

If B is longer than A but each A[i] = B[i], that is B differs from A only by being longer, then A is considered first in sorting order and R is 1 ¯1.

If A is longer than B, but each B[i] = A[i] then R is 2 ¯1.

DOMAIN ERR is reported if L is not a two-element list or if A or B is not a text vector or scalar.

LENGTH ERR is reported if A or B has a length of less than one or more than 255.

APPENDIX A. ERROR MESSAGES

Table A-1 is an alphabetic listing of possible APL error messages. The first column contains the message and the second column contains explanatory details and recovery procedures where appropriate. The effects of error detection on APL processing are described in more detail in Chapter 10; see also "Sidetracking on Errors and Breaks" following Table A-1.

Table A-1. Error Messages

Message	Description
name NOT COPIED	The item named has the same name as a pendant function in active workspace.
name NOT ERASED	The item named in an)ERASE command was not erased because it was a pendant function.
name NOT FOUND	The item named in a)COPY command was not found (the item may have been a local variable).
ABORTED BY BRK OR CTRL-Y	An enqueue request has been aborted by user (pertains to shared APL indexed files).
BAD CHAR	A bad input character was detected. This is usually the result of a transmission error or the input of an illegal overstrike. In the case of nonstandard I/O devices, the message can also indicate the input of a character which is "illegal" for that device.
BAD COMMAND	An improper command construct was detected.
BAD FILE REF	A bad reference to an existing file name was made during a)SAVE command. This could occur, for example, if the workspace name specified in the)SAVE referenced some existing workspace that was protected by a password. The)SAVE should be respecified using a different workspace name. This message will also result on)CONTINUE if a passworded CONTINUE workspace already exists.
COMP. ALREADY HELD	An attempt was made to enqueue a record already enqueued in a shared APL indexed file.
COMP. NOT HELD	An error was detected on attempting to dequeue an unheld record in a shared APL indexed file.
DAMAGED WS, ERROR TYPE: *n*	Workspace damage has been detected in executing)LOAD,)SAVE, or)COPY. In any case, the filed version of the damaged workspace exists. The error type is a number indicating the type of damage detected. This message, and the name and account of the damaged workspace, should be reported to your time-sharing service. The information may be used by Xerox to correct an error in the APL processor. If the error occurred during a LOAD or SAVE, a COPY of the damaged workspace may allow retrieving the global data and functions.
DEFN ERR	This message is output for any sort of error in function definition, such as a misplaced del symbol (∇), improper syntax of a function header as a result of header editing, or an attempt to edit a pendant function.

Table A-1. Error Messages (cont.)

Message	Description
DMS ERR	An error was detected during a DMS operation in a system supporting interface between APL and EDMS data bases.
DOMAIN ERR	The indicated argument is of the wrong type or out of the proper range for this specified operator or for the other argument. Examples are character data input for a numeric operation, or numbers input for a logical operation which do not reduce to 0 or 1. See the domain tables in Chapter 5 for examples of acceptable types of argument data for each APL operator.
ENQ FULL	The CP-V Enqueue stack is full (pertains to shared APL indexed files).
ENQ UNAUTHORIZED	User is not authorized for Enqueue operations on APL shared files.
FILE ACCESS ERR	This file I/O error often means a password is missing or is inaccurate.
FILE DAMAGE	This file I/O error indicates some damage, but not necessarily to every record or component in the file. Recovery is often possible by copying undamaged material to a new file, replacing damaged items.
FILE IN USE	The file named in a)SAVE command is currently in use, i.e., another user may be simultaneously executing a load of that file. Since this situation is a momentary timing conflict, the user should retry the command after a short wait. This type of timing conflict may also occur when exercising file I/O.
FILE INDEX ERR	This file I/O error may mean that an index (record identifier, sometimes called a key) is incorrect, or an attempt has been made to read beyond the limits of a file.
FILE I/O ERR xxxx	This is a general file I/O error message. The "xxxx" stands for a hexadecimal error or abnormal condition, with the first pair of digits indicating a code and the second pair a subcode. A code 00 indicates that APL has detected an error (see Appendix B, File Input/Output). Other codes indicate errors detected by the monitor and correspond to I/O error codes shown in monitor reference manuals, for example the CP-V Reference Manual, 90 09 07.
FILE NAME ERR	This file I/O error may mean that a file identifier is improperly formatted, an attempt has been made to use a file that doesn't exist, or an attempt has been made to create a file that already exists.

Table A-1. Error Messages (cont.)

Message	Description
FILE SPACE TOO LOW	Either the user's or the system's file granule limit is surpassed. This can occur when workspaces are being saved or during file I/O operations. Recovery is usually possible; the user drops unneeded files from his account and retries the aborted step.
FILE TBL FULL	This file I/O error means that the maximum permissible number of files have been "tied" (designated).
FILE TIE ERR	This file I/O error may mean either that the file has not yet been tied (designated to an input or output stream), or that the file being tied has already been tied, or that an attempt has been made to write into a file owned by another user.
FORMAT SYNTAX ERR	A syntax error was detected in the left argument of a ΔFMT expression. See Chapter 9 for an explanation of correct ΔFMT syntax.
I/O ERR	This message indicates that an irrecoverable system I/O error occurred and an error exit has been made from the APL processor. A system I/O error should be reported to the user's field representative along with the conditions under which it occurred (see also SYS ERR).
I/O ERR xxxx	If blind I/O was being used, this message indicates that the requested blind write could not be executed for some reason. The user may retry the I/O or otherwise continue operation. This general I/O error includes hexadecimal error or abnormal condition "xxxx". The first pair of digits indicates a code and the second pair a subcode. These correspond to I/O error codes shown in the Xerox CP-V reference manuals.
INDEX ERR	The index subscript specified in an expression is out of the range of the particular array to which it is applied. For example, if A is a four-element vector, the expression A[6] would generate an INDEX ERR since the requested sixth element does not exist.
LENGTH ERR	The length(s) of the indicates argument(s) are not conformable or are incorrect for the operator used. For example, the expression 9 7 8 + 5 3 results in a LENGTH ERR because the two vectors do not have the same number of elements.
LINESCAN ERR	An obvious error in form (leading right bracket, misplaced colon, line ending with an operator, etc.) was detected in the scan of a line input for execution or function definition. No part of the line is executed. On direct input, the user is prompted with the partial line for correction unless the device is nonstandard, in which case he must retype the entire line. In function definition mode, the incorrect line is entered as part of the function and may then be replaced or edited.

Table A-1. Error Messages (cont.)

Message	Description
NO RESULT	A user-defined function which generated no result was used in a syntax which requires a result.
NOT APL FILE	This file I/O error means that a component read failed because it did not have the structure required by APL.
NOT ENQ SYSTEM	An attempt was made to use shared file enqueue feature, which is not supported by the local installation.
NOT GROUPED	The group name specified in a)GROUP or)GRP command already references an item which is not a group. A different group name must be used.
NOT SAVED, THIS WS IS name	If "name" = CLEAR WS, either there is nothing to save or the SAVE command did not specify a name for the saved workspace. Otherwise, the save command named an existing saved workspace when the active workspace name is different. Change the name or drop the saved workspace.
OPEN QUOTE	The Execute operator has been used on an argument that has an odd number of quotes before the end of line (or first embedded carriage return).
PRIVATE PACK UNAVAIL, CALL OPR.	This file I/O error usually means that the user has referenced a private disk pack that has not yet been mounted by the computer operator. A message should be sent to the operator to mount the correct disk pack.
RANK ERR	The rank of the indicated argument is incompatible with the operator or with that of the other argument.
REQ. WOULD CREATE DEADLOCK	An enqueue request has been made, pertaining to APL shared files, which, if honored, would create a deadlock stopping further activity of two or more users.
SEALED WS	Attempt was made to save or display functions of a sealed workspace.
SI DAMAGE	A suspended function has been erased or replaced, and the state indicator has been modified to delete all calls to it from its active list. This may occur in function definition, or upon execution of an)ERASE or a)COPY command.
SI DAMAGE WILL RESULT: TYPE'GO'TO CLOSE	This warning message is output in function definition when the header of an existing active function is changed. It indicates that references to this function in the state indicator list will be damaged if the header change is implemented. In order to avoid SI damage, the user may restore the header to the old form or change the function name in the header before closing the function. If he wishes to close the function and accept the SI damage, he may do so by typing GO in reply to the warning message.
SING. MATRIX	The right argument of a matrix divide operation (⌹) is a singular matrix, i.e., it has no inverse.

Table A-1. Error Messages (cont.)

Message	Description
SYM TBL FULL	The internal symbol table is full. This may occur during execution, in function definition, or because of a)GROUP command. If the user wishes to extend the symbol table at this point, he must)SAVE the workspace, issue a)CLEAR command, use the)SYMBOLS command to request an extended allocation, and then issue a)COPY, not a)LOAD, to restore the workspace. Further information on the use of the)SYMBOLS command may be found in Chapter 8.
SYNTAX ERR	Improper syntax was detected in the executed line. Examples of improper syntax are unbalanced parenthesis, two variables not separated by an operator or by a function, or an attempt to assign a value to a label.
SYS ERR	An irrecoverable system error of indeterminate origin has occurred and an error exit has been made from the APL processor. If APL is reaccessed, it starts again with a clear workspace and should operate correctly unless the conditions which led to the SYS ERR reoccur. Error should be reported to field representatives with accompanying data.
TOO BIG	A)COPY command accessed more material than would fit in the current workspace; no items were copied.
TOO BIG TO LOAD	The workspace specified in a)LOAD command was saved by a user with larger memory allocation than the present user, and there is insufficient space for the workspace to be loaded (in some cases it cannot even be copied). See also the description of the)COPY command, Chapter 8.
TOO MANY SYMBOLS	The symbol table would have overflowed if the requested)COPY command were performed; no items were copied.
TRAP	Similar to SYS ERR except error was detected via machine trap rather than by the APL system. See description of SYS ERR, above.
TRUNCATED INPUT	The input line was too long.
UNDEFINED	The indicated symbol has not been assigned a value.
WRONG TERMINAL	The user has attempted to perform graphics input, graphics output, or graphics service (△GRF) with a terminal type not appropriate to graphic I/O. Terminal type 13 should be used for Tektronix 4013 or 4015. Terminal type 3 should be used for Tektronix 4010.
WS FULL	The active workspace is full. This may occur during execution, in function definition, or because of a)GROUP command. Depending on his particular situation, the user may choose to)ERASE unneeded objects from the workspace, clear the state indicator, or)CLEAR the entire workspace in order to free up space.
WS NOT FOUND	The workspace file specified in a)LOAD or a)COPY command was not found.

Sidetracking on Errors and Breaks

In some APL applications, the programmer would like to bypass APL's standard error and break procedure. He may, for instance, wish to substitute his own messages or institute corrective actions. Computer-assisted learning programs and commercial business aids are applications where this may be desired. Users of such applications may have little knowledge of APL, and messages such as DOMAIN ERR or WS FULL frustrate rather than help.

Xerox APL has been enhanced to allow the programmer to overcome this problem. This enhancement is called "sidetracking"; also the term "error control" is sometimes used (with an understanding that break control is included).

Suppose a DOMAIN ERR has been detected by APL. Under the enhancement, Xerox APL starts looking back through the state indicator searching for active functions for which the programmer has designated sidetracking. If a sidetracking function wants control over DOMAIN ERR, then APL sidetracks (branches) to the line in the function specified for DOMAIN ERR. If no active function wants such control, then APL issues the standard diagnostic message.

Sidetracking is both flexible and dynamic. Different errors can be sidetracked to distinct lines of a function. Certain sidetracking functions may control some errors while other sidetracking functions control others. Sidetracking functions can also compete for control of the same error. In this case, the most recently invoked function gets control, and its competing predecessors never become aware that the error occurred. Sidetrack specifications can be changed at will. They can be turned on and off, the error selection can be altered, and the sidetrack branches can be changed; the application program itself can modify sidetracking specifications throughout its execution. This capability permits a simple or comprehensive treatment at the programmer's discretion.

Table A-2 shows errors that are subject to sidetracking. Errors not listed in the table include:

SYS ERR, **TRAP**, *BAD WS*, and (irrecoverable) I/O ERR.

Since recovery from these errors is impossible for an applications program, APL retains exclusive control. See the discussion following Table A-2 for details concerning certain unique errors.

Associated with each item in Table A-2 is an error number. Error numbers are informally grouped by common classifications: statement execution errors, input-translation errors, command errors, file input/output errors, etc. Gaps are provided in the error number sequence to accommodate future diagnostics. It should be pointed out that in the case of input/output errors two conditions must exist. One, the error must be recoverable. Two, APL must acknowledge the error – if a file I/O subsystem uses the file primitive `14⍑2`, APL will acknowledge file I/O errors. If file primitive `14⍑1` is used, however, the error data is passed on to the file I/O subsystem (as a scalar integer), and APL will neither display a standard diagnostic nor attempt a sidetrack.

The items in Table A-2 contain four cases in which APL gives up control somewhat grudgingly:

SI DAMAGE
name NOT COPIED
name NOT FOUND
name NOT ERASED

These cases, in which command processing or function definition is in effect, must reach an orderly conclusion to avoid workspace damage. Therefore, APL unilaterally displays the messages and proceeds to conclusion. (Nevertheless, sidetracking is still possible, and an application program might issue explanatory messages after the APL messages, as one alternative.) As APL proceeds in these four cases, a series of such messages could be displayed (but this would be unusual). APL permits sidetracking only with regard to the latest error known at the conclusion of this kind of processing. Using the execute-operator, for example, suppose a)COPY command occurs while sidetracking is in effect. Suppose also that some object, X, is missing – X NOT FOUND is displayed; furthermore, suppose that the data found will not fit in the active workspace. Then the)COPY command concludes without copying anything and would ordinarily issue a TOO BIG message. Sidetracking would then apply to this example to the TOO BIG error and, the NOT FOUND error would be "forgotten".

Note in the foregoing example that copying was attempted by means of an execute-operation. This was a necessity. A function can obtain sidetracking only while it is actively in execution. Thus command and function definition errors can be sidetracked only when the function (or some function invoked by the sidetracking function) actually executes a command or function definition. Evaluated-input might seem to provide another way in which a function could, indirectly, invoke command or function definition activity. However, for the reason given below, evaluated input is not considered capable of being sidetracked (except while that input has itself invoked a sidetracking function).

Table A-2. Items Subject to Sidetracking

Error Number	Standard Error Message or Break	Error Number	Standard Error Message or Break
1	WS FULL	46	TOO BIG
2	SYNTAX ERR	47	TOO MANY SYMBOLS
3	UNDEFINED	48	name NOT COPIED
4	DOMAIN ERR	49	name NOT FOUND
5	RANK ERR	50	name NOT ERASED
6	LENGTH ERR	51	NOT GROUPED
7	INDEX ERR	55	COMP. NOT HELD
8	NO RESULT	56	COMP. ALREADY HELD
9	SYM TBL FULL	59	ABORTED BY BRK OR CTRL-Y
15	SING. MATRIX	61	REQ. WOULD CREATE DEADLOCK
16	FORMAT SYNTAX ERR	62	ENQ FULL
20	BAD CHAR	64	ENQ UNAUTHORIZED
21	LINE SCAN ERR	65	NOT ENQ SYSTEM
22	TRUNCATED INPUT	70	FILE SPACE TOO LOW
23	OPEN QUOTE	71	FILE I/O ERR xxxx
30	I/O ERR xxxx	72	FILE DAMAGE
31	WRONG TERMINAL	73	FILE NAME ERR
35	DEFN ERR	74	NOT APL FILE
36	SI DAMAGE	75	FILE TBL FULL
40	BAD COMMAND	76	FILE ACCESS ERR
41	NOT SAVED, THIS WS IS name	77	FILE TIE ERR
42	FILE IN USE	78	PRIVATE PACK UNAVAIL, CALL OPR.
43	BAD FILE REF	79	FILE INDEX ERR
44	WS NOT FOUND	99	DMS ERR
45	TOO BIG TO LOAD	100	Break

In Xerox APL, the execute-operator makes it possible to execute via quote-quad input anything that could be entered via evaluated-input. Quote-quad input has an advantage from the standpoint of error recovery. The input text can be assigned to a variable before it is executed. Thus, a sidetrack function can analyze this text to determine correct recovery action. Evaluated-input is not susceptible to this analysis; it is immediately interpreted by APL.

Setting Sidetracks

A sidetrack setting resembles setting a stop vector. The similarity is so strong that it must be noted that they are totally independent of one another. The syntax for setting sidetracking in a function is

S∆name ← table

where:

name is the name of the function and

table is the 2-column matrix in which the first column contains line numbers and the second contains error numbers.

The function must be defined when this syntax is executed; it could be a statement in the function. Sidetracks can be set even when a function is locked.

In effect, the sidetrack table becomes part of the function's definition and is copied or loaded if the function is copied or loaded. Function editing has no influence on the sidetrack setting. Since the sidetrack table contains line numbers, the following precaution should be observed. If editing a sidetracking function alters the position of a line specified by the sidetrack table, a correct setting must be reissued. This is necessary to ensure that the proper line will be branched to if the sidetrack does take place.

Erasing a sidetracking function erases its sidetrack setting. A sidetrack setting can also be removed by being replaced with an empty matrix, as in the following example.

```
S∆FUN←0   2⍴0
```

Note: Assigning an empty vector removes the stop vector, but not a sidetrack setting. Only matrix assignments affect sidetracking.

When a (nonempty) table is assigned for sidetracking, it consists of one or more rows. Each row contains a pair of integers — a line number and an error number. The line number designates which line of the function is to be sidetracked to (branched to) if the indicated error occurs. The following sample sidetrack setting specifies a branch to line number 9 in case of a DOMAIN ERR (error 4 in Table A-2).

```
S∆FUN←1   2⍴9   4
```

A new sidetrack setting for a function entirely replaces any previous setting. Thus, the following example would remove FUN's control over DOMAIN ERR.

```
S∆FUN←2   2⍴9   3   9   2
```

In this example, FUN sidetracks to line 9 for UNDEFINED or SYNTAX ERR. This illustrates that a sidetrack table can contain duplicate line numbers; however, it is useless to duplicate an error number in the same table. Only the first such number would be effective.

In the above example, FUN sidetracks on only two of the possible errors. If other errors occur, APL handles them in the standard manner unless some other function has specified sidetracking for those errors.

A special error number, 0, exists for sidetracking on all items in Table A-2 except the break (number 100). In the following example, FUN sidetracks:

- to line 8, if a break is detected
- to line 7, if WS FULL occurs
- to line 9, for any other error subject to control.

```
S∆FUN←3   2⍴8   100   7   1   9   0
```

Error number 0 should always be the last number in the table, anything after it would be ignored.

Breaks are sidetracked only if the sidetrack explicitly includes error number 100.

The following example shows a compact way of setting several different error numbers to the same line:

Suppose ERRLAB is the label of the desired line and sidetracking is set within the function FCN containing ERRLAB.

```
SΔFCN←ERRLAB, [1.5]2 3 5 8 21
```

sets the indicated error no sidetracks to ERRLAB (see "Lamination").

The above examples illustrate how to set sidetracks. This does not imply that the function (FUN) immediately receives control if an error occurs. If FUN is not actively in execution, its sidetracking is disregarded. Even if FUN is being executed, it may still not be given control. The error may have occurred in evaluated-input, or FUN may have called another function which has a competing sidetrack.

The Dynamics of Sidetracking

A step-by-step outline reveals significant aspects of sidetracking dynamics. Assume a controllable error or break has occurred and APL is ready to check for sidetracking.

Step 1: APL designates and saves the current error number, replacing any previously recorded error number. For the moment, it initializes the error location to be line zero and an empty function name. APL points to the top (latest) entry in the state indicator.

Step 2: The state entry is examined. If it is a pendant function, APL proceeds to Step 3. If it is an execute-operation state, APL points to the next entry and repeats Step 2. Otherwise, sidetracking is not applicable; so APL issues the standard diagnostic.

Step 3: (Pendant function state.) The error location is tested. If still initialized (see Step 1), the line number and name of the pendant function are recorded. The function's definition is tested for sidetrack setting. If it features some sidetracking, APL proceeds to Step 4. Otherwise, APL points to the next state indicator entry and repeats Step 2, attempting to find a function with a sidetrack setting.

Step 4: (Sidetrack setting present.) The sidetrack table is tested sequentially versus the recorded error number. If a match is found or the table has a zero error number, APL proceeds to Step 5. Otherwise, APL points to the next state entry and repeats Step 2, attempting to find a function interested in the current error.

Step 5: (Sidetrack acknowledged.) APL removes from the state indicator any entries it bypassed in reaching Step 5. This puts the sidetracking function at the top of the state indicator. APL then branches to the specified line number (which may or may not actually exist in the function).

Considerations After Gaining a Sidetracks

Once APL performs a sidetrack, it has no further interest in handling the break or error. Responsibility falls to the application programmer, depending on the line number dictated and statements supplied for the sidetracking function. Caution is advised.

If a mistake occurs in statements entered via a sidetrack, a new error may confuse the intended recovery procedure. It is possible for that statement to generate the error being considered, leading to the same sidetrack, the same mistake, and so on indefinitely. WS FULL can be particularly troublesome. In some cases, the statement reached by the sidetrack will itself cause another WS FULL. There is no general solution to this potential problem, but it is a rare difficulty for two reasons. First, intermediate results may be discarded after any error, freeing up sufficient workspace for recovery. Second, more workspace may become available if state indicator entries are removed in reaching the sidetrack (see Step 5 above).

The application programmer should also be cognizant of the monitor's role in handling breaks. If the monitor detects four consecutive breaks between two terminal input requests, it abandons APL processing and enters the TEL subsystem. Thus it is recommended that when sidetracking on breaks, some input from the user should be requested. This clears the monitor's break count before the threshold is reached.

Aids for Sidetrack Users

Three niladic intrinsic functions have been added to APL which are of particular interest after a sidetrack has occurred:

```
ERRN←16⊤0
ERRF←16⊤1
ERRX←16⊤2
```

ERRN produces a 2-element integer vector; the first element is the latest error number recorded and the second is the line number for the error location (see Step 3 above). ERRF produces a text vector containing the name of the function for the error location (it might not be the same as the sidetracking function). Initial values are zero for error number and line number, and the function name is an empty vector. These are re-initialized by a)LOAD or)CLEAR command.

ERRX produces a 4-character text vector containing the latest I/O "error" information available to APL. These characters are hexadecimal digits. They represent input-output error (or abnormal) codes and subcodes as shown in monitor reference manuals or, in the case of file I/O, APL file I/O error subcodes. The latter case is distinguished by two zero characters in the "code" field (i.e., the first two of the four characters).

Note: APL sometimes expects error or abnormal I/O codes. Thus, the value reported when using ERRX does not necessarily indicate erroneous processing.

APPENDIX B. NONSTANDARD INPUT/OUTPUT

Standard and Nonstandard Devices

As defined in Chapter 3, standard APL input/output refers to the use of the APL typeball with the IBM 2741, or a functionally equivalent terminal such as the DATEL 20-31; the DURA 1021 and 1051; the NOVAR 5-50; and the TST 707. Without attempting a truly comprehensive I/O capacity, APL nevertheless accommodates certain other devices. These include Tektronix 4013, Teletype Model 33, non-APL equipped 2741 terminals for on-line processing, and card reader format input or line printer format output for batch processing.

Blind I/O and File I/O are also discussed in this appendix since they are considered to be modes of nonstandard I/O.

Using Nonstandard Input/Output

Nonstandard input/output provides the user with a set of substitute characters and mnemonics as replacements for APL characters which are either illegal or missing on his particular input or output device. In some cases the substitution is one-to-one (APL ⍺ is input or output as @ on nonstandard devices), but most substitutions for APL symbols involve mnemonics of one, two, or three characters preceded by a dollar sign key. Standard APL operators and their substitute character mappings for each nonstandard device are listed in Table B-3 at the end of this Appendix.

Terminal Declaration

The character mapping to be used for a particular I/O device is determined by the user's terminal declaration. This declaraction is made by means of the system command)TERMINAL n where n can have the following values:

1 - 2741 (or equivalent) with APL typeball.

2 - 2741 (or equivalent) with non-APL typeball.

3 - Teletype Model 33 (or equivalent).

4 - Line printer format output and card reader format input.

13 - Tektronix 4013 or other typewriter-paired APL/ASCII terminal.

14 - APL/ASCII bit-paired mapping terminals.

Type 1 is the standard terminal. In on-line runs the APL processor assumes a default terminal declaration of Type 1 if no declaration is made by the user. Type 2 typically features both uppercase and lowercase letters, although special-purpose typeballs are also available. These can be used, but translation correspondences are the responsibility of the user. Type 3 indicates Teletype Model 33, but this declaration may be used for compatible terminals, such as Teletype Models 35 and 37 (restricted usage). Type 4 is primarily applicable to batch jobs and is the default declaration in CP-V batch runs. Type 13 is strictly applicable to the graphics terminal, Tektronix 4013, and must not be declared for any other device.

If the input/output device is nonstandard, the user should make the proper terminal declaration immediately after calling APL. He may then proceed with normal APL processing, using substitute characters and mnemonics as required for his particular device.

Changing Terminal Declaration

The importance of making the correct terminal declaration should be evident to the user. In addition to character mapping differences among the various devices, certain operational idiosyncracies exist for each device, as will be discussed later. A false terminal declaration will very likely result in output discrepancies. However, new terminal declarations are acceptable at any time during an APL run as long as the user can tolerate the consequences. One example of these consequences is error message discrepancies. A diagnostic message (e.g., LENGTH ERR) will usually display correctly, but the offending APL statement may appear badly garbled – in fact, some characters may even be omitted. For the sophisticated user, however, there may be legitimate applications in which a false terminal declaration can be useful. The following discussion of translation processes is intended to aid this type of user.

Input/Output Translation

It is vital that the user correctly identify his terminal to the CP-V communications handler at log-on time. All input and output, whether standard or nonstandard, is ultimately constrained by the communications handler and the input/output equipment. The initial translation on input and the final translation on output are always made by CP-V according to the code set corresponding to the type of terminal indicated at log-on time. In the case of 2741 or equivalent terminals, this translation is complicated by the fact that two fundamental code sets are available for these terminals – Selectric® and EBCD. In addition there are APL and non-APL versions for both of these code sets. Unless deceived, the communications handler reconciles the different code sets; otherwise the confusion between EBCD and Selectric codes can be a considerable problem. Whatever the device, this level of translation will occur regardless of the terminal declaration applied by APL. Thus APL exercises only second-level control in the translation process.

The actual effect of the APL terminal declaration is to cause translation tables called Input/Output Mapping Tables to be brought into memory. The particular set of tables is determined by the)TERMINAL command. Each character input from the communications handler is then translated according to the resident Input Mapping Table. Certain one-character translations are made at this point – converting an @ to APL internal α for Type 2, 3, and 4 tables for example.

Illegal characters and certain key characters will be detected by APL for any terminal declaration. The backspace key character indicates that overstrikes follow, and the $ key character indicates that mnemonic equivalents to APL characters may follow. Once an overstrike has been accumulated, it is compared to a table of valid overstrikes. This either results in the proper translation to internal APL form or the detection of an illegal overstrike character. When a $ is detected as a single input character, the next three characters are acquired if available. These are compared to a table of three-character mnemonics. If no match is found, the rightmost test character is ignored and the remainder is compared to a table of two-character mnemonics. Again if no match is found, only the character following the $ is used for comparison to a table of one-character mnemonics. If a match occurs in any of the above comparisons, the proper translation to internal form takes place. If no match is found, the $ itself is translated to internal APL form. Thus, if a sequence of characters following a $ looks like a mnemonic, it is translated accordingly.

The overstrike and mnemonic tables reside permanently within the APL processor and are unaffected by the terminal declaration. This means that the Teletype user could actually input overstrikes, assuming he could simulate the backspace and "strange" characters. This flexibility is not possible for output, however. The Output Mapping Table determines the resulting display regardless of the manner of input.

Output processing is similar to input processing. For each internal character, one of four possible translations will occur:

- Conversion to a Single EBCDIC character.
- Conversion to an overstrike triplet (character-backspace-character) given in the overstrike table.
- Conversion to a mnemonic form given in the mnemonic table.
- No conversion (for instance, when the character is not a known internal APL character).

Final output translation will then be made by the CP-V communications handler, again according to the type of terminal indicated at log-on time.

The following code tables may be of some interest to the user. Figure B-1 illustrates standard EBCDIC codes and APL codes. Table B-1 shows Xerox line printer graphic codes.

False Terminal Declaration

As noted earlier, false terminal declaration does have potential applications. A batch user, for example, may wish to create a file for later on-line output. He may declare a Type 1 terminal in this case in order to avoid having the output file cluttered with mnemonics.

®Registered trademark of the IBM Corporation.

XEROX STANDARD 8-BIT COMPUTER CODES (EBCDIC)

Hexadecimal	Binary	Most Significant Digits 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
		0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
0	0000	NUL	DLE	ds		SP	&	-									0
1	0001	SOH	DC1	ss				/		a	j		\ [1]	A	J		1
2	0010	STX	DC2	fs						b	k	s	{ [1]	B	K	S	2
3	0011	ETX	DC3	si						c	l	t	} [1]	C	L	T	3
4	0100	EOT	DC4							d	m	u	[[1]	D	M	U	4
5	0101	HT	LF NL			Will not be assigned				e	n	v	] [1]	E	N	V	5
6	0110	ACK	SYN							f	o	w		F	O	W	6
7	0111	BEL	ETB							g	p	x		G	P	X	7
8	1000	EOM BS	CAN							h	q	y		H	Q	Y	8
9	1001	ENQ	EM							i	r	z		I	R	Z	9
A	1010	NAK	SUB			¢ [2]	!	~ [1]	:								
B	1011	VT	ESC			.	$	,	#								
C	1100	FF	FS			<	*	%	@					Will not be assigned			
D	1101	CR	GS			(	)	_	'								
E	1110	SO	RS			+	;	>	=								
F	1111	SI	US			| [2]	¬ [2]	?	"								DEL

Least Significant Digits

3 4 5

NOTES:

1 The characters ^ \ { } [ ] are ANSCII characters that do not appear in any of the EBCDIC-based character sets, though they are shown in the EBCDIC table.

2 The characters ¢ | ¬ appear in the 63- and 89-character EBCDIC sets but not in either of the ANSCII-based sets. However, Xerox software translates the characters ¢ | ¬ into ANSCII characters as follows:

EBCDIC	=	ANSCII
¢		` (6-0)
|		¦ (7-12)
¬		~ (7-14)

3 The EBCDIC control codes in columns 0 and 1 and their binary representation are exactly the same as those in the ANSCII table, except for two interchanges: LF/NL with NAK, and HT with ENQ.

4 Characters enclosed in heavy lines are included only in the standard 63- and 89-character EBCDIC sets.

5 These characters are included only in the standard 89-character EBCDIC set.

APL CODES

Hexadecimal	Binary	Most Significant Digits 0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
		0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
	0000			Index		SP	⍝	⎓	∧				⍀				0
1	0001					!	⌶	/	⍥	A	J		\	A	J		1
2	0010					⊥	⍞	⌸	⍫	B	K	S	⍁	B	K	S	2
3	0011					⍋	⍂	⍒	⍎	C	L	T		C	L	T	3
4	0100	EOT				⌊	⍑	+	≤	D	M	U	[	D	M	U	4
5	0101	HT	NL			∈	⍕		⍟	E	N	V	]	E	N	V	5
6	0110		IDLE			⍙	○	⍵	≥	F	O	W	⍬	F	O	W	6
7	0111					⍙	●	⍹	⍲	G	P	X	⍞	G	P	X	7
8	1000	BS				Δ	⍬	⍜	⍱	H	Q	Y	⍰	H	Q	Y	8
9	1001					⍳	⍝		∨	I	R	Z		I	R	Z	9
A	1010					⊂	∘	↑	:							⍛	×
B	1011					.	∪	,	≠							⊢	÷
C	1100					<	*	⍴	⍺							⊣	→
D	1101	CR				(	)	_	'							{	←
E	1110					+	;	>	=							}	⍞
F	1111					|	~	?	⍵								

Least Significant Digits

NOTES:

1 The CP-V communications handler ignores codes for the unmarked positions in the tables.

2 EOT is input when ATTN key is used in input mode.

3 The CP-V communications handler converts both upper and lowercase letters to APL (capital) letters on output.

4 The ⍛, ⊢, ⊣, { , and } characters are valid only for the Tektronix 4013 terminal.

Note: Characters bounded by heavy lines are overstrike combinations in APL.

Figure B-1. EBCDIC and APL Codes

Table B-1. Xerox Line Printer Graphic Codes

Character	6-Bit Code	Hex. Code	Character	6-Bit Code	Hex. Code	Character	6-Bit Code	Hex. Code
Blank	00 0000	40	I	00 1001	C9	<	00 1100	4C
0	11 0000	F0	J	01 0001	D1	(	00 1101	4D
1	11 0001	F1	K	01 0010	D2	+	00 1110	4E
2	11 0010	F2	L	01 0011	D3	|	00 1111	4F
3	11 0011	F3	M	01 0100	D4	&	01 0000	50
4	11 0100	F4	N	01 0101	D5	$	01 1011	5B
5	11 0101	F5	O	01 0110	D6	*	01 1100	5C
6	11 0110	F6	P	01 0111	D7	)	01 1101	5D
7	11 0111	F7	Q	01 1000	D8	;	01 1110	5E
8	11 1000	F8	R	01 1001	D9	-	10 0000	60
9	11 1001	F9	S	10 0010	E2	/	10 0001	61
A	00 0001	C1	T	10 0011	E3	,	10 1011	6B
B	00 0010	C2	U	10 0100	E4	%	10 1100	6C
C	00 0011	C3	V	10 0101	E5	>	10 1110	6E
D	00 0100	C4	W	10 0110	E6	:	11 1010	7A
E	00 0101	C5	X	10 0111	E7	#	11 1011	7B
F	00 0110	C6	Y	10 1000	E8	@	11 1100	7C
G	00 0111	C7	Z	10 1001	E9	'	11 1101	7D
H	00 1000	C8	.	00 1011	4B	=	11 1110	7E

As an example of the potential mix-ups when a false terminal declaration is made, however, consider the on-line 2741 user who issues the line printer declaration)TERMINAL 4. Suppose he requests display of a function containing the characters shown in the first column of the Table B-2. APL will transmit the characters shown on the second column, since APL thinks it is outputting to a line printer. The CP-V communications handler then transmits data according to the translation codes for a 2741. The results are shown in the remaining columns based on the type of 2741 used.

Table B-2. Examples of Problems with False Terminal Declaration

Character in Function	Character Sent to Handler	APL	Result	
			Non-APL (EBCD)	Non-APL (Selectric)
×	#	≠	#	#
÷	%	⍴	%	%
$	$	∪	$	$
^	&	∩	&	&
|	|	|	ignored	ignored
<	<	<	<	ignored
>	>	>	>	ignored

Teletype Usage

ESCAPE Key Sequences and APL

An important consideration for Teletype users is that standard CP-V ESCAPE key sequences will still be effective during APL runs. These should generally be used with caution, since some of the ESCAPE sequence capabilities may have strange effects on APL input. Applications and restrictions for the more useful sequences are described below.

Line Editing

The ESC RUBOUT sequence for erasing the last character input, and the ESC X sequence for erasing the entire input line are the recommended APL line editing procedures for Teletype users.

For Teletype terminals with true backspacing capability, the CONTROL L key combination (Form Feed) will be accepted as an alternate input editing signal. The response will be to generate a line feed and continue accepting line input. CONTROL L (followed by carriage return) is used to delete a function line.

Tab Usage

Since Teletype Model 33 does not perform true tabbing, CP-V provides tab simulation. APL then allows input and output of TAB characters on the assumption that the user has invoked tab simulation mode. However, APL will always replace input TAB characters with one or more spaces, depending on whether or not tab settings have been specified via the)TABS command.

Teletype users should generally avoid the)TABS command, however, because its effect is nullified by CP-V tab simulation. On output, for example, when)TABS have been specified, APL replaces strings of spaces with actual TAB characters where appropriate. On detecting the TAB characters CP-V will change them back to spaces in order to simulate mechanical tabulation. The net result is slower output than would have resulted if)TABS had not been used.

Operational Differences

Prompt Character

The Teletype displays a double colon in place of the quad-colon sequence to indicate evaluated input prompt. Quote-quad prompt is indicated by a single colon.

Illegal Character

The uppercase (shift) N is considered to be an illegal input character on Model 33 Teletypes because of the potential confusion for APL users. On some keyboards this key represents an up-arrow (the APL "take" operator), while on other keyboards it represents a caret (the APL "and" operator). Mnemonics will have to be used to represent either of these characters.

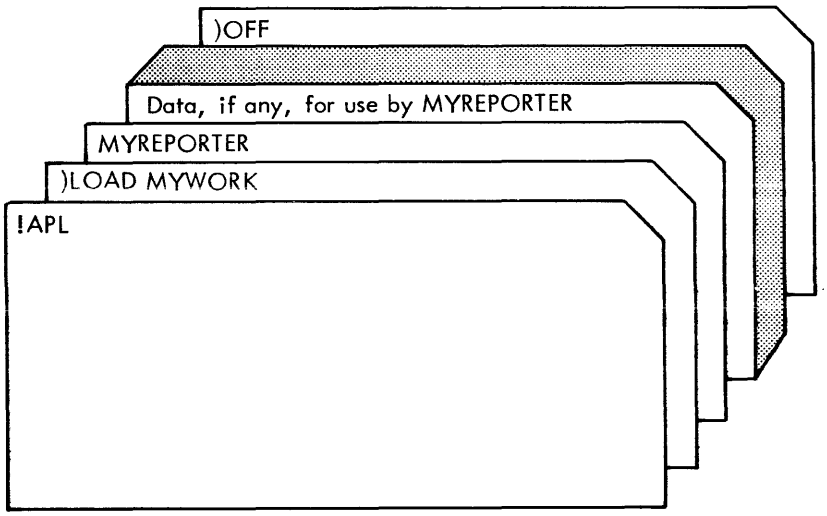
Error Marker

The ampersand will be output on Teletypes as APL's error marker.

Batch Operation

Intended Usage

The batch mode in Xerox APL is provided for the operation of programs with either extensive output or long execution times. Its usage is not intended for simulation of on-line operations. The anticipated operation is to load extant workspaces or copy functions and data from extant workspaces and execute them. Below is an example of a typical card sequence which would follow the Monitor control cards.



In this example the workspace MYWORK contains the master function MYREPORTER which drives the APL processing. Quad input will access data cards as appropriate.

Input/Output Device Assignments

In CP-V batch processing mode, input (DCB = F:APL) and output (DCB = F:OUT) that would normally go to the user's terminal are defaulted to the card reader and line printer, respectively, unless otherwise assigned by a Job Control card. Xerox APL will then assume a character mapping default of)TERMINAL 4 for card reader input and line printer output. The)TERMINAL command can be used to modify character mappings if desired.

The F:APL DCB is used for APL source input, and F:OUT is used for output of results and diagnostic messages. Either or both of these may be reassigned via the Monitor !ASSIGN card prior to entering APL. For example,

```
!ASSIGN  F:APL, (FILE, APLINPUT), (IN)
```

This card will cause APL input to be read from a user's file. That file should normally start with a)TERM command, unless the user wishes to use the default terminal type 4, and should end with one of the)OFF or)CONTINUE commands. If the following card is also used, the output will be routed to the file APLOUT rather than to the line printer:

```
!ASSIGN  F:OUT, (FILE, APLOUT), (OUT), (SAVE)
```

See the Xerox CP-V/BP Reference Manual, 90 17 64, for use of the !ASSIGN card and batch deck setups.

Card Input Format

When)TERMINAL 4 is used, the following will be observed:

- When input is via records other than cards (e.g., F:APL assigned to a file), APL automatically inserts a new-line character following the last actual character read.
- When input is via cards (or, in !BATCH runs with F:APL not assigned to a file), all input lines consist of 80 columns, including trailing blanks.
- Each card or record represents an individual line of APL input; there is no provision for continuation.

Error Response

Xerox APL batch operation is necessarily success-oriented. APL normally expects any input errors to be corrected as they are detected, which is not possible in this mode of operation. Therefore, any error will terminate the run. If a run is aborted, however, the processor will attempt to save the user's workspace by simulating the)CONTINUE HOLD command, if appropriate. Sidetracking (see Appendix A) of errors can be used in batch runs, which will continue until a non-sidetracked error occurs.

Operational Differences

Prompt Character

The line printer displays a double colon in place of the quad-colon prompt character to indicate evaluated input prompt. Quote-quad input is indicated by a single colon. The card reader is accessed for the actual input data in both cases, unless otherwise assigned by a Job Control card.

Error Marker

When output is to the line printer, the ampersand is used in place of the caret for APL's error marker.

Tektronix 4013 Usage

The Tektronix 4013 terminal is used in conjunction with the on-line curve plotting facilities described in Chapter 11. It employs a keyboard for input and a storing CRT display tube for output. Character I/O may be performed in either of two modes selectable at the terminal: ASCII or APL.

ASCII Mode

In ASCII mode, recommended for use in communicating with CP-V outside of APL, the 4013 acts like a Model 33 teletype with a true backspacing capability. The character set and keyboard layout is similar to that of the Model 33. Characters that differ from the APL characters shown on the key tops are indicated on the fronts of the keys. ASCII mode is established by (1) placing the 'ASCII-APL'/'APL' switch in the 'ASCII-APL' position, (2) illuminating the TTY LOCK switch, and (3) depressing SHIFT and RESET simultaneously to activate the ASCII character set. This procedure should be done initially, before logging on, and whenever leaving APL to use other CP-V facilities.

APL Mode

This mode provides a full APL character set with a keyboard layout similar to a 2741, as indicated by the key tops. This, therefore, is the mode recommended for use in APL. A ")TERMINAL 13" command identifies the 4013 to the APL processor. Since this command will ordinarily be issued immediately upon entering APL, with the terminal still in ASCII mode, it will be necessary for the user to switch to APL mode after typing ")TERMINAL 13". The user does this by turning the TTY LOCK switch illumination off. Thereafter, I/O is done with the APL character set shown on the key tops. In addition, the specification of terminal type 13 permits use of the terminal's graphics capabilities via ⎕I/O and the ΔGRF intrinsic, as described in Chapter 11.

Logging On

The log-on procedure is quite similar to that described for ordinary terminals at the beginning of Chapter 2. With reference to the steps there listed, the procedure is as follows:

1. Preparing the 4013 terminal for use:

   a. As shown – no change.

   b. As shown – no change.

   c. As shown – no change.

   d. Set the terminal to ASCII mode as follows:

      (1) Position the ASCII-APL/APL switch to ASCII-APL.

      (2) Depress the TTY LOCK key to illuminate that key (each depression of this key reverses the TTY LOCK state).

      (3) Press the SHIFT and RESET keys simultaneously.

2. Logging on to CP-V:

   a. As shown, except it is not necessary to type * (RET) . Once communication has been established, CP-V initiates the dialog by typing the log-on message.

   b. As shown, except the prompt symbol is ! instead of o.

   c. As shown, except the prompt symbol is ! instead of o.

   All input characters are ASCII and are shown on the key fronts, if different from the APL characters on the key tops.

3. Calling APL:

   As shown. When APL prompts for its first input, type the command

)TERMINAL 13

   (using the ASCII right parenthesis located over the 0), then switch to APL-mode by depressing the TTY LOCK key once to turn off its illumination. Thereafter, all characters indicated on the key tops are included in the APL set.

Line Editing

Line editing is done with the standard BACKSPACE-ATTN sequences, where the function of the nonexistent ATTN key is supplied by D^C (i.e., the CTRL and D keys depressed simultaneously). A function line may be deleted by striking D^C followed by RETURN. In addition, the standard ESCAPE key sequences will be effective, but should be used with caution.

Strapping Options

The APL graphics software defaults to expect the terminal to transmit seven characters in "gin-mode". If a terminal is strapped for five or six characters, users of APL graphics must invoke the STRAPIS function for proper response (see Chapter 11). Unless conflicts with other operations preclude seven-character strapping, it is recommended that the 4013 terminal be set in this mode.

Data Transmission Rates

The 4013 is available with interfaces to operate at several data transmission (baud) rates. The slowest rate, 110 baud, is compatible with Model 33 Teletype. If higher baud rates are used, different phone numbers are required. Check with local installation management on transmission rate capabilities and corresponding phone line. (Note that the higher baud rates, such as 300, are preferred for graphics work.)

Non-APL 2741 Terminals

Applications

For normal users, the addition of the standard APL typeball to the 2741 terminal is highly recommended. Nonstandard I/O imposes a considerable overhead on the user because of the additional input and output characters required for any operation. Furthermore, there are only certain applications in which nonstandard I/O capability would be useful on the 2741. For example, standard APL might be used to generate output intended to be run with a specialized typeball with dots and lines for plotting, or with a typeball which could output lowercase letters. The common case, however, involves input and output on the same terminal, and standard I/O use is much more practical here.

Operational Considerations

Functions such as tabbing and backspacing can always be performed on the 2741 terminal regardless of the typeball used. The prompt for quote-quad input – unlocking the keyboard with the carrier at the left margin – is the same for both APL and non-APL terminals. The BACKSPACE-ATTN sequence for current line editing is also used for both types. Operational differences are described below.

Illegal Characters

The following characters are considered to be illegal because they are not standard on all non-APL 2741 keyboards. Mnemonics must be used in place of each.

<	Less than
>	Greater than
|	Vertical bar
!	Exclamation point
¬	Not sign
]	Right bracket
[	Left bracket
°	Degree

The cent sign (¢) is also considered illegal because it has no reasonable APL interpretation.

Quad Input Prompt

A double colon is displayed in place of the quad-colon prompt character to indicate evaluated input prompt.

Error Marker

The ampersand is output in place of the caret as APL's error marker on nonstandard 2741 terminals.

Blind I/O

Blind I/O is a specialized capability which may be of value to the advanced APL user. Essentially, it provides a means by which characters may be input and output either to a specific device or to a file without undergoing any sort of translation or validity check by the APL processor. Blind I/O could enable the user to generate special characters which would otherwise be illegal under APL, for example, and route them to or from a unique device such as a CRT or a plotter.

Using Blind I/O

APL provides two DCBs – F:Q1 and F:Q2 – to be used for blind I/O, but performs no special set-up on them. It is assumed that the user will assign F:Q1 or F:Q2 to devices or files, according to his needs, using the)SET command (Chapter 8).

Within the APL processor, the special input/output characters ⍞1 and ⍞2 (quad overstruck with 1 and 2) supplement the quad and quote-quad characters. They are used to access the F:Q1 and F:Q2 DCBs when blind input or output is desired.

The default limit on record size for blind input is 512 bytes. This limit can be modified to any value from 1 to 32,767 by the SIZE option of the)SET command. Values higher than 512 may be needed for file input, and very small values are useful to control input from special devices. Input from ⍞1 or ⍞2 always creates a text vector result. If the source data is actually logic values, integer values, or double-precision floating point values, then file I/O operator 25, 26, or 27 may be used to correct the erroneous data type after input.

Blind output may be used to output any data. It should be noted, however, that large output records routed to physical devices with maximum length constraints will be truncated on output. In particular, records output to the user's console should be limited to 140 bytes, and records output to a line printer to 132 bytes. Note also that blind output of non-character data to a printing device will lead to useless results.

APL bypasses all translation sequences and legality checks for blind input and output – overstrikes are not even resolved, for instance. If an end-of-file condition is encountered by a blind-input request, APL returns an empty integer vector result.

Blind I/O on a Device

In the following examples, ⍞2 is assigned to the user's console prior to calling APL, as follows:

```
)SET ⍞2 INOUT UC
```

Quad 2 is then used for blind input and for blind output. In the example below, the blind input functions much the same as a quote-quad input, since the terminal itself is the blind input device.

```
      A←⍞2
NOW IS THE TIME FOR ALL GOOD MEN.
      A
NOW IS THE TIME FOR ALL GOOD MEN.
```

Consider the next example, however, in which blind input is used to input illegal overstrike characters, which cannot be done with quote-quad input.

```
      C←⍞2
■
      C
■
```

The examples below illustrate blind output to the terminal. Note that the data to be output was specified as a literal. When the RETURN key is struck, the data is output to the terminal exactly as it was input.

```
      ⍞←'1234567890+×QWERTYUIOP←ASDFGHJKL[]ZXCVBNM,∘/'
1234567890+×QWERTYUIOP←ASDFGHJKL[]ZXCVBNM,∘/


      ⍞←'¨¯<≤=≥>≠∨∧-÷?ωϵρ~↑↓⍳○*→⍺⌈⌊_∇∆∘⎕()⊂⊃∩∪⊥⊤|;:\'
¨¯<≤=≥>≠∨∧-÷?ωϵρ~↑↓⍳○*→⍺⌈⌊_∇∆∘⎕()⊂⊃∩∪⊥⊤|;:\


      ⍞←'ASDF'
ASDF
```

The user should note that the CP-V communications handler still performs its normal translation at the level of the I/O device. Bypassing the APL translation routines can result in the output of characters which are unrecognizable to the handler. For example, the handler maps most overstrikes as bad characters (≠ on the 2741) as shown below.

```
      ⍞←'●⍋⍒⍀⌿⊖'
≠≠≠≠≠≠
```

The SET command may also be used to bypass character translation at the CP-V level by using the options BIN and DRC. These options should be used only by a user with detailed knowledge of the characteristics of the particular I/O device.

Blind I/O for Files

In the following examples, [1] is assigned to a test input file which was built using the CP-V EDIT subsystem:

```
∘EDIT
EDIT HERE
*BUILD BLINDIN
   1.000 BLINDIN, RECORD 1.
   2.000 RECORD 2. TEST BACKSPACING.
   3.000 LAST RECORD.
   4.000
*END
```

Record 2 of the file contains a series of blanks and backspaces such that the total number of characters in the record is considerably more than the example shows. Record 3 contains an "illegal" overstrike character.

After APL is called:

)SET [1] IN DC/BLIND IN

In the next example, blind input is used to input the file records to the processor. Note that an attempt to use blind I/O to access the nonexistent fourth record results in empty integer vector.

```
A←⍞
B←⍞
C←⍞
D←⍞
```

A, B, C, and D now contain the data from the file records, as shown below. Note that the length of B reflects the blanks and backspace characters that were a part of the file record. C contains the illegal overstrike character just as it was originally entered.

```
      ρA
18
      A
BLINDIN, RECORD 1.

      ρB
57
      B
RECORD 2. TEST BACKSPACING.

      ρC
15
      C
LAST RECO←RD.

      ρD
 0
```

When blind output to a file is used, records are output as text data – scalar, vector, or array – without any sort of header data such as the record keys output by CP-V EDIT when it builds a file.

File Input/Output

The Xerox APL processor provides a set of intrinsic operators for file I/O. Normally, each installation provides its users with a file I/O system made up of locked functions using these operators. These systems may vary with the installation and are documented at the installation level. The file I/O operators used in such systems are described below. Direct use of these operators by programmers who have no prior I/O programming experience is discouraged. A workspace, FILEIO, containing a set of user file I/O functions is distributed to each Xerox APL installation. This workspace includes a user function, DESCRIBE, which describes the contents of the workspace.

Creating the Set of File I/O Operators

The file I/O operators are a set of 29 "operators" defined by one of three dyadic functions. These functions are created with the forms

```
fname ←14 ⍑1
fname ←14 ⍑2
fname ←14 ⍑4
```

where "fname" is the function name (fname will be used in all succeeding format examples in this section to indicate that a function name is to be supplied). The first two forms produce identical processing in successful usage. They differ only in the method of handling errors. The first type transmits error data (in the form of a scalar integer) back to the statement using that type of fname. The second type is used to have APL acknowledge an error; this permits sidetracking on file I/O errors or else APL will display one of the file I/O error messages (see Appendix A). The second type is preferred. The first type has been retained primarily to avoid incompatibilities in workspaces that used that type in the past.

For operators 14⍑1 and 14⍑2, the files built and accessed by APL are CP-V keyed files. Usually, they are built and accessed with a particular key system common to Xerox EDIT, BASIC, and APL.

An alternate form of file can be accessed if the file I/O operator is the third form. These files are built using the CP-V random file capability, and are referred to as APL indexed files. A file I/O package using the 14⍑4, form of operator can access either keyed or indexed files and the indexed files have been designed to appear essentially the same as the keyed files to the end user. The indexed file capability has been added principally to increase file access efficiency and to allow for shared access to files, for large data bases.

Appendix D contains information concerning the creation of APL indexed files, the design of indexing structures, and trade-offs between indexed and keyed files. New APL indexed files cannot be created directly within APL. Check with your installation for availability of, and initiation of, indexed files.

The following usage allows executing sequences of these operations in order in a single line (each use of this form yields an empty vector as a result unless otherwise specified):

A fname B

where

A is the I/O operator number (ranging from 1 to 29).

B is the argument applicable to the I/O operation.

Structure of APL Files

Xerox APL uses two forms of CP-V file structure, keyed files and random files. The random file use by APL simulates keyed access, with certain restrictions and certain additional capabilities.

Keyed files normally use a numeric key system, which is essentially that used by Xerox EDIT. Provision is also made for accessing text keys of up to 31 characters in length.

Files created by or accessed by APL can be considered in three categories:

- APL component files. A component consists of two physical records, an ID record and a data record. Numeric keys are used. The key of a data record is always one plus the key of its ID record. Each data record consists of one APL variable, including the information required to indicate its shape and type. No records which are not parts of 'components' may be included.

  APL indexed files are always component files.

  Component files relieve the user of concern for housekeeping items such as record size, and carry information concerning when and by whom each component was created or most recently changed. The cost is in the extra file storage and I/O accesses for the ID record of each component. APL indexed files reduce I/O accesses but not space required for ID information.

- APL non-component files. These files are similar to component files in that data records are APL variables and numeric keys are used. ID records are omitted. The user must maintain information concerning maximum record size, and no information is maintained concerning the source or time of creation of individual records.

  Non-component files reduce I/O access by half compared with component files, and eliminate file space requirements for ID records. They should be preferred for use when file structure will be fairly simple and detailed ID information on individual records is not valuable.

- Non-APL files. Provision is made to access any keyed or sequential file created outside APL. Such files may be accessed sequentially, or by key. If keyed access is used, two forms are permitted. Numeric keys are appropriate for accessing numeric keyed files created by EDIT, BASIC, or FORTRAN. For these files, the key as seen by the originating user is multiplied by 1000 to form the real key.

  Example:

  EDIT record 1.25 has a real key of 1250. APL file functions accessing such records should apply this conversion factor. Text keys are required to access some files created by processors such as COBOL.

  If APL is used to create non-APL files, two modes are allowed: numeric keys or text keys. Numeric keys should be used if the files are to be accessed by EDIT, BASIC, or FORTRAN. In this case, the multiple of 1000 should be applied to the key value as noted above. Numeric keys are 3-byte keys. If text keys of more than 3 bytes are to be used in creating a new file, the maximum key length must be specified.

Opening and Creating Files

Following are the forms for the set of operators required to establish parameters prior to opening a DCB to a file and to carry out the open CAL:

- Establishing "file number":

  1 fname B

  where B is a positive integer specifying the file number to be used for subsequent file operations. Up to eight file numbers may be in use (OPEN) at any time.

- Establishing file name:

  2 fname B

  where B is a character vector specifying the file name for the currently set file number. A file name may contain up to 11 characters. If B is not a character vector, DOMAIN ERR is reported. If B is more than 11 characters, LENGTH ERR is reported.

- Establishing or resetting account:

  3 fname B

  where B is either zero or a character vector specifying the account for the currently set file number. An account can contain up to eight characters. Using a zero for the B argument resets the assignment to the user's account.

- Establishing or resetting password:

  4 fname B

  where B is either zero or a character vector specifying the password for the currently set file number, consisting of up to eight characters. Using the number zero for the B argument resets the password control to indicate no password. In the above cases, if B is neither an integer of value 0 or a text vector of letters and digits, DOMAIN ERR is reported. If the length exceeds eight characters, LENGTH ERR is reported.

- Establishing file identification as a single primitive:

  21 fname B

  where B is a character vector up to 40 characters in length specifying file ID for the currently set file number in the same formats permitted for system commands such as)LOAD. The permitted forms are:

  name

  account name

  name:key

  account name:key

  name. account

  name. account. password

  name. . password

- Assigning serial numbers for private pack utilization (check with local installation for availability of private pack use) or setting maximum key length:

  20 fname B

  where B is a text vector of up to 12 characters, the numeric value 0, or a positive integer from 3 to 31. Other forms give a rank, length, or domain error. Zero resets the pack control authorization, and should be issued, after a private pack has been opened, before opening public files. Numeric values of 3 to 31 set maximum key length (applicable only to new output files).

If B is a character vector, it is used to set the serial number field of the user's DCB for private pack use. If B is zero, it resets serial number control. The form 20 fname B applies to the currently set file number.

- Opening DCB in indicated mode:

  5 fname B

  where B is an integer specifying the mode of DCB for the currently set file number, as follows:

  1 indicates IN.

  2 indicates OUT.

  4 indicates INOUT.

  8 indicates OUTIN.

  17 indicates INABN, which means OPEN in IN mode but take error exit if file is found. It is used to verify that a file to be opened for OUT or OUTIN does not replace an existing file.

  20 indicates INOUT, shared, and applies to APL indexed files only.

Unless otherwise indicated, files are created as private files, with read and write access by user only. When opening new files (OUT or OUTIN), they may be declared public for READ access by adding 32 to the argument and for WRITE access by adding 64 to the argument. For example,

5 fname 8+32+64

This example opens the specified file as OUTIN, with public access for READ and WRITE. If the value of B is not consistent with stating a specified I/O mode, DOMAIN ERR is reported. The result of "5 fname B" is an empty vector unless an error or abnormal return results from the CP-V M:OPEN call. Error reporting is covered later in this section.

As an example of file creation, suppose for function F that N contains the file number, NAM contains the file name, A contains zero (indicating the user's own account), and P contains a password.

5F 8, 5F 17, 4F P, 3F A, 2F NAM, 1F N

This example opens a private, passworded file to the user's account, or detects an error if the file already exists. The file is open for output and input. Note that several operations are carried out in a single line. The result of each (except for error conditions) is an empty vector, which is catenated to the argument of the next operation.

APL indexed files may be opened only in IN or INOUT mode. IN mode is used for input only. INOUT may be in exclusive or shared mode. See operators 28 and 29 for control of shared update access.

Closing Files

- Closing and saving file for indicated file number:

  6 fname B

  where B is an integer specifying the file number. The result is an empty vector.

- Closing and releasing the file for indicated file number:

  7 fname B

  where the argument B is the same as above. (This form is used to delete files. A file is opened for input, using 5 fname, and then closed for release.) The result is an empty vector.

  APL indexed files may not be released by using 7 fname B.

Maintaining Component Range and Current Component Value

When files are created by APL or accessed in other than sequential mode, primitives are required to find the key range of an existing file, to change the range as the file is appended, and to set keys for access or creation of specific records. Keys are usually numeric but may be text. It is very inadvisable to mix the forms in a single file (text keys may of course consist of text numerals, but such keys are not handled as numeric keys). When a file is opened in OUT or OUTIN mode, values for the 'first component' and 'last component' are initialized to artificial values which will be updated when the first record is written. These keys will be numeric unless a maximum key length greater than 3 has been set. APL indexed files do not use true keys but simulate numeric keyed operation from the user's viewpoint.

- Returning the value of a designated key for the currently set file number:

  8 fname B

  where B is 1, 2, or 3, specifying which key the value is to be returned for (the key returned will be that for the currently open file, if any, of the most recently referenced file number):

  1 indicates that the value of the first key in the file is to be returned as an integer scalar or text vector.

  2 indicates that the value of the current key is to be returned as an integer scalar or text vector.

  3 indicates that the value of the highest key is to be returned as an integer scalar or text vector.

- Getting highest index number or index to component ratio.

  8 fname B

  where B is 4 or 5. Used only if APL indexed files are in use.

  4 indicates highest permitted index (or real key) value. For keyed files, returns 9,999,998.

  5 indicates ratio of index number to component number for a particular indexed file. Returns 0 if not an APL indexed file. May be used to test if current file is indexed.

- Setting the value of the current key for the currently set file number:

  9 fname B

  where B is an integer or text vector specifying the value for the current key. The key value is established for the next record to be read or written on the most recently referenced "file number". The result is the empty vector. If B is numeric and not in the range 1 to 9,999,998, DOMAIN ERR is reported. If B is a text vector and its length is not in the range 3 to 31, LENGTH ERR is reported. If the key length is too great for the current output file, an error will occur when the next record write is attempted.

Key Values Versus Component Values

In order to permit record insertion into existing numerically keyed files in APL, the 'component number' is generally chosen as a multiple of the real key value used. This is accomplished by the user functions which set and examine keys. A typical form of such a system is to use a multiple of 1000. As noted earlier, this provides the same kind of numbers, from the user's viewpoint, as Xerox EDIT, BASIC, or FORTRAN numerically keyed files.

If the APL file is a 'component file', such a system allows component numbers of up to 9999.998 and the insertion of components down to the .002 level.

Example:

'Component' 1.106 consists of an ID record with key 1106 and a data record with key 1107.

If the file is a non-component APL file or non-APL file, insertion to the .001 level is allowed, as in EDIT and BASIC. Note that FORTRAN will provide keyed access only for records whose true keys are multiples of 1000.

In the case of APL indexed files, the ratio of component number to index (corresponding to true key) is set at the time the file is created. User functions must use that ratio to relate component to 'key' values. The ratio may be found using 8 fname 4 as noted earlier. The ratio will generally be much lower than the 1000 used for the keyed file examples here. Providing that extensive an insert capability causes prohibitive file storage costs for APL indexed files.

Writing APL Records

- Writing a record containing the value of an expression:

  10 fname expression

  The currently set key value and file number are used. The result is an empty vector unless an error is detected.

  Not permitted for APL indexed files.

- Writing a component:

  11 fname expression

  The first record is an identification record containing the time, the user's account, and the size of the data block associated with that identification record. The currently set key value and DCB number are used. The remaining record is as would be written by the "10 fname expression" form above, and has a key value of one greater than that of the identification record. The result is an empty vector unless an error is detected.

As an example, suppose that a function named WRITE exists to write a component, and that the function call has the form

```
A WRITE X
```

where A is the value to be filed, X[1] is the file number, and X[2] is the component number (there is no checking for length errors in the function call or for prior existence of the record to be written). The function consists of the single line

```
[1]    11fname A, 9fname ⌈1000×X[2], 1fname X[1]
```

which sets the file number, generates the key, and writes the identification and data records. It is assumed that the DCB has been opened for output.

A function similar to WRITE but adding to the end of the file might be the function APPEND,

```
[1]    11fname A, 9fname 1000×1+⌈(8fname 3) ÷ 1000, 1fname X
```

The file number is set to X. The highest key is found, the next multiple of 1000 is created, and the new key is set. The identification and data records are then written.

Writing Non-APL Records

Data records may be written that do not retain the APL internal attributes of 'shape' and other internal reference data. Such records may not be written as 'components' with paired ID records, and may not be written on APL indexed files.

- 22 fname B

  where B is any APL expression. The data represented by B is written as a single record in ravel order. If B is a logic vector the length is rounded up to a multiple of 8 bits, since the smallest I/O unit is one byte (8 bits). Data is not converted on output but APL header information is excluded. The purpose of this primitive is to allow writing records and files for use by processors other than APL. The record is written using the current key setting and file number.

Reading APL Records

Reading records in APL requires that the record size be known, in order to permit allocating space or indicating WS FULL. If the user has created files with records of known fixed length, the read may be made directly, specifying length. If records of variable length are used, the data records should be preceded by identification records, or a 'safe' length used for reads.

- Reading a data record:

  12 fname B

  where B is an integer specifying the size of the data record in bytes. The data record is read, if space is available, using current key and file numbers. If the record size is larger than specified or if the read results in an error or abnormal return, the result is an error (see Error Reporting in this section). If the record is smaller, the operation is completed. In this case, the unused portion of space reserved for the data is made available for other use.

  Not permitted for APL indexed files.

- Reading an identification record and a data record:

  13 fname B

  where B is an integer specifying the key value. The identification record is read; and if space is available, the data record is read using size information from the identification record. As an example, suppose that function READ is called with A READ X and that it consists of a single line:

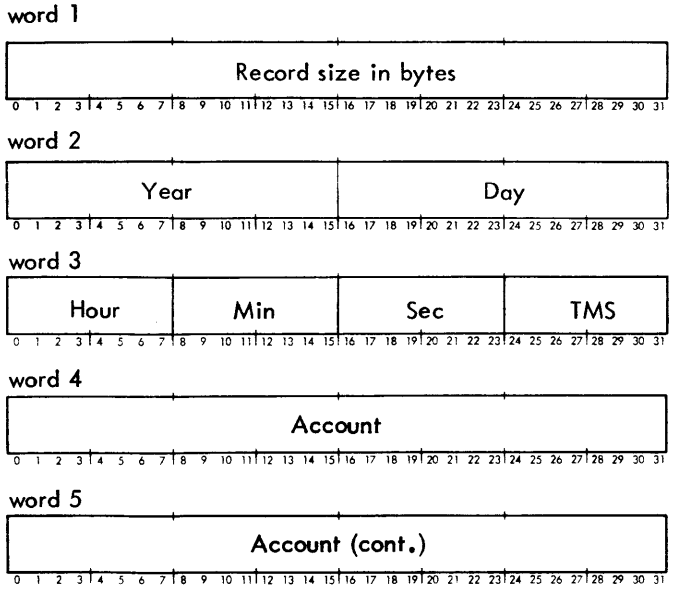
  ```
  [1]    A←13fname ⌈1000×X[2], 1fname X[1]
  ```

  This example sets the DCB number and the key value and then reads the identification and data records The data block is assigned to dummy variable A.

- Reading an identification record into a data block:

  14 fname B

  where B is an integer specifying the key value. The identification record is read in the form of a numeric integer vector with fixed format:



where

year is a binary value; for example, 1970 is represented as X'46'.

day is the Julian day of the year represented in binary; for example, September 14 is represented as X'101'.

hour is the hour of day (0-23).

min is the minute of hour (0-59).

TMS is the number of two millisecond units since the last 1/1000 of a minute (0-29).

account is the form contained in DCBs, left-justified, blank-filled to the right, even though it is delivered as two words of an integer vector. The following APL expression may be used to convert the integer values into a character vector. Assume the ID record has been read into variable X.

```
R←,0 2⍉(4⍴256)⊤ X[4 5]
```

Reading Non-APL Records

- 23 fname B

Reads a non-APL record using currently set key and file number. B is an integer specifying the record size in bytes. The data is arbitrarily treated as a text vector and an appropriate APL data block is formed. If the record is larger than indicated by B, an error results. If the record is smaller, unused space is made available. Separate primitives (25 to 27) may be used to change the data type of the result to logical, integer, or real numbers.

Not permitted for APL indexed files.

Deleting Records or Components

- Deleting a specified record:

  15 fname B

where B is an integer specifying the key value. The current DCB number is used in deleting the record. The result is an empty vector unless an error is detected.

Not permitted for APL indexed files.

- Deleting identification and data records:

  16 fname B

where B is an integer specifying the key value. This primitive allows construction of functions for the deletion of selected components or ranges of components. The result is an empty vector unless an error is detected.

Sequential Access to Existing APL Files

If a file has been created and modified such that there are gaps and inserts in the range of "component" values, it may be difficult to read the file in a keyed form without excessive errors for missing components or without inadvertently skipping existing records. The following operations cause sequential reads:

- 17 fname B

where B is an integer specifying the size of the record in bytes. Records are read sequentially, using the current file number. After each read, the current key value is updated and may be accessed. If an integer of zero is specified, the record is accessed but data is not read, regardless of actual record size. If the integer specified is greater than zero and the record size is larger than that specified, an error is reported.

Not permitted for APL indexed files unless B = 0 (skip record).

- 13 fname 0

  This is similar to "13 fname B" except that it reads the next identification record and associated data record. If the next record is not an identification record, records are skipped until an identification record is reached. At end of read, the current key is set to that of the last record read. If no identification record is found, an error is reported.

- 14 fname 0

  This is similar to "14 fname B" except that it skips forward to next identification record. The current key is updated. If no identification record is found, an error is reported.

Sequential Access to Non-APL Files

- 24 fname B

  where B is an integer specifying size of record in bytes. Records are read sequentially, using the current file number. Operation is analogous to 23 fname B except that the read is sequential rather than keyed.

  Not permitted for APL indexed files.

Converting Data Types

Primitives 23 and 24, for reading non-APL records, arbitrarily create text vector results. The records read may, in fact, be logic values, character, integer, or real (double-precision floating point) values.

- Convert character to logical vector.

  25 fname B

  where B is a character vector. The result is a logic vector consisting of the actual data in B. The length value is multiplied by 8 and the data type is switched to logic.

  If B is an expression rather than a named variable, this operator does not require that a copy of B be created. This primitive would typically be used in conjunction with a keyed or sequential read of a non-APL data record.

  Example: C ← 25 fname 23 fname 100

  If, in this example, the record read consists of 100 bytes, C is an 800-element logic vector.

- Convert character to integer vector.

  26 fname B

  where B is a character vector. The length must be a multiple of 4. The result is an integer vector consisting of the actual data in B. The length value is divided by 4 and the data type switched to integer. If B is an expression, the operator does not require that a copy of B be created.

- Convert character to real (double-precision floating point) vector.

  27 fname B

  where B is a character vector. The length must be a multiple of 8. The result is a real-number vector consisting of the actual data in B. If B is an expression rather than a named variable, the operator does not require that a copy of B be created.

Controlling Access to Shared APL Indexed Files

If an APL indexed file is opened in the shared mode, multiple updates are permitted concurrent access. The following features are provided to permit the user to lock out portions of such a file for purposes of reading a set of records without other intervening updates or completing a set of updates without interference. These features use the CP-V Enqueue-Dequeue facility, and an installation supporting these features must have reserved queue space.

- Locking out a record or block of records.

  28 fname B

  B is an index (key) value. Causes the designated record to be enqueued for exclusive use. Operates only if file is in shared INOUT mode. Successive use of 28 fname B can be made to enqueue a contiguous set of records, but not to enqueue records not in a contiguous block.

- Releasing a blocked record.

  29 fname B

  B is an index value. If a block of records is queued, they must be released in sequence from the ends, that is, release of a record may not be used to split a contiguous block of held records into two blocks.

  If B = 9999999, all records are released.

Error Conditions Unique to Enqueue-Dequeue Operations

The following error conditions may be reported on attempting to use Enqueue-Dequeue features. (These errors may be sidetracked.)

Message	Code-Subcode Values (CP-V) and Cause	
DOMAIN ERR	No code-subcode. File is not shared or result would create non-contiguous blocks of held records.	
COMP. NOT HELD	3100	Tried to dequeue an unheld record.
COMP. ALREADY HELD	3101	Tried to enqueue a held record.
ABORTED BY BRK OR CTRL-Y	3104	User aborted queue request.
REQ. WOULD CREATE DEADLOCK	5800	Queuing would deadlock access.
ENQ. FULL	5801	CP-V queue stack is full.
ENQ. UNAUTHORIZED	5803	User not authorized for Enqueue.
NOT ENQ. SYSTEM	AE00	Enqueue-Dequeue not supported.

Listing File Names and Numbers

These operations may be used in functions designed to list file components by number, with or without contents of the records.

- File names in a specified account

  18 fname B

  where B is a text vector specifying a user account. Result is a character matrix. Each row has account in columns 1 through 10 and a file name in columns 12 through 24. The matrix is a list of files in the specified account. Because of the general file I/O capability in CP-V, these files will not all be the result of APL file I/O and the matrix may include other passworded or protected files. Non-APL files and APL workspaces that are not passworded or read-protected will not be reported in the result.

- Names or numbers of currently open files

    19 fname B

    where B is an integer specifying the structure of the result as follows:

    1 indicates a character matrix with names of currently open files, one file per row.

    2 indicates a numeric vector with the currently open file numbers.

    If B is not 1 or 2, DOMAIN ERR is reported.

Error Reporting

The use of file I/O primitives may lead to a variety of errors, which are reported similarly to errors for other APL operations. The following common errors are of course included: DOMAIN, LENGTH, RANK, WS FULL, and SYNTAX. Errors are also reported for inability to open specified files or find specified records, to read or write records on a DCB that is closed or not open in the appropriate mode, or to use files and DCB in an inconsistent combination. The error codes returned by the monitor are listed in the Xerox CP-V/BP Reference Manual, 90 17 64. The error code in the APL file I/O subsystem is an integer scalar, related to monitor error codes as follows:

result = (128xcode) + subcode

The subcode and code may be separated for checking by using the encode operator. For example, if the result is 2561, the expression

V←0 128⊤result

gives the two-element vector where

V[1] = 20 and V[2] = 1

Notice that the code is 20 (hexadecimal 14) and the subcode is 1. (An attempt is made to open for output when the file is currently open to another user or DCB.)

If the code value is zero (that is, the result is less than 128), the subcodes are as follows:

0	indicates INABN set and OLD FILE found.
1	on read of identification record, indicates invalid record format.
2	on read of data record, indicates record is not a valid APL data block.
3	indicates file tie table is full. No new file numbers may be used until an open file is closed.
4	indicates attempt was made to release file from an account other than the user's.
5	indicates attempt was made to open a file with a tie number not in the file tie table.
6	indicates attempt was made to release a file with a number not in the file tie table.
7	indicates attempt was made to close and save a file with a number not in the file tie table.
8-9	indicates attempt was made to query or set key values for a file that is not currently open.
10-17, 22-24	indicates attempted I/O operation on a file not currently open. Error number is same as primitive number.
20	indicates attempt to delete record when file not in update mode.
21	indicates bad file ID format.

In general these operations will be used in locked functions and the error report will only indicate the type of error and the line number of the function.

The above form of error reporting applies only when the 14⍑1 intrinsic function is used; the error code is produced as a scalar integer result to be analyzed solely by the file I/O subsystem using that intrinsic. (If 14⍑2 is applicable, the subsystem may use sidetracking to process the error – see Appendix A – otherwise APL will handle

the error in the standard manner. The latter case relieves the subsystem from any responsibility for analyzing errors, and it can be designed largely as if only successful operation were possible.)

A list of standard file I/O error messages with corresponding code-subcode value (hexadecimal) follows:

Message	Code-Subcode Values
FILE I/O ERR xxxx	Any values not specified below
FILE NAME ERR	0000, 0015, 0300
FILE DAMAGE	0001, 7500, 7501, 7502, 7503, 7504, 7505, 7506
NOT APL FILE	0002
FILE TBL FULL	0003
FILE ACCESS ERR	0004, 0014, 1400
FILE IN USE	1401
FILE SPACE TOO LOW	5600, 5700
FILE INDEX ERR	0600, 0D00, 4200, 4300
PRIVATE PACK UNAVAIL, CALL OPR.	2001, 2002, 2003, 2004
FILE TIE ERR	0005, 0006, 0007, 0008, 0009, 000A, 000B, 000C, 000D, 000E, 000F, 0010, 0011, 2E00, 4400, 5100, 2500, 0016, 0017, 0018

Corresponding error messages for Enqueue-Dequeue features are listed in the section on controlling access to shared files.

Generation of File I/O Subsystems

A file I/O subsystem is not included as an integral module in the APL processor. File I/O subsystems may be tailored to an installation's needs, within the capabilities of the provided file I/O primitive operators. A sample workspace, FILEIO, is distributed to APL installations.

Table B-3. Translation Equivalences for Nonstandard Devices

2741 APL	2741 non-APL	TTY Model 33	Line Printer Output (Card Read Form Input)
1	1	1	1
2	2	2	2
3	3	3	3
4	4	4	4
5	5	5	5
6	6	6	6
7	7	7	7
8	8	8	8
9	9	9	9
0	0	0	0
+	+	+	+
×	#	#	#
Q	Q	Q	Q
W	W	W	W
E	E	E	E
R	R	R	R
T	T	T	T

Table B-3. Translation Equivalences for Nonstandard Devices (cont.)

2741 APL	2741 non-APL	TTY Model 33	Line Printer Output (Card Read Form Input)
Y	Y	Y	Y
U	U	U	U
I	I	I	I
O	O	O	O
P	P	P	P
←	_	← or _	$IS
A	A	A	A
S	S	S	S
D	D	D	D
F	F	F	F
G	G	G	G
H	H	H	H
J	J	J	J
K	K	K	K
L	L	L	L
[	$(	[	$(
]	$)	]	$)
Z	Z	Z	Z
X	X	X	X
C	C	C	C
V	V	V	V
B	B	B	B
N	N	N	N
M	M	M	M
,	,	,	,
.	.	.	.
/	/	/	/
¨	$DRS	$DRS	$DRS
¯	-	-	- (Negative Sign)
<	$LT	<	<
≤	$LE	$LE	$LE
=	=	=	=
≥	$GE	$GE	$GE
>	$GT	>	>
≠	$NE	$NE	$NE
∨	$OR	$OR	$OR
∧	&	&	&
-	$-	$-	$- (Subtract Operator)
÷	%	%	%

Table B-3. Translation Equivalences for Nonstandard Devices (cont.)

2741 APL	2741 non-APL	TTY Model 33	Line Printer Output (Card Read Form Input)
?	?	?	$RND
⍵	$W	$W	$W
∈	$E	$E	$E
⍴	$R	$R	$R
~	$NOT	$NOT	$NOT
↑	$TAK	$TAK	$TAK
↓	$DRP	$DRP	$DRP
⍳	$I	$I	$I
○	$O	$O	$O
*	*	*	*
→	$GO	$GO	$GO
⍺	@	@	@
⌈	$MAX	$MAX	$MAX
⌊	$MIN	$MIN	$MIN
–	$U	$U	$U (Underscore)
∇	"	"	$DEL
∆	$DLT	$DLT	$DLT
∘	$SC	$SC	$SC
'	'	'	'
⍞	$Q	$Q	$Q
(	(	(	(
)	)	)	)
⊂	$CPL	$CPL	$CPL
⊃	$CPR	$CPR	$CPR
∩	$CAP	$CAP	$CAP
∪	$CUP	$CUP	$CUP
⊤	$ECD	$ECD	$ECD
⊥	$DCD	$DCD	$DCD
|	$ABS	$ABS	|
;	;	;	;
:	:	:	:
\	$XPD	\	$XPD
SPACE	SPACE	SPACE	SPACE
TAB	TAB	TAB	$TAB
BACKSPACE	BACKSPACE	$BS	$BS
RETURN	RETURN	NL, CR	NL, CR
INDEX	INDEX	LF	LF (See also Notes at the end of Table B-3)
$	$	$	$
⍎	$REV	$REV	$REV

Table B-3. Translation Equivalences for Nonstandard Devices (cont.)

2741 APL	2741 non-APL	TTY Model 33	Line Printer Output (Card Read Form Input)
⍞	$TPS	$TPS	$TPS
⊖	$RV1	$RV1	$RV1
⍟	$LOG	$LOG	$LOG
⍱	$GD	$GD	$GD
⍋	$GU	$GU	$GU
!	$FCT	!	$FCT
⌶	$IB	$IB	$IB
⍁	$QQ	$QQ	$QQ
⌸	$MDV	$MDV	$MDV
⍝	$COM	$COM	$COM
⍲	$NOR	$NOR	$NOR
⍱	$NND	$NND	$NND
⍒	$LOK	$LOK	$LOK
⍀	$RD1	$RD1	$RD1
⍉	$XP1	$XP1	$XP1
⍐	⍐	$OUT	$OUT
A	a	$UA	$UA
B	b	$UB	$UB
C	c	$UC	$UC
D	d	$UD	$UD
E	e	$UE	$UE
F	f	$UF	$UF
G	g	$UG	$UG
H	h	$UH	$UH
I	i	$UI	$UI
J	j	$UJ	$UJ
K	k	$UK	$UK
L	l	$UL	$UL
M	m	$UM	$UM
N	n	$UN	$UN
O	o	$UO	$UO
P	p	$UP	$UP
Q	q	$UQ	$UQ
R	r	$UR	$UR
S	s	$US	$US
T	t	$UT	$UT
U	u	$UU	$UU
V	v	$UV	$UV
W	w	$UW	$UW

Table B-3. Translation Equivalences for Nonstandard Devices (cont.)

2741 APL	2741 non-APL	TTY Model 33	Line Printer Output (Card Reader Form Input)
X	x	$UX	$UX
Y	y	$UY	$UY
Z	z	$UZ	$UZ
⍙	$UDL	$UDL	$UDL
⍑	$TBR	$TBR	$TBR
[0]	$Q0	$Q0	$Q0
[1]	$Q1	$Q1	$Q1
[2]	$Q2	$Q2	$Q2

Notes:

- For TTY terminal the INDEX character is equivalent to LF only for output; for input the Form-Feed (Control-L) is equivalent to INDEX. This is most useful when desiring to delete a function line.
- For the Tektronix 4013, there is no INDEX or ATTN key. In order to delete a function line, the user may substitute Control-D for an ATTN.
- The Tektronix 4013 features five unique characters: ◊, ⊢, ⊣, ⌊, ⌋. There are no translation equivalents for these characters.
- As normally provided, Xerox APL accepts only [0], [1], and [2] and their mnemonic equivalents. Individual installations may modify their APL processors to allow [3] through [9]; they could also allow only a subset of these, and it is possible to allow none of the blind-quads.
- For TTY operation, APL assumes that the terminal has been designated as a Model 33 TTY at TEL level. Some installations default to other types, such as 7015. CP-V maps the bracket characters differently in this case and they may then not be used in APL. It is advisable for TTY users to be sure that they are identified as terminal type 33 at monitor level.

APPENDIX C. INTRINSIC FUNCTIONS

The intrinsic functions described in Chapter 7 can be created by using the dyadic T-Bar operator. Used dyadically, this operator creates a special data block that identifies a particular intrinsic function (coded within the APL processor). The data block may be assigned to any name, which then will be the name for the intrinsic function. In the examples below, the names selected are the standard names for the existing intrinsic functions (other names could be used).

The following intrinsic function assignments already exist in the WSFNS or GRAF (for △GRF) workspaces (which accompany the Xerox APL processor when it is delivered to an installation – see the installation manager).

```
ΔFMT←14 ⍑0
FIO←14 ⍑1
FIOE←14 ⍑2
ΔGRF←14 ⍑3
FIOI←14 ⍑4
ΔCR←14 ⍑5
ΔWM←14 ⍑6
ΔTE←14 ⍑7
ΔXL←14 ⍑8
ΔDMS←14 ⍑9

ORIGIN←15 ⍑0
WIDTH←15 ⍑1
DIGITS←15 ⍑2
TABS←15 ⍑3
SETLINK←15 ⍑4
SETFUZZ←15 ⍑5
DELAY←15 ⍑6
HEADER←15 ⍑7
VFCHAR←15 ⍑8

ERRN←16 ⍑0
ERRF←16 ⍑1
ERRX←16 ⍑2
PAGE←16 ⍑3
NLINES←16 ⍑4
```

The left argument of the T-bar operator indicates the type of the resulting intrinsic function:

14 for dyadic function,

15 for monadic function, and

16 for niladic function.

APPENDIX D. DESIGNING AND CREATING APL INDEXED FILES

The following material describes how to design an APL indexed file for a proposed data base and how to originate such a file. Indexed files are created using the CP-V random file capability. Their availability is subject to individual installation control.

Limits and Trade-Offs

Several characteristics of the keyed APL filed system, of CP-V random files, and of sample uses of APL files, have been considered in designing APL indexed files. Some of these characteristics should also be noted in designing applications. For many data bases, keyed access may be better than indexed files.

- Total Secondary Storage Occupancy

  Since indexed files are CP-V random files, they require dedication of a fixed block of contiguous secondary storage from the time of creation. When such files are filled, they cannot be dynamically expanded. Indexed files are thus suitable only for data bases with a reasonably predictable total size, and occupancy of a reasonable fraction of that space soon after creation.

- Component Identification and Size

  Since indexed files are to look, to the user, like current APL keyed files, each 'component' includes five words of identity information — size, date, time, and account (2 words). Each APL variable also includes a minimum of 2 words of 'header' information. In both the keyed and the indexed file systems there is considerable overhead associated with small records.

  Several design decisions for indexed files have been made to minimize the overhead cost of small records. It is still true, however, that applications using small records are relatively inefficient compared to those using primarily large records.

- Record Size vs. Secondary Storage Granularity

  The smallest addressable unit in secondary storage is the 'granule' of 512 words (2048 bytes). All reads and writes start on granule bounds. If records are not aligned on granule bounds, the impact is as follows: Reads must be buffered in core and the relevant data moved to its target location. Writes must be preceded by reads so that the new data is merged, in a core buffer, with the old data. The full granule is then written. It would clearly be advantageous to restrict data records to start on granule bounds. This is impracticable, however, if a file includes many small records. The design of APL indexed files compromises on the granule bound question. Records which approach or exceed one granule in size are written on granule bounds. Smaller records are packed.

- Scalars and 'Empty' Components

  Existing applications of APL files make extensive use of 'empty' components — records with keys but occupied by 'empty' APL variables. These components each require an identification record (5 words) and a data record (4 words). In the indexed file system, each index entry consists of 8 words. Any APL variable which requires a 4-word data record in the keyed file system is stored directly in the index in the new system. Records stored directly in the index include

  Empty vectors

  Empty matrixes

  Scalars

  Logic vectors of length 32 or less

Text vectors of length 4 or less

Integer vectors of length 1

This approach increases the size of the file index but significantly speeds treatment of empty and 'very small' components.

- Insertion Capability vs. Index Size

  In keyed APL files, the design encouraged the use of key values which were a multiple of 'component' numbers. The 'standard' described in the reference manual is 1000 to 1 and allows component numbers to the .002 level. The ratio is actually dependent on user-defined functions and may be varied by individual installations and users. For keyed files, there is no particular extra overhead in allowing extended insertion between components. For indexed files, there is the fixed overhead of the index itself. If M is the maximum number of records allowed and IG is the number of granules used for the index:

  $IG = M \div 64$

  It is impractical, for indexed files, to use a high ratio of component number to index number because of the fixed index granule overhead that would be incurred. This is particularly the case if the average record size is small. A realistic maximum is probably 10 to 1 — assuming that some significant number of insertions may be made.

File Structure

The indexed file capability employs standard CP-V random files. Structural aspects of these files are described below:

Granule Zero

The first granule (offset zero) has the following fixed structure:

Word 0	TEXT 'APLI'	Identifies as APL indexed file.
Word 1	M	Number of index entries.
Word 2	R	Ratio of component number to index number.
Word 3	O	Granule offset to start of index.
Word 4	L	Lowest index number in use.
Word 5	H	Highest index number in use.
Words 6-11		Spares (set to zero).
Words 12-511		Free segment chain.

Free Segment Chain

The free segment chain is a variable length table, or tables, of unused file space. The table starts in granule zero. Each entry requires two words as follows:

Word 1	Size, in words, of unused block.
Word 2	Offset, in words, from start of file.

If word 2 = 0, this is the last entry in the table.

If word 2 = 0 and word 1 is not zero, word 1 is the granule offset to another table of free segments.

If word 2 < 0, entry is currently not in use.

Index Granules

The file index begins at a granule offset specified by word 3 of granule 0. The index occupies contiguous granules and all entries are initialized to zero when the file is created. Each entry consists of eight words.

Case I. No record with this index number.

Word 1	Zero.
Words 2-8	Unused.

Case II. Records of APL variables requiring 4 words or less in primary storage.

Words 1-4	APL variable, including header.
Words 5-8	Date-Time-Account in same format as for keyed file ID records.

Case III. Records of APL variables requiring more than 4 words in primary storage.

Word 1	Physical record size in words.
Word 2	Word offset from start of file.
Word 3	First word of data block header.
Word 4	Unused.
Words 5-8	Same as for Case II.

Distinguishing Cases I, II, and III.

For Case I, Word 1 equals 0.

For Case II, Word 1 (bits 16-31) equals 4.

For Case III, Word 1 (bits 16-31) greater than 4.

Data Granules

Each data record in the file is a copy of the core image of an APL variable. Records which are 400 words or longer always start at granule bounds. Shorter records may start mid-granule but may not cross granule bounds. Note that the physical size of records may exceed the data block size. This is because physical size for larger records is always rounded up to a multiple of 512 words and because small records may be rounded up to avoid leaving 'free segments' smaller than 6 words.

Efficiency Considerations

This addition to APL file I/O has been motivated by specific needs which relate to extensive resource demands, and require explicit concern for optimization. The file structure has been designed primarily to minimize the number of disc accesses required to read and write APL file components. A second consideration has been to minimize in-core data transfers.

In particular, the design choice of placing the component ID data in the index rather than with the data block was dictated by the desire to allow direct moves of the APL data blocks between core and secondary storage. The additional increase of index entry size to allow direct access of 'empty' and scalar variables was dictated by the fact that existing data bases of potential users contain a high proportion of 'empty' values.

The design goal of minimal disc access has been compromised for records of sizes between 6 and 400 words. These records may start in mid-granule and require read-merge prior to write. They also require in-core data transfer following reads.

Fixed Overhead

The fixed overhead in granules associated with an APL indexed file is 1 + M ÷ 64, where M is the number of index entries.

Strictly speaking, this is not all overhead. The index includes identification data for all occupied entries and complete data for empty and scalar values.

Variable Overhead

As a file is modified, particularly by record deletions and replacements of records by larger or smaller versions, the free segment chain may grow and a number of unusable small segments of granules may accumulate. Each 255 entries in the free segment chain will require one granule (after the 249 'free' entries in granule zero). An extensive free segment chain slows processing associated with writing or deleting records but has no timing impact on reads. If a file has become badly fragmented, it should be transferred to a new file, copying in index order, to create a clean version.

Estimating Granule Requirements

An approximate formula for estimating granule requirements for an APL indexed file is:

G = IG + DG + 2.

IG = M ÷ 64 (index granules).

M = number of index values.

DG = (1 ÷ E) x NR x ARS ÷ 512 (data granules).

E = efficiency of packing. Depends on record sizes.

If record sizes unpredictable, assume E = .75.

NR = number of non-empty, non-scalar records when file is fully occupied, including anticipated inserts.

ARS = average record size, in words, including data block header.

Procedure for Creating APL Indexed Files

APL indexed files are created by the following procedure:

1. Execute APL workspace, SETPARS. This workspace requests information on the following parameters:

   Number of granules

   Number of index entries

   Ratio of index number to component number

   Offset to first index granule

If the inputs are logically consistent, SETPARS creates a file, APLIPARS, which will be used to initialize the random file.

2. Build a job file, similar to APLISAMP, shown below, specifying file name, and number of granules. This job file may optionally, as shown, specify READ accounts, WRITE accounts, SN (for private packs), and a password. The file may be created by the TEL BUILD command or by copying and editing APLISAMP, using EDIT.

   This job consists of an ASSIGN card, with extensions, followed by a RUN of APLILMN, which create and initialize the file. APLILMN was created from APLISI, also shown below.

APLISAMP sample of job to create APL indexed file:

```
1.  !JOB
2.  !LIMIT (TIME, 5), (CORE, 10)
3.  !PCL
4.  DELETE DC/fid
5.  END
6.  !ASSIGN M:BO, (RANDOM), (OUT), (SAVE), ;
7.  !              (FILE, fid), ;
8.  !              (READ, 'ALL'), ;
9.  !              (WRITE, 'NONE'), ;
10. !              (PASS, name), ;
11. !              (SN, serial number(s)), ;
12. !              (RESTORE, limit)
13. !RUN(LMN, APLILMN)
14. !EOD
```

Notes:

Lines 3 to 5 are to delete any prior file with the name of that being created.

Lines 8 and 9, as shown, are default Read-Write access. Up to 16 individual accounts can be specified for read access and write access.

Line 10 is optional, for passworded files.

Line 11 is optional, for private packs. If private packs are used, the !LIMIT card should also include a MOUNT option specifying the serial numbers to be used.

Standard CP-V error diagnostics will be issued if this job fails to create the specified random file.

```
    ∇ SETPARS
[1]   ⍞←'HOW MANY GRANULES?'
[2]   →ERROR1×⍳(NG≠⌈NG)∨(1≠⍴,NG)∨(NG←⍞)<6
[3]   ⍞←'HOW MANY INDEX ENTRIES?'
[4]   →ERROR1×⍳(NIE≠⌈NIE)∨(1≠⍴,NIE)∨(NIE←⍞)<1
[5]   ⍞←'RATIO OF INDEX NO. TO COMPONENT NO.(INTEGER RATIO)=?'
[6]   →ERROR1×⍳(RIC≠⌈RIC)∨(1≠⍴,RIC)∨(RIC←⍞)<1
[7]   ⍞←'OFFSET TO FIRST INDEX GRANULE=?'
[8]   →ERROR1×⍳(IO≠⌈IO)∨(1≠⍴,IO)∨(IO←⍞)<1
[9]   LIG←IO+NIG←⌈NIE÷64
[10]  →ERROR2×⍳LIG>NG
[11]  →ERROR3×⍳NIG>NG÷4
[12]  →IOEQ1×⍳IO=1
[13]  O1←512;S1←512×IO-1
[14]  O2←512×NIG+IO;S2←(512×NG-NIG+1)-S1
[15]  ←CATEN×⍳S2≠0
[16]  O2←0
[17]  →CATEN
[18]  IOEQ1:O1←512×NIG+1;S1←512×NG-NIG+1
[19]  O2←S2←0
[20]  CATEN:GRAN0←(26 F'APLI'),NIE,RIC,IO,NIE,(7⍴0),S1,O1,S2,O2,(496⍴0)NIG,IO
[21]  5 F 2,21 F'APLIPARS',1 F 1
```

```
[22]  6 F 1,22 F GRAN0,9 F 1
[23]  →
[24]  ERROR1:'INPUT PARAM. NOT SINGLE ELEMENT,NON-INTEGER, OR OUT OF RANGE'
[25]  →
[26]  ERROR2:'INDEX OFFSET TOO HIGH'
[27]  →
[28]  ERROR3:'TOO MANY INDEX ENTRIES'
    ∇

*

* APLISI-SOURCE FOR APLILMN,WHICH WILL CREATE AN APL INDEXED FILE

*        ASSEMBLED BY APLIMETA,WHICH CREATES APLIBO
         APLILOAD CREATES APLILMN
* SEE SETPARS, AN APL WORKSPACE, WHICH CREATES THE FILE 'APLIPARS'
*        USED BY APLILMN.
* SEE ALSO APLISAMP, WHICH IS AN EXAMPLE OF THE JOB FILE,USING
         APLILMN, TO CREATE AN APL INDEXED FILE
*
         SYSTEM   BPM
         SYSTEM   SIG7F
         REF      M:BO
         REF      M:SI
         CSECT    0
GRAN0    RES      514
ZEROS    RES      0
         DO1      32
         DATA     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
         CSECT    1
START    M:OPEN   M:SI,(IN),(FILE,'APLIPARS')
         M:OPEN   M:BO,(OUT),(SAVE)
         M:READ   M:SI,(BTD,0),(BUF,GRAN0),(SIZE,2056),(WAIT)
         M:WRITE  M:BO,(BLOCK,0),(BTD,0),(BUF,GRAN0),(SIZE,2048)
         LW,1     GRAN0+512          NUMBER OF INDEX GRANULES
         LW,2     GRAN0+513          INDEX OFFSET
WRITEZ   M:WRITE  M:BO,(BLOCK,*2),(BTD,0),(BUF,ZEROS),(SIZE,2048)
         AI,2     1
         BDR,1    WRITEZ
         M:CLOSE  M:BO,(SAVE)
         M:EXIT
         END      START
```

APPENDIX E. APL/EDMS INTERFACE

Installations which support the Xerox Extended Data Management System (EDMS) may also support a version of APL which provides an interface to EDMS data bases. Xerox EDMS is described in the EDMS Users Guide (90 30 37) and EDMS Reference Manual (90 30 12). The EDMS Reference Manual includes a complete description of the APL/EDMS Interface.

Table E-1 is a summary of the EDMS interface function call formats. These functions are contained in the DMSFNS workspace, which normally resides in the DMSLIB account. Table E-1 is ordered alphabetically by function name.

Table E-1. EDMS Interface Functions

Function Call Format	Action/Result
BERRCODE	Result is contents of CCB ERR-CODE cell.
BGRPNO	Result is contents of CCB GRP-NO cell.
BREFCODE	Result is contents of CCB REF-CODE cell.
CLOSAREA 'area-name'	Named area is closed.
CLOSEDB	All areas are closed.
CREATE 'area-name[.[account][.[password][.cipher-key]]]'	Named area is opened in create mode.
CURRGRP 'group-name'	Result is contents of current-of-type cell for named group.
CURRSET 'set-name'	Result is contents of set table for named set.
DCDREF encoded-reference-codes	Result is matrix of decoded reference codes.
DELETANT 'group-name'	†
DELETE 'group-name'	†
DELETSEL 'group-name'	†
'group-name' DELINK 'set-name'	†
DMSABORT 'function-name'	†
DMSCHKPT	DBM buffers are flushed and lockout bit is reset.
DMSEND	Dynamic memory is released and EDMS public library is disassociated.
DMSERCOD	Result is code of most recent APL-level EDMS error.

†No explicit result is returned. See EDMS reference manual for a description of the action taken.

Table E-1. EDMS Interface Functions (cont.)

Function Call Format	Action/Result
DMSLOCK 'function-name'	†
DMSPASS 'password'	Password is placed into CCB PASSWORD cell.
DMSPKSN 'serial-numbers'	Serial numbers are placed into most recently referenced DCB.
DMSRECV	†
DMSRLSE	†
DMSSUB 'subschema-name[.[account][.password]]'	Named subschema is identified for subsequent use.
DMSTRACE	Procedural trace is initiated.
ECDREF decoded-reference-codes	Result is vector of encoded reference codes.
ENDTRACE	Procedural trace is terminated.
FINDC 'group-name'	†
FINDD	†
FINDDUP 'group-name'	†
FINDFRST 'group-name'	†
FINDG 'group-name'	†
FINDLAST 'group-name'	†
FINDM 'set-name'	†
FINDN {'group-name' / 'set-name'}	†
FINDP {'group-name' / 'set-name'}	†
FINDS	†
FINDSI	†
FINDX 'item-name[{OF / IN} group-name]'	†
FROMDMS 'item-name[{OF / IN}{group-name / set-name}]'	Result is contents of item working storage.
FRSTREF encoded-reference-code	CCB FRST-REF cell is set to the value of the argument.

†No explicit result is returned. See EDMS reference manual for a description of the action taken.

Table E-1. EDMS Interface Functions (cont.)

Function Call Format	Action/Result
GET 'group-name'	t
HEAD 'set-name'	t
LASTREF {encoded-reference-code / integer scalar}	CCB LAST-REF cell is set to the value of the argument.
'group-name' LINK 'set-name'	t
MODIFY 'group-name'	t
{OPENRET / OPENUPD / OPRETSHD / OPUPDSHD} 'area-name[.[account][.[password][.cipher-key]]]'	Named area is opened in indicated mode.
REFCODE encode-reference-code	CCB REF-CODE cell is set to the value of the argument.
'group-name' RELINK 'set-name'	t
REMOVE 'group-name'	t
REMOVSEL 'group-name'	t
RESETERR integer-scalar-or-vector	Error control for indicated data-dependent errors is reset.
integer-scalar-or-vector SETERR 'function-name'	Error control for indicated data-dependent errors is set to function-name.
STORE 'group-name'	t
value TODMS 'item-name[{OF / IN} {group-name / set-name}]'	Item working storage is set to argument value.

†No explicit result is returned. See EDMS reference manual for a description of the action taken.

APPENDIX F. APL SYMBOLS

Table F-1. APL Symbols and Names

Symbol	Name(s)	Page(s)
+	Identity or Addition	54 54
×	Signum or Multiplication	56 56
←	Specification Arrow	40
[	Left Bracket	21, 68
]	Right Bracket	21, 68
,	Ravel or Catenation or Lamination	75 75 76
.	Period	71, 73
/	Reduction or Compression	68 84
¨	Dieresis	7
¯	Negative Sign	13
<	Less Than	63
≤	Less Than or Equal	63
=	Equal	64
≥	Greater Than or Equal	64
>	Greater Than	65
≠	Not Equal	65
∨	Or	66
∧	And	66
-	Negation or Subtraction	55 55

Table F-1. APL Symbols and Names (cont.)

Symbol	Name(s)	Page(s)
÷	Reciprocal or Division	56 57
?	Random	74
ω	Omega	7
∈	Membership or Execute	87 87
ρ	Dimension or Restructure	77 78
~	Not	68
↑	Take	86
↓	Drop	86
⍳	Index Generator or Index Of	74 75
○	Pi Times or Circular Functions	61 61
*	Exponential or Exponentiation	57 58
→	Branch Arrow	96
α	Alpha	7
⌈	Ceiling or Maximum	59 59
⌊	Floor or Minimum	59 59
–	Underscore	7
∇	Del	101
Δ	Delta	7
∘	Small Circle	73

Table F-1. APL Symbols and Names (cont.)

Symbol	Name(s)	Page(s)
'	Quote	14
⎕	Quad	43, 47
(	Left Parenthesis	48
)	Right Parenthesis	48, 122
⊂	Left Cap	7
⊃	Right Cap	7
∩	Cap	7
∪	Cup	7
⊤	Encode	83
⊥	Decode	82
|	Absolute Value or Residue	60 60
;	Semi-Colon	21, 47, 93
:	Colon	105
\	Scan or Expansion	70 85
$	Dollar Sign	212-215
⌽	Reversal or Rotation	78 78
⍉	Transpose	79
⊖	Reversal or Rotation	78 78
⍟	Natural Logarithm or Logarithm	58 58
⍒	Grade Down	82
⍋	Grade Up	82

Table F-1. APL Symbols and Names (cont.)

Symbol	Name(s)	Page(s)
!	Factorial or Combination	62 62
⌶	I-Beam	92
⍞	Quote-Quad	44
⌹	Matrix Inversion or Matrix Division	90 90
⍝	Comment	10
⍱	Nor	67
⍲	Nand	67
⍎	Locked Function	119
/	Reduction or Compression	68 84
\	Scan or Expansion	70 85
⍙	Underscored Delta	7, 15
⍑	T-Bar	93
⎕0	Quad-Zero	166
⎕1	Quad-One	198
⎕2	Quad-Two	198

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

A

absolute value operator, 60
account, 17
active workspace, 123
adding characters to end of line, 113
addition operator, 54
affixture codes, 156
ampersand, 198
and operator, 66
APL codes, 191
APL exponential notation, 13
APL features, 1
APL operators, 25, 52
APL terminal keyboard, 6, 7
argument characteristics, 52
arguments, 24
arithmetic group (operators), 54
arrays, 19
arrays of two or more dimensions, 46
assigning a value to an array, 23
assignment, 40
assignment statement, 99
asterisk after an entry, 118
ATTN key, 159
autostart, 2
AUTOSTART, 141
auxiliary plotting functions, 165

B

base value or decode operator, 82
batch operation, 193
blind I/O, 198, 2
blind I/O for files, 199
blind I/O on a device, 198
blind output, 47
branch statements, 96
breaks, 50

C

canonical representation, 171
card input format, 194
CATCH command, 128
catching assignments, 3
catenation, 76
catenation and lamination operator, 75
CENTER function, 164
changing a function header, 114
changing suspended functions, 105
changing terminal declaration, 189
character set, 7
circular operator, 61
CLEAR command, 130
CLEAR option, 147
closing files, 203
combination operator, 62
command statements, 95
commands, 124
comments, 10
communications commands, 122
composite operators, 68
compound statements, 2, 100
compression operator, 84
constants, 13
CONTINUE command, 6, 130
CONTINUE HOLD command, 6, 130
CONTINUE workspace, 123
control keys, 11
converting data types, 208
COPY command, 131
COS function, 166
△CR function, 171, 120, 121
creating the set of file I/O operators, 200

D

data list (right argument), 153
data transmission rates, 197
default terminal output, 49
defined functions, 101, 12
defined functions, displaying and editing, 108
defined functions, examples, 102
defined functions, syntax, 102
DELAY, 120
deleting a line, 109
deleting characters, 112
deleting records or components, 207
devices, standard and nonstandard, 189
DIGITS, 120
DIGITS command, 45, 133
dimension operator, 77
direct control of graphic I/O, 166
direct input, 42
direct-line prompt, 9
directives, 105
display function, 89
displaying and editing defined functions, 108
displaying user-defined functions, 106
division operator, 57
domain, 52
double colon, 198
DRAW function, 163
DROP command, 133
drop operator, 86
dummies, 104
dyadic function, 40
dyadic scalar operators, 53
dyadic transposition operation, 80

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

E

EBCDIC codes, 191
editing a line number, 114
editing user-defined functions, 107
empty arrays, 46
empty vectors, 46
equals operator, 64
ERASE command, 134
ERRN, ERRF, and ERRX, 121
error and break control, 3
error exits, 157
error marker, 193, 195, 198
error messages, 179
error reporting, 171, 210
error response, 195
error stop, 161
errors, 50
ESCAPE key sequences and APL, 192
evaluated input, 43
execute operator, 2, 87
execution and definition modes, 9
execution break, 159
execution stops, 159
EXP, 166
expansion operator, 85
exponential notation, 45
exponential operator, 57
exponentiation operator, 58
expression evaluation, 48
expunge, 172, 173

F

factorial operator, 62
false terminal declaration, 190
false terminal declaration, problem examples, 192
fast formatted output, 1
file I/O subsystems, 211
file identifier (FID), 16
file input/output, 200, 1
FIO, 121
FIOE, 121
floor and ceiling operators, 59
△FMT, 152, 216
FMT operation, 153, 157
FNS command, 134
format specifications, 152
format statement (left argument), 153
formats for branching, summary, 98
formatted output function (FMT), 152
formatting aids, 157
fractional number, 45
function copying, 2
function creation, 89
function definition mode, 88
function editing, 105
function editing in evaluated input and execute mode, 3
function execution, 116
function line appendage, 2
function name, 15
function references, 39
function-line prompt, 10
functions, 12
fuzz, 45

G

general input/output, 42
generalized logarithm (base A) operator, 58
gin-mode, 197
global variables, 17
grade down operator, 82
grade up operator, 92
GRAF workspace, 167
graphic functions, 163
graphic I/O, 166
graphic input functions, 165
graphics capability, 3
greater than operator, 65
greater than or equal to operator, 64
GRF calls, 168
△GRF intrinsic function, 167
GROUP command, 135
group name, 15
GRP command, 136
GRPS command, 136

H

HEADER function, 157, 120
higher-order array, 20
home terminal, 149

I

I-beam functions, 92
identity operator, 54
illegal character, 193
illegal characters, 197
index generator operator, 74
index of operator, 74
indexed assignment, 41
indexing, 19
indexing an indexed argument, 24
indexing of arrays, 21
inner product operator, 71
input scaling, 170
input/output, 42
input/output device assignments, 194
input/output devices, 42
input/output translation, 190
inquiry commands, 122
inserting a line, 110
inserting characters, 112
INT (interval) function, 163

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

intrinsic functions, 120, 216
issuing system commands, 115
items subject to sidetracking, 185

K

key values versus component values, 204

L

labels, 105
lamination, 25, 76
left argument, 153
length, 52
less than operator, 63
less than or equal to operator, 63
LIB command, 137
line corrections during input, 8
line deletion, 89
line editing, 192, 196
line insertion, 89
line numbers, 110
line printer graphic codes, 192
line replacement, 89
linelist, 171
listing file names and numbers, 209
LOAD command, 137
local variables, 17, 18
locals, 105
locking function, 119
log on/log off procedures, 4
logging off, 6
logging on, 4, 196
logging on and logging off APL system, example, 5
logical operator, 65

M

maintaining component range and current component value, 204
mathematical notation, 13
matrix, 20
matrix arguments, 155
matrix divide operator, 90
matrix inversion operator, 90
membership operator, 87
minimum and maximum operators, 59
mixed operators, 74
mixed output, 46
modification function, 90
modifying a line, 112
monadic function, 40
monadic scalar operators, 53
monadic transposition operation, 79
multiple assignment, 41
multiplication operator, 56

N

name format, 15
name usage, 15
namelist, 171
names, 7, 15
nand operator, 67
natural logarithm (base e) operator, 58
negation operator, 55
negative symbol, 13
niladic function, 40
NLINES function, 157, 120
non-APL 2741 terminals, 197
nonassignment statement, 99
nonstandard input/output, 189
nor operator, 67
normal stop, 159
NOSCALE function, 164
not equal to operator, 65
not operator, 68
numeric and character vectors, 46
numeric constants, 13

O

observation of intermediate results, 3
OBSERVE command, 138
OFF command, 6, 139
OFF HOLD command, 6, 139
OFF option, 147
ON option, 147
on-line and batch operation, 1
opening and creating files, 202
operation without APL characters, 1
operators, 11, 24, 25
OPR command, 140
OPRN command, 140
or operator, 66
order of evaluation, 48
ORIGIN, 120
ORIGIN command, 140
outer product operator, 73
output, 44
output scaling, 169
output value forms, 155
overstriking a character, 114

P

PAGE function, 157, 120
parentheses, 48
password, 17
passwords, 124
PCOPY command, 141
pendant function, 118
pi times operator, 61
plotting functions, 163
precedence of operators, 48

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

primitive functions (see "APL operators")
prompt character, 193, 195
prompts, 9

Q

QCOPY command, 141
QLOAD command, 141
quad input prompt, 198
quad or quote-quad input, 44
quad output, 47
quad prompt, 10
quad zero input, 165
quad zero output, 166
qualifiers, 156
quiet load and copy commands, 2, 141
quitting line editing, 114
quote-quad input, 44
quote-quad prompt, 10

R

rank, 52
ravel operator, 75
reading APL records, 206
reading non-APL records, 207
reciprocal operator, 56
recursive function, 116
reduction operator, 68
referencing a single element, 21
referencing more than one element, 22
replacing a line, 111
replacing characters, 113
report formatting, 152
representation or encode operator, 83
reshape operator, 78
residue operator, 60
reversal operator, 78
right argument, 153
right parenthesis, 122
roll operator, 74
rotation operator, 78

S

SAVE command, 141
saved workspace, 123
scalar arguments, 153
scalar operators, 53
scalar output, 166
SCALE function, 164
scaling functions, 164
scan operator, 70
SEAL command, 142
sequential access to existing APL files, 207
sequential access to non-APL files, 208
SET command, 142
SETFUZZ, 120
SETLINK, 121
shadowing, 18
shape, 52
SI CLEAR command, 119
SI command, 118, 146
SI-damage protection, 2
sidetrack setting, 186
sidetracking considerations, 187
sidetracking dynamics, 187
sidetracking on error and breaks, 184
sidetracking, aids for users, 188
sidetracking, items subject to, 185
significant digits, 45
signum operator, 56
simple assignment, 40
SIN function, 166
SIV command, 119, 146
standard 8-bit computer codes (EBCDIC), 191
standard file I/O error messages, 211
state indicator, 118
statement label, 15
statement labels, 98
statements, 11, 95
stop control vector, 160
stop of user input, 159
stopping a display, 47
stopping execution, 117
STRAPIS, 166
strapping options, 197
subtraction operator, 55
suspended function, 118
suspending execution, 118
SYMBOLS command, 147
syntax considerations, 49
syntax conventions, 52
system commands, 11, 115, 122
system commands, summary, 124

T

T-bar functions, 93
tab usage, 193
TABS, 120
TABS command, 148
take operator, 86
△TE function, 171, 121
Tektronix 4013 graphics terminal, 163
Tektronix 4013 usage, 195
teletype usage, 192
TERMINAL command, 149
terminal declaration, 189
text constants, 14
text editing functions, 175
tracing execution, 116
translation equivalences for nonstandard devices, 211
transparent scaling, 169
transposition operator, 79
types of input, 42

Note: For each entry in this index, the number of the most significant page is listed first. Any pages thereafter are listed in numerical sequence.

U

unequally spaced tabs, 2
unscaled graphic I/O, 169
user accounts, 124
user input versus computer output, 8
user-defined functions, 101

V

value of variable versus its name, 48
variable, 15
variables, 11, 16
variables local to defined function, 104
VARS command, 150
VCHAR function, 157, 120
vector, 20
vector arguments, 154
VS function, 163

W

WHATCHAR function, 165
WHATCOORD function, 165
WHATSCALE function, 164
WHATWINDOW function, 164
WIDTH, 120
WIDTH command, 150
width of line, 44
window functions, 164
△WM function, 171, 172, 121
wordspace control commands, 122
workspace concept, 123
workspace management functions, 171
workspace name, 15
workspace WSFNS, 120
writing APL records, 205
writing non-APL records, 205
WSID command, 151

X

△XL function, 158, 120

Xerox Corporation
701 South Aviation Boulevard
El Segundo, California 90245

XEROX

Reader Comment Form

We would appreciate your comments and suggestions for improving this publication.

Publication No.	Rev. Letter	Title	Current Date

How did you use this publication?

☐ Learning ☐ Installing ☐ Sales
☐ Reference ☐ Maintaining ☐ Operating

Is the material presented effectively?

☐ Fully Covered ☐ Well Illustrated ☐ Well Organized ☐ Clear

What is your overall rating of this publication?

☐ Very Good ☐ Fair ☐ Very Poor
☐ Good ☐ Poor

What is your occupation?

Your other comments may be entered here. Please be specific and give page, column, and line number references where applicable. To report errors, Please use the Xerox Software Improvement or Difficulty Report (1188) instead of this form.

Your Name & Return Address

2190(12/72)

Thank You For Your Interest. (fold & fasten as shown on back, no postage needed if mailed in U.S.A.)

Staple

Staple

Fold

First Class
Permit No. 229
El Segundo,
California

BUSINESS REPLY MAIL

No postage stamp necessary if mailed in the United States

Postage will be paid by

Xerox Corporation
701 South Aviation Boulevard
El Segundo, California 90245

Attn: Programming Publications

Fold

XEROX

701 South Aviation Boulevard
El Segundo, California 90245
213 679-4511

XEROX® is a trademark of XEROX CORPORATION.