A PLUM

REFERENCE MANUAL

By
Clark Wiedmann

Second Edition

APLUM Reference Manual

by
Clark Wiedmann

APL Group
University of Massachusetts Computing Center
Graduate Research Center
Amherst, Massachusetts 01002

© 1975 by Clark Wiedmann
All rights reserved.

SECOND EDITION

First Printing, September 1975
Second Printing, February 1976

Printed by
Hamilton I. Newell, Inc.
Amherst, MA.

PREFACE

This manual describes a version of the programming language APL developed by the University Computing Center of the University of Massachusetts at Amherst. The system described here, known as APLUM, was developed under the direction of James H. Burrill, who was responsible for the overall system design, planning, and coordination, and who wrote a considerable portion of the system. Other programmers on the project included Rick Mayforth, who was responsible for system commands, system functions, and system variables; Sheldon Gersten, who did work on mixed functions and shared variables; Brian Arnold, who worked primarily on output, format functions, and shared variables; Jeff Dean, who worked on mixed functions, workspace conversion and system functions; Judy Smith, who wrote inner product; and Clark Wiedmann, who worked on function definition, the file system, and mixed functions. In addition, Pat Driscoll and Wendy Mayfield assisted with documentation. The Raytheon Corporation provided assistance with many of the scalar functions, as did the Canadian Development Division of Control Data Corporation. The project was supported in part by a grant from Control Data Corporation. Special thanks are due to the entire staff of the University Computing Center for their assistance with the APL project. Credit is also due to the developers of several other versions of the APL language for the features and careful design that guided us in the development of APLUM.

Primary objectives of the design for APLUM were: to achieve a very high level of performance on the CDC 6600 and CYBER computers under the KRONOS operating system, to provide a flexible file system, to incorporate system functions and variables, to provide all system command capabilities to user-defined functions, to allow communication with programs in other languages, and to allow all workspace areas (including the symbol table and file buffers) to change size dynamically according to changing needs. The system was also designed with a storage management scheme that will enable future implementation

of arrays of arrays or future extensions to allow functions and variables not in use to reside on secondary memory devices, thus allowing an almost unlimited workspace size.

This edition is intended to describe version 2.12 of APLUM. For information about recent changes, see the APLNEWS workspace on the system. This is accessed by typing the APL command ⎕LOAD '*APL1 APLNEWS'.

This book is arranged as a reference manual and not as a teaching manual. The intent is to accurately describe particular details of APLUM, but not to teach APL to the novice. Hence the reader will find that this book lacks the wealth of examples, problems, and exercises that are usually found in a teaching manual. Some previous knowledge of APL is almost essential in order to make use of this manual, and the following books are recommended as possible introductions to the language:

APL\360 Primer, Paul Berry, IBM Technical Publications, New York, 1969. Covers in detail function definition and many primitive functions and system commands, but barely touches on matrices, higher-order arrays, and many mixed functions. Recommended for the occasional user of APL.

Handbook of APL Programming, Clark Wiedmann, Petrocelli Books, N.Y., 1974. A comprehensive introduction for the more advanced user of APL, who will make extensive use of the capabilities available.

Although this reference manual is intended for the reader who already has some knowledge of APL, it is recognized that all too often programmers are introduced to a new language through a reference manual. Consequently, a short introduction has been provided so that the following chapters will make some sense to the APL novice. The introduction also attempts to emphasize some of the more important features of the language to an extent that the organization of later chapters does not allow.

Amherst, 1975

Clark Wiedmann

CONTENTS

Introduction. A Sample Terminal Session

This short introduction to APL shows a sample terminal
session from the time of signing on until the time of signing
off. This section attempts to emphasize some of the important
facilities of APL, and attempts to show the dynamic nature of APL
(which may not be evident from the following chapters).

SIGNING ON

The first step is to establish a telephone connection
between the terminal and the computer. This procedure varies
somewhat according to the installation and the equipment used.
Further information about telephone numbers, types of terminals
that are supported, accounting procedures, and restrictions
placed on use of computer resources should be obtained from the
installation. The following discussion assumes that an acoustic
coupler will be used and that the terminal is capable of printing
the APL symbols (a Selectric or an ASCII-APL terminal). Other
terminals, such as a Model 33 Teletype, can be used (see
Appendix C) but they are much less satisfactory.

1. Turn on the terminal and the coupler (sometimes one
switch activates both). Dial the phone number for the
computer. At most installations the phone number
varies according to the type of terminal and the data
rate to be used. However, at the University of
Massachusetts a single phone number is used
(413-545-1600). You should soon hear a high-pitched
tone indicating the computer has answered the phone.
Place the telephone handset in the acoustic coupler.
Usually, one end of the acoustic coupler is marked
"cord" to indicate which end of the telephone handset
should be placed there. It is important to match the
correct ends.

2. At most installations the system will begin to print after a pause of a few seconds. However, at the University of Massachusetts the user must take the initiative by typing I= and RETURN for a Selectric terminal, or) and RETURN for an ASCII-APL terminal. These first characters identify the type of terminal being used and the data transmission rate to the TEMPO mini-computer that handles communications. Failure to use the correct characters to identify the terminal is likely to result in failure to communicate.

3. The system will print something like the following:

```
75/08/14. 16.14.58.
UMASS CYBER 74, KRONOS 2.1.7 *C*.
USER NUMBER:
```

The first line is the current date and time, and the second line identifies the installation and version of the operating system in use. Type your user number where the system has requested it and press RETURN. The system will then request your password. Type it over the blackened squares the computer provides (or merely press RETURN if your user number has no password). The system responds with something like:

```
TERMINAL               22,COR
RECOVER/SYSTEM:
```

4. Type APLUM to begin APL processing if you are using an ASCII-APL terminal (e.g., Teletype 38, Tektronix 4013, or Memorex 1240 with APL keyboards). At the University of Massachusetts other terminals that print API characters (e.g., Selectric terminals) may be used without special considerations. However, for most installations the use of such terminals requires the use of APLUM commands of the form APLUM,TT=COR. See Appendix C and Appendix D for details. The APL system will identify itself as follows:

```
APLUM2.11  75/08/12.  09.57.22.
NEW APLNEWS 75/08/06
CLEAR WS
```

The time and date on the top line indicate when the present version of APL was generated. The message NEW APLNEWS indicates when a news item about changes in the APL system was entered. To access the news item, type the command □LOAD'*APL1 APLNEWS'. The message CLEAR WS indicates that you have begun with a clear active workspace.

IMMEDIATE EXECUTION MODE

You can now type APL expressions. What you type is evaluated immediately. For example,

```
      3+5
8
```

(You type this and press RETURN.)
(This is the computer's response.)

Pressing the RETURN key is your signal to the computer that you have finished typing the line. The computer will not process the line until you press RETURN. The expressions you type are interpreted as they appear on the paper. This is called the principle of visual fidelity. You can space forward or backward as much as you please as long as the final appearance of the paper is what you intended. If you make a typing mistake you can cancel the line by pressing ATTN and RETURN (for a Selectric terminal) or ESC (for an ASCII terminal). You can revise the line if you have not pressed RETURN. To revise the line, press ATTN (for a Selectric terminal) or LINE FEED (for an ASCII terminal) then backspace until the type element is positioned below the left-most character to be corrected. This cancels the part of the line directly above and to the right of that position. Then type any characters to replace those that were removed.

The following examples show some simple calculations being performed.

```
      2×3
6

      3÷2
1.5
```

Note that the APL system indents six spaces before allowing you to type, but the system prints its response at the left margin. This clearly distinguishes what you type from what the computer types. The following example shows how arithmetic can be performed with several numbers at the same time

```
      2×1 2 3 4
2 4 6 8
```

The series of numbers on the right is called a vector. Each element of the vector was multiplied by 2.

Values can be given a name and saved for later use. The names are called variables. The process of giving a variable a value is called assignment. The following examples show assignment of values to variables A and B.

```
A←4.8
B←1 2 3 4
A+B
5.8 6.8 7.8 8.8
```

Note that when the result of a calculation is not assigned to a variable it is printed. The sum of the elements in a vector can be found as follows:

```
B←1 2 3 4
+/B
10
```

Any symbols on the keyboard can be used as values if they are surrounded by quotes. For example,

```
GRADES←'ABADCABAABADB'
```

The = symbol can be used to compare values. The result is 1 where a match is found and 0 otherwise. For example,

```
'A'=GRADES
1 0 1 0 0 1 0 1 1 0 1 0 0
```

The following example shows how a table of comparison values can be produced:

```
'ABCD'∘.=GRADES
1 0 1 0 0 1 0 1 1 0 1 0 0
0 1 0 0 0 0 1 0 0 1 0 0 1
0 0 0 0 1 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 1 0
```

There is one row for each value in 'ABCD' and there is one column for each value in GRADES. To find the number of A's, B's, C's, and D's, add up the 1's in the four rows as follows:

```
+/('ABCD'∘.=GRADES)
6 4 1 2
```

Below is an example of another comparison table using ≤ instead of =. Also, +/ is used instead of using +/ to add the rows as in the last example. +/ is used to add up the columns. The symbol /, called an overstrike, is formed by typing /, backspacing, and then typing -. (Actually the two symbols comprising the overstrike can be typed in either order.)

```
V←2.1 3.2 .08 8.1 4.6 1.2 2.3 4.2 1.6
2 4 6 8∘.≤V
1 1 0 1 1 0 1 1 0
0 0 0 1 1 0 0 1 0
0 0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0 0
```

```
+/(2 4 6 8∘.≤V)
1 1 0 4 2 0 1 2 0
```

As shown, this operation classifies each value in V according to the number of values in 2 4 6 8 it exceeded. That is, a value between 2 and 4 is in class 1, a value between 4 and 6 is in class 2, and a value between 6 and 8 is in class 3. A user defined function can be written to perform this operation:

```
∇Z←A CLASSIFY B
[1]  Z←+/A∘.≤B∇
```

The first ∇ signals to the computer that you wish to define a function. The first line shows that the function takes two arguments (input values) and gives a result. The computer numbered the next line with [1]. The ∇ at the end indicates that you have completed typing the lines of the function. The function can be used as follows:

```
2 4 6 8 CLASSIFY V
1 1 0 4 2 0 1 2 0
```

Now it might be interesting to tabulate how many 0's, 1's, 2's, and so forth were in the last result. An APL function can be written to do this but it requires two more APL operations: The largest value in a vector V is given by ⌈/V; and ιN gives the integers 1 2 3 ... N. We use both of these as follows:

```
∇Z←TAB B
[1]  K←ι((⌈/B)+1)
[2]  K←K-1
[3]  Z←+/(K∘.=B)∇
```

```
TAB 0 1 0 1 2 1 3
2 3 1 1
```

```
TAB(2 4 6 8 CLASSIFY V)
3 3 2 0 1
```

The following function will give a crude histogram of these results:

```
∇Z←HIST B
[1]  P←⌈/B
[2]  Z←((P+1)-ιP)∘.≤B ∇
```

```
HIST 3 3 2 0 1
1 1 0 0 0
1 1 1 0 0
1 1 1 0 1
```

A neater histogram can be produced by adding another line to the HIST function:

```
        ∇HIST
[3]     Z←' □'[Z+1]            (Function definition is opened.)
[4]     [□]                    (Another line is added.)
        ∇ Z←HIST B             (Display is requested.)
[1]     P←⌈/B
[2]     Z←((P+1)-⍳P)∘.≤B
[3]     Z←' □'[Z+1]
[4]     ∇
        ∇                      (Definition is closed.)
```

Note that to add more to the function you first type ∇ and the name (but not Z←HIST B). The computer numbered the line [3]. Typing [□] on the next line caused the computer to list the function. Finally, the ∇ was typed to indicate that no more lines were to be added. This function can now be used with the two others as follows:

```
HIST TAB 2 4 6 8 CLASSIFY V
□□
□□
□□□ □
```

You can display the names of defined functions and variables as shown below:

```
        )FNS
CLASSIFY HIST TAB

        )VARS
A   B   GRADES   K   P   V
```

To save the functions and variables for use at some other session, type

)SAVE MYWORK

The collection of functions and variables constitutes a workspace. Here a workspace named MYWORK was saved. It is advisable to save the workspace often if you are changing it in order to minimize the amount of work that will be lost in the event of a serious computer malfunction. (See Chapter 12 for the procedure to follow to avoid losing work after a telephone disconnect or minor computer malfunction.) To remove all functions and variables from the workspace you are now working with, type

)CLEAR
NEW APLNEWS 75/08/06
CLEAR WS
)FNS   (No functions.)
)VARS   (No variables.)

You can retrieve the MYWORK workspace as shown below:

)LOAD MYWORK
MYWORK 75/08/08 16:18:28
)FNS
CLASSIFY HIST TAB

To terminate the session and sign off the computer, type)OFF.

)OFF
A123456 LOG OFF. 18.12.07
A123456 CP 0.012 SEC

Although this sample session was short and only used a small fraction of the APL operations, it illustrates how well adapted APL is to experimentation. Programs can easily be developed in small parts and put together to do useful work. The flexibility in using functions in new combinations makes many problems much easier to solve. Many users of APL begin with the habit (formed by familiarity with other computer languages) of writing large monolithic programs in one piece. It should be evident that the modular approach illustrated above is better.

Function definition mode allows the user to enter function lines one at a time, remove lines, change lines, insert lines, or display the function. In function definition mode, APL statements entered are not executed or checked for errors, nor are system commands executed. Most errors will be detected when the statement is executed for the first time. System commands are illegal in the body of a function.

CREATING A FUNCTION

To enter function definition mode, type ∇ and the function header. The form for the function header should be determined by how the function is used. The six possible forms are shown in the following table.

Number of Arguments	0	1	2
No Result	name	name B	A name B
Result	Z←name	Z←name B	Z←A name B

The name of the function (represented by name in the table above) can consist of any number of letters A to Z, underscored letters A to Z, digits 0 to 9, or the symbols _, Δ, or Δ, but must not begin with a digit. The function name must not be in use for another global function or global variable. In the table, Z is used as the result variable, A is the left argument, and B is the right argument. Any other names could be used instead, provided they are used consistently in the body of the function. Names of system functions or variables must not be used as the result variable or argument variables.

After any of the forms in the table, there can be a semicolon and additional names separated by semicolons. The additional names declare variables and functions to be local to

Creating a Function

∇Z←A NAME B

Reopening Definition

∇NAME

Insert a line between [2] and [3]

[2.1] P←15

Display

```
[□]       (Display all.)
[□2□]     (Display from 20.)
[20□]     (Display line 20.)
```

Delete line [3]

[△3]

Replace line [3]

[3] P←Q+5×ιN

Line Editing for line [3]

```
[3□8]       (Line 3, column 8.)
P←Q+5×ιN    (The line is printed.)
     /1   1 (Type / to remove, 1 to insert 1 space.)
P← +5×ι N   (Type additions in the spaces.)
```

Extending line [5]

[5□0]

Context editing for line [3]

```
[3] /.old phrase.new phrase   (To replace.)
[3] /.old phrase.             (To delete.)
[3] /..old phrase.            (To extend.)
[3] /..new text               (To display the line and then
[3] ..                         extend it.)
```

1-2

the function. (Local variables and functions are discussed later in this chapter.)

The function header is line [0] of the function. After entering a ∇ and a header, function definition is said to be open. The system then types [1] on the next line to invite the user to enter line [1] of the function. The user can then type the function lines, and the system continues to number lines. When the last line has been entered, function definition mode can be terminated by typing a ∇ at the end of a line or on a line by itself. The ∇ is recognized as long as it is the last nonblank character on the line, even if the line is a comment.

Upon an attempt to close definition, the function header is checked for duplicate use of names, and statement labels are checked for duplication with names used in the header or names used for labels on other statements. In addition, the form of the header is checked for correctness. Any of these errors causes the message DEFN ERROR and display of the header or the line with the incorrect label. The error should be corrected, then ∇ should be typed to attempt to close definition again.

REOPENING DEFINITION

To add more lines to a function, first reopen definition by typing ∇ and the name. No other header information should be used--use of other header information causes the system to assume you are mistakenly attempting to create a new function having the same name as an old function. (The header can be changed after definition is open by treating it as line [0] and revising it as described below.) After definition of the function has been opened, the system types the number of the next line the function will have. The user can type additional lines in the same manner as when the function was created.

OVERRIDING THE LINE NUMBER

After the system types a line number, the user can override that line number by providing a different one. For example, assume the system printed [4] because line [4] was expected. The user could type [2] to override the [4] if he wants to enter a new line [2]. He could type the new line [2] on the same line he types the line number, or, he can type only the overriding line number and press RETURN, after which the system would type [2]. After line [2] is provided, the system would continue numbering the next line with [3].

To insert a new line between lines, use a fractional line number. For example, [3.2] could be used to insert a line between lines 3 and 4. No more than 4 digits are allowed after the decimal point. The system continues to number subsequent lines by incrementing the last position of the overriding line

1-3

number until another overriding line number is used. Thus, after line [3.98] would follow [3.99], [4], [4.01], and so forth.

To remove a line, use a request of the form [∆3]. The tilda before the overriding line number indicates that the line should be deleted. More than one line number can be provided (e.g., [∆3 9 1.6]). Note that a line cannot be replaced by a blank line by overriding a line number with the number of the line to be deleted and pressing RETURN.

Line [0] (the header) can be replaced like any other line, but it cannot be deleted. If the new line [0] causes the name of the function to change, the old function remains as it was when the function definition was opened, and a function having the new function definition is closed. The function name name is produced when definition is closed. The function name cannot be changed to the name of a global function or variable.

When function definition is closed, all lines are renumbered with consecutive integers. Because line numbers can change, use of labels for all branching is recommended.

If a WS FULL error occurs during function definition, all function definition mode is closed automatically, and all functions and variables remain as they were when function definition was opened.

DISPLAY OF FUNCTIONS

When function definition mode is open, the entire function can be displayed by typing [□]. To display only line 3 of the function, type [3□]. To display all lines from line [3] on, type [□3]. If you interrupt the display (see Appendix C), function definition remains open unless a closing ∇ appeared in the same line as the request for display.

LINE EDITING

Line editing can be used to change individual characters in a line. To begin line editing, type something of the form [3□8], where 3 is the number of the line to be revised, and 8 is the approximate position in the line where the first change is to be made. The system then prints the line and unlocks the keyboard below the 8th character. Use spaces or backspaces to position the typeball to the position to be changed. Type / under a character to delete it, or type a digit 1 to 9 to insert 1 to 9 spaces before the character, or type A below it to insert 5 spaces, B for 10 spaces, C for 15 spaces, and so on up to H for 40 spaces. To replace a character, you must delete that character (which closes up the line leaving no new space) and type a 1 before the next character to provide space for the replacement character. After the changes are specified and

RETURN is pressed, the system prints the revised line and waits at the position of the first inserted space or at the end of the line if no spaces were inserted. Type in any new characters in the spaces and then press RETURN.

If line editing causes the line number to change, the old line remains intact, and a new line with the new number is inserted. To extend a line, use the form [3□0]. The zero as a position in the line causes the line to be printed and the keyboard to unlock at the end of it.

Note that line editing should not be used with a terminal that does not print the APL character set. The APL system does not take account of single APL characters printing as more than one character on such terminals, so the position where changes are made can be somewhat unpredictable.

CONTEXT EDITING

Context editing allows replacement of the first occurrence of a given phrase by another phrase. Context editing is often more convenient than line editing when the changes are localized in a small part of the line and prior display of the line is not required. The line that is changed is the line the system is expecting next. The editing command has the form

/.old phrase.new phrase

The / signals that what follows is an editing request. The symbol immediately after the / is the symbol chosen by the user to mark the end of the first phrase and the beginning of the second phrase. Any symbol can be used as long as it occurs in neither phrase. The delimiter can optionally be used at the end of new phrase, but if it is not, the new phrase is assumed to end at the rightmost excursion of the type element (except that a ∇ at the end is not considered part of the phrase). The system first searches for the old phrase. If the old phrase is found, it is deleted and the new phrase is inserted at that point. If the old phrase is empty, the new phrase is inserted at the end of the line. After replacement, the system prints the corrected line. If both the old phrase and new phrase are empty, the system prints the line and unlocks the keyboard to allow the line to be extended. If the old phrase is not found, the error message

13: PHRASE NOT FOUND

is printed. (Note that the line number can be changed using context editing. When the line number is changed, the old line remains intact and the revised line is added to the function.) If the editing request is incorrectly formed, DEFN ERROR is printed. The following examples illustrate useful editing requests:

```
/.FOUR.SIX        (FOUR is replaced by SIX)
/.3,5,4.5         (A comma is used as the delimiter
                   because periods occur in the phrase.)
[3]  /.X1Y+.      (Deletion of X1Y+; [3] was used to
                   override the line number that had
                   been printed by the system.)
/...;C            (To extend the line with ;C)
/...              (To extend the line with information from
                   the keyboard.)
```

FUNCTION DEFINITION SHORTCUTS

In general, a line you type in function definition mode is used up before you are required to type another line. For example, you can type [□3]∇ to display line [3] and then close function definition. Or, you can type ∇FN[3]P←ιN∇ to open definition, override the line number with [3], provide a new line [3], and close definition. A ∇ at the end of a statement is always recognized, but other editing requests at the end are interpreted as being part of the line. Hence ∇FN[3]P←ιN[4]□∇ would cause line [3] to be P←ιN[4□]. It would not cause display of line [4] after replacing line [3].

LINE SEPARATOR

You can use the diamond symbol (the overstrike ⋄ for a Selectric terminal) as an input line separator for function definition mode. The parts separated by diamonds are used as if they were entered consecutively from the keyboard except that the normal line number prompt is suppressed. However, input lines for line editing requests must still be entered separately from the keyboard. Any diamonds preceded by an odd number of quotes are considered to be part of character constants and not line separators. If an error occurs, any remaining lines are discarded and input is again requested from the keyboard. The following example shows use of the line separator to define a function and then display it:

```
     ∇Z←NEXTLINE N ⋄ Z←CFREAD N ⋄ Z←(ν\Z≠' ')/Z ⋄ □∇
[1]  ∇Z←NEXTLINE N
[2]  Z←CFREAD N
     Z←(ν\Z≠' ')/Z
     ∇
```

The purpose of the line separator is to reduce waiting time when the computer responds slowly. The diamond is allowed as a line separator only in function definition mode and should not be confused with the use of the same symbol in other versions of APL to allow multiple executable APL statements on a line.

LOCALIZATION OF VARIABLES AND FUNCTIONS

The variables local to a function include all variables appearing in the function header and all statement labels. Variables that are not local to any function are called global variables. When execution of a function begins, the local variables take precedence over any other functions and variables having the same names. Other variables that were in effect before this function was called (that is, those not local to this function, which are called variables global to the function) remain accessible. When execution of the function is completed, the variables local to it vanish, thus releasing storage space for other uses, and any variables or functions global to the function become accessible again.

As execution of the function begins, the argument variables are assigned the values of the arguments in the expression invoking the function. If the function modifies the arguments, it is actually changing a copy of the original arguments. (See Chapter 11 for storage implications.) The label variables are also assigned scalar integer values of the line numbers on which they appear. These variables are locked to prevent them from being assigned inappropriate values. (However, they can be given improper values if they are first erased and then given a value.) The result variable and any other variables listed after the first semicolon in the header have no initial value.

A function can also have another function local to it if it has the second function's name in its header. As for local variables, the local function is undefined as execution of the main function begins. The local function can then be defined by use of □FX or □COPY with □ENV having 1 as its value (the normal case --see Chapter 7 for details about □FX, □COPY, and □ENV). When execution of the main function completes, the function local to it will vanish, just as a local variable would, and any temporarily inaccessible function or variable having the same name would again become accessible.

FUNCTION EXECUTION

Function execution begins when the name of the function is encountered in an expression being executed and any arguments have been evaluated. The system must save information about how far execution has progressed in the calling line in order to be able to eventually return to it and continue processing. The state indicator is a summary of this information and is available to the user. Execution of a function begins with establishment of local variables as discussed in the last section. Then, except for branching, the statements are executed in order from first to last. After the last statement has been executed, the value last assigned to the result variable is returned to be used in the calling expression, and all local variables vanish.

Branching can be used to control which statement will be executed next. A branch statement consists of a branch arrow followed by an expression that returns a result. The value must be a scalar or a vector, and unless it is an empty vector, the first value must be a nonnegative integer. If an empty vector is used, the next statement is performed. If the value is a scalar or vector, its first element is used as the number of the line to be executed next. If the value is 0 or exceeds the largest line number, the function exits. The following examples show useful branch statements. Close examination of the expressions to the right of the arrows should show how they generate appropriate line numbers:

→5×1A<14   (Branch to line 5 if A is less than 14. Note that this will not work in 0-origin.)

→(A=3)/8   (Branch if A equals 3 to line 8.)

→(L1,L2,L3)[2+×B]   (Branch to L1 if B is negative, to L2 if zero, or to L3 if positive.)

→(A>20 18 13 2)/L5,L4,L3,L2 (Branch to L5 if A is greater than 20, to L4 if greater than 18 but not 20, to L3 if greater than 13 but not 18, to L2 if greater than 2 but not 13, or go to the next line if A is less than or equal to 2.)

STATE INDICATOR

Any lines that call for execution of another function cannot be completed until the other function has exited. Such unfinished lines are called pendent lines. If an error causes a halt at a line of a function, that halted line is said to be suspended. The state indicator is a record of all pendent and suspended lines of functions. It omits partially executed lines entered in immediate execution mode, lines entered for quad input, and lines used as arguments to the execute function. The state indicator with variables, displayed by the system command)SIV, shows what lines are pendent or suspended and also shows variables local to functions. An abbreviated form, displayed by the system command)SI, omits names of label variables and names appearing in the header after the first semicolon. For example:

)SIV
[3]*Z←PRINT B;X;K;LIMIT:L1:L2
[4] SIMU K:L3

)SI
[3]*Z←PRINT B
[4] SIMU K

In both examples above, the most recently invoked line is shown first. An asterisk marks a line that is suspended. Here, line [4] of SIMU called PRINT, and execution of PRINT halted at line [3] because of an error. The)SIV display shows the full function header followed by a colon and names of statement labels separated by colons. If the function has no statement labels, no colons appear.

The)SIV display shows that the variable K currently accessible is the one local to PRINT. The other K local to SIMU is no longer accessible. However, the label variable L3 is local to SIMU still has its value because no variable L3 is local to PRINT. In general, the current value associated with a variable name is that for its first occurrence on the state indicator. If it does not appear on the state indicator, the current value is that of any global variable having that name.

A branch in immediate execution mode can be used to restart execution of the most recent suspended function. For example, →5 would cause execution of PRINT to continue at line 5. Usually, the function would be corrected or values of variables would be changed before proceeding. To remove the most recent suspension and the pendent lines that led to it, type a branch arrow with nothing to the right. A beginning user of APL often begins a new execution of a function without removing the old one, causing a large number of suspensions to accumulate. These unnecessary suspensions waste space and can lead to confusion by allowing local variables to make global variables inaccessible. When a suspension occurs, it is a good practice to either make corrections and continue execution or clear the state indicator by use of the niladic branch (see Chapter 2). An excessive number of suspensions can be eliminated by use of 0 □SAVE 'name' (see Chapter 7).

The information the system keeps about pendent lines can become invalid if the pendent functions are altered, replaced, or erased. The system responds by printing 14: SI DAMAGE and surrounding with brackets the names of the affected functions on the state indicator display. Execution of the affected functions cannot be resumed. Experienced users are expected to avoid SI DAMAGE if they intend to continue execution of a halted function. Certain changes to suspended functions can also lead to SI DAMAGE --specifically, altering the function header or changing the number or relative order of statement labels.

HALTING A FUNCTION

While a function is running, it can be halted by an interrupt (see App. C). However, when the keyboard is unlocked,

use of the interrupt on some terminals is interpreted as an attempt to revise the line being entered. To halt a function requesting quote-quad input, type the overstrike ⍝ (formed from O, U, and T). This results in suspension as if an error had occurred. To halt a function requesting quad input and remove it and all related pendent lines from the state indicator, use a branch arrow with nothing to the right.

TRACE AND STOP CONTROLS

Any stop, trace, and timing controls in effect for a function are cleared if function definition mode is used to change the function in any way.

LOCKED FUNCTIONS

A function can be locked by using ⍫ (∇ overstruck by ~) in place of ∇ when opening or closing function definition. Locking a function prevents display of the function and prevents its definition from being reopened. An attempt to open definition of a locked function results in the error message DEFN ERROR. A locked function cannot be unlocked; if you will want to change a locked function at a later date, keep an unlocked copy of the locked function in another workspace protected by a password or keep a printed listing of the function. If closing definition of the function results in SI DAMAGE, the request to lock the function is ignored.

Chapter 2. Statement Form and Order of Evaluation

This chapter discusses the form of legal APL statements and the order of evaluation of statements. Restricting the discussion to "APL statements" means that system commands (which are distinguished by beginning with a right parenthesis) are not of interest here. The meaning of a statement is determined in part by its form, but mainly by the functions used and the environment in which they are used. This chapter discusses the influence of form on meaning and leaves the functions and environment to be discussed in several other chapters.

SPACES

The use of spaces in an APL statement is usually unimportant to the meaning of the statement except for a few cases:

(1) Names must be separated from other names by spaces, and names must be separated from digits of a number to the right by spaces. (Also, a name beginning with E must be separated from digits to the left.) Otherwise, they would run together and appear to be all one name. Conversely, spaces in the middle of a name would make it appear to be two names.

(2) Numbers next to one another must be separated by spaces, and spaces cannot appear within a number.

(3) Spaces within a character constant are treated as any other character in the constant and affect the value of the constant.

(4) Spaces in a comment (except for trailing spaces) are preserved by the system. Although they have no meaning to the APL system, they may be important to the reader of the comment.

As execution begins for statements entered in immediate execution mode, entered in response to quad input, or used as arguments to the execute function (but excluding statements in the body of a function), a check is made to determine if the first nonblank character on the line is ∇, ⍫, or). In these cases the statement is preconverted to become a call to the function ⎕FD (a program that performs function definition mode) or ⎕SY (a program that performs system commands) with the original line as a character argument. For example, ∇FN[6]⎕ becomes ⎕FD '∇FN[6]⎕'. To preserve the original meaning, any quotes in the original statement become double quotes after the conversion. Any comment at the end of the original statement becomes part of the argument to ⎕FD or ⎕SY. The discussion that follows assumes that any such preconversion has already been performed.

COMMENTS

A comment may be entered in immediate execution mode or may appear in a function line. Comments begin with the symbol ⍝ and extend to the right to the last nonblank on the line. The part of the line following the comment symbol is not executed. This allows the user to intersperse descriptive text with APL statements. The following example shows a comment used in immediate execution mode to add a description to the printed transcript of the session:

K←2×⍳N ⍝ TO GENERATE 2 4 6, ETC.

The following discussion makes no further mention of comments, although a comment may appear at the end of any line, or the comment may constitute the entire line.

CONSTANTS

Constants represent numbers or characters. For example, .14 5.2 9 is a numeric constant-vector, and 'ABCD' is a character constant-vector. Constants consisting of one character or number are scalars, while those having more components or no components are vectors.

An *unsigned-number* is defined to be any of the following:

    digits
    digits.digits
    .*digits*

where *digits* represents one or more of the digits 0123456789. The italic notation used here is used throughout this book to

denote a term having a special definition. Here, *digits* represents a sequence of digits, not the letters *d i g i t s*. Hence the following numbers are examples of *unsigned-numbers*:

3.4
.05
58

However, an *unsigned-number* cannot end with a decimal point. Hence, 3. would not be legal.

A *number* has any of the following forms:

    unsigned-number
    ⁻*unsigned-number*
    unsigned-number exponent
    ⁻*unsigned number exponent*

The symbol ⁻ is used to express a negative number--the minus symbol cannot be used in its place. An exponent has one of the following forms:

    *E*digits
    E digits

The *E* can be read "times 10 to the power." So, 1E23 means 1×10*23, and 3.2E⁻3 is the same as .0032. A *numeric-constant* is formed from one or more *number*, separated by spaces.

A *character-constant* is of the form:

    '*symbols*'

where *symbols* represents any number of APL symbols, including no symbols. The symbol ' in a *character-constant* is represented by two quotes. For example,

    'IT''S'

is read

    IT'S

Quotes must always appear in pairs. An expression with an odd number of quotes results in a *SYNTAX ERROR*.

The term *constant* means either a *numeric-constant* or a *character-constant*.

FUNCTIONS

Functions are of three kinds:

(1) System functions, which have names that begin with ⎕ or ⍞, are used to communicate with the APL system.

(2) User-defined functions, which have names formed in the same way as variable names, are the only ones the user can define.

(3) Primitive functions (except those produced by operators) are symbolized by single characters such as +, ×, ÷, etc.

For the purposes of this chapter, the important features of functions are the number of arguments they require and whether they return results. Functions can be monadic (one argument), dyadic (two arguments), or niladic (no arguments). If *rfunction* is used to denote a function that returns a result and *function* is used to denote one that does not, the six possible forms are:

dyadic-rfunction   (Dyadic, returns a result.)
monadic-rfunction   (Monadic, returns a result.)
niladic-rfunction   (Niladic, returns a result.)
dyadic-function   (Dyadic, no result.)
monadic-function   (Monadic, no result.)
niladic-function   (Niladic, no result.)

For some primitive functions and system functions the same symbol or name is used for two distinct functions--one monadic and the other dyadic. The dyadic function is used if there is a left argument, and the monadic function is used if there is no left argument.

Dyadic user-defined functions can be used without a left argument, but if the function requires a value for its left argument, a *VALUE ERROR* results. The following example is a function that can be used without a left argument provided its right argument is not negative:

```
     ∇Z←A F1 B
[1]    Z←2×B
[2]    →(B≥0)/0
[3]    Z←Z+A  ∇

      F1 5
10

      5 F1 ¯1
3

      F1 ¯1
05: VALUE ERROR
F1[3] Z←Z+A
      ∧
```

The function ⎕NC, described in Chapter 7, can be used to check whether the left argument has a value. This could be used to write user-defined functions that have distinct monadic and

dyadic forms in analogy to distinct primitive functions having the same symbol.

Whether a name refers to a function or a variable is a matter that can be decided only when the line begins to execute. Also, whether a function actually returns a result may depend on circumstances. For example, if a user-defined function was defined to return a result, but the result variable was not assigned a value prior to exit from the function, a *VALUE ERROR* results if the expression calling the function requires a result.

OPERATORS

An *operator* is a special kind of function that takes functions as arguments and produces functions as results. Following are examples of four types:

A+.×B   (Inner Product.)
A∘.×B   (Outer Product.)
+/B   (Reduction.)
+\B   (Scan.)

The operators are the period, /, and \. In place of the ÷ and × in the above examples, any dyadic scalar function symbols could be used. These operators are discussed in detail in Chapter 6, but for the present, it is important to note that the forms exemplified by ÷.× and ∘.× represent dyadic functions that return results, and ÷/ and ÷\ represent monadic functions that return results.

The axis operator is used to specify the coordinate along which an operation is to be performed. Only a few functions can be used with the axis operator and further details are discussed with those functions. The operator is used in the form *function-symbol[value]*. For example:

Φ[2]B
÷/[1]B

VARIABLES

A variable is a name that might be associated with a value. The *variable-name* is formed from any sequence of the letters A to Z, underscored letters A̲ to Z̲, digits 0 to 9, or the symbols ∆, ∆̲, or ⍞, but the name cannot begin with a digit. System variables are special variables with names that begin with ⎕ or ⍞. The rest of the name can be composed in the same way as normal variable names. Only the system variables recognized by the system can be used--the user cannot invent new ones.

An indexed-variable is of the form:

variable[list]

A variable-name having no value associated with it can be used only immediately to the left of an assignment arrow; otherwise a VALUE ERROR will result.

VALUES

A value is any of the following:

    variable
    constant
    indexed-variable
    monadic-rfunction value
    left-argument dyadic-rfunction value
    niladic-rfunction
    left-argument
    variable-name←value
    indexed-variable←value
    (value)
    →value

The last case has the further restriction that the → may appear only as the first character of a line.

Use of an indexed-variable to the left of a specification arrow sets the values of elements of the variable without changing the shape of the variable. Used elsewhere, the index returns parts of a value.

The assignment arrow can be used to give a value to a variable or to change the value of a variable. The result of the assignment (not to be confused with the value of the variable) is the value used on the right. Consequently, A←B[1 2]←3 is the same as the two statements B[1 2]←3 and A←3. Similarly, A←□←B is the same as □←B and A←B; but A←□←B is not the same as □←B and A←□.

Hence,

A←3
(A←4)×A

The operations to find a value occur in right to left order. Hence, 3×2÷4 means 3×(2÷4). When a dyadic function is encountered, the right argument is preserved while the expression producing the left argument is evaluated. Hence,

12

However, it is poor programming practice to take advantage of the right argument being preserved; some APL systems produce a different result in cases like the one above. More generally,

any value encountered in the right to left scan is preserved. For example,

A←4 5 6
A[3 2 1]←A
A
6 5 4

(On some APL systems the result would be 4 5 4 or 6 5 6 because the variable on the right is not preserved, while on other systems such operations are prohibited.) However, the following example shows a case where the value is not preserved because the scan has not reached the variable:

A←2
A+A←3
6

LEFT ARGUMENTS

A left-argument is any of the following:

    variable
    constant
    constant[list]
    indexed-variable
    (value)
    (value)[list]
    niladic-rfunction

For example, 3 can be used as a left argument, ALPHA can be used as a left argument, and '123'[2] can be used as a left argument, but 2×3 cannot unless it is enclosed in parentheses. In fact, in 2×3*5, the 3 would actually be used as the left argument to *.

EXPRESSIONS

An expression is the same as a value except that it need not return a result that can be used for subsequent operations. An expression is any of the following:

    monadic-function value
    left-argument dyadic-function value
    niladic-function
    value
    →

The last case, called niladic branch, can be used only as the leftmost character of a line. The branch with no value or expression to the right causes exit from the executing function and from all other functions on the state indicator up to any

previous suspension.

LISTS

A *list* is of the form:

list-element
list-element;list-element
list-element;list-element;list element ...

The list, if used for an index, must have one list element for each dimension of the array being indexed.

A *list-element* can be:

    vacant
    value
    expression

An expression that does not give a result can be used in a list used for indexing and is treated as if the list element were vacant. A list element is *vacant* if there is nothing at all in that position. For example, $F[3;]$ illustrates a list having a vacant *list-element*.

The elements of a list are evaluated in right to left order.

Hence

    $A \leftarrow 3 ; A \leftarrow 5$

gives A a final value of 3. Note that the semicolon is not an APL function. Lists can only be used for indexing and output. Expressions like the following are illegal:

    $3\rho(A;B)$

Also, the statement

    $3\rho A;B$

is equivalent to

    $(3\rho A);B$

not

    $3\rho(A;B)$

The expressions separated by semicolons are evaluated separately, then their results constitute the list.

LINES AND IMPLICIT OUTPUT

A *line* is any of the following:

    value
    expression
    list
    vacant

When a *line* is a *list*, the list elements are printed in left to right order. The list can contain a mixture of character and numeric values as shown below:

    $X \leftarrow 34$
    'THE VALUE OF X IS: ';X
    THE VALUE OF X IS: 34

Scalar and vector list elements are printed on the same line (if $\square PW$ has not been exceeded), but printing of a matrix or array of higher rank begins on a new line, and any subsequent vector or scalar begins on a new line. List elements that are vacant or that produce no results are ski̇pped over.

If the first list element is a niladic branch, no output is produced. If the first list element is a branch with a value to the right, the value of the branch is printed along with the other list elements, then the branch is taken.

When the line is a *value*, the value is printed unless a specification or branch occured as the last operation. Hence, $3+2$ would print a result, but $A\leftarrow 3+2$ or even $(A\leftarrow 3+2)$ would not.

STATEMENTS

A *statement* is either a *line* or a *line* with a *label*. The *label* is a *variable-name* and colon placed before the *line*. For example:

    $REPEAT:\rightarrow 4 \times \iota X=Y$

A label on a statement entered in immediate execution mode, for quad input, or in the argument to the execute function is ignored.

QUAD AND QUOTE-QUAD

The system variables $\square$ and $\square$ are used for input and output. When they are assigned a value, the system prints the value. When their values are used in an expression, the system reads input from the keyboard to provide the value.

When $\square$ input is requested, the keyboard unlocks (normally

with the type element at the left margin). Any characters typed are returned as a vector, except that a single character gives a scalar.

   When ▯ input is requested, the system prints ▯: and then on the next line indents six spaces and unlocks the keyboard. Any APL expression that returns a result can be entered. If the expression is incorrect or does not produce a result, an error message is printed and the input request is repeated. For example,

```
▯:
      B
      B
05: VALUE ERROR
      B
      /
```
        (This is the input line.)

```
A←▯

▯:
      2×14
      A
2 4 6 8
```
        (The input request is repeated.)

A branch in quad input does not actually effect a branch.

   The ▯ can be used for output to conserve lines in a program. The statement ▯←A+B has the same effect as the two statements A←B and ▯←B.

   The ▯ symbol, when used for output, is slightly different from ▯ used for output. Ordinarily, APL output is followed by a carriage return so that the next input or output will begin on a new line. However, when quote-quad is used for output, the extra carriage return is suppressed. This allows the program to continue output on the same line or to give output and then request input on the same line. For example,

```
    ∇Z←ASK B
[1]  ▯←B
[2]  Z←▯ ∇

P←ASK 'AGE?'
AGE? 38
38
P
38
```
        (Note leading blanks in the result.)

   The leading blanks show where the typeball was positioned when the keyboard was unlocked. The person who was typing could have backspaced and replaced the blanks with other characters. Any leading blanks can be removed by using (v\B≠' ')/B←▯. Note that ▯ is not ignored when ▯ output is used. If the number of printed characters reaches ▯PW, the system inserts a carriage return in the output and indents 6 spaces before continuing the output.

Chapter 3. Scalar Functions

   The class of scalar functions includes those functions that can be defined for scalar arguments and then can be extended to other arguments through element-by-element extension. That is, if the function is monadic, the result has the same dimensions as the argument, and the elements of the result are found by applying the function to all elements of the argument. For the dyadic functions the following rules apply:

   1. If the arguments have the same shape, the result has that shape and is formed by applying the function to the corresponding elements of the arguments.

   2. If one argument is a one-element array and the other is not, the result has the shape of the one that is not one element. The one-element argument is used with each element of the other argument to form the result.

   3. If both arguments are one-element arrays, the result has the larger of the ranks of the arguments.

For the dyadic functions, the arguments must either have identical shapes or at least one must be a one-element array. Any other arguments produce a RANK ERROR if their ranks differ, or a LENGTH ERROR if their ranks match but dimensions differ. The following examples illustrate some of these rules:

```
      ▯←A←3 3ρ19
1 2 3
4 5 6
7 8 9

      -A
-1 -2 -3
-4 -5 -6
-7 -8 -9
```
        (A monadic scalar function.)

Summary of Scalar Functions.

Dyadic Function		Monadic Function	
A+B Addition	Sum of A and B. 3+5↔8	+B Plus	Same as 0+B.
A-B Subtraction	A minus B. 3-4↔¯1	-B Additive Inverse	Same as 0-B.
A×B Times	Product of A and B. 2×4↔8	×B Signum	Sign of B. Same as (B>0)-B<0 ×3 0 ¯2↔1 0 ¯1
A÷B Divide	A divided by B. Division by 0 is not allowed except that 0÷0 is defined to be 1. 3÷2↔1.5	÷B Reciprocal	Same as 1÷B. Not allowed if B is 0. ÷.2↔5
A⌈B Maximum	Larger of A and B. 3⌈5↔5 ¯1⌈¯5↔¯1	⌈B Ceiling	If B is an integer, the result is that integer. Otherwise the smallest integer greater than B. ⌈2.5 3↔3 3
A⌊B Minimum	Smaller of A and B. 3⌊5↔3 ¯1⌊¯5↔¯5	⌊B Floor	If B is an integer, the result is that integer. Otherwise the largest integer less than B. ⌊2.5 3↔2 3
A*B Power	A to the B power. A may be zero if B is not negative. 0*0 is defined to be 1. If A<0, B must be representable as a rational fraction with an odd denominator.	*B Exponential	e to the B power (e is 2.71828182...)
A⍟B Logarithm	Base A logarithm of B. A must be positive and must not be 1.	⍟B Natural Logarithm	Natural (base e) logarithm of B.

Summary of Scalar Functions, Continued.

Dyadic Function		Monadic Function	
A|B Residue	The remainder of B divided by A. More precisely, B-A×⌊B÷A+A=0	|B Magnitude	Absolute Value of B. |3 0 ¯3↔3 0 3
A!B Combinations	Number of combinations of B things taken A at a time for positive integer arguments. More generally A!B ↔ (!B)÷(!A)×!B-A	!B Factorial	Factorial of B for nonnegative integers. Otherwise the gamma function of B+1. Not defined for negative integers.
A○B Circular	The argument A determines which function from the following table is applied to B. A must be an integer in the range ¯7 to 7	○B Pi times	Pi times B ○1↔3.14159...
		?B Roll	A random choice from ⍳B. Depends on current origin.
		~B NOT	B must consist of 1's or 0's. ~1↔0 ~0↔1

N	N○B	(-N)○B
0	(1-B*2)*.5	(1-B*2)*.5
1	sin B	arc sin B
2	cos B	arc cos B
3	tan B	arc tan B
4	(1+B*2)*.5	(¯1+B*2)*.5
5	sinh B	arc sinh B
6	cosh B	arc cosh B
7	tanh B	arc tanh B

A=B	Equal	Result is 1 if the relation holds, 0 otherwise.
A≠B	Not equal	
A<B	Less than	
A>B	Greater than	
A≤B	Not greater than	
A≥B	Not less than	3≥5 3 1↔0 0 1

A∧B	AND	Elements of A and B must be 1's or 0's.
A∨B	OR	
A⍲B	NAND	A⍲B↔~A∧B
A⍱B	NOR	A⍱B↔~A∨B

A	B	A∧B	A∨B	A⍲B	A⍱B
1	1	1	1	0	0
1	0	0	1	1	0
0	1	0	1	1	0
0	0	0	0	1	1

```
¯2  4   6
 8 10 12
14 16 18

        2×A

 2  4  6
 8 10 12
14 16 18

        A+A

1  1
1  1

        ρ(1 1ρ4)÷(1 1 1ρ3)   (The larger rank prevails.)
```

(Two arguments with identical shapes.)

The table at the beginning of this chapter describes most of the functions in complete detail. Most of these functions are familiar mathematical functions or incorporate very simple concepts. Therefore, the discussion below deals with only a few of the less familiar functions or special cases.

FLOOR AND CEILING

The functions floor and ceiling always return an exact integer. The result depends on the value of ⎕CT as follows: If (|B-NINT B)≤⎕CT×(1+|NINT B) the result is NINT B, where NINT B is the nearest integer to B. Otherwise, the result is the least integer larger than B for floor, or the largest integer smaller than B for ceiling. Note that B-⌊B can be negative in cases where ⎕CT is not zero and B is slightly less than an integer.

POWER

In keeping with proper mathematics, the power function does not allow taking square roots of negative numbers (e.g., ¯1*.5), but it does allow taking cube roots of negative numbers (e.g., ¯1*÷3). To distinguish these cases, the power function attempts to represent the right argument P as a rational number N÷M, where N is an integer and M is the least integer such that (N÷M)=|P. Note that (N÷M)=|P depends on ⎕CT. If the left argument is negative and the rational representation has an even denominator, the power function gives a DOMAIN ERROR. If the left argument is negative and the rational fraction has an odd denominator, the result is negative if the numerator is odd and is positive if the numerator is even.

RESIDUE

The residue function is slightly more sophisticated than the definition in the table. For example, 2|2-.5×⎕CT would give the improper negative result ¯.5×⎕CT. The actual algorithm returns zero if B-A×⌊B÷A+A=0 would give a result having a sign opposite to the sign of A.

COMBINATIONS-OF

The combinations-of function returns limit values of A!B if A, B, or B-A are negative integers. That is, the result is zero if A, B, and B-A are all negative integers or if B is not a negative integer but either A or B-A is a negative integer.
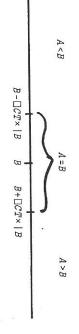
CIRCULAR FUNCTIONS

The domains and ranges of the circular functions are given below.

N	NOB	Domain	Range	(-N)OB	Domain	Range
0	(1-B*2)*.5	1≥|B	1≥Z			
1	sin B		1≥|Z	arc sin B	1≥|B	(o.5)≥|Z
2	cos B		1≥|Z	arc cos B	1≥|B	(○0)∧Z≤○1
3	tan B			arc tan B		(o.5)≥|Z
4	(1+B*2)*.5	1≤Z		(¯1+B*2)*.5	1≤|B	0≤Z
5	sinh B			arc sinh B		
6	cosh B	1≤Z		arc cosh B	1≤B	1≥|B
7	tanh B	1≥|Z		arc tanh B	1≥|B	0≤Z

RELATIONAL FUNCTIONS

The functions = and ≠ are the only scalar functions that can be used with arguments of character type. Characters can be compared with numbers, but the result always shows inequality. For numeric A and B, the result for A=B is 1 if |B-A is not greater than ⎕CT×|B. The three conditions A<B, A=B, and A>B are always exclusive. For example, if A=B gives 1, then A>B and A<B give 0. The range where two numbers are considered equal is illustrated below:

```
 A<B              A=B              A>B

          B-⎕CT×|B    B    B+⎕CT×|B
                 _____/
```

Note that when B is zero, A=B gives 1 only if A is exactly zero.

Chapter 4. Array Concepts and Indexing

An APL array can be visualized as an arrangement of values along n orthogonal coordinates, where n is 0 to 75 for this particular APL system. The positions along the coordinates are numbered 1, 2, 3, etc. in 1-origin, and they are numbered 0, 1, 2, etc. in 0-origin. The number of elements along a coordinate can be 0 or more. The lengths of the array along the coordinates are called the dimensions of the array, and the number of coordinates is called the rank of the array. The names scalar, vector, and matrix are used to denote arrays of rank 0, 1, and 2, respectively. No special names exist for arrays of rank greater than 2. The APLUM system has an arbitrary limit of 75 as the maximum rank of an array, but in practice, this limit is so large that it is not restrictive. Contrary to common casual practice in mathematics, an APL array has a definite rank--a one-element vector is not the same as a scalar, and a matrix with one row or column is not a vector.

The last coordinate of an array is conventionally considered to be the column coordinate, the second from last coordinate is the row coordinate, and the third from last coordinate is the plane coordinate. The following examples show how various arrays can be formed and displayed:

        3   (A scalar.)

        3
        ι4   (A vector.)
    1 2 3 4

        2 3ρι6   (A matrix.)
    1 2 3
    4 5 6

        2 3ρ'ABCDEF'   (A matrix of characters.)
    ABC
    DEF

Function	Description
ρB Size	Returns a vector containing the dimensions of B. The result has 0 elements for a scalar B, 1 element for a vector, and 2 elements for a matrix.
VρB Reshape	Forms a result having the dimensions specified by the left argument and having elements taken from the right argument in odometer order.
,B Ravel	The result is a vector containing all elements of B in odometer order.
R←B[I1;I2;I3;...;IN] Indexed selection	The result has as dimensions (ρI1),(ρI2),(ρI3),...,(ρIN) and contains those elements of B for which their first index is in I1 and their second index is in I2, etc. If a list element is vacant, all possible index values are used.
R←B[I1;I2;I3;...;IN]←B Indexed specification	The indicated elements of R are set to corresponding values from B. Either B must be a one-element array, or the dimensions of B must match (ρI1),(ρI2),(ρI3),...,(ρIN) except that dimensions of 1 are ignored. If a list element is vacant, all possible index values are used.

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

13 14 15 16
17 18 19 20
21 22 23 24
```

(Two planes, three rows, four columns.)

The last example shows that a rank-3 array is printed as a number of matrices separated by 1 blank line. A rank-4 array would be printed as a number of rank-3 arrays separated by two blank lines, and in general, a rank-N array is displayed as a number of arrays of rank $N-1$ separated by $N-2$ blank lines. An empty array prints as a blank line.

One often visualizes an array as a spatial arrangement of values. The spatial conceptualization leads to use of terms like "shape of array" and "vector along the Kth coordinate." These terms are important enough to give precise meanings for them. We define the "shape of an array" to be the result given by the size function (to be discussed in this chapter). As a consequence, a vector and a one-row matrix have different shapes, even though they may be visualized to look the same (and in fact, the system prints them identically). We define "a vector along the Kth coordinate" to be a vector of those elements in the array for which the coordinates other than the Kth are the same, and the Ith element of the vector has I as its Kth coordinate in the array--that is, a line of values aligned in the direction of the Kth coordinate.

RESHAPE: $R←VρB$

The reshape function was used in some of the previous examples to form arrays. The function forms a result having the dimensions specified by the vector (or scalar) left argument and having elements taken from the right argument. Elements are taken in first to last order, and if they are exhausted, they are used again beginning with the first. The right argument must not be empty unless the result will be empty--"reshape" never makes something out of nothing.

ORDERING OF ELEMENTS

The elements of an array are considered to be ordered. The reshape function takes elements according to this ordering. The ordering is the same as the order in which the elements are printed by a terminal. The order is called odometer order because the indices (coordinate positions) vary in the same way as the digits of an odometer. For example, for an array A3 having dimensions 2 3 4 the elements in odometer order are:

A3[1;1;1]
A3[1;1;2]
A3[1;1;3]
A3[1;1;4]
A3[1;2;1]
A3[1;2;2]
  .
  .
  .
A3[2;3;4]

SIZE: $R \leftarrow \rho B$

The size function returns a vector of the dimensions of its right argument. Because there is one element in the result for each dimension of B, the result has 0 elements for a scalar B, 1 element for a vector, 2 elements for a matrix, and so forth. Note that because ρB has one element for each dimension of B, $\rho\rho B$ gives the rank of B as a one-element vector. The following examples illustrate the size function for arrays of various ranks:

      ρ3   (A scalar.)
                                  (A blank line indicates an empty vector
                                  result.)

      ρρ3
0

      ρι3
3   (A vector.)

      ρρι3
1

      ρ2 3ρι6
2 3   (A matrix)

      ρρ2 3ρι6
2

      ρ2 3 5ρι30
2 3 5   (A rank-3 array.)

      ρρ2 3 5ρι30
3

RAVEL: $Z \leftarrow ,B$

The ravel function returns a vector result containing all the elements of the right argument in odometer order. For example:

      ,2 3ρι6
1 2 3 4 5 6   (Changing a matrix to a vector.)

      ρ,3
1   (Changing a scalar to a vector.)

The ravel function can be used to determine the number of elements in an arbitrary array. The number of elements in B is ρ,B. (Note that the ravel function could be omitted in this expression if B were always a vector.)

INDEXED SELECTION: $R \leftarrow B[I1;I2;I3; \ldots ;IN]$

Indexed selection chooses those elements of an array for which all indexes occur in the respective list elements. For example, if M is a matrix, M[3;4] gives the element having 3 as its row index and 4 as its column index. Similarly, M[2 3;4 5] gives those elements in the second and third rows that are also in the fourth and fifth columns. If a list element is vacant, ιN is used, where N is the length along that coordinate. The index values must be integers in the range of coordinates of elements in B. The index list for an array of rank K must have K-1 semicolons. The result R has the dimensions $(\rho I1),(\rho I2),(\rho I3), \ldots ,(\rho IN)$. Hence the rank of R is the sum of the ranks of the indices. If the indices are vectors, the result satisfies

$$R[K1;K2;K3; \ldots ;KN]=B[I1[K1];I2[K2];I3[K3]; \ldots ;IN[KN]]$$

When the indices are not all vectors, the result is:

$$((\rho I1),(\rho I2),(\rho I1), \ldots ,(\rho IN))\rho B[,I1;,I2;,I3; \ldots ;,IN]$$

Indexed selection cannot be applied to a scalar. The following examples show indexed selection applied to vectors and matrices:

      V←3 6 9 12
      V[1]
3

      V[4]
12

      V[5]
07: INDEX ERROR
      V[5]
        /   (An error results from a request
                                  for an element that does not exist.)

      V[5;6]
06: RANK ERROR
      V[5;6]
        /   (Because V is a vector, its rank is
                                  incompatible with the index list.)

```
V[1 2 1 1 2]
3 6 3 3 6

         1  2  3  4
□←M←3 4ρι12
         5  6  7  8
         9 10 11 12

M[2;3]
7

M[2; ]
5 6 7 8

M[ ;3]
3 7 11

M[2;3 4]          (Row 2, all columns.)
7 8

M[1 2 1;3 1]      (All rows, column 3.)
3  1
7  5
3  1

□←K←3 5ρ2 3 2 3 4 2 3 5 3 2 1 3 1 3 1
2 3 2 3 4
2 3 3 5 2
1 3 1 3 1

'_|Q'[K]
_|_|Q
_|X|_
_|_|_

                  (A matrix of characters.)
```

INDEXED SPECIFICATION: R[I1;I2;I3; ... ;IN]←B

Indexed specification allows setting of selected elements of
R. The index list indicates elements to be set in the same way
as for indexed list selection (see previous section). The
restrictions on list elements are also the same as for indexed
selection. The array B must be a scalar (or one-element array)
or must have dimensions (ρI1),(ρI2),(ρI3), ... ,(ρIN) except that
dimensions of length 1 are ignored in the comparison. If B is
not a scalar (or one-element array), the elements of B are taken
in odometer order and placed in appropriate locations in R. If
two elements of B are placed in the same position in R, the last
one in odometer order in B prevails. Both R and B must be of the
same type (i.e., character or numeric). The shape of R is not
changed by the operation. R must not be a scalar.

```
V←3 6 9
V[2]←-1
V
3 -1 9
```

```
V[2 3]←10 12
V
3 10 12

V[3 3]←15 16
V
3 10 16

□←M←2 3ρ'*'       (A scalar is used repeatedly.)
***
***

M[1;1 2 3]←'O'
OOO
***

M[ ;1]←'□'        (All rows, column 1.)
□OO
□**

M[1;2 3]←'+×'
□+×
□**
```

Chapter 5. Mixed Functions

The class of mixed functions includes all functions that are not system functions, composite functions, or scalar functions. Because few patterns exist between the mixed functions, they must be discussed individually to describe the arguments they allow and the results they produce. Chapter 4 already discussed the three mixed functions reshape, size, and ravel.

EXCEPTION RULES

Most of the mixed functions have "normal" cases for which the results are relatively simple to express in terms of the arguments. They also generally have additional special cases that are convenient but are treated as exceptions. The following are some of the reasons these exceptions are allowed:

Exceptions to overcome notational difficulty. There is no way to represent an empty numeric vector constant in an expression, and 10 is inconvenient to use as a left argument because it must be surrounded by parentheses. Hence ''ρB is allowed in place of (10)ρB. However, the only other case where an empty character argument is allowed, where a nonempty character argument would not be is the catenate function. (However, the system functions $\Box STOP$, $\Box TRACE$, and $\Box LTIME$ also allow empty character left arguments.) Another class of exceptions to overcome notational difficulty arises because it is not possible to type a one-element vector constant. Because a constant consisting of a single character or number is a scalar, many functions allow a scalar in place of a one-element vector. However, the left argument for index-of and the arguments to grade up and grade down are not allowed to be scalars.

Function	Description, Examples
ιB Index generator	Produces a vector of the first B integers. ι5↔1 2 3 4 5
VιB Index-of	For each element of B gives the first index in the vector V where the element is found or 1+ρV (in 1-origin) if the element is absent from V. 5 6 7 8ι6 5 2↔2 1 5
AεB Membership	Returns 1 for each element of A that occurs in B and returns 0 for other elements of A. 1 3 5ε2 3↔0 1 0
S1?S2 Deal	Chooses S1 random numbers from ιS2 without any duplications.
ΔV Grade up	The Ith element of the vector result is the index in V of the Ith smallest value in V. V[ΔV] gives V sorted in increasing order. Δ3.3 5.2 1.1↔3 1 2
▽V Grade down	The Ith element of the vector result is the index in V of the Ith largest value in V. V[▽V] gives V sorted in decreasing order.
A,[K]B Join	Joins A and B along the Kth coordinate. 1 2 3,4 5↔ 1 2 3 4 5
V/[K]B Compress	The result includes elements along the Kth coordinate of B for which there are corresponding 1's in V and does not include elements in V and does not include elements for which there are 0's in V. 1 0 1/1 2 3↔1 3 1 0 1/'ABC'↔'AC'

Function	Description, Examples
V\[K]B Expand	Expands by inserting zeros (if B is numeric) or blanks (if B is of character type) where there are 0's in V and selects consecutive elements along the Kth coordinate of B where there are 1's in V. 1 0 1 0\3 4↔3 0 4 0 1 0 1\'AB'↔'A B'
A↑B Take	Selects the first (if A[K]>0) or last (if A[K]<0) |A[K]| elements along the Kth coordinate of B. If |A[K]| exceeds (ρB)[K], zeros or blanks are used as the extra elements. 3↑1 2 3 4 5↔1 2 3 ‾3↑'ABCDE'↔'CDE'
A↓B Drop	Drops the first (if A[K]>0) or last (if A[K]<0) |A[K]| elements along the Kth coordinate of B. If |A[K]| exceeds (ρB)[K], the Kth dimension of the result is zero. 3↓1 2 3 4 5↔4 5 ‾3↓'ABCDE'↔'AB'
φ[K]B Reverse	Reverses the order of elements along the Kth coordinate of B. φ5 6 7↔7 6 5 φ'ABCD'↔'DCBA'
Aφ[K]B Rotate	Shifts vectors along the Kth coordinate of B in a negative direction (for A>0) or positive direction (for A<0). 2φ1 2 3 4 5↔3 4 5 1 2 2φ'ABCDE'↔'DEABC'
⍉B Monadic transpose	Reverses coordinates of B. ρ⍉B↔φρB
A⍉B Dyadic transpose	Interchanges coordinates of B according to A. The Kth coordinate of the result corresponds to the (A=K)/ιρA coordinate of B.

Function	Description, Examples
A⊥B Base value	Evaluates B as a number represented in a number system having radices A. 2 2 2⊥1 0 1↔5 10 10 10⊥2 3 4↔234
A⊤B Represent	Represents B in the number system having radices A. 2 2 2⊤5↔1 0 1 10 10 10⊤296↔2 9 6
⍎B Execute	Executes the character vector B as an APL statement. ⍎ '⍳5'↔1 2 3 4 5
⍕B Monadic format	Produces a character array representation of B. Except for treatment of lines longer than ⎕PW, ⍕B looks exactly like B when printed.
A⍕B Dyadic format	Represents columns of B according to the format specified by pairs of numbers on A. The first element of a pair in A is the width of the field (0 to have the system choose a width), and the second element of the pair gives the number of digits beyond the decimal if positive. If the second element of the pair is negative, its absolute value determines the total number of digits, and exponential format is used.
⌹B Matrix inverse	Matrix inverse of B. Same as I⌹B where I is an identity matrix.
A⌹B Matrix divide	Solution to a system of equations (for a square matrix B) or least squares regression coefficients (if B has more rows than columns). Same as (⌹B)+.×A.

Exceptions to ignore dimensions of 1. At times it is convenient to treat a row or column of an array as a vector, while at other times it is more convenient to treat it as a matrix. Consequently, some flexibility has been built into functions to allow extra or missing dimensions of 1.

Generalized scalar extension. The dyadic scalar functions allow a scalar argument to be used repeatedly with all elements of the other argument. More generally, some mixed functions allow a single vector, plane, etc. to be used repeatedly with parts of the other argument.

ARRAY TYPES

An array, even if it is empty, is either of character type or numeric type. Those mixed functions that rearrange elements of an array or select elements of an array always return a result having the same type as the right argument. For example, 0ρ'ABCD' gives an empty result of character type.

AXIS OPERATORS

For several of the mixed functions (and composite functions) an axis operator can be used to specify the coordinate along which the operation is to be performed. If no axis is specified, the last coordinate is assumed. Alternate symbols can be used to perform the operations along the first coordinate. These forms are:

Last coordinate	First coordinate	Kth coordinate
A,B	A⍪B	A,[K]B
A/B	A⌿B	A/[K]B
A\B	A⍀B	A\[K]B
⌽B	⊖B	⌽[K]B
A⌽B	A⊖B	A⌽[K]B

Note that the symbols for performing the operations along the first coordinate are not allowed to be used with an axis operator. For example, ⊖[K]B would produce a SYNTAX ERROR.

The value used for an axis operator must be a one-element array, and for functions other than join, it must be an integer in ⍳ρρB (except that if B is a scalar, it may be an integer ¯1+⍳1). For the join function (e.g., A,[K]B) the value of K may be an integer in ⍳(ρρA)⌈(ρρB)⌈1 or a half integer obtained by adding or subtracting .5 from one of those integers.

INDEX GENERATOR: $R \leftarrow \iota B$

The index generator function produces a vector of length B containing the first B integers. The result depends on the current origin.

<u>Requirements for B.</u> B must be a one-element array containing a nonnegative integer.

<u>Examples.</u>

```
      ι3
1 2 3    (In 1-origin.)

      □IO←0
      ι5
0 1 2 3 4    (In 0-origin.)

      ι0

    (Blank line indicates ι0 is empty.)
```

INDEX-OF: $R \leftarrow V \iota B$

The index-of function returns for each element of B the least index I in the vector V for which $V[I]$ equals the element of B. If no value in V is equal, the result element is $1+\rho V$ in 1-origin, or ρV in 0-origin. The comparisons use $\square CT$ so that elements of V and B may be considered equal even if they differ slightly.

<u>Requirements for V and B.</u> V must be a vector--a scalar is not allowed. B may be of any shape and the result will have that shape.

<u>Examples.</u>

```
4 5 6 12 5
4 2

      □←M←2 3ρ 'DEFGHI'
DEF
GHI

      'HIDE DOG'ιM
3 4 9
8 1 2    (A matrix result for a matrix right argument.)

7 8 9ι'AB'
4 4 4    (Characters never equal numbers.)
```

```
      'ABA'ι'ABAB'
1 2 1 2

      □IO←0
      'ABA'ι'ABAB'
0 1 0 1    (The 0-origin result is 1 less.)
```

MEMBERSHIP: $R \leftarrow A \in B$

The membership function returns 1 for each element of A that occurs in B. For numeric arguments the comparisons use the current value of $\square CT$, so values may differ slightly and still be considered equal.

<u>Requirements for A and B.</u> A and B may have any shape. The result has the same shape as A.

<u>Examples.</u>

```
      1 2 3∈3 1 6 4 9
1 0 1

      'ABCD' ∈ 'BACKS'
1 1 1 0

      □←A←2 3ρ'CATDOG'
CAT
DOG

      A∈'GOAT'
0 1 1
0 1 1    (The result has the shape of the left argument.)

      'GOAT'∈A
1 1 1 1

      'ABC'∈1 2 3 4
0 0 0
```

DEAL: $R \leftarrow S1?S2$

The deal function chooses at random $S1$ values from $\iota S2$ without repetitions.

<u>Requirements for $S1$ and $S2$.</u> Both $S1$ and $S2$ must be one-element arrays containing nonnegative integers such that $S1 \leq S2$. The result is a vector of length $S1$.

Examples.

```
3?5
3 1 4

3?5
4 5 3

5?5
1 2 5 3 4

□IO←0
5?5
3 2 0 1 2    (0-origin.)
```

GRADE UP AND GRADE DOWN: $R←⍋B$ and $R←⍒B$

The Ith element of the vector result R is the index in B where the Ith smallest (for grade up) or the Ith largest (for grade down) element of B occurs. The comparisons do not use □CT. If a value occurs more than once in B, the indices of those values occur together in R in increasing order.

Requirements for B. B must be a numeric vector. The result R is a numeric vector of the same length as B.

Examples.

```
⍋3.3 1.1 2.2 4.4 1.1 5.5
2 5 3 1 4 6

⍒3.3 1.1 2.2 4.4 1.1 5.5
6 4 1 3 2 5

V←3.3 1.1 2.2 4.4 1.1 5.5
V[⍋V]   (To sort in increasing order.)
1.1 1.1 2.2 3.3 4.4 5.5

V[⍒V]   (To sort in decreasing order.)
5.5 4.4 3.3 2.2 1.1 1.1

P←3 4 5 1 2
('ABCDE'[P])[⍋P]   (⍋P is the inverse of a permutation
ABCDE                vector P.)

X←'ABC'
Y←'DEF'
Z←'GHI'
(X,Y,Z)[⍋⍋ 0 2 1 1 2 0 0 2 1]
AGDEHBCIF   (Select next from X for a 0, Y for a 1,
             Z for a 2.)
```

```
□IO←0
⍋3.3 1.1 2.2
1 2 0    (0-origin.)
```

JOIN: $R←A,[K]B$

The join function connects A and B along a coordinate already existing in A or B or along a new coordinate of length 1 inserted into each. The first elements along the coordinate come from A and the rest come from B. When K is an integer, the operation is called catenate. When K is not an integer, the operation is called laminate and the new coordinate of length 1 is inserted into each argument between the existing ⌊K coordinate and ⌈K coordinate.

Requirements for A and B. Except for the special cases below, A and B must have the same rank, and dimensions other than the Kth must be the same. The types of A and B must be the same unless one or both are empty arrays. (Warning: some APL systems do not allow empty arrays to have a different type. It is recommended that differing types be avoided for compatibility.) The shape of the result is the same as the shape of the two arguments except that the Kth coordinate of the result is (ρA)[K]+(ρB)[K]. If both arguments are empty and of differing types, the result is numeric.

Exception cases. If A or B is a scalar (but not both), it is reshaped to have the shape of the other argument except that the Kth dimension is 1 for catenate. If both arguments are scalars, they are treated as one-element vectors for catenate. For catenate, one argument may have a rank 1 less than the rank of the other argument. In this case a new coordinate of length 1 is inserted to become the Kth.

Examples

```
1 2 3,4 5 6
1 2 3 4 5 6    (Joining two vectors.)

□←M←2 3ρ'*'
***
***

□←M←3 3ρ'O'
OOO
OOO
OOO
```

`M,[1]N`

```
***
***
ooo
ooo
ooo
```

`□←L←2 4ρ'□'`

```
□□□□
□□□□
```

`M,L`

```
***□□□□
***□□□□
```

`M,'+'`   (The scalar is treated as a one-column matrix.)

```
***+
***+
```

`M,'34'`   (A vector is treated as a one-column matrix.)

```
***3
***4
```

`M,[1]'345'`   (A vector is treated as a one-row matrix.)

```
***
***
345
```

`1 2 3,[.5]4 5 6`   (Laminate along a new first coordinate.)

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

`1 2 3,[1.5]4 5 6`   (Laminate along a new last coordinate.)

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

`1 2 3,[1.5]4`

```
1 4
2 4
3 4
```

COMPRESS: `R←V/[K]B`

The compress function shortens B along the Kth coordinate by omitting those elements for which there are corresponding 0's in V.

Requirements for V and B. V must be a vector and all elements of V must be 1's or 0's. The length of V must be the same as $(\rho B)[K]$. The result has the same dimensions as B except that the Kth dimension is $+/V$.

Exception cases. If V or B is a scalar it is treated as a one-element vector. Then if V is a one-element vector, it is extended to the length of B along the Kth coordinate. If B is a one-element vector, it is extended to the length of V.

Examples.

```
1 0 1 0 1/1 2 3 4 5
1 3 5

1 0 1 0 1/'ABCDE'
ACE

1/'ABCDE'
ABCDE

0/'ABCDE'
```
    (Blank line indicates an empty result.)

```
□←M←3 4ρ112
1 2 3 4
5 6 7 8
9 10 11 12

1 0 1 1/[2]M
1 3 4
5 7 8
9 11 12
```
    (Same as `1 0 1 1/M`.)

```
1 0 1/[1]M
1 2 3 4
9 10 11 12

1 0 1/4
4 4
```

EXPAND: `R←V\[K]B`

The result is formed by expanding B along the Kth coordinate by filling with zeros (if B is numeric) or blanks (if B is of character type) in those positions in R for which there are corresponding 0's in V.

Requirements for V and B. Ignoring the special cases, V must be a vector containing only 1's and 0's such that $(+/V)=(\rho B)[K]$. The result R has the same dimensions as B except that the Kth dimension is ρV.

Exception cases. If V or B is a scalar, it is treated as a one-element vector.

Examples.

```
1 0 1 0 1\1 2 3

1 0 2 0 3


□←M←2 3ρι6
1 2 3
4 5 6


1 0 1 0 1\M
1 0 2 0 3
4 0 5 0 6


1 0 1\[1]M        (Same as 1 0 1\M.)
1 2 3
0 0 0
4 5 6


1 0 1\[1]M
1 2 3
0 0 0
4 5 6


ρ1/2
1


' '=0\''          (An empty array can be expanded.)
1


0\ι0              (A vector result.)
0
```

TAKE: R←V↑B

The take function selects |V[K] first elements (for V[K]>0) along the Kth coordinate of B. If or last elements (for V[K]<0). If |V[K] exceeds (ρB)[K], zeros (if B is numeric) or blanks (if B is of character type) are used to provide the extra elements.

Requirements for V and B. Ignoring the special cases below, V must be a vector having an integer for each dimension of B. That is, (ρV)=ρρB. The result R has dimensions |V.

Special cases. If V is a scalar, it is treated as a one-element vector. If B is a scalar, it is treated as a one-element array of rank ρV.

Examples.

```
3↑1 2 3 4 5
1 2 3


-3↑1 2 3 4 5
3 4 5
```

```
3↑'ABCDE'
ABC


5↑1 2 3
1 2 3 0 0


□←M←3 4ρι12
1  2  3  4
5  6  7  8
9 10 11 12


2 -5↑M (First 2 rows, last 5 columns.)
0 1 2 3 4
0 5 6 7 8


3↑ι0              (Take can be applied to an empty array.)
0 0 0
0 0 0


2 3↑5             (5 is treated as a 1 by 1 matrix.)
5 0 0
0 0 0
```

DROP: R←V↓B

The drop function forms its result by omitting |V[K] first elements (if V[K]>0) or last elements (if V[K]<0) along the Kth coordinate of B.

Requirements for V and B. Ignoring the special cases below, V must be a vector of integers, and ρV must be the same as ρρB. The result has dimensions 0⌈(ρB)-|V.

Special cases. If V is a scalar, it is treated as a one-element vector. If B is a scalar, it is treated as a one-element array of rank ρV.

Examples.

```
3↓1 2 3 4 5
4 5


-2↓'ABCDEF'
ABCD


10↓1 2 3
                  (Blank line indicates an empty result.)


□←M←3 4ρι12
1  2  3  4
5  6  7  8
9 10 11 12
```

1 ¯2 ↓M   (First row and last 2 columns are dropped.)
5 6
9 10

      ρ0 0↑3
1 1

      (,10)↑3
3

(The scalar was treated as a matrix.)

REVERSE: R←⌽[K]B

The reverse function reverses the order of elements along the Kth coordinate of B. The result has exactly the same shape as B.

Examples.

      ⌽3 4 5 6
6 5 4 3

      ⌽'ABCDEF'
FEDCBA

      □←M←3 4ρι12
1 2 3 4
5 6 7 8
9 10 11 12

      ⌽M
4 3 2 1
8 7 6 5
12 11 10 9

      ⊖M
9 10 11 12
5 6 7 8
1 2 3 4

(Same as ⌽[1]M.)

ROTATE: R←A⌽[K]B

The rotate function shifts elements of B along the Kth coordinate a number of positions specified by A. For positive elements of A, the elements move so that their indices decrease, and for negative elements of A their indices increase. Elements shifted beyond the end are replaced at the other end. The absolute value of the elements in A gives the number of positions shifted. The corresponding vector along the Kth coordinate of B is shifted.

Requirements for A and B. Ignoring the exception below, A must have one element for each vector in B along the Kth coordinate. That is, ρA must be (K≠ιρρB)/ρB. Thus the dimensions of A must be like those of B except that the Kth dimension of B is absent from A. The result has the same shape as B.

Special cases. If A is a scalar, it is extended to become an array having dimensions suitable for B. Rotation of a scalar is allowed, but the left argument must be a scalar, and the result is the same as B.

Examples.

      2 ⌽ 1 2 3 4 5   (Rotation by 2 positions to the left.)
3 4 5 1 2

      ¯2 ⌽ 1 2 3 4 5   (Rotation by 2 positions to the right.)
4 5 1 2 3

      2 ⌽'ABCDE'
CDEAB

      □←B←3 4ρι12
1 2 3 4
5 6 7 8
9 10 11 12

      0 ¯1 2 ⌽ B
1 2 3 4
8 5 6 7
11 12 9 10

(Rows are shifted.)

      0 ¯1 1 2 ⊖B   (Same as 0 ¯1 1 2 ⌽[1]B.)
1 10 7 12
5 2 11 4
9 6 3 8

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

(All rows are shifted by 1.)

MONADIC TRANSPOSE: R←⍉B

The monadic transpose function reverses the coordinates in B. Thus the last coordinate in R corresponds to the first in B, the second to the last corresponds to the second in B, and so forth. For a vector or scalar, the result is the same as the argument. For a matrix, the result is the usual matrix transpose. For an array of rank 3, R[I;J;K] is the same as B[K;J;I]. The shape of the result is ⌽ρB.

Examples.

```
□←M←3 4ρ⍳12
1  2  3  4
5  6  7  8
9 10 11 12

⍉M
1 5  9
2 6 10
3 7 11
4 8 12

□←C←3 4ρ'FOURFIVEFORT'
FOUR
FIVE
FORT

⍉C
FFF
OIO
UVR
RET

□←R3←2 3 4ρ⍳24
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24

ρ⍉R3
4 3 2

⍉R3
1 13
5 17
9 21

2 14
6 18
10 22

3 15
7 19
11 23

4 16
8 20
12 24
```

DYADIC TRANSPOSE: $R←V⍉B$

The dyadic transpose function interchanges coordinates of B according to the integer values in the vector V.

Requirements for V and B. Ignoring the special case below, V must be a vector having one element for each dimension of B--that is, V and B must satisfy $(ρV)=ρρB$. The elements must be integers such that $(⍳⌈/V)∊V$ and $V∊⍳⌈/V$ (all integers up to the largest element in V but no other values). The rank of R is $⌈/V$ in 1-origin or $1+⌈/V$ in 0-origin. The Ith coordinate of R is $⌊/(V=I)/ρB$. The Ith coordinate of B becomes the $V[I]$th coordinate of R. If two or more coordinates of B map into the same coordinate of R, the length along that coordinate is the least of the related dimensions in B.

Special case. If V is a scalar, it is treated as a one-element vector.

Examples.

```
□←M←3 4ρ'ABCDEFGHIJKL'
ABCD
EFGH
IJKL

2 1 ⍉ M
AEI
BFJ
CGK
DHL
          (R[I;J]=B[J;I].)

1 1⍉M     (R[I]=B[I;I]. The diagonal of the matrix.
AFK        Note that the length is the shorter
           of the two dimensions of the matrix.)

□←A3←2 3 4ρ⍳24
 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 ⍉ A3
 1  2  3  4
13 14 15 16

 5  6  7  8
17 18 19 20

 9 10 11 12
21 22 23 24
```

```
5  6  7  8       (R[I;J;K]=B[J;I;K].)
17 18 19 20
9  10 11 12
21 22 23 24

                 1 1 1⌽A3
1 18             (R[I]=B[I;I;I].  The main diagonal.)

1  5  9          1 2 1⌽A3
14 18 22         (R[I;J]=B[I;J;I].  A diagonal slice.)

1  14            2 1 2⌽A3  (R[I;J]=B[J;I;J].)
5  18
9  22
```

The expressions to the right which relate elements of B, V, and R are formed as follows: The indices applied to R are (⍳V)ρ'IJKL...', and the indices applied to B are 'IJKL...'[V].

BASE VALUE: R←A⊥B

The base value function evaluates its right argument as a representation of a number in a general number system described by its left argument. For example, 2 2 2⊥1 0 1 gives 5; the vector 1 0 1 is evaluated as a number represented in base 2. The left argument, 2 2 2, contains the radices of the number system. (Radices are ratios between the weightings of the positions.) For the simple case of a vector left argument A, the Kth weighting (in 0-origin) is ×/(-K)↑A. That is, the Kth weighting is the product of the last K elements of A. If W is a vector of these weightings, the result for A⊥B is W+.×B. Thus for the case 2 2 2⊥1 0 1 the result is +/4 2 1×1 0 1.

Requirements for A and B. Except for the special cases below, A and B must satisfy (¯1↑ρA)=1↑ρB (the last dimension of A must be the same as the first dimension of B). For arrays A and B, the vectors along the last coordinate of A are used to find vectors of weightings, and each vector along the first coordinate of B is evaluated according to each vector of weightings. The weightings are W←ϕ×\(ρA)↑ϕ,1. The result is then W+.×B (same as the dimensions of A+.×B).

Special cases. If A or B is a scalar, it is treated as a one-element vector. If the last dimension of A does not match the first dimension of B but one of the two dimensions is 1, that dimension is extended to match the other.

```
3723     24 60 60⊥1 2 3
                         (One hour, two minutes, and
                          3 seconds is 3723 seconds.)

3723     0 60 60⊥1 2 3  (The first element in the
                          left argument has no effect.)

         □←A←2 3ρ2 2 2 10 10 10
2 2 2
10 10 10

         □←B←3 4ρ1 1 3 2 0 1 4 0 1 0 5 3
1 1 3 2
0 1 4 0
1 0 5 3

           A⊥B
5   6   25  11
101 110 345 203   (Each vector is evaluated
                   along the last coordinate of A.)
                   (Each vector along the first coordinate of B
                   is evaluated according to each vector
                   along the last coordinate of A.)

7.75     .5⊥3 4 5   (Evaluates the polynomial (3×.5*2)+
                     (4×.5)+5.  The left argument is extended
                     to become a 3-element vector.)
```

REPRESENT: R←A⊤B

The represent function represents its right argument in the number system described by its left argument. For example, 2 2 2⊤5 gives 1 0 1. The left argument contains the radices of the number system. For a vector left argument and a scalar right argument, the result is given by the following function:

```
[1]  VR←A SREP B;N
[2]  N←ρA
[3]  R←Nρ0
[4]  L1:→(N=0)/0
[5]  R[N]←A[N]|B
[6]  →(B=0)/0
[7]  B←B-R[N]
[8]  B←B÷A[N]
[9]  N←N-1
     →L1 ▽
```

This function is a generalization of the usual method of converting between number systems by dividing and finding remainders.

Requirements for **A** and **B**. A and B may have any shape. Each element of B is represented according to the radices in each element of A. (If A is a scalar, it is treated as a one-element vector for this operation.) The vector along the first coordinate of A. (If A is a scalar, it dimensions of the result are $(\rho A),\rho B$ (i.e., the same as for outer product.)

Examples.

```
      10 10 10⊤273
2 7 3

      24 60 60⊤3723      (3723 seconds is 1 hour,
1 2 3                     2 minutes, and 3 seconds.)

      2 2 2⊤31           (High-order information is lost.)
1 1 1

      0 2 2⊤31           (High-order information is
7 1 1                     intercepted by using a zero.)

      10 10 10⊤34 281
0  2
3  8
4  1

      □←A←3 2ρ0 10 2 10 2 10
0 10
2 10
2 10

      A⊤281 323
70 80
 2  3

 0  1
 8  2

 1  1
 1  3
```

EXECUTE: $R←⍎B$

The execute function performs APL statements in the vector or scalar B.
  A result is returned only if the expression produces a result. When the execute function is performed as the last operation on a line, any result is automatically printed unless specification or a branch was the last operation within the argument. Branching in the argument has no effect, and any statement label is ignored. If execute is applied to a character

---

argument representing a list, the list is printed and the first list element is returned as the result of the execute function. Note that the present system does not allow expressions exceeding 150 characters to be executed. (Some statements of as few as 87 characters give a *LIMIT ERROR*.)

Examples.

```
      ⍎'1 24 48'      (Converting characters to numbers.)
1 24 48
```

MONADIC FORMAT: $R←⍕B$

The monadic format function returns a character array that when printed looks exactly like B (except possibly when $□PW$ is exceeded, in which case numbers in $⍕B$ could be split between lines). An argument of character type is returned unchanged. For a numeric argument, each column of B becomes several columns in the result (depending on $□PP$ and the numbers in the column), but the other dimensions of the result are the same as for the argument. Thus the rank of R is the same as the rank of B, and ρR matches ρB except that $\bar{1}↑\rho R$ (the last dimension of R) is generally greater than $\bar{1}↑\rho B$ (the last dimension of B). Note that the exact output format differs between APL systems and may even differ between versions of the same system. Programs should be written to be independent of such differences.

Examples.

```
      X←1 2 3
      +/⍴⍕'X'×X*2'
20

      P←⍕'A←⍳3'
      P
1 2 3

      P←⍕'2;3;4'
234

      ⍕2
2
```

Special case. A scalar numeric argument is treated as a one-element vector and thus produces a vector result.

Examples.

```
      ⍕1 2 3
1 2 3

      ρ⍕1 2 3
5
```

⍒3 4ρ⍳12
1 2 3 4
5 6 7 8
9 10 11 12

AB

      ⍒'AB'

      ρ⍒3 4ρ⍳12
3
      10

DYADIC FORMAT: Z←V⍒B

The dyadic format function represents columns of B according to pairs of integers in the vector V. The first number of a pair in V gives the width, and the second number gives the precision to be used in representing the number. The width is the number of character positions to be used for the column, and if 0 is used, the system chooses a width so that at least one blank will separate that column from the preceding column. The result R has the same dimensions as B except that the last dimension of R is usually greater than the last dimension of B. Character arguments are not allowed. The precision has the following significance:

precision≥0
The numbers are represented in decimal format. The precision is the number of digits beyond the decimal point. If the precision is zero, no decimal point appears.

precision<0
The numbers are represented exponential format. The number of digits shown is the absolute value of the precision, and if the precision is 1, no decimal point appears. Five columns are required for the exponent, unless the system chooses the width, in which case the number of columns required for the exponent depends on the magnitude of the numbers.

Exception cases. If B is a scalar, it is treated as a one-element vector. (Hence the result is a vector, never a scalar.) If V is a scalar or one-element vector, it is extended to become V←0,V. (Thus the width of the columns will be chosen by the system.) Then if ρV is 2 but ¯1↑ρB (the last dimension of B) is not 1, V is replicated to become V←(2×¯1↑ρB)ρV so that the pair of numbers in V will be applied to all columns.

Note that a domain error will result if a number cannot be represented in the space provided. However, there is no requirement that spaces separate numbers in the same row.

Examples.

      7 3⍒.3456 2.8 928
0.346 2.800928.000

      4⍒.3456 2.8 928
0.3456 2.8000 928.0000

      1 0⍒2 3ρ1 0
101
010
101

      10 ¯3⍒3 1ρ2.34567 4.23E18 ¯5.3E¯6
  2.35E0
  4.23E10
 ¯5.30E¯6

      ¯1⍒1E5 1.2E6 1.8E2
1E5 1E6 2E2

MATRIX INVERSE: R←⌹B

The matrix inverse function returns the inverse of a matrix B. The inverse is such that

$$(⌹B)+.×B ↔ I$$

where I is an identity matrix (i.e., a matrix with 1's along the diagonal and 0's elsewhere) having 1↑ρB rows and columns. Note that this uniquely defines R only as long as B has the same number of rows and columns. However, if B has more rows than columns, the result R can be uniquely defined by R←(⌹(⍉B)+.×B)+.×⍉B. The result is related to the result for the dyadic matrix divide function according to ⌹B↔I⌹B, where I is an identity matrix having ¯1↑ρB rows and columns.

Requirements for B. Ignoring the exceptions below, B must be a matrix such that (1↑ρB)≥¯1↑ρB (B must have at least as many rows as columns) and B must have an inverse. Note that some matrices do not have inverses and produce a DOMAIN ERROR if an inverse is requested. In particular, a square matrix with two identical rows or with one row that can be produced by multiplying other rows by factors and adding them has no inverse. Actually, there is no precise distinction between matrices that have inverses and those that do not, and ⎕CT is used in the test. Increasing the value of ⎕CT may prevent a DOMAIN ERROR, but the result so produced is less reliable and may be completely meaningless. The dimensions of the result are ⍉ρB (i.e., ρ⍉B).

Special cases. If B is a scalar, the result is the scalar ÷B. If B is a vector, the result is B÷/B*2. Except for the result rank, the scalar case is the same as if the scalar were treated as a one-by-one matrix, and the vector case is the same as would be produced by treating the vector as a one-column matrix.

Examples. See the examples at the end of the following discussion of matrix divide.

MATRIX DIVIDE: R←A⌹B

The matrix divide function solves systems of simultaneous equations or finds least-squares regression coefficients. When the matrix B has the same number of rows and columns, R is the solution to linear equations represented by the constant vector A and the coefficient matrix B. When B has more rows or columns, the result R contains the regression coefficients for a dependent variable A and independent variables in the columns of B. Note that the result is the same as (⌹B)+.×A.

Requirements for A and B. Ignoring the special cases below, B must be a matrix such that $(1↑ρB)≥ ̄1↑ρB$, and B must have an inverse (see the preceeding discussion of the matrix inverse function). Also, A must be a matrix such that $(1↑ρA)=1↑ρB$ (they must have the same number of rows). When A has more than one column, the result R has a solution column for each column of A. The result has the dimensions $(1↓ρB),1↓ρA$ (one row for each column of B and one column for each column of A). The result R satisfies B+.×R←→A if B is a square matrix. When B is not a square matrix, the result minimizes each element of

+/(A-B+.×R)*2

That is, B+.×R gives predicted values for the regression cofficients R, and A-B+.×R gives the residuals; so the sum of the squared residuals is minimized.

Special cases. The arguments may also be scalars or matrices. A scalar is treated as a one-by-one matrix, and a vector is treated as a one-column matrix. After this extension, $1↑ρB$ must match $1↑ρB$. The dimensions of the result are $(1↓ρB),1↓ρA$ where A and B here are the original arguments before extension.

Example 1. To solve the system of equations:

5 = x + 2y
4 = 5x + 3y

Use:

```
□←M←2 2ρ1 2 5 3
1 2
5 3
```

```
 ̄1  5  4 ⌹M      (A vector result.)
 3

     (2 1ρ5 4)⌹M      (A matrix result.)
 ̄1
 3
```

The answer is x= ̄1, y=3.

Example 2. Given V1←.8 .9 1.0 2.2 3.1, V2←1 2 3 1 2, and Y←4.5 6.6 9.2 8.3 7.1 find the values of A1 and A2 that most nearly satisfy Y=(A1×V1)+A2×V2 in the least squares sense.

```
□←Q←V1,[1.5]V2
0.8 1
0.9 2
1   3
2.2 1
3.1 2

□PP←3
□←Z←Y⌹Q
1.37 2.56
```

The predicted values for Y are:

```
Q+.×Z
3.66 6.35 9.04 5.58 9.37
```

and the residuals are:

```
Y-Q+.×Z
0.845 0.252 0.159 2.72  ̄2.27
```

Example 3. Using V1 and V2 from Example 2 and Y2←6.5 8.6 11.2 10.3 9.1, find A1, A2, and A3 that most nearly satisfy Y2=A1+(A2×V1)+A3×V2. This problem is like the previous one except that we imagine A1 to be the coefficient of a vector of 1's. The solution is given by:

```
□←B←1,Q
1 0.8 1
1 0.9 2
1 1   3
1 2.2 1
1 3.1 2

Y2⌹B
5.76 0.593 1.35
```

Chapter 6. Composite Functions

As described in Chapter 2, an operator is a special function that takes functions as arguments and produces a function as a result. Except for the result of the axis operator, these resulting functions are the composite functions. A few examples will help to illustrate this. The expression $+/12$ 3 ("the plus reduction of 1 2 3") is the same as 1+2+3. Similarly, $-/1$ 2 3 is 1-2-3 or 2 (remember that it is performed from right to left). The function symbol to the left of the slash indicates the particular dyadic scalar function to be used. The forms for composite functions are d/B (reduction), $d\backslash B$ (scan), $A\circ.dB$ (outer product), and $Ad.DB$ (inner product), where d and D represent symbols for any dyadic scalar functions.

REDUCTION: $R\leftarrow d/[K]B$

Reduction applies a dyadic scalar function repeatedly between elements in vectors along the Kth coordinate of B. For a vector B, the reduction is of the form

$$B[1]dB[2]d \dots dB[N]$$

For higher order arrays the same sort of operation is performed for each vector along the Kth coordinate. When the axis operator is omitted the operation is performed along the last coordinate. The alternate symbol $\not$ can be used to indicate the operation should be performed along the first coordinate.

Requirements for B. Elements of B must be in the domain of the scalar function used. Thus, character arguments are allowed only for the functions = and $\neq$. Except for the special cases below, the result has a rank that is one less than the rank of B and the dimensions of the result are $(K\neq i\circ\rho B)/\rho B$ (the same as the dimensions of B except that the Kth dimension of B is missing).

6-1

Special cases. A scalar is treated as if it were a one-element vector, and the result is then a scalar. If the length of B along the Kth coordinate is 1, the result is the same as the argument except that one dimension is removed. No operation is actually performed in this case, so no check is made to see whether the values are in the domain of the function, except that arguments of character type are still illegal for functions other than = and ≠. When B is empty but the result is not empty, the result contains the identity element for the function if one exists. The following table shows the identity elements used. Note that in some cases the identity elements are identities in a rather loose sense. Some are both, and some are identities only for logical arguments. Functions for which there is no identity in the table produce DOMAIN errors when applied along a coordinate of length 0.

```
Function   Identity          Function   Identity
  +           0                 *           1
  -           0                 ∨           1
  ×           1                 ∧           0
  ÷           1                 <           1
  ⌈        ¯1.26E322            ≤           0
  ⌊         1.26E322            >           1
  |           0                 ≥           1
```

Examples.

```
⌈/3 1 9 15
15                 (Largest element.)

⌊/3 1 9 15
1                  (Smallest element.)

×/1 2 3 4 5
120                (Product.)

+/1 2 3 4 5
15                 (Sum.)

-/1 2 3 4 5
3                  (Alternating sum; same as
                   1+(¯2)+3+(¯4)+5.)

÷/1 2 3 4 5
1.875              (Alternating product; same
                   as (1×3×5)÷2×4.)

□←P←3 4ρ112
1  2  3  4
5  6  7  8
9 10 11 12
```

```
⌈/P
4  8 12            (Largest element in each row.)

⌈⌿P
9 10 11 12         (Largest element in each column.)

+/3
3                  (A scalar is treated as a vector.)

∧/5
5                  (No domain check for one element.)

+/⍳0
0                  (An identity if the length is zero.)

+/3 0ρ0
0 0 0              (An identity for each of the 3 rows.)

∧/AεB
1 1 0 1 0 1        (Gives 1 if all elements of the vector A
                   occur in B.)

∨/AεB
1 1 0 0 0 0        (Gives 1 if any elements of the vector
                   A occur in B.)
```

SCAN: R←⍺\[K]B

Scan performs a series of reductions. For example, +\1 2 3 4 5 returns 1 3 6 10 15; that is, the Ith element is +/I↑B. For arrays other than vectors, the result has the same shape as the argument, and the elements along the Kth coordinate are produced by performing a reduction over the first I elements. Arguments of character type are not allowed. If the axis operator is absent, the last coordinate is assumed. The alternate symbol ⍀ can be used to indicate the operation is be performed along the first coordinate.

```
∨\0 1 0 0 1 0
0 0 1 1 1 1

∧\1 1 0 1 0 1
1 1 0 0 0 0

×\1 2 3 4 5 6
1 2 6 24 120 720

□←P←3 4ρ112
1  2  3  4
5  6  7  8
9 10 11 12
```

```
        +\P
1  3   6  10
5 11  18  26
9 19  30  42

        +\P
1  2   3   4
6  8  10  12
15 18 21  24
```

(Same as +\[1]P.)

OUTER PRODUCT: R←A∘.dB

Outer product applies a scalar dyadic function using all elements of A as left arguments and all elements of B as right arguments.

The rank of the result is (ρρA)+ρρB and the dimensions of the result are (ρA),ρB. Each result element has as its first ρρA indices the indices of the element used from A and has as its last ρρB indices the indices of the element used from B.

Examples.

```
1 2 3∘.×4 5 6 7

 4  5  6  7
 8 10 12 14
12 15 18 21
```
(Each element of the left argument is multiplied by each element of the right.)

```
1 2 3∘.=3 1 3

0 1 0
0 0 0
1 0 1
```
(The number of 1's, 2's and 3's in the right argument.)

```
+/1 2 3∘.-3 1 3

1 0 2
```

INNER PRODUCT: R←Ad.DB

Inner product applies the scalar function D between each vector along the last coordinate of A and each vector along the first coordinate of B, then performs a reduction using d to that result. The usual matrix product is A+.×B.

Requirements for A and B. Ignoring the special cases below, the last dimension of A must match the first dimension of B. The dimensions of the result are (¯1↓ρA),1↓ρB (all dimensions of A except the last and all dimensions of B except the first).

Special cases. If A or B is a scalar it is treated as a one-element vector. Then if the last dimension of A does not match the first dimension of B but one of the two dimensions is 1, that dimension is extended to match the other (thus allowing the array having the 1 as a dimension to be used repeatedly).

Examples. The following table shows examples for arguments of various ranks.

ρρA	ρρB	ρρR	Result
2	2	2	R[I;J]=d/A[I;]DB[;J]
2	1	1	R[I]=d/A[I;]DB
1	2	1	R[I]=d/ADB[;I]

The following examples illustrate useful inner products:

```
□←A←2 3ρ1 0

1 0 1
0 1 0

□←B←3 3ρ⍳9

1 2 3
4 5 6
7 8 9

A+.×B
(Matrix Product.)

 8 10 12
 4  5  6
```

```
'ABCD'+.='XZCD'  (Counts matches in corresponding
                  positions.)
2
```

```
□←TABLE←3 4ρ'FOURFIVESIX'

FOUR
FIVE
SIX
```

```
TABLE∧.='FIVE'     (Gives 1 for a row that matches
0 1 0               'FIVE'.)
```

This chapter discusses system functions and variables other than ⎕ and ⍞ (which were discussed in Chapter 2), other than the ones related to shared variables (which have not been implemented in the present version of APLUM), and other than ⎕FD and ⎕SY. System functions and variables allow communication with the APL system, and, to some extent, with the KRONOS operating system under the control of which the APL system runs. In most respects system functions and variables behave as other APL functions and variables except that: their names are distinguished by beginning with the symbol ⎕ or ⍞, they control the APL environment in ways that other functions and variables cannot, and the values of system variables can change between settings. For example, ⎕AI, which is a vector of accounting information, may be set by the user to any desired value, but the next time he requests its value, it will correctly reflect current accounting information--that is, the system resets the value of ⎕AI before it is read. Similarly, ⎕AI can be erased by the user, but the system gives it a value whenever its value is requested.

The system variables that affect operation of the APL system have restricted shapes and domains. For example, ⎕IO, the origin for indexing, must have a value of, 1 or 0. Any attempt to set ⎕IO to an improper value will result in a RANK ERROR or a DOMAIN ERROR. However, the user can erase ⎕IO or declare ⎕IO to be local to a function and then fail to assign it a value. When a system variable is undefined and its value is required for an operation, an IMPLICIT ERROR results. For example:

```
      ∇Z←IOTA B;⎕IO
[1]   Z←⍳B ∇

      IOTA 3
01: IMPLICIT ERROR
IOTA[1] Z←⍳B
        /
[1]   Z←⍳B ∇     (⎕IO is a local variable.)
```

Output Control

□PP←integer (1 to 15)
  Printing precision--maximum number of significant digits used for numeric output.

□PW←integer (30 to ¯1+2*17)
  Maximum printing width used for output.

□PL←pagesize,linecount (0 to ¯1+2*17)
  Print lines. Print lines to be used before a halt to allow the terminal operator to intervene, and count of lines used. If □PL[1] is 0, output will be uninterrupted.

Indicators affecting Primitive Functions

□CT←number (0 to .01)
  Comparison tolerance used for relational functions, membership, index-of, and domain tests.

□IO←0 or 1
  Index origin. Determines base for counting.

□RL←integer (1 to ¯1+2*47)
  Random link used by random number functions.

Function Definition

□ENV←0 or 1
  Environment control. Affects □CR, □FX, □EX, □NC, □NL, □STOP, □TRACE, □LOCK, □NAMES, and □COPY. If □ENV is 0, the global environment is used, and if □ENV is 1, the current environment is used.

matrix←□CR 'name'
  Canonical representation of a function in the form of a matrix.

Z←□FX matrix
  Fixes the function represented by the character matrix argument. The result returned is a vector containing the name of the function, or, if the operation failed, a numeric scalar line number for the erroneous statement.

vector←□EX 'names'
  Expunges (erases) objects named by the right argument. The result contains 1's for names that are now available, 0's for others.

vector←□NC 'names'
  Returns the name class for each name--0 for available, 1 for locked variable (label or group), 2 for unlocked variable, 3 for function, or 4 for distinguished name.

matrix←□NL V
matrix←'letters' □NL V
  The namelist functions return matrices of names in use. Which names are returned depends on class numbers in V--locked variables (labels or groups) if 1∈V, unlocked variables if 2∈V, functions if 3∈V, and defined distinguished names if 4∈V. The left argument of the dyadic form should contain letters to further restrict names to those beginning with those letters.

vector←□LOCK 'names'
  Locks functions and variables named by the right argument. The result is a vector containing 1's for success, 0's for failure.

Stop, Trace, and Timing Control

V □STOP 'name'
  Sets stop controls for lines specified by V and clears controls for other lines.

Z←□STOP 'name'
  Returns a vector of line numbers for which stop controls are set.

V □TRACE 'name'
  Sets trace controls for lines specified by V and clears controls for other lines.

Z←□TRACE 'name'
  Returns a vector of line numbers for which trace controls are selected.

V □LTIME 'name'
  Selects accumulation of line timing information for lines specified by V and clears time accumulation for other lines. Also clears time totals to 0.

Z←□LTIME 'name'
  Returns a matrix with line numbers in column 1 and times (in milliseconds) in column 2.

Program Library Functions

□WSID←'name'
  □WSID contains the workspace identification of the active workspace. This name is used when no name is given for □SAVE.

Z←□SAVE 'wsname [:passwd][/options]'
Saves a copy of the active workspace under the name specified. □SAVE'' (no name given) uses the name in □WSID.

A □SAVE 'wsname [:passwd][/options]'
Same as above except that A controls the state indicator of the active and stored workspaces. If A is 0 or 1 the state indicator is cleared or backed up to the last suspension, respectively.

□LX←'expression'
The latent expression is executed immediately after the workspace containing it is loaded.

□LOAD '[*account] wsname [:passwd]'
Activates a copy of a stored workspace and then executes the latent expression if one is defined.

matrix←V □NAMES '[*account] wsname [:passwd]'
Lists names used in a stored workspace. The result is a matrix of names of objects in the name classes specified by elements of V--locked variables (labels or groups) if $1 \in V$, unlocked variables if $2 \in V$, functions if $3 \in V$, and distinguished names if $4 \in V$.

matrix←□NAMES '[*account] wsname [:passwd]'
Returns a matrix of all names of classes 1, 2, and 3 in the workspace.

matrix←'names' □NAMES '[*account] wsname [:passwd]'
Returns a matrix of all names of classes 1, 2, and 3 from the stored workspace.

matrix←□COPY '[*account] wsname [:passwd]'
Copies all objects of classes 1, 2, and 3 from the workspace.

matrix←'names' □COPY '[*account] wsname [:passwd]'
Copies specified objects into the active workspace from a stored workspace.

□DROP '[*account] wsname [:passwd]'
Removes the stored workspace or file named by the right argument from the indicated library.

Z←□LIB '[*account] [name]'
Returns a matrix containing names, types, and sizes of files in a library. If an account number is given, information is given only for the files that are public or semiprivate or for which the user has access permission. If a name is given, detailed information about that one file is returned.

Error Processing

□TRAP integer
Specifies that errors are to be intercepted by a forced branch to the specified line of the currently executing function.

Z←□ERR
□ERR is a 3-row matrix of the last error message, the line having the error, and a pointer to the position of the error in the line.

matrix←□SIV vector
The result is a character matrix containing the rows of the state indicator with variables display specified by the right argument. □SIV ι□LC gives the entire display (in either origin).

V←□LC
□LC is a vector of⍺all line numbers appearing on the state indicator.

Miscellaneous System Communication

V←□AI
□AI is a vector of accounting information. □AI [1 2 3 4] gives: an encoding of the user's account number, accumulated central processor time, accumulated connect time, and accumulated keying time.

V←□AV
Atomic vector of all 256 APL characters.

V←□TS
Time stamp: current year, month, day, hour, minute, second, and millisecond.

V←□TT
Terminal type.

□WA←V
Working area: □WA[1] is the part of the maximum field length available for use, □WA[2] is the current field length, □WA[3] and □WA[4] are the minimum and maximum field lengths the user wishes APL to use.

□TM 'command'
Terminal node: commands are SYSTEM, OFF and ABORT, to return to KRONOS command processor, sign off, or abort batch job.

S←□DL seconds
Causes execution to delay for the specified number of seconds.

However, three system variables are so important that when they are undefined the system uses default values. Thus, when `⎕PW` is undefined the system uses 30 as the printing width. When `⎕PP` is undefined, normal output uses a value of 1. When `⎕CT` is undefined, the system uses zero as the comparison tolerance for domain tests, although numerical comparisons still give implicit errors. For example,

```
      ⎕EX'⎕CT'

1
      3=3
01: IMPLICIT ERROR   (Because ⎕PP is undefined for comparison.)

1 2 3
      ι3
      ι3+1E¯12
03: DOMAIN ERROR   (Because ⎕CT is zero for domain tests.)
      ι3+1E¯12
      /
```

Certain system variables are not stored in the workspace. These session variables remain in effect if another workspace is loaded and always have their normal values when an APL session begins. The session variables are `⎕WA`, `⎕PL`, `⎕TT`, `⎕TS`, and `⎕AI`.

NAME LISTS

Some system functions require arguments consisting of lists of names. In all cases such name lists can be either a vector of names separated by spaces, or a matrix of names with one name in each row. In either form extra spaces are allowed before or after names. When a system function returns a list of names as a result, the list is always in the form of a matrix because the matrix form is usually more convenient for manipulation by the program.

WORKSPACES

An APL workspace comprises variables, user-defined functions, the state indicator, and system variables that are currently defined. A <u>clear</u> workspace comprises the following:

```
an empty state indicator
⎕PP←10 (printing precision of 10 digits)
⎕PW←120 (up to 120 characters are printed per line)
⎕CT←5E¯11 (comparison tolerance is 5E¯11)
⎕IO←1 (index origin is 1)
⎕RL←16807 (random link is 16807)
⎕ENV←1 (local environment)
⎕ERR←3 0ρ''
```

In addition, the clear workspace also presently includes the APL functions `⎕FD` (which performs function definition mode), `⎕SI` functions (which performs system commands), and the mixed functions ⊟, ⍎, ⊤, and ⊥. Although some of these may eventually become incorporated into the APL system itself (and thus be removed from the clear workspace), they presently occupy considerable space in active and stored workspaces. See Chapter 11 to learn how to eliminate or restore these functions.

As functions and variables are defined, they become part of the <u>active workspace</u>. A copy of an active workspace can be saved. To use it at a later time, a copy of the saved workspace can be <u>activated</u> (that is, made active).

A stored workspace is a special kind of KRONOS "file." Under an account number (or user number) can be stored as many files as are allowed by restrictions imposed on the account number. The collection of files is known as a <u>library</u>.

APL workspaces and data files are ordinarily KRONOS *private* files, which means that other users cannot use them. A user may optionally save a workspace as a *semiprivate* file or *public* file by use of commands of the form `⎕SAVE'name/S'` or `⎕SAVE'name/PU'`. This allows other users to access the workspace but does not allow them to alter it. Other users can be given permission to access a *private* file by use of the KRONOS PERMIT control card (see Chapter 12). This gives selected user numbers permission to access the particular file. Further details about these file categories can be found in Chapter 9 and Chapter 12.

Passwords can be given to workspaces for additional security. When a workspace is given a password, other users must provide the password in order to access the workspace. However, the owner of the workspace need not provide the password in order to use it.

The first time a workspace is saved it can be given a password or a category (i.e., *private*, *semiprivate*, or *public*). Thereafter, the file password and category remain unchanged for subsequent save commands that replace the stored workspace. (Thus, the password and category options should not be provided for subsequent save commands.) To change the password or category you must load the workspace, drop the stored one, and then resave it with the new options. Alternatively, you can use the CHANGE control card (see Chapter 12).

Workspaces can optionally be saved in *indirect access* form (ordinarily they are saved in *indirect access* form). This option is chosen by using a command of the form `⎕SAVE'name/DA'`. the first time the stored file is established. *Direct access* workspaces are faster to load, save, or copy, but require more disk space.

The direct access option is appropriate for unusually large workspaces that are loaded or saved very often. A workspace can be changed to direct access form by loading it, dropping it, then resaving it using the DA option.

Workspace names must begin with a letter, which may be followed by additional letters and digits. However, the name must not exceed seven characters. Passwords may consist of 1 to 7 letters or digits.

NOTATION

Throughout this chapter, brackets are used to surround optional portions of expressions. The brackets themselves should not be used. For example,

)LOAD '[*account] wsname [:passwd]'

means that the account number and password are optional. Any of the following commands are of the correct form:

)LOAD 'ALGEBRA'
)LOAD '*A123456 ALGEBRA:SESAME'
)LOAD 'ALGEBRA:SESAME'
)LOAD '*A123456 ALGEBRA'

SYSTEM VARIABLES FOR OUTPUT CONTROL

Printing precision. □PP←integer (1 to 15) determines the maximum number of significant digits to be used for numeric output. The value of □PP determines the ... The result is rounded to □PP digits; hence if □PP is 3, 0.34567 would be printed as 0.346. See Appendix B for further details of numeric output format.

Printing width. □PW←integer (30 to ¯1+2*17) The value of □PW determines the line width used for output. When a line of output requires more character positions than □PW, the remaining characters are indented and continued on successive lines. Output of numbers will not cause individual numbers to be split between two lines, but output of character data representing numbers may cause numbers to be split between lines.

Print lines. □PL←pagesize, linecount □PL is primarily intended to facilitate the use of CRT terminals having a screen smaller than the total amount of output generated. Appropriate setting of □PL causes output to pause when the screen has been filled to allow the screen to be examined or cleared (if required) before more output is sent.

The first element of □PL should be set to the number of lines that will be used for actual output. The second element of □PL is a count of the number of lines actually used for input and output. When each output line or input line has been completed, □PL[2] is incremented by 1. If □PL[1]=□PL[2], the system prints ? on the next line and suspends further output until RETURN is pressed. (Any other input is treated as if RETURN has been pressed.) The program requesting input can be halted by use of an interrupt (see Appendix C). When RETURN is pressed, □PL[2] is reset to 0, and further output is sent. The value of □PL[2] can be reset to compensate for screen repositioning caused by graph mode output. The elements of □PL are restricted to positive integer values. If an attempt is made to set □PL[1] to 1, it actually is set to 0. If the last line on the screen is used for input, the ? is suppressed and normal input can be entered on that line. (The input request gives a pause to allow the screen to be read.)

□PL has a different meaning when API output is sent to a file rather than to ¯a terminal. Specifically, if API is not being used from a terminal or is being used from a terminal but the output file name is not OUTPUT, and if the shifted output option is in effect (see Appendix D), a page eject carriage control character is sent at the beginning of the next output line whenever the page size has been exhausted.

VARIABLES AFFECTING PRIMITIVE FUNCTIONS

Comparison tolerance. □CT←number (0 to .01) The comparison tolerance is used when comparing numeric values and when testing whether values are sufficiently close to integers:

1. Two numbers A and B are considered equal only if

   $(|A-B| \le \Box CT \times B$

2. A number B is considered to be in the integer domain if

   $(|(NINT\ B)-B| \le \Box CT + |\Box CT \times NINT\ B$

where $NINT\ B$ is the nearest integer to B, defined by:

   ∇Z←NINT B
[1] Z←(×B)×⌊.5+|B∇

The value actually used for the operation is $NINT\ B$. If □CT is undefined, zero is used as □CT.

Random link. ⎕RL←integer (1 to ¯1+2*47) determines the next random number to be produced by roll or deal. Each time a random number is requested, the value of ⎕RL changes. A series of random numbers can be recreated by setting ⎕RL to the same initial value and repeating the same requests. Because the value of ⎕RL is saved with the workspace, it may be desirable to reset it after the workspace is loaded to a value based on the current time of day so that the random numbers produced will not be the same as for the last session; for example, ⎕RL←+/⎕TS.

Index origin. ⎕IO←0 or 1 determines the origin for counting coordinates or elements along coordinates. In 0-origin the elements of a vector would be numbered 0, 1, 2, etc. All indexing should use values that are 1 less in 0-origin than in 1-origin. In addition, the following functions produce results that are 1 less in 0-origin than in 1-origin: A1B,ιB, ⍷B,?B, and ?B. In addition, the left argument for dyadic transpose should be 1 less for 0-origin, and all axis operators require values that are 1 less. That is, K should be 1 less in expressions like A/[K]B and ⌽[K]B.

FUNCTION DEFINITION

Environment. ⎕ENV←0 or 1 controls whether the functions ⎕CR, ⎕FX, ⎕EX, ⎕NC, ⎕NL, ⎕TIME, ⎕NAMES, ⎕COPY, ⎕STOP, ⎕LOCK, and ⎕TRACE refer to the global environment or to the current environment. When ⎕ENV is 0, the global environment is used, and when ⎕ENV is 1, the current environment is used. The normal value of ⎕ENV is 1, so the system functions listed above may refer to local variables and functions. However, when function definition mode is entered or when a system command is performed, only the global functions and variables are used. When the state indicator is empty, the current environment and the global environment are the same and ⎕ENV has no effect.

Canonical representation. matrix←⎕CR 'name' Canonical representation of a function. The right argument contains a character vector or scalar containing the name of the function to be returned. The result will have one row for each line of the function, including the function header. Lines will be indented one space unless they have labels or begin with a comment. If the argument does not name a function in the environment, a NAME NOT FOUND error is given. If the function named by the argument is locked, the result will have 0 0 as its shape.

Fix. Z←⎕FX matrix ⎕FX establishes the function represented by the character matrix argument. If the attempt to establish the function is

successful, Z will be a vector containing the name of the function. Replacement of previously existing functions is allowed and may result in SI DAMAGE if the function is halted. The SI DAMAGE error is processed as a normal error, except that if the state indicator entry for the currently executing function was damaged, error trapping is not allowed to take place. In this case the error is considered to be located at the last line entered in immediate execution mode. An attempt to establish a replace objects other than functions, ⎕FX cannot be used to function may also fail as a result of statement labels or local function header or duplicate use of an incorrectly formed variables. If the attempt fails, Z will contain a scalar row index of the line that was improper. Functions created by ⎕FX can be declared local to other functions.

Expunge. Z←⎕EX 'names' ⎕EX expunges (erases) functions and variables named by the argument. The result Z is a logic vector containing 1's in positions corresponding to names in the argument that are now free, and 0's in positions corresponding to names that remain unavailable for new uses. Erasure of a function or other (i.e.; a defined distinguished name), respectively. Incorrectly formed names in the argument cause a DOMAIN ERROR.

A locked function that is on the state indicator does not take effect until the function can erase itself longer on the state indicator. Thus a function that is no and not actually be erased until it exits. The unfinished execution can complete, but the name is immediately available for new uses.

Name class. vector←⎕NC 'names' ⎕NC returns information about use of the names in the right argument. The result Z contains 0, 1, 2, 3, or 4 according to whether the name is available (not in use), a locked variable (label or group), unlocked variable, a function, or other.

Name list. matrix←⎕NL V or Z←'letters' ⎕NL V The name list functions return lists of names in use. The right argument is a numeric vector such that ∧/V∊1 2 3 4. V indicates the classes of names for which information is desired--1 for locked variables (labels or groups), 2 for unlocked variables, 3 for functions, and 4 for distinguished names. The result Z is a matrix of the names. The left argument of the dyadic form may contain any number of letters, and names appear in the result only if they begin with those letters.

Lock. vector←⎕LOCK 'names' The variables and functions specified by the right argument are locked. A locked function cannot be displayed, and a locked variable cannot be reset using specification. (However, a locked variable can be reset by erasing it and then using specification.) Locking a variable is a very useful way to find where the variable is reset. When the variable has been locked, the next assignment to it will cause an error halt. Label

variables and groups are automatically locked to prevent them from having improper values. The result returned by □LOCK contains 1's in positions corresponding to names that are now locked and contains 0's for other names.

STOP, TRACE, AND TIMING CONTROL

The functions □STOP, □TRACE, and □TIME are closely related. In each case the right argument is a character vector or scalar that names a function, and the left argument for the dyadic form must contain nonnegative line numbers for which the control is to be set. Setting controls for any lines clears all controls of the same type for the other lines of the function. Elements of the left argument not in the range of line numbers are ignored. In all cases, an empty vector of line numbers can be used to clear the controls. An empty character vector is allowed as a left argument for notational convenience (e.g., '' □STOP 'PLOT'). The monadic forms of the functions return information about controls that are currently set.

Stop control. V □STOP 'name' and Z←□STOP'name'. When the stop control is set for a particular line, execution of the function suspends before execution of the line begins, and the system prints STOP SET, the name of the function, and the line number. To continue execution where it stopped, issue a branch to the line number just printed. Stop control at line 0 causes suspension just prior to exit from the function. The monadic form returns a vector of line numbers for which stop controls are currently set.

Trace control. V □TRACE 'name' and Z←□TRACE 'name'. Setting trace control for a line causes the function name and line number to be printed each time after the line has been executed, and if the result of the line was used for a branch or assignment, the result is printed even though it ordinarily would not be. Setting trace control for line 0 causes tracing of the exit from the function and causes printing of the explicit result of the function (if it has one). The monadic form returns a vector of line numbers for which trace controls are set.

Line timing control. V □LTIME 'name' and Z←□LTIME 'name'. Setting the line timing control for a line causes the Central Processor time for that line to be accumulated. The time for a line is accumulated until line timing controls for the function are reset, at which time all accumulated times are set to zero. An attempt to set line timing control for line 0 of a function causes a DOMAIN ERROR. The result returned by the monadic form is a 2-column matrix—the first column contains the line numbers for which the line timing control is set, and the second column contains the total times for the lines. Because the time clock has a resolution of one millisecond, each parcel of time used by the line is measured with limited accuracy, and lines consuming

very little time or lines consuming time in small parcels can be expected to show relatively large inaccuracy in accumulated times. Note that the times accumulated for a recursive function can count the time more than once.

PROGRAM LIBRARIES

Workspace identification. □WSID←'name'
The variable □WSID contains the name of the active workspace. The name of the active workspace is used as the name for storing the workspace if no name is specified, when □SAVE or)SAVE is used. The name must begin with a letter, which may be followed by additional letters or numbers. No spaces are allowed within the name, but spaces may precede or follow the name. The name must not exceed seven characters.

Save. Z←□SAVE 'wsname [:passwd][/options]'
□SAVE saves a copy of the active workspace under the specified name and attaches to the saved workspace the password if one is used. If a password is used, it must be separated from the name by a colon. The name itself may be omitted, and in this case the value of □WSID is used as the name. When □SAVE is executed from a function, the state indicator of the saved workspace will show suspension where □SAVE was executed. The options may include S, P, or PU (for semiprivate, private, or public category) or may include DA or IA for direct access or indirect access. The list of options may include any desired number of options, separate by spaces, as long as the options do not include contradictory choices. The options and password may be specified only when the saved workspace is first established. If no options are specified, the workspace is saved as a KRONOS indirect access private file if the saved workspace is being created; otherwise it is saved in the same form as before.

The result returned is a vector of the workspace name and the current date and time. However, when □SAVE is used immediate execution mode, the name, date, and time are printed rather than being returned as a result.

Dyadic save. A □SAVE 'wsname [:passwd][/options]'
The dyadic save function is like the monadic form except that it permits control over the state indicator in both the active and the saved workspace. The argument A may be a numeric scalar or vector. If A is 0, a clear state indicator results, and if A is 1, the state indicator is backed up to the point of the most recent suspension (or cleared if there have been no previous suspensions). Note that a function calling the dyadic □SAVE function always ceases to execute because of the change in the state indicator, unless an error prevented completion of the operation. Dyadic save prints the workspace name and the current date and time.

Latent expression. `□LX←'expression'` The latent expression in a workspace is executed immediately when the workspace containing it is loaded. When a workspace has no latent expression, the keyboard unlocks for the user to specify the first operation to be performed. A successful load operation ordinarily causes the time and date when the workspace was saved to be printed, but when the workspace contains a latent expression this message is absent.

Load. `□LOAD '[*account] wsname [:passwd]'` The function `□LOAD` activates a copy of a stored workspace. The right argument must contain the name of the workspace to be loaded, the password for the workspace (if it requires one), and the account number under which the workspace is stored (if different from the user's own). A successful load results in execution of the latent expression if the workspace being loaded has one. If the workspace has no latent expression, the time and date when the workspace was saved are printed.

Name list for stored workspaces. `matrix←V □NAMES '[*account] wsname [:passwd]'` The function `□NAMES` returns a matrix list of the names used in a stored workspace. The list returned is controlled by □ENV in the active workspace. The right argument is the same as the right argument for □LOAD. The vector V may contain the integers 1, 2, 3, or 4 to specify what classes of names should be returned--locked variables (labels or groups) if 1∊V, unlocked variables if 2∊V, functions if 3∊V, and distinguished names if 4∊V.

Copy. `matrix←'names' □COPY '[*account] wsname [:passwd]'` The function □COPY copies functions and variables from a stored workspace to the active workspace. The account number, workspace name, and password are the same as described for □LOAD. The list of names in the left argument specifies objects to be copied. However, if copying the object would cause replacement of objects already in the active workspace, the copying process is inhibited. If □ENV is zero, copying will be from the global environment of the stored workspace to the global environment of the active workspace, and if □ENV is 1, the current environments will be used. The result from □COPY is a matrix of names of objects not copied because they were not found, because WS FULL occurred, or because they already were in use in the active workspace.

Monadic copy. `error←□COPY '[*account] wsname [:passwd]'` Like dyadic copy except that all objects of classes 1, 2, and 3 (see □NC) are copied.

Drop. `error←□DROP '[*account] wsname [:passwd]'` The function □DROP removes a stored workspace (or other KRONOS file) from the user's library. A password must be specified if an account number is specified and differs from the one used to sign on to the system and if the file has a password.

Library list. `list←□LIB '[*account] [name]'` The function □LIB returns the names of files stored under the specified account number (or the user's own account number if no account number is specified). The list is a matrix such that each row has the following fields:

File name: 7 characters
File type: 2 characters
File size (in words): 6 characters

One space separates adjacent fields. When a file name is given, detailed information about that particular file is returned. The format when a name is provided is illustrated below:

```
□LIB'*APL1 FILESYS'
FILESYS WS   2059
IA  S   RD
75/05/12 11:46:58   (When created.)
75/05/30 13:03:30   (Last change.)
75/07/31 12:30:59   (Last access.)
```

The first row gives the name, type of file (WS for workspace, F for APL file, blank for all others), and the size in words. The second row indicates the file is indirect access (the other possibility would be DA for direct access), the file category (S for semiprivate, P for private, and PU for public), and the mode of access permitted for other users, (RD for read, WR for write, RM for read-modify, MD for modify, AP for append, RA for read-append).

ERROR PROCESSING

Some system functions respond to certain error conditions by returning a result to indicate the error. However, APL handles most errors by suspending execution at the point of the error, printing a message, and unlocking the keyboard for a new command to be entered. Note that a keyboard interrupt (see Appendix C) is treated as an error, as is typing ⍉ (O, BACKSPACE, U, BACKSPACE, T). However, halts due to stop controls are not errors. Special exceptions arise when the error is in an argument to the execute function, in a quad input entry, or when □TRAP has been used to intercept errors.

Errors in an argument to the execute function normally cause two error messages to be printed. The first shows the execute argument, and the second shows the line where the error was called (more precisely, the most recent pendent line other than lines of locked functions or arguments to execute).

Errors in lines entered for quad input cause the request for input to be repeated. If the error was encountered in a function called by the input line, the request for input is not repeated and normal error processing ensues.

For security reasons, lines of a locked function are not shown in error messages. Any error in a line of a locked function is treated as if it were situated in the line where the locked function was called (more precisely, the most recent pendent line other than lines of locked functions or arguments to execute).

The function ⎕TRAP can be used to designate a line of the currently executing function to intercept errors. Once this has been done, error trapping is in effect and an error in any line of the function causes a forced branch to the trap line. The error trap is in effect for functions called by that function or for functions that are in turn called by those it calls, etc.

The scope of error trapping is analogous to the scope of local variables. A function with a trap line remains in control of errors unless a function called by it sets its own trap line. The newer trap line takes precedence over the old one until the called function completes execution or clears its trap. The trap also takes precedence over the normal processing of errors in quad input lines.

When a workspace is loaded, an interrupt may be acted upon as an error before the latent expression has been executed and the error trap has been enabled. To prevent this situation, a function with a trap can be halted using a stop control before the workspace is saved. The latent expression can then deliberately cause an error in order to invoke the trap line.

Error matrix. ⎕ERR
The character matrix ⎕ERR contains the last error message. Row 1 has the type of error. Row 2 has the name of the function (truncated to 8 characters for long names) the line number (surrounded by brackets), and the line itself. Row 3 of ⎕ERR has a slash to indicate where the error was found in row 2. The number of columns in ⎕ERR varies according to the longest of the three rows.

The first row always shows the type of error actually encountered, but the location of the error as shown in rows 2 and 3 can be different from the actual location of the error under the following conditions:

1. If error trapping is in effect, the error is treated as an error in the pendent line of the trapping function.

2. If error trapping is not in effect and the error occurred in a line of a locked function or in an argument to execute, the location of the error is considered to be the most recent pendent line that is not an argument to execute or a line of a locked function. However, an error in a locked function uses trapping causes ⎕ERR to contain a line of the locked function. It is advisable for the locked function to localize ⎕ERR in order to protect its security.

Trap set. ⎕TRAP integer
The ⎕TRAP function sets, resets, or clears the trap line for the currently executing function. Use of ⎕TRAP from immediate execution mode has no effect. The argument must be an integer. If the integer is within the range of line numbers, that line becomes the trap line. If the number is 0 or exceeds the number of lines, trapping causes exit from the function. The trap can be cleared by ⎕TRAP←0. Once trapping is in effect, an error in that function, in ⎕ input, or any function invoked by it causes a forced branch to be taken to the trap line, and the trap state is cleared.

Note that ⎕TRAP must be used to set the trap again before additional errors can be intercepted by that function. Hence a second error during processing of the trap routine results in either normal error processing or error processing by a function that invoked this one. If trapping is in effect, execution of functions can still halt as a result of a stop control. However, the trap then remains in effect for errors in immediate execution mode.

When a forced branch to the trap line occurs, at least one function will execute before an interrupt is detected. For complete security, the trap line can immediately reset the trap.

Location counter. ⎕LC
The variable ⎕LC contains a vector of all line numbers appearing on the state indicator. The numbers appear in the same order as in the)SIV display--that is, the numbers of the most recently invoked lines appear first. The first element is the number of the function line currently executing.

State indicator and variables. matrix←⎕SIV vector
The function ⎕SIV returns rows of the state indicator, including local variables. The argument must be a vector or scalar containing integers. The value returned is a character matrix containing a portion of the)SIV display selected by the right argument. ⎕SIV prints the entire SIV display (in either origin). If a value in the argument exceeds the range of appropriate row indexes for the SIV display, a blank line

appears in the corresponding row of the result. Note that only entries for function lines appear on the state indicator--not execute arguments, quad input lines, or immediate execution lines.

MISCELLANEOUS SYSTEM COMMUNICATION

Accounting information. ⎕AI
The variable ⎕AI is a numeric vector of the following accounting information:

⎕AI[1] - A numeric encoding of the user's account number. For a character vector V containing the 7-character account number, the value of ⎕AI[1] is generated in one origin by 100⊥'ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'ιV
⎕AI[2] - Central processor time used.
⎕AI[3] - Total connect time.
⎕AI[4] - Total time the keyboard has been unlocked. Includes part of the time required for the system's response.

Times are in milliseconds and are cumulative since signing on to APL.

Atomic vector. ⎕AV
The vector ⎕AV contains all 256 characters manipulable by APL. The ordering is such that the first 128 characters are in ASCII order. Note that the ordering of characters in ⎕AV is system dependent, and programs that depend on the ordering of characters in ⎕AV cannot be easily transferred to other APL systems. See the table in Appendix C to find positions of particular characters.

Time stamp. ⎕TS
The value of ⎕TS is a 7-element numeric vector expressing the current point in time. The elements are in the following order: the year (e.g., 1975), month (1 for January), day of the month, hour (0 to 23), minute, second, and millisecond. The last element is always 0 because the operating system does not report the time of day to millisecond precision.

Terminal Type. ⎕TT
The value of ⎕TT identifies the type of terminal in use. The value is a numeric scalar as follows:

1 - Correspondence
2 - Type-pairing
3 - Bit-pairing
4 - ASCII-APL
5 - Teletype Model 33

6 - Full ASCII
7 - Batch ASCII
8 - Batch 501 Printer
9 - Teletype 38, arrangement 3

Working area. ⎕WA
The value of ⎕WA is a 4-element vector of: the part of the maximum field length available, the current field length, the minimum field length the user wishes used, and the maximum field length the user wishes used. The field length is the actual memory space occupied by the APLUM system and the workspace. The user can set constraints on the field length to be used in order to optimize performance (see Chapter 11). Attempts to reset the first two elements of ⎕WA have no effect. The maximum field length cannot be set to less than that which is currently required. Setting ⎕WA[4] to more than the user's validation limit (see Chapter 12) or more than the field length limit imposed by the computer operator may actually increase the chance of a WS FULL error.

Terminal mode. ⎕TM
The terminal mode function allows the following operations:

⎕TM'SYSTEM'   Returns control to the KRONOS command processor.

⎕TM'OFF'   Signs the user off.

⎕TM'ABORT'   Terminates job with KRONOS abort error flag set.

Note that these commands do not cause the active workspace to be saved.

Delay. Z←⎕DL seconds
Causes execution to delay for the number of seconds requested. The delay does not involve consumption of Central Processor time. The result returned is the actual delay that occurred (usually slightly more than requested). The delay cannot be interrupted.

Chapter 8. System Commands

System commands in APLUM provide the same capabilities as some of the system functions and variables. The system commands are provided for compatibility with other APL systems. There are some advantages to using system functions and variables instead of system commands--the system functions and variables can be used in programs (system commands cannot) and system functions and variables are more efficient. For more complete discussions of the operations performed by system commands, see the related system functions in Chapter 7.

GROUPS

The APLUM system, unlike some other APL systems, does not have a distinct data type for "groups." However, the APLUM system commands allow a character matrix of names to be used for the same purposes as groups in the other systems. For example, if *GRPX* is a matrix of names, the command)*ERASE* .*GRPX* would erase *GRPX* and any objects referenced by the names in *GRPX*. The period in the command is required to indicate that objects referenced by *GRPX* are to be erased, not just *GRPX* itself. The general system convention for distinguishing groups is that all group names should begin with *GRP*. Matrices of names that do not begin with *GRP* can be used as groups, but they will not be listed by the command)*GRPS*. The names in the group definition can be preceded by a period, which causes them to be interpreted as a reference to another group.

)*CLEAR* (Equivalent to □*LOAD*'*APL0 CLEARWS*')

The command)*CLEAR* activates a clear active workspace (described in Chapter 7) and erases all indirect access files and unties all direct access files.

)CLEAR
Activates a clear workspace.

)SAVE [wsname] [:passwd] [/options]
Saves specified functions and variables.
Saves a permanent copy of the active workspace.

)ERASE names
Erases specified functions and variables.

)LOAD [*account] wsname [:passwd]
Activates a copy of the specified workspace.

)DROP [*account] wsname [:passwd]
Removes a permanent workspace from the library.

)COPY [*account] wsname [:passwd] [names]
Protected copy of all global objects of classes 1, 2, and 3 or selected global objects from a stored workspace to the active workspace.

)UCOPY [*account] wsname [:passwd] [names]
Unprotected copy of all global objects of classes 1, 2, and 3 or selected global objects from a stored workspace to the active workspace.

)LIB [*account] [name]
Displays names, types, and sizes of all files, or displays detailed information about a single file.

)SYSTEM
Returns control to KRONOS command processor.

)OFF
Signs a user off.

)SI
Displays the state indicator.

)SIV
Displays the state indicator along with names of variables.

)FNS [letter]
Displays the names of functions.

)VARS [letter]
Displays names of variables.

)GRPS [letter]
Displays names of groups.

)GRP group-name
Displays names in a specified group.

)GROUP group-name names
Forms a group having specified names.

)ERASE names (Equivalent to ☐EX 'names')
Erases all global objects specified by the list of names. If a name is preceded by a period, the name is treated as the name of a group. The erasure erases the group itself (actually a matrix of characters) and the objects referenced by the group.

)SAVE [wsname] [:passwd] [/options]
(Equivalent to ☐SAVE '[wsname] [:passwd] [/options]')
The)SAVE command saves a copy of the active workspace under the name specified or under the name in ☐WSID if no name is given.

)LOAD [*account] wsname [:passwd]
(Equivalent to ☐LOAD '[*account] wsname [:passwd]')
The)LOAD command activates a copy of a stored workspace. A password is required if the workspace has a password and is stored under another user number. After the workspace has been loaded, the system executes ☐LX if ☐LX is defined.

)DROP [*account] wsname [:passwd]
(Equivalent to ☐DROP '[*account] wsname [:passwd]')
The)DROP command removes a stored workspace or other KRONOS file from a library. If the workspace is in another user's library, a matching password must be given if the stored workspace has a password. The user must also be authorized to alter the existing file.

)COPY [*account] wsname [:passwd] [names]
)UCOPY [*account] wsname [:passwd] [names]
The)COPY command performs a protected copy of global functions and variables from a stored workspace to the active workspace. The)COPY command will not replace objects in the active workspace with objects from the stored workspace having the same names. The)UCOPY command performs an unprotected copy and will replace objects having the same names. If no list of names is given, all objects of classes 1, 2, and 3 are copied. If a name in the list is preceded by a period, the name is assumed to refer to a group and objects named in the group are also copied. The ☐COPY function can be used instead of)COPY if groups are not to be copied. The form is ['names'] ☐COPY '[*account] wsname [:passwd]'.

)LIB [*account] [name] (Equivalent to ⎕LIB '[*account][name]')

The)LIB command displays names, types, and sizes of all files the user is authorized to access, or, if a file name is specified,)LIB displays detailed information about that particular file. The format is the same as for ⎕LIB (see Chapter 7).

)SYSTEM (Equivalent to ⎕TM'SYSTEM')

The command)SYSTEM causes the user to leave APL control and allows the KRONOS command processor to execute subsequent commands. The active workspace is not saved.

)OFF (Equivalent to ⎕TM'OFF')

The)OFF command signs a user off the system.

)SI
)SIV (Equivalent to ⎕SIV ιρ⎕LC)

The command)SI lists the state indicator, and the command)SIV lists the state indicator and all local variables. See Chapter 2 for the format of the display.

)FNS [letter] (Equivalent to ⎕NL 3)
)VARS [letter] (Equivalent to ⎕NL 2)
)GRPS [letter] (Roughly equivalent to 'G' ⎕NL 1)

These commands list the names of defined global functions, variables, and groups, respectively. If a letter is included, only names beginning with that letter or letters that follow that letter in the alphabet are shown. The command)GRPS lists variable names that begin with GRP.

)GROUP group-name names

The command)GROUP defines a group, extends a group, or erases a group definition. Groups are actually represented as character matrices. If the group-name itself is the first name in the list of names, any previously defined group is extended by the addition of the remaining names. If no names are given, the group definition is erased but objects named by the group are not erased. Names listed in the command can be preceded by a period in order to include a period in the group definition (to indicate the name refers to another group).

)GRP grpname (Equivalent to)GRP grpname)

The command)GRP displays the definition of the indicated group. If the group is not defined or is not a character matrix, an error message is given.

Chapter 9. File System

This chapter discusses files from the APL user's point of view. The APLUM system supports two distinct types of files: APLUM files, and KRONOS coded files. The use of files enables programs to deal with large quantities of data that would not fit into a workspace, and files also provide a convenient way for programs to communicate with one another.

Further information about KRONOS files can be found in the KRONOS 2.1 Reference Manual.

APLUM FILE CONCEPTS

An APLUM file is a collection of APL arrays with each array identified by a nonnegative integer. The following example shows creation of a file and writing and reading a few records (arrays) of the file.

```
)LOAD  *APL1 FILESYS       (File system functions are
                            loaded from APL1.)

'SAMPLE' FCREATE 9         (The FCREATE function is used
                            to create a file with the name SAMPLE
                            and with 9 as its number.)

'RECORD 3' FWRITE 9 3      (The left argument is written
                            to file 9 as record 3.)

(3 3ρ⍳9)FWRITE 9 1

(2 3ρ'□') FWRITE 9 28

FREAD  9 1                 (The records can be read in
1 2 3                       any order.)
4 5 6
7 8 9
```

9-1

Summary of File Functions.

'[*account] filename [:passwd] [/options]' FCREATE fnum
    Creates a file. Options are DA, C, WR, S, or PU.

array FWRITE fnum[,rnum]
    Writes array on file number fnum as record rnum.

result←FREAD fnum[,rnum]
    Reads the record numbered rnum from the file numbered fnum.

FRDEL fnum[,rnum]
    Deletes record rnum from file fnum.

rnum←FFREE fnum
    Returns the least record number not presently in use in file fnum.

FPOS fnum,rnum
    Sets position of file fnum to rnum.

result←FSTATUS fnums
    Returns the status of all files specified by the right argument. The result is a vector or matrix according to whether the argument is a scalar or vector. Columns are: (1) largest record number, (2) current position, (3) file size, (4) unused space, (5) lost space, (6) space not used because record sizes not divisible by 64, (7) 1 if coded file, (8) 1 if DA type, (9) 1 if absent record encountered by last read attempt.

PSTATUS
    Prints status information (with discriptive headings) for all active files.

result←FNUMS
    Returns a matrix of user numbers and names for all tied files.

result←FNAMES
    File numbers in use for tied files.

FRETURN fnums
    Unties specified direct access files and erases specified indirect access files.

FUNTIE fnums
    Unties files in right argument. This leaves a permanent copy.

FERASE fnums
    Erases all files specified by right argument. Erasure affects active file and for DA type also affects permanent file.

'[*account] file-name [:passwd] [/options]' FTIE fnum
    Ties a file with specified options--RD for read only (other users can read at the same time), and RM for read-modify (another user can modify at the same time).

result←CFREAD fnum
    Coded read. Result is character vector or numeric scalar--1 for end of record, 2 for end of file, 3 for end of information.

array CFWRITE fnum
    The left argument is written to the coded file fnum. The argument should be a character scalar, vector, or matrix, or integers--1 to write end of record or 2 for end of file.

integers CFPOS fnum
    Positions file. Operations indicated by first integer are: 0 for rewind, 1 for skip record, 2 for skip file, 3 for skip to end. Second integer for skip record or skip file may be included as repetition count.

jobname←CSUBMIT fnum
    Submits the coded, indirect access file fnum as a batch job and erases the active copy.

      FREAD 9 28
□□□
□□□

      FREAD 9 3
RECORD 3

      FUNTIE 9)
    After the above steps, the user can store the file (using FUNTIE 9), an operation analogous to saving a workspace. The user could then sign off the system. The information in the file would remain intact and could be accessed or modified at a later time.

File limits. Individual file records are allowed to be as large as desired. However, user numbers have associated restrictions that may limit the total number of files, the total size of all files, the size of individual files, and whether the user can create direct access files. This information must be obtained from the installation or by executing the LIMITS control card (see Chapter 12).

Tied files. It is usually more convenient to use numbers within a program to identify a file rather than using the file name. All file operations require this file number. The number is tied to the file when the file is created using FCREATE or when a previously stored file is accessed using the FTIE function. Once a file has been assigned a number, the file is said to be tied. The file can be released by using the FUNTIE operation, the FRETURN operation, by erasing the file using FERASE, by signing off from APL, or by typing)CLEAR. However, files remain tied when another workspace is loaded.

Accessing file functions. The functions described in this chapter are ordinarily stored under the user number APL1 in the workspace FILESYS. Before file operations can be performed, the functions must be obtained from APL1 by loading the entire FILESYS workspace or by copying selected functions from FILESYS. All functions in FILESYS are independent, and you need copy only those functions you intend to use. The following examples show various ways that copies of the file functions can be obtained.

```
)LOAD *APL1 FILESYS
)LOAD '*APL1 FILESYS'
)COPY *APL1 FILESYS .GRPPRIM    (A group that excludes'
                                 documentation)
)COPY *APL1 FILESYS FTIE FREAD
```

The file functions use the system function □FI to perform all file operations. The function □FI could actually be used directly, but it is usually more convenient to use the functions in the FILESYS workspace. Most of the functions in the FILESYS workspace are locked so that error processing will be more convenient. Users who wish to learn how to use □FI directly can discover all details about □FI by studying the definitions of the locked FILESYS functions below:

```
∇A FCREATE B [1] A □FI 1,B∇

∇A FWRITE B [1] A □FI 2,B∇

∇Z←FREAD B [1] Z←□FI 3,B∇

∇FERASE B [1] B □FI 4∇

∇FRDEL B [1] □FI 5,B∇

∇Z←FSTATUS B [1] Z←B □FI 6∇

∇Z←FNAMES [1] Z←□FI 7∇

∇Z←FNUMS [1] Z←□FI 8∇

∇FUNTIE B [1] B □FI 9∇

∇A FTIE B [1] A □FI 10,B∇
```

based on the functions in FILESYS and perform more complicated file operations.

In addition to the basic functions in the FILESYS workspace, the workspace FILES2 contains additional file functions that are based on the functions in FILESYS and perform more complicated file operations.

```
∇FPOS B [1] □FI 11,B∇

∇A CFWRITE B [1] A □FI 12,B∇

∇Z←CFREAD B [1] Z←□FI 13,B∇

∇A CFPOS B [1] □FI 14,B,A∇

∇Z←CSUBMIT B[1] Z←□FI 15,B∇

∇Z←FFREE B [1] Z←□FI 16,B∇
```

Active and stored files. APLUM files are ordinarily KRONOS indirect access files unless the user specifies otherwise at the time of creation. This means that when the file is tied, the system makes a copy of the stored file. All reads and writes actually interact with this active copy. To save the file as a permanent stored file, an FUNTIE is required. Signing off from APL, typing)CLEAR, or a telephone disconnect (assuming the RECOVER command is not used) causes the active file to be erased. One advantage of having a separate active copy of file is that no damage can be done to a stored file if a series of file updates is not completed. For example, suppose that a program writes a record to indicate that a transfer of funds was made from one account to another on a certain date, then the program revises two records containing the balances of those accounts. If the program were to halt in the middle of the sequence of operations (due to a system problem or telephone disconnect), the transactions recorded in the file would be inconsistent with the balances in the file. This causes no problem when indirect access files are used because the inconsistent information is in a temporary file and the stored file is in the same state it was when it was tied.

For some applications that use indirect access files, it may be desirable to perform an FUNTIE and an FTIE at intervals of about every ten minutes in order to minimize the amount of new information that would be lost in the event of a system problem.

Forms for file names and passwords. File passwords and file names must be composed of 1 to 7 of the letters A to Z and digits 0 to 9, must begin with a letter, and must not contain any embedded blanks. File names should be distinct from names used for other files or workspaces. Use of the same name will result in an error message when an attempt is made to untie the newly created file. (For a direct access file, the error occurs when FCREATE attempts to create the new file.)

Range for file numbers. File numbers can be any nonnegative integers not greater than ¯1+2*17.

Range for record numbers. File record numbers can be any nonnegative integers not greater than ¯2+2*17.

FILE SECURITY

A file is owned by the user who created it. The owner is allowed to alter the file in any desired manner, but the owner can control access by other users through the following controls:

1. The file category is ordinarily *private*. Private files cannot be accessed by other users unless their user numbers have been given explicit access permission by use of the KRONOS PERMIT command (see Chapter 12). Alternatively, the file can be assigned a category of *semiprivate* or *public*. Either of these categories allows other users to access the file if they know the password. The ⎕LIB command will reveal to another user the names of files that are *semiprivate*, *public*, or that are *private* and have been explicitly made accessible to the other user. To make a file *public* or *semiprivate*, use the options PU or S when the file is created, or use the KRONOS CHANGE command to change the category. When the ⎕LIB function is used with a file name, the result shows when the file was created, when it was last changed, and when it was last accessed. In addition, for *semiprivate* and *private* files the system retains the number of accesses and the time of the last access for each user of the file. This information can be displayed by use of the KRONOS CATLIST command (see Chapter 12).

2. The file can be given a password. Only users who know the password can use the file; however, the owner of the file is never required to provide the password. The password can be assigned when the file is created, or the password can be assigned or changed by use of the KRONOS CHANGE command (see Chapter 12).

3. The file *mode* can be used to control the type of operation another user can perform. For files created by APL (including workspaces) other users are ordinarily allowed to read the file (assuming the password and *category* do not exclude them) but are not allowed to alter or destroy the file. Other users can be given permission to alter the file by specifying the WR option (for write) when the file is created. For *private* files, this mode has no significance because when other users are given explicit access permission via the KRONOS PERMIT command, the permitted access

mode for each user becomes that expressed in the PERMIT command. For *semi-private* files, the general access *mode* is applicable to most users of the file, but an overriding access *mode* can be specified for individual users by use of the PERMIT command. For example, most users might be allowed to read the file, while a few selected users might be allowed to alter it. The general *mode* allowed for other users can be changed after the file has been created by use of the KRONOS CHANGE command. For APLUM files the mode should be write or read-modify, while for coded files it should be write or read.

4. Files can be accessed by other users through locked functions which can provide extremely general control over the permitted operations. For example, the locked function can prohibit alteration of the first five records of the file, or, it can prohibit adding records that are not vectors of 4 integers. The success of locked functions as a security measure rests on preventing the user from learning the file name, the user number, or the password, and preventing him from accessing the file directly. To assure this, the locked function should not call other functions (except those local to itself) lest someone substitute a subversive function having the same name. In particular, ⎕FI should be used directly rather than using FTIE. (A subversive FTIE could print its arguments and thus reveal the file password). Also, ⎕ input should not be used while the file is tied, and the file should be untied prior to exit from the function. To ensure that the file will be untied, use ⎕TRAP to specify a trap line that will release the file prior to exit.

Note that the file *category*, *password*, and *mode* are independent restrictions on access by other users. Each of these further restricts the type of access permitted to others. Unless different options are specified when the file is created or the controls are changed, the APLUM system selects *private* as the file *category*, assigns no *password*, and selects read or read-modify *mode* (depending on whether the file is coded or APLUM type, respectively) so that other users may only read the file.

APLUM FILE OPERATIONS

Sequential file operations. The file operations that ordinarily require a record number can also be used without specifying the record number. When this is done the record number used is the current file position (available in the result of FSTATUS). The file position can be reset using FPOS and is incremented by each

successful read, write, or deletion. When a file is tied or created, the position is initially zero. For example:

```
'XRAY' FTIE 5      (The file position is zero.)
Z←FREAD 5          (Record 0 is read; the position becomes 1.)
K FWRITE 5         (Record 1 is read)
Y FWRITE 5         (Record 2 is written)
W FWRITE 5         (Record 3 is written)
```

When a record number is provided for the operation, the position will be set to one greater than that number if the operation succeeds.

File create: 'file-name [:passwd] [/options]' FCREATE fnum The file create function can be used to create a file and specify options about the type of file. When the file is created, it is tied to the file number fnum. In addition to the name of the file, the left argument may include the password the file is to have. Examples of file creation follow:

```
'FILE1' FCREATE 11      (A file named FILE1 with 11
                         as its number.)

'FILE2: SESAME' FCREATE 2      (A file with SESAME as its
                                password.)
```

The list of options can include any of the following separated by spaces: DA, C, WR, S, or PU (to specify direct access, coded, write mode, semiprivate, or public). The option RM is appropriate for APLUM files, while RD is the corresponding option for coded files. These are discussed in other sections of this chapter.

File read: result←FREAD fnum[,rnum] The FREAD function reads from the file having fnum as its file number that record having rnum as its record number. If that record does not exist, an empty numeric vector is returned, and the file status (see FSTATUS) will indicate that the last read attempt encountered a nonexistent record.

File write: array FWRITE fnum[,rnum] The FWRITE function writes its left argument on the file having fnum as its number the record having rnum as its record number. This will replace any existing record in that file number, previously having that record number.

File record delete: FRDEL fnum[,rnum] The FRDEL function deletes the record rnum from file fnum. If the record was absent already, nothing is done (except that the file position changes) and no error results.

Free record number: rnum←FFREE fnum The FFREE function returns the first free (unused) record number for file fnum. This is a useful way to select the record number for a new record when the application does not require a particular ordering of the records.

File positioning: FPOS fnum,rnum The function FPOS sets the position of the file identified by fnum to record number rnum.

File status: result←FSTATUS fnums The file status function returns various information about the condition of files identified by file numbers in the right argument. If the argument is a vector, the result is a matrix having a row for each file number in the right argument. If the argument is a scalar, the result is a vector of information about the single file. The columns of the result contain:

Column	Contents
1	largest record number currently in use or $^-1$ if the file is empty.
2	Current file position.
3	File size in words.
4	Unused space in words.
5	Lost space in words.
6	Space not used because of record sizes not being divisible by 64. (This space is called "tails" because it resides at the tail ends of physical record units.)
7	0 if APLUM type file, 1 if coded type file.
8	0 if indirect access file type, 1 if direct access file type.
9	0 if last read attempt succeeded, 1 if the record was absent.

Note that only columns 7 and 8 are meaningful for coded files. All columns will be zero if the file is not tied.

The largest record number does not take account of records that have been deleted. That is, the largest record number is the largest number currently in use for records that actually exist. The largest record number is convenient to know when adding a new record to the file. Adding 1 to the largest record number gives a safe record number to use to append a new record.

Print status: PSTATUS

The PSTATUS function prints the information returned by FSTATUS. The FNUMS along with the file names. The information is given in a descriptive format and is thus a convenient way to discover the status of all tied files if you do not remember the meanings of the columns in the result. The following example illustrates the format used.

PSTATUS

NAME	NUMBER	LAST R	POS	SIZE	UNUSED	LOST	TAILS
COMTIME	14	8	0	768	64	0	397
LIB	2	1	14	256	0	98	DA
*A123456 SYSGEN	45	CODED FILE		80384	2496	7744	3233 DA
REFMAN7	1	94	0 FILE				

File names: result←FNAMES

The FNAMES function returns a matrix of names (and user numbers) of files currently tied. The number of columns in the matrix is always 16. For example,

```
      FNAMES
SAMPLE1
ALGEBRA
*A123456 FILE1
```

File numbers: result←FNUMS

The FNUMS function returns a vector of numbers in use for tied files. The order is the same as the order of file names in the result from FNAMES.

File untie: FUNTIE fnums

The FUNTIE function unties all files for which their file numbers appear in the vector or scalar right argument. This produces a permanent stored copy of each file. The new permanent copy will replace any previously existing file having the same name, unless the active file was newly created. To untie a newly created file when the same name is already in use for another stored file, first use DDROP to remove the old file. If any of the files specified in the argument is not tied, nothing is done and an error message results. To untie all tied files, use FUNTIE FNUMS.

File return: FRETURN fnums

The FRETURN function behaves as FUNTIE for direct access files and behaves as FERASE for indirect access files. This frees the number of a currently tied file for other uses with a minimal impact on stored files. The use of this function is recommended for cleaning up any files that may have been accidentally left tied. File numbers in the argument that are not in use for tied files are ignored.

File erase: FERASE fnums

The FERASE function erases the active copy of the file but leaves any stored copy of the file. (See the section on direct access files for exceptions.) To remove a stored copy, use DDROP.

File tie: 'file-name [:passwd] [/options]' FTIE fnum

The FTIE function gives the number fnum to the previously stored file having the indicated name. If no previously stored file having that name is found, an error message is given and no file having that name is found. If a user number is given, the stored file is sought under that user number rather than the one used when signing on to the system. The password need be given only if a another user number was provided and a password was given to the file when it was created using FCREATE. Examples using FTIE follow:

```
'FILE5' FTIE 7          (A user ties one of his own files.)
'*A0Q1234 FILE6'FTIE 8  (A user ties a file belonging
                         to another user.)

'*A123456 FILE7 :SESAME' FTIE 9
```

Note that the options DA and C (for direct access or coded files) must not be provided to the FTIE function. These options are chosen when the file is created and can be altered only by making a copy of the file. If the file number or file name is in use for another tied file, an error message results. The list of options can include either of the options RD or RM. These options are discussed in later sections.

SPECIAL CONSIDERATIONS FOR CODED FILES

Coded files are the standard type of file on the KRONOS system for information interchange between programs, card readers, printers, and so forth. APLUM can access any coded files provided they contain lines no longer than 1280 characters. Coded files are essentially intended for sequential access; replacement of records, except at the end, is not practical. Instead, such changes would ordinarily be made by copying the file and making the changes as the new file is produced.

Coded files consist of lines (essentially vectors of characters) which can be separated into groups by end of record marks. These groups can in turn be separated by end of file marks. At the end of the file is an end of information mark. The characters in a line of a coded file are restricted to 63 characters. The 256 APLUM characters are translated into these 63 characters as shown in Appendix C. Briefly, the letters A to Z become A to Z, all symbols with approximate equivalents for an ASCII printer are translated into those equivalents, and all others become @. When translating from the KRONOS character set

Characters, all symbols are represented by equivalents,
◼ is represented as ◼ (the symbol used for illegal
overstrikes).

The functions $FTIE$, $FUNTIE$, and $FRETURN$ have essentially the same meanings for coded files as for APLUM files. However, special functions must be used for reading, writing, and repositioning coded files.

Creating a coded file.

A coded file can be created using $FCREATE$ by including C as an additional parameter. For example,

`'PRINT:XXX/C'FCREATE 9`

Coded read: $result \leftarrow CFREAD\ fnum$

The result returned by $CFREAD$ is a character vector containing the next line from the file, or if an end of record, end of file, or end of information was encountered, the result is the scalar integer 1, 2, or 3, respectively. The file position changes after each read so that the next read will give the next line of the file. The $FREAD$ function cannot be used in place of $CFREAD$ with a coded file.

Coded write: $array\ CFWRITE\ fnum$

The left argument to $CFWRITE$ is written at the current position of the file. The left argument must be a character vector, a scalar or vector containing the integers 1, 2, or 3. A character scalar or vector produces one line in the file, whereas a matrix produces one line for each row of the matrix. Trailing blanks in a line are removed. The integers 1 or 2 produce an end of record mark or end of file mark, respectively. A vector of integers can be used to produce a series of these marks. The file position is altered after each write so that subsequent writes will add information after that produced by the present one. Anything written to the file is automatically followed by an end of information mark. This has the effect of truncating the file if the write was not performed at the end of the file. The function $FWRITE$ cannot be used for a coded file in place of $CFWRITE$.

Coded file positioning: $integers\ CFPOS\ fnum$

The function $CFPOS$ repositions the file according to integers in the scalar or vector left argument. The first element in the left argument indicates the action to be taken, and the optional second element may contain a repetition count.

Operation	Value
Rewind	0
Skip record	1
Skip file	2
Skip to end	3

The rewind operation positions the file at its beginning. The rewind and skip-to-end do not allow use of a repetition count. For the skip record or skip file operations, the repetition count may be negative to skip towards the beginning of the file. If no repetition count is given, a count of 1 is assumed. The skip record operation counts end of file marks as records. The skipping never goes past the end of information mark or the beginning of the file, even if the repetition count has not been satisfied.

Batch job submission: $z \leftarrow CSUBMIT\ fnum,type$

The coded file $fnum$ is submitted as a batch job. The $type$ may be 0 if the batch output produced by the job should be discarded, or 1 if it should be printed or punched at the central batch site. The file must be a properly constructed job file (see KRONOS 2.1 Reference Manual). In particular, the first two lines must be a job card and account card. The file must not be direct access type. If the operation is successful, the active file vanishes as if $FERASE$ had been used. The result returned is the job name assigned to the job. This name can be used with KRONOS ENQUIRE command (see Chapter 12) to determine whether the job has completed. Note that the number of concurrently executing deferred batch jobs allowed for a given user number is controlled by the system. The number can be determined by use of the LIMITS control card (see Chapter 12).

SPECIAL CONSIDERATIONS FOR DIRECT ACCESS FILES

A direct access file differs from an indirect access file in that all operations interact with the permanent file itself, not with an active copy. This has both advantages and disadvantages. One advantage is that a copy of the entire file need not be made by the system when the file is tied. One disadvantage is that a program can stop executing due to a system problem in the middle of a series of file writes, and the stored file can end up with contradictory information. Another disadvantage of direct access files is that write operations take a little longer (because the APLUM system does less buffering of information due to the risk of a system problem freezing the file in a temporary state).

To create a direct access file, include the parameter DA in the left argument to $FCREATE$. A direct access file may also be a coded file if desired--these two options can be chosen independently. The following are examples of direct access file creation:

`'FILEX/DA' FCREATE 4`
`'FILEY: XYZ/DA S WR' FCREATE 5`
`'FILEZ/C DA' FCREATE 6`

All operations with direct access files take the same form as for indirect access files, but because of the differences

between the two file types the file tie, untie, and file erase
operations behave differently: A file tie to a direct access
file does not make a copy of the file. An untie does not create
the permanent copy, it merely releases the file itself for use
with other files and releases the file itself for access by other
users. An erase removes both the active and stored copy of the
file because they are the same thing. In addition,)FUNTIE or a
telephone disconnect cause an automatic)FUNTIE of a direct access
file (thus leaving a stored file) whereas an indirect access file
would be erased.

If an telephone disconnect occurs, the file remains tied for
10 minutes. The operations that were in progress can be
continued by use of the KRONOS RECOVER command (Chapter 12).
However, an attempt to sign on without using the RECOVER command
will leave the file tied until the 10 minute period is over,
possibly causing an error message indicating the file is busy.

SYNCHRONIZED FILE OPERATIONS

At present, it is not very practical for two users to update
a single file at the same time. With an indirect access file the
two users are actually updating separate copies of the same file,
and whichever user unties the file last will create a stored file
with his updates, but will replace any stored file just produced
by the other user. The KRONOS operating system does not allow
two users to be tied to the same direct access file in write mode
at the same time, so no conflicts can occur, but an error occurs
if a second user attempts to tie the file. However, users can
tie a direct access file in read mode (which allows other users
to read the file at the same time) or read-modify mode (which
means the user desires only to read the file but has no objection
to another user writing to the file at the same time). To tie a
file in read mode or in read-modify mode, include RD or RM (but
not both) in the left argument to the FTIE function. For
example,

'FILE1/RD'FTIE 9   (Read mode.)

'FILE2: SECURE/RM' FTIE 10   (Read-modify mode.)

These modes have meaning only for direct access files and must
not be used unless the file is direct access. Read mode can be
used for APLUM or coded files while read-modify mode is allowed
only for APLUM files.

FILE EFFICIENCY

Although many users need not concern themselves with the
information presented here on file efficiency, users of very
large files will find this information important. Use of a few

fairly simple techniques can result in improved speed and reduced
storage requirements.

First of all, each APLUM file has an initial size of 64
words, used for a table of available space. In addition, one word
is required for each record number up to the last record number
in use. This space is allocated in multiples of 64 words. These
two factors combine to make it inefficient to store many files
with only a few records in each rather than one file with many
records. Also, it is inefficient to leave large gaps between
record numbers as the unused numbers require an average of one
word each.

Indirect access files grow in multiples of 64 words, but
direct access files grow in multiples of a logical track (usually
several thousand words, depending on the storage device used).
There is consequently a considerable space advantage to using
indirect access files for files smaller than several thousand
words.

File records require 2 words more than they require in the
workspace, and the space for records is allocated in multiples of
64 words. However, large records of character or logical type
require only about half as much space in the file. Because
records require multiples of 64 words, there is some saving in
space if many little arrays can be packed together and written as
a single record.

When records are erased or replaced by records of a
different size, the APL system keeps track of any unused gaps in
the file where records can be placed in the future. The total
amount of this space in words is in column 4 of the result
returned by the FSTATUS function. It may happen that the number
of gaps exceeds the size of the table, in which case the number
of gaps is removed from the table. This results in a certain amount
of space becoming unusable, and the total amount of this lost
space is in column 5 of the result returned by the status
function. Lost space can also result in a direct access file if
a telephone disconnect or system problem prevents the file from
being untied (⎕TM'SYSTEM', ⎕TM'ABORT', and ⎕TM'OFF' untie files
properly), and if the RECOVER command is not or can not be used.
All lost and unused space can be recovered by copying all records
to a new file, last record first, and using the new file as a
replacement for the old one. Because each record occupies a
multiple of 64 words, some space is generally left unused. This
space is returned in column 6 of the result from FSTATUS.

Details of the space required for coded files can be found
in the KRONOS 2.1 Reference Manual. Coded files have a speed
advantage over APLUM files when the information is accessed
sequentially, the records are small, and the limitations of the
63 character set are not restrictive.

INTEGRITY OF DIRECT ACCESS APLUM FILES

File integrity refers to the ability of a file to retain internal consistency. Some file access methods render a file practically useless if a program operating on the file does not complete properly (due to a flaw in the program or a system problem). Every effort has been made in the design of the APLUM file system to minimize the chance of such damage.

All alterations to an APLUM file are performed immediately and thus occur in exactly the order requested. When multiple files are being updated, one file will not be several transactions ahead of another. A checksum is computed for each file record so that if the storage device corrupts the information and is unable to detect the error, the error will still be detected by the APLUM system. A system halt, program halt, or telephone disconnect will leave the file in a satisfactory state except that in the rare event of a system halt requiring a level zero deadstart within a minute of extending a direct access file, there is some chance of damage to newly created or replaced records.

File damage will cause an error message to be printed at the time it is detected. The damage will usually affect only one record of the file. If the file cannot be reconstructed, installation personnel can assist with restoring the file to its state the last time files were dumped to magnetic tape.

Note that a telephone disconnect or system problem that results in failure to untie the file may cause the information on file space utilization (unused space, lost space, and tails) to be incorrect. This does not hinder utilization of the file and can be corrected by copying the file.

FILE EXAMPLES

The following sample functions taken from the workspace FILES2 under user number APL1 illustrate simple file operations. The first function, FCOPY, can be used to copy an APLUM file. Such a copy might be made to convert the file from indirect access to direct access form or to compact the file by minimizing unused space. The left argument should be the character argument required to tie the old file, and the right argument should be the character argument required to create the new file. Note that the first line illustrates a simple way to select a file number that is not already in use.

```
∇FCOPY[□]∇
∇A FCOPY B;P;K;I;J
[1]    A FTIE I←1+⌈/0,FNUMS
[2]    B FCREATE J←I+1
[3]    K←(FSTATUS I)[1] A GET LARGEST RECORD NUMBER
[4]  L1:→(K<0)/L3
[5]    P←FREAD I,K A READ RECORD K FROM FILE I
[6]    →(FSTATUS I)[9]/L2 A IF ABSENT RECORD
[7]    P FWRITE J,K A WRITE RECORD K TO FILE J
[8]  L2:K←K-1
[9]    →L1
[10] L3:FUNTIE I,J A UNTIE BOTH FILES
[11]   'COPY COMPLETE'
∇
```

The next function is useful for listing a coded file. The right argument may be the name of a stored file or the number of an active file. If a name is given, the file is tied, listed, then untied. If a number is provided, the file is listed beginning at its current position and is left tied.

```
∇CLIST[□]∇
∇CLIST B;K;L
[1]    →(0=0\0ρK←B)/L1 A IF FILE ALREADY TIED
[2]    B FTIE K←1+⌈/0,FNUMS
[3]  L1:L←CFREAD K
[4]    →(0=0ρρL)/L2 A SCALAR INDICATES SPECIAL MARK
[5]    L
[6]    →L1
[7]  L2:→L3+2×L-1
[8]  L3:'-END OF RECORD-'
[9]    →L1
[10]   '-END OF FILE-'
[11]   →L1
[12]   '-END OF INFORMATION-'
[13]   FUNTIE(0≠0\ρB)/K
∇
```

The next two functions are useful when a file is too large to list at a terminal but it is necessary to learn the general structure of the file. The function FMAP prints the structure of an APLUM file, and the function CMAP prints the structure of a coded file. Both functions allow a character argument or a numeric argument in the same manner as CLIST. If the file is already tied (for numeric arguments) the mapping begins at the current file position. FMAP prints record numbers and the types (C or N for character or numeric) and shapes of records that exist, or ABSENT for absent records. CMAP prints the number of lines in records and prints EOR, EOF, or EOI when an end of record, end of file, or end of information is encountered.

```
∇FMAP[]∇
∇FMAP B;K;P
[1]   →(0=0\0ρK←B)/L1  ⍝ IF B IS NUMERIC
[2]   B FTIE K←1+⌈/0,FNUMS
[3]   L1:→(¯1=1↑FSTATUS K)/L2  ⍝ IF FILE NOT EMPTY
[4]   'NO RECORDS'
[5]   →0
[6]   L2:'NUMBER, TYPE, DIMENSIONS'
[7]   L3:→(</2↑FSTATUS K)/L5  ⍝ IF FINISHED
[8]   P←FREAD K
[9]   →(FSTATUS K)[9]/L4  ⍝ IF READ FAILED
[10]  ¯1+(FSTATUS K)[2];0 1 0/'CN'[1+0=0\0ρP];ρP
[11]  →L3
[12]  L4:¯1+(FSTATUS K)[2];' ABSENT'
[13]  →L3
[14]  L5:FUNTIE(0≠0\0ρB)/K
∇

∇CMAP[]∇
∇CMAP B;K;P;C
[1]   →(0=0\0ρK←B)/L1  ⍝ IF B IS NUMERIC
[2]   B FTIE K←1+⌈/0,FNUMS
[3]   L1:C←0
[4]   L2:→(0=ρP←CFREAD L)/L3
[5]   C←C+1
[6]   →L2
[7]   L3:→(C=0)/L4
[8]   C;' LINE',(C≠1)/'S'
[9]   L4:'EO','RFI'[P]
[10]  →(P>3)/L1
[11]  FUNTIE(0≠0\0ρB)/K
∇
```

Chapter 10. APL Public Libraries

The standard APLUM release includes the following workspaces stored under the user number APL1:

APLNEWS   News about the changes in the APLUM system as well as a list of reported bugs and requests for system changes.

WSFNS   Workspace utility functions. Includes functions SHORT, LONG, and SHORTSAVE for saving a workspace without ⎕FD, ⎕SY, ⌽, ⊖, ↓, and ⍉.

FILESYS   File system functions.

FILES2   Contains functions from FILESYS for primitive file operations as well as additional functions for more elaborate file operations.

PLOTFNS   Function to produce an X-Y plot of multiple sets of data on a standard terminal.

CATALOG   A guide to workspaces in the APL public libraries.

STP1, STP2, STP3, STP4, STP5, STP6
           A collection of mathematical and statistical programs developed by K. W. Smillie of the University of Alberta. Primary documentation is in STP1. Capabilities include: descriptive statistics, regression and correlation analysis, analysis of variance, linear programming, and critical path.

To learn how to use any of these workspaces, type a command of the form ⎕LOAD'*APL1 CATALOG' and then type DESCRIBE.

APL PROGRAM LIBRARY STANDARDS

It is suggested that installations reserve the user numbers *APL1* to *APL999* for APL public libraries. Although these user numbers need not be defined in the system, they should not be used for other purposes. It is suggested that programs placed in these public libraries be of fairly general interest so that users will find it rewarding to browse through the various workspaces. Workspaces of interest only to a specialized group or course should be stored elsewhere.

Programs placed in the public libraries should be well documented. The available documentation may be entirely in the workspace or partly in the workspace and partly in a manual. In any case, the documentation should be readily available. The advantage of having the documentation in the workspace is that it will be immediately accessible. The disadvantages are that the documentation is slow to print and therefore tedious to read, and the format of the documentation is constrained by the APL character set. Generally, the amount of documentation determines whether it is practical to put the documentation in the workspace.

Documentation in the workspace should consist of functions or variables that describe the workspace. The documentation should be able to be printed with a standard APL terminal and should print within a standard 65 column page width. The following documentation variables or functions are suggested. Typing the name of the function or variable should cause the information to be printed.

ABSTRACT. Should contain a brief description of the contents of the workspace.

DESCRIBE. This should give the user further details than that provided in the *ABSTRACT*. This should print the names of all functions intended for the user to use as modules along with a short description and names of related *HOW* functions (see below). If groups are defined in the workspace, describe them and their purposes.

HOW functions. If a function has the name *NAME*, detailed documentation of that function should have the name *NAMEHOW*. There is no point in giving a line-by-line description of the function. The APL program is already an excellent description of the separate steps. The *HOW* function should tell what the function does and how to use it as a module. In some cases it should outline major steps in the processing and describe the method used. References might be appropriate. Special limitations of the function should be discussed.

SOURCE. Should give the author's name, an inquiry name, and an inquiry address. The date when the workspace was contributed should be included.

CHANGES. Changes should be documented by a function or variable having a name of the form *CHANGES092675* (so that the name includes the date of the changes).

GRPDOC. The group (locked matrix of names) *GRPDOC* should include names of all documentation variables and functions so that the user can readily erase them to make more space available in the workspace or reduce disk storage charges.

Even when most of the documentation is in a separate manual, the following variables or functions are required: *ABSTRACT*, *SOURCE*, *GRPDOC*, and *DESCRIBE*.

Chapter 11. Optimization of APL Programs

This chapter discusses some of the techniques that can be used to make APL programs perform better and run with lower demands on computer resources. It may seem out of place to discuss efficiency in an APL manual--after all, APL should free the user from being concerned with the nature of the particular computer being used--but the techniques discussed here may yield efficiency improvements as large as a factor of a hundred. To neglect discussing efficiency could leave many users with the mistaken impression that APL cannot perform well enough to be used for their problems.

Often, the question of efficiency calls to mind the fanatical programmer who constructs a program he considers efficient but who in doing so produces a totally incomprehensible collection of operations. It should be remembered that for many programs the programming time is so great that the only kind of efficiency worth considering is the sort that makes the program easy to understand, free of errors, and easy to change. Fortunately, a simple program is usually an efficient program. However, when improving the performance of the program does not coincide with simplifying it, the optimization should not be applied unless it is very important for the program to perform well.

As a very blatent example of misguided optimization, consider the following statement:

$$K \leftarrow 1, 0 \rho P \leftarrow 0, \lceil / L \leftarrow \iota Q \leftarrow \rho R$$

This statement was probably contrived by someone who believed that the most efficient program was the one that required the smallest number of lines. The fact is, execution proceeds from one line to the next very rapidly compared to the time required to perform the extra steps needed to fit the operations in one line. The following statements are a more straightforward way to achieve the same results:

```
K←1
L←1Q←ρR
P←0,Q
```

One way to estimate the relative time required for an expression is to count the number of operations required. (This method is fairly valid when the number of elements in arrays is less than about 20.) For this method of estimation, specification is not counted at all (it takes relatively little time). The one line version totals 6 operations while the three line version requires only 3 operations. The efficiency expert who wrote the one line version devoted extra time to adding three operations, which double the time required for execution. The one line version is harder to understand, is more likely to contain errors, and when changes are made, the rest of the line hinders revision. The one liner is thus a poor example of efficiency in all respects.

At this point it must be stated that much of the information in this chapter is relevant only to this particular APL system. Also, it may occur that something that is particularly slow now will become particularly fast in later versions of the system. Other versions of APL on other computers will often show quite different characteristics. In fact, according to Paul Berry, the popular belief that one line programs are more efficient is based on a system for which this is true. A version of APL on the IBM 1130 actually requires considerable time to change from one line to the next because only one line at a time is kept in main memory.

Although very few present users of APL ever used that particular system, its influence persists.

STORAGE REQUIREMENTS

Although the APLUM system allows a workspace of up to about 90,000 words (provided the user is validated to use that much main memory and the installation has that much), equivalent to 675,000 8-bit bytes, there are practical reasons to keep a workspace much smaller. The KRONOS operating system uses computer resources much more effectively when it runs programs requiring minimal amounts of central memory. Also, the "response time" for an interactive program to respond to a command requiring a trivial amount of processing increases somewhat with central memory requirements. In addition, minimizing storage requirements improves the chances that the same program will be able to run under another version of APL or on a computer with less central memory.

The vector ⎕WA contains information about the memory currently in use for the APLUM interpreter and the active workspace. The field length is the amount of memory space currently in use. The APLUM system manages that memory space and at any given time some of the space may not be in use for

functions, variables, and other information kept by the APLUM system. The APLUM system evaluates storage requirements from time to time and resets its actual field length according to current needs. The user can set ⎕WA to specify the maximum and minimum field lengths to be used. Increasing the maximum field length generally reduces the central processor time used by APLUM to reorganize its storage, but as discussed previously, reduces the operating system efficiency. As a general rule of thumb, leave the minimum field length at its normal value, and set the maximum field length large enough to avoid WS FULL plus a little extra to prevent frequent storage reorganization. Incidentally, referencing the value of ⎕WA in a statement causes the APLUM system to reorganize its storage, so programs should not alter or read the value of ⎕WA too often or performance will be degraded.

Obvious techniques for minimizing storage requirements include using algorithms that minimize temporary storage, using local variables and local functions to assure automatic erasure of unneeded objects, and, using ⎕EX to erase other functions that are no longer needed. ⎕EX can also be used to erase variables, but respecification (e.g., $A←''$) is faster. Files can be used to store functions and variables until they are required. ⎕LOAD can be used to load another workspace of functions and variables. Any variables that must be communicated from one workspace to the next can be placed in files--files remain tied when another workspace is loaded. Of course, any of these techniques can be overdone. Do not let the time spent performing these operations outweigh the time they save.

The space in words required for an APL array A is

$$2+(\rho\rho A)+\lceil(\times/\rho A)\div D$$

where D is the number of elements packed per word--1 for floating point values, 4 for characters, and 32 for logical. Clearly, there is an advantage to using the internal logical representation if the values are ones and zeros. The system does not always use the logical representation when it could. For example, the scalar constants 1 and 0 are floating point, and $1+0$ is floating point. However, the following functions always produce a logical result: $A∧B$, $A∨B$, $A⍲B$, $A⍱B$, $A=B$, $A≠B$, $A<B$, $A≤B$, $A≥B$, and $A∈B$. Also, the functions that restructure or rearrange their arguments always preserve the same type of representation, so $N\rho 0$ is floating point, while $N\rho 1\ 0$ is logical (because vector constants consisting of ones and zeros are packed as logicals). To assure that a result is logical, apply 1= to it.

Expressions like $A←B←C←ι100$ do not cause three copies of ι100 to be produced. Actually, only one copy is kept. However, subsequently altering an element of A, B, or C, (e.g., $A[3]←9$) will cause a separate copy to be made. Similarly, arguments to functions are not actually copied unless an attempt is made to

alter them using indexed specification. Unlike most other APL systems, using function arguments rather than global variables incurs no storage penalty.

Storage requirements for programs are too complicated to discuss in detail. As a rule of thumb, unless you make a special effort to put a lot on each line, figure that an average statement takes about 10 words of storage. The first time a statement is executed it is converted to an internal form for more efficient execution. In the internal form the function almost always requires somewhat more space. The storage overhead per line of a function averages about 3.5 words for lines without labels and 4.5 words for lines with labels.

In version 2.12 of APLUM, function definition mode and system commands are performed by functions written in APL. When a user types)expression or ∇expression the system translates these into □SY')expression' and □FD'∇expression', respectively. The functions □SY and □FD are stored in the so-called "clear" workspace. In addition, some of the mixed functions are written in APL and take space in the "clear" workspace. It may be desirable to erase these if they are not required. For example, anyone who has mastered the equivalent system functions can eliminate □SY from his workspaces. Similarly, after a workspace has been developed, □FD can often be erased before saving the workspace. To minimize disk storage use of the programs in the workspace, □FD and □SY can be erased before saving the workspace.

In the workspace WSFNS in library APL1 there is a function named SHORT that can be used to erase all of these functions, and there is a function named LONG that will copy in all of the functions. Future system improvements will eliminate all of these functions from the clear workspace.

The APL system keeps a "symbol table" in the workspace containing all names of functions, variables, and labels. Once a name has been used (even if the use resulted in a VALUE ERROR) the name remains in the symbol table. The space used by names that are no longer needed can be recovered by copying all objects into a truly clear workspace. The recommended procedure is:

```
□LOAD '*APL0 EMPTY'                (A clear workspace.)
□ENV←0                             (Copy global objects.)
(1 2 3 4 □NAMES 'OLDWS')□COPY 'OLDWS'
□SAVE 'NEWWS'
```

This procedure will also recover space in workspace areas other than the symbol table in some circumstances.

Space can be conserved in the symbol table by using names consisting of a single symbol whenever possible. Space can also be conserved by using the same name in several functions for local variables or labels. A common convention is to use the letters A to Z for local variable names and use L1, L2, and so forth for labels.

CENTRAL PROCESSOR TIME

For many programs the main optimization problem is to minimize central processor time. First of all, one of the primary determinants of central processor time is the appropriatness of the algorithm used. The algorithm should be appropriate to the data to be processed and appropriate to APL. Computer literature is filled with algorithms that are "efficient" for other languages but which perform miserably in APL. Often a straightforward translation of a program from another language gives a program that performs poorly because it fails to take advantage of the more powerful APL functions.

For most operations in APL the time required for the operation can be separated into a per-element time required to process each element of the arguments and a setup time required for interpreter overhead, to check the arguments for compatibility of dimensions, to compute the result dimensions, and allocate space for the result. The time per element varies considerably with the complexity of the operation. The sine function, for example, requires far more time per element than addition. The time also depends on the way the values are stored; operations defined only for logic values perform better if their arguments are internally represented as logical type, and arithmetic operations are far faster for the floating point internal type. The setup time varies far less from function to function than the time per element.

For many functions the setup time is on the order of 25 times as great as the time per element. This means that the setup time is negligible when thousands of elements are to be processed, but the setup time constitutes about 95 percent of the time when only one element is being processed. For most programs, the setup time limits speed more than the time per element. Thus the first step to optimization is to minimize the number of operations to be performed. For example, if ρX is used many times in a function, it would be worthwhile to assign the value of ρX to a variable (assignment requires negligible time). Often a branch statement can be added to skip steps that are not required except in special cases.

When the arrays used have a large number of elements, the operations should be chosen to minimize the number of elements processed. For example, if V is a vector of 5000 characters, a few elements can be selected from V using $N \uparrow M \downarrow V$ (which might process about 5000 elements) or using $V[J + \iota K]$ (which would process only a few elements). The second approach is much more efficient. Similarly, rather than extending a vector by catenating one element at a time, it might be preferable to extend it with a large number of elements and then respecify the elements one at a time using indexed specification.

It is commonly believed that APL branching for looping is slow. Actually, looping is fairly fast by itself but is usually a sign that the program is performing operations one element at a time--the amount of time required is mainly due to the number of operations being performed. Actually, looping is sometimes a very efficient way to perform an operation, especially if the number of iterations required for normal cases is small and the alternative requires more operations than are used in the loop.

The evaluation of constants other than 1, 2, ¯1, 0, .5, and ¯1 takes about half as much time as the setup time for functions. Some time can be saved by using variables to store the values of commonly used constants other than the constants just mentioned.

On some APL systems central processor time can be saved by catenating output together and then printing it in a batch rather than as it is generated. However, on the APLUM system it is more efficient to print the output as it is produced.

The following chart gives approximate timings for various operations. Be forewarned that these timings are approximate and will vary with the version of APLUM in use, the particular computer used, and the internal workspace configuration. Times are expressed in terms of T, the time per element for addition.

Time range	Operations
0 to T	Time per element for $A \wedge B$, $A \vee B$, and $\sim B$ for logical internal representation
	Setup time per statement to be evaluated
	Time per element for most scalar and mixed functions
T to $5 \times T$	Time per element for complicated functions such as $A \circ B$, $A \otimes B$, $A \circledast B$, and $A[B]$
	Time required for an unnecessary set of parentheses in a statement
	Time required to evaluate a constant other than 0, 1, 2, .5, ¯1, and ''
$5 \times T$ to $25 \times T$	Extra time per local variable for a function call
	→B
	A←B
$25 \times T$ to $125 \times T$	Call to a user defined function with a few local variables
	Setup time for primitive functions

Chapter 12. KRONOS Features for APLUM Users

This chapter discusses a few KRONOS commands of interest to users of APLUM. The discussions cover only the more important details. Further information can be found in the KRONOS 2.1 Reference Manual or in the KRONOS Time-sharing User's Manual. Most of the commands discussed can be used as timesharing commands or batch job control cards. However, they cannot be used while in APL. Use the commands before issuance of the APLUM command, or use ⎕TM'SYSTEM' to leave APL to use these commands. Note that none of these commands allow embedded spaces.

HELLO

The HELLO command allows you to sign on again with a different account number.

BYE

The command BYE is the correct way to sign off the system when not in APL. This is equivalent to the APL command ⎕TM'OFF' or)OFF.

RECOVER,number

The RECOVER command can be used to return to the state just before a disconnect or system malfunction occurred. The use of this command prevents loss of the active workspace or active files. The command is allowed only when the system prints RECOVER/SYSTEM at the end of the sign-on procedure. If you have already proceeded beyond that point and wish to initiate recovery, type HELLO to begin the sign-on procedure anew. The number you provide in the RECOVER command should be the terminal number that was printed after the previous sign on. (That is, the terminal number in effect for the session that terminated

abnormally.) After you type the RECOVER command, the system may print RECOVERY IMPOSSIBLE, which indicates that the system malfunction was too serious to allow recovery, that too much time has elapsed (recovery information is retained for ten minutes), or you signed on with a different user number, or that you gave an incorrect terminal number. When the RECOVER command is successful, the recovery information is destroyed and the system prints various information about the status at the time of disruption. Press the RETURN key to continue (or type STOP to exit from APL). The recovery is sometimes imperfect. Some output may be lost, and the next input request may cause a question mark to be printed, and any special APL symbols used in the input may be translated incorrectly. Do not perform the recovery on a different type of terminal from that in use when the disruption occurred or the APL system will translate input and output incorrectly for that terminal.

LIMITS

The LIMITS command causes validation limits for the account number currently in use to be printed. Any numbers in the output that are followed by a B are expressed in octal (base 8). The APL functions base-value and represent can be used to convert between octal and decimal. For example, 70000B can be converted to decimal using $8\perp7\ 0\ 0\ 0\ 0$, and 32768 can be converted to octal using $(6\rho8)\top32768$. The following are the limits that are important to APLUM users:

TL = CPU time limit in 10's (octal) per session. Append a zero to the right of the number to find the CPU time limit in octal seconds. In addition, there may be a smaller time limit per session. This other time limit per session can be overridden by using the SETTL command or by using the T,number command after a *TIME LIMIT* error occurs. If you have consumed your entire CPU time limit for the session, you can use the KRONOS HELLO command to get a new CPU time allotment, or hang up, sign on again, and use the KRONOS RECOVER command.

CM = Maximum central memory field length. Append two zeros to the right of the number to find the central memory limit in octal words. Note that a more stringent restriction can be imposed on all timesharing users by the computer operator. This second restriction may vary according to the time of day.

DB = The number of jobs allowed for the given user. The CSUBMIT function (see Chapter 9) is not allowed to submit additional jobs if the total number of jobs for that account number already equals or exceeds this parameter. The count of jobs includes the program attempting to use the CSUBMIT function.

FC = Maximum number of stored indirect access files allowed.

CS = Total storage in PRU's allowed for all stored indirect access files. (One PRU is 64 words or 640 six-bit bytes.)

FS = Maximum size in PRU's allowed for individual stored indirect access files. (One PRU is 64 words or 640 six-bit bytes.)

AW = Access word. If the last digit is 4 or greater, the user is allowed to create direct access files.

The following limits are not in effect for version 2.1.1 of the KRONOS operating system but may be in effect in later versions:

NF = Number of local files allowed. This includes active APLUM files and coded files. Allow one extra file when saving or loading a workspace.

MS = Maximum number of mass storage tracks allowed for files, including active copies of indirect access files. One track may be anywhere from 1024 words to 54,784 words depending on the storage device used. Contact installation personnel for details.

In some cases SYSTEM may be printed as one of the above limits, which indicates that there is no limitation on the individual account number, but there may be a general limitation on all timesharing users. Contact installation personnel to determine this limitation.

SETTL,number

Sets the CPU time limit to number. This can be used before entering APLUM to prevent a *TIME LIMIT* error from occurring. The number should be the desired time limit in octal. In order to be meaningful, the time limit should be at least 10 (octal) and the last digit should be a zero. The time limit must not be set to more than the remaining allowance for the session. (You can use the HELLO command to start a new session and get a fresh allotment of CPU time.)

T,number

This command is meaningful only immediately after the system has printed *TIME LIMIT*. The number has the same significance as for the SETTL command. If you type anything other than T,number, a forced exit from APL will occur and the active workspace will be lost. If you have used up the entire CPU time

allotment for the session, hang up the phone and then sign on again and use the RECOVER command.

DISPOSE

The DISPOSE command is used to send a coded file to a batch printer. Note that the first character on each line in the file should be a printer carriage control symbol. A blank is usually used, but a 1 can be used to skip to the next page, or a 0 can be used to skip a line before printing. Numerous other options exist (see KRONOS 2.1 Reference Manual). The following steps would be used to print a file at the central batch site:

BATCH
GET,filename
DISPOSE(filename=PR/BC=account)

where filename is the name of an indirect access coded file and account is your account number. If the coded file does not include carriage control characters in the first column, the following procedure should be used instead:

BATCH
GET,filename
RETURN,TEMP
COPYSBF,filename,TEMP
DISPOSE(TEMP=PR/BC=account)

CHANGE

The CHANGE command can be used to change the name of a file (which includes workspaces), its password, category, or access modes permitted to other users. The following examples show simple forms of the command.

CHANGE,newname=oldname
Changes the name from oldname to newname.

CHANGE,filename/CT=category
Changes the category. The category specified may be P for private, S for semiprivate, or PU for public.

CHANGE,filename/M=mode
Changes the mode. The mode specified may be R for read, W for write, MODIFY for modify, or RM for read-modify. (Other modes exist but are not of interest for APLUM users.)

CHANGE,filename/PW=password
Sets the file password. The password may consist of 1 to 7 letters or digits.

ENQUIRE,J=jobname

This command can be used to determine the status of a job submitted using the CSUBMIT function (discussed in Chapter 9). If the response indicates the job is not in the system, this usually indicates that it has completed or is presently being printed.

PERMIT

The PERMIT command can be used to give another user access to a private file or to specify the permitted access mode for a particular user of a semiprivate file. The form of the command is:

PERMIT,filename,account=mode,account=mode,...

The mode for each account number determines the type of access allowed. Meaningful modes for APLUM users are R for read, W for write, RM for read-modify, or MODIFY for modify.

CATLIST

The CATLIST command can be used to examine access information about an individual file. The following examples show how to find information not provided by the APL ⎕LIB function:

CATLIST/LO=F,FN=filename
Similar to ⎕LIB'filename' but also gives the password and count of the number of accesses.

CATLIST/LO=FP,FN=filename
Gives access information for each user who accessed the specified private or semiprivate file. The information printed includes the number of accesses by each user, the access mode allowed for each user, and the date and time of the last access by each user.

The following list describes the APL error messages and their meanings. It should be noted that most of these cause execution to halt (unless □TRAP is used to intercept the error processing), but function definition mode prints its error messages and then may continue processing.

00: INTERRUPT
    This indicates that an interrupt has been received from a terminal or that the overstrike ⍉ has been entered as the first nonblank symbol for quote-quad input.

01: IMPLICIT ERROR
    An implicit argument to a primitive function is not defined.
The system variable □CT is required for the functions A=B, A>B, A<B, A≥B, A≤B, A≠B, A∈B, A⍳B, and ⍺B. The variable □IO is required for indexing, the axis operator, A⌽B, A⍺B, ⍺B, ⍝B, and A?B. The variable □WSID must be defined for □SAVE''. □RL is required for A?B and ?B, and □PP is required for monadic format. □ENV is required for □CR, □EX, □FX, □NC, □STOP, □TRACE, □LOCK, □LTIME, □NAMES and □COPY.

02: SYNTAX ERROR
    Incorrectly formed statement. Check to be sure the statement has matched quotes, parentheses, and brackets. A common error is to forget to place an operation symbol between two variables when catenation is intended (e.g., (M N)ρQ instead of (M,N)ρQ). Other causes include failure to provide a right argument to a function, and use of a branch arrow other than at the left end of a statement. Check the state indicator to be sure a local variable or label is not obscuring a function having the same name.

03: DOMAIN ERROR
    The argument is not in the domain of the function or is an improper value for a system variable being specified. The following are examples of ways that domain errors can arise:

13.5 (an integer is required), □IO←14 (the index origin must be 1 or 0), 1,+3 (character arguments are not allowed for many operations, even if the argument is empty), □PP←45 (printing precision must be between 1 and 15). When □CT is not defined, 11+1E¯14 would not be zero is used as □CT in domain checks. zero is allowed because exact integers are required when □CT is zero.

04: *LENGTH ERROR*
The lengths of the arguments to a function are incompatible, or the operation is not defined for arguments of that length.

05: *VALUE ERROR*
A variable used in an expression has not been assigned a value, a dyadic function has been used without a left argument, the result variable of a function that returns a result was not assigned a value, or a function was used for which there is no current definition. Check the state indicator to see if a local variable has obscured a global variable or function.

06: *RANK ERROR*
The ranks of the arguments are incompatible or the operation is not defined for arguments of that rank. For example: 1 1 1 is not defined for an argument of that rank. For vectors unless they have one element (not defined for vectors unless they have one element), A[1;2] (if A is a vector it has the wrong rank for the index applied), ⊞ 3 4 5ρ0 (not defined for ranks greater than 2).

07: *INDEX ERROR*
Index out of range. For example, if A is a three-element vector: A[4] in 1-origin, A[3] in zero origin, or A[0] in 1-origin. To find the current origin, display □IO.

08: *LIMIT ERROR*
The operation exceeds limitations of the computer or the APLUM system. Limit errors can result from: attempts to generate a result greater in magnitude than about 1E322, attempts to execute a line longer than 120 characters (in a function, or entered as input), attempts to execute function, or entered as input), attempts to produce an array having a rank greater than 75, or an attempt to produce a result requiring more storage than the entire central memory of the computer.

09: *LOCKED OBJECT*
Attempt to specify a value for a locked variable (label or group). Locked variables can be redefined only by erasing them and then specifying them.

10: *WS FULL*
Insufficient space remains in the workspace for the operation. Erase unneeded functions and variables to make more space available, or reset □WA to allow a larger workspace. However, do not set □WA to a greater field length than you are validated for or you might actually increase the chance of *WS FULL* (see LIMITS in Chapter 12). If there is insufficient room for executing system commands, try using system functions

instead. If that fails, try using specification to reduce the space required for larger variables (e.g., A←''). Some space can usually be reclaimed by executing a niladic branch (e.g., →''). If more than one suspension is on the state indicator, use a niladic branch for each suspension.

12: *DEFN ERROR*
Incorrect request in function definition mode. May result from providing header information other than the function name when reopening the function, use of a function name already in use for another global function or variable, or an illegal display or line editing request. Another cause is an attempt to close definition of a function having an incorrectly formed header or duplication of names used in the header or as labels.

13: *PHRASE NOT FOUND*
The phrase specified was not found in the line where it was sought. Be sure to specify the correct line number. Display the line to determine the correct phrase.

14: *SI DAMAGE*
Information on the state indicator has been lost due to changing a pendent function, by altering a function that is suspended more than once, or by changing the number or relative order of local variables in the header or label variables for a suspended function. This message is a warning—no corrective action is required. The pendent or suspended functions on the state indicator that are affected by *SI DAMAGE* are indicated by enclosing brackets. The affected functions cannot be continued, but they remain on the state indicator as long as other suspensions are above them. When the state indicator collapses to the affected suspension, the system automatically removes that suspension.

15: *NAME NOT FOUND*
No function or variable having that name exists.

16: *NAME IN USE*
A function or variable already has that name.

18: *MIXED FUNCTION*
A mixed function has been used where a dyadic scalar function is required as an argument to an operator. For example: A+.1B, A∘.1B, φ/B.

19: *UNDEFINED FUNCTION*
No such primitive function exists. For example: αB, ≠B (no monadic ≠ function).

20: *operating system error message*
This message is a message from the KRONOS operating system and usually concerns some sort of operation with a file or with a workspace. See the list of common errors under OPERATING SYSTEM ERROR MESSAGES below.

21: *FILE DAMAGE*
Usually indicates that one record of the file has been damaged. If an attempt to tie the file causes this message, the entire file may have been damaged. Most installations periodically copy all files to tape, and files can be restored to their condition when the last copy was made. Contact installation personnel for assistance. File damage may be reported erroneously when reading a direct access file in *RM* mode if repeated interference is encountered from another user writing the same record.

22: *WRONG TYPE FILE*
An attempt was made to use *CFREAD, CFPOS, CSUBMIT,* or *CFWRITE* on an APLUM file, or an attempt was made to use *FREAD* or *FWRITE* on a coded file. Note that the KRONOS COPY commands do not preserve the type with a copy made from an APLUM file. This error also occurs if an attempt is made to alter a direct access file that was tied in read or read-modify mode.

23: *FILE TIE ERROR*
An attempt was made to use a file number or file name that was already in use, or an attempt was made to perform an operation (e.g., *FREAD, FWRITE*) that requires the file to be tied.

OPERATING SYSTEM ERROR MESSAGES

20: *filename BUSY*
The specified direct access file is tied in an incompatible mode. This may be caused by a system problem or telephone disconnect, in which case the file will be released in 10 minutes or can be accessed by using the KRONOS RECOVER command to resume the session that terminated abnormally. Occasionally a file will be left busy due to an operating system error and will remain busy until a level zero deadstart (usually done at the start of the day). An APLUM direct access file can usually be retrieved from this condition by using *RM* mode to make a new copy of the file.

20: *filename ALREADY PERMANENT*
A file having the indicated name already exists. This error may result if a workspace is being saved and a password, category, mode, file type (i.e., *IA* or *DA*), or a name different from ⎕*WSID* was specified. This error can also occur when *FCREATE* attempts to create a direct access file having the same name as a file already in existence or when *FUNTIE* attempts to store a copy of an indirect access file that was created during the session. If the old file is no longer needed, use ⎕*DROP* to eliminate it; otherwise, copy the new file to change its name.

20: *filename NOT FOUND*
The file does not exist under the specified user number, the user is not allowed to access the file, or the user did not provide a correct password for a file requiring a password.

20: *ILLEGAL USER ACCESS*
The user is either not allowed to create direct access files or is not allowed to create indirect access files.

20: *FILE TOO LONG*
The file is too large for the limits associated with the account number.

20: *PF UTILITY ACTIVE*
The computer operations staff is using a permanent file utility program that prevents users from performing operations involving permanent files. Try the operation again.

20: *CATALOG OVERFLOW - SIZE*
The operation would cause the user's limit on total size of all indirect access files to be exceeded.

20: *CATALOG OVERFLOW - FILES*
The operation would exceed the limit on the number of files allowed for the account number.

20: *PARITY ERROR*
20: *ADDRESS ERROR*
20: *DEVICE STATUS ERR.*
20: *6681 FUNCTION REJ.*
20: *DEVICE RESERVED*
20: *DEVICE NOT READY*
Any of these messages indicates a malfunction in the computer or a storage device. Try the operation again, and if the problem persists, notify installation personnel.

20: *TRACK LIMIT*
There is no space available on the device where the file resides. Be sure you have not accidentally created a gigantic file. If you use very large files, you may need to make special arrangements with the installation personnel.

ABNORMAL EXITS FROM APLUM

PARAMETER ERROR
This error indicates the APLUM command was incorrect in form or that a parameter was specified incorrectly.

TIME LIMIT
A CPU *TIME LIMIT* occurred and the *T,number* command was not used to continue processing (see Chapter 12).

PP ABORT
A peripheral processing unit requested that the program be terminated.

OPERATOR DROP
The computer operator intervened and terminated the program.

FILE LIMIT
More active files were used than are allowed by the user's validation limits (see LIMITS in Chapter 12).

SYSTEM ABORT
The operating system terminated the program. This presumably indicates a defect in the operating system.

APL SYSTEM ERROR (or *EXCHANGE PACKAGE*)
This indicates a defect in the APIUM system or a computer or operating system malfunction. Please report this error to installation personnel along with work that led to the problem and any further output from the APIUM system. Unlike most error messages, this is not an indication of an error by the APL programmer.

OTHER MESSAGES

DEL
This indicates that the input line was cancelled.

OVL
This indicates that the preceding input line was too long for the operating system.

Appendix B. Output Format

Character output is sent to a terminal unaltered except for character translation required for the particular type of terminal and omission of trailing blanks in rows of a matrix. This omission of trailing blanks in character output speeds the printing of the result from $\square CR$, the printing of tables of names, and so forth.

Numeric output is ordinarily shown in decimal form unless decimal form would not be sufficiently compact. When decimal form is used, up to $\square PP$ significant digits are shown, but trailing zeros beyond the decimal point are omitted, as is the decimal point itself if no digits follow. Numbers with a magnitude less than 1 are shown with a zero before the decimal point (e.g., 0.025, 0.125). All numbers in a column have their decimal points aligned.

Exponential form is used if decimal form would require more than 3 zeros after the decimal point before the first significant digit, if aligning decimal points in the column would require more than $1.5 \times \square PP$ characters positions, or if more than $\square PP$ digits would appear to the left of the decimal point. If any number in a column requires exponential format, the entire column is shown in exponential format with the decimal points and exponents aligned. All numbers in the column are shown with the same number of digits in the mantissa. The number of mantissa digits is less than $\square PP$ according to how many trailing zeros would otherwise appear in all numbers in the column. If no numbers in the column have digits beyond the decimal point, the decimal point is omitted.

Numbers in adjacent columns are separated by at least one space. However, no more spaces than necessary are used.

Appendix C.   Character Sets and Terminals

Many different types of terminals can be used with the APLUM system. In addition, card readers, printers, and files can be used for input and output. The characters available on these various devices are shown in the table at the end of this section. Many of these devices cannot print the full set of APL characters. APL characters are translated so as to print the same whenever possible. In some cases characters having a similar appearance are substituted (e.g., " for ¨ and @ for ⍝). In the table where two characters appear in the same column, either character may be used for input, but all output uses the second character. Note that the APLUM system assumes the same terminal type for input and output. Where there is a blank entry in the table, the "bad character symbol," □AV[220], is used. Note that the □AV indices are for 0-origin. The first 128 elements of □AV, when used in output, cause APLUM to issue the corresponding ASCII code. However, further translation may occur within the operating system for some types of terminals.

TERMINAL CONTROLS

Table C.1 shows the characters used to cancel an input line, correct an input line, stop a program not requesting input (called an "interrupt" elsewhere in the text), and to stop a program requesting input. The two entries for halting a program requesting input are for quote-quad and quad input, respectively.

TERMINAL TYPES

From the point of view of the user, the three APL terminal types are almost identical except for the control functions shown in Table C.1. The only other difference is that the two ASCII terminals have the extra symbols {}◇⊢⊣ which are represented on Selectric terminals as the overstrikes ⊢ ⊣ ≍ ≁ ↔. The Selectric terminals are distinguished from the others in that they are distinguished from the others in that they are

C-1

based on the IBM Selectric print mechanism. These terminals are further divided into correspondence and EBCDIC terminals. The APLUM system allows either correspondence or EBCDIC terminals to be used. (The APLUM system does not need to distinguish between EBCDIC and correspondence types because the operating system compensates for the differences). The Teletype Model 38 is an example of an ASCII-APL terminal, and the CDI 1030 is an example of a bit-pairing terminal. All of these terminals are shown in the character code tables in the column showing APL symbols.

Full-ASCII terminals are characterized by having upper and lower case letters. Because these terminals do not have all the APL symbols, many APL symbols must be represented by a dollar sign followed by two additional symbols.

ASCII terminals also include the Teletype Model 33 and Model 35. These terminals also require the use of the dollar sign sequences to represent most APL symbols.

The BATCH column is for ASCII line printers having 63 printing characters. The B501 column is for the CDC 501 printer which differs in a few character positions from ASCII. The 501 printer prints the KRONOS display code symbols.

The file system translates coded file characters using an ASCII coding in the file, regardless of the terminal type specified on the APLUM control card. If the 501 type printer is used instead of an ASCII printer, the APL symbols ¯-:!×'?@\~ will come out as ≡≠←∨∧↑↓≤≥⍕ instead of #'"_!&'?@\^ .

TABLE C.1.   TERMINAL CONTROLS

	CANCEL INPUT	CORRECT INPUT	STOP PROGRAM	STOP INPUT	*SIGN-ON CHARACTERS
ASCII with APL print	ESC	LINE-FEED BACKSPACE	BREAK	∅ or →	) RETURN
Full ASCII	ESC	LINE-FEED BACKSPACE	BREAK	$G. or $GO	RETURN RETURN
ASCII	ESC	LINE-FEED CTRL-H	BREAK	$G. or $GO	RETURN RETURN
Selectric APL	ATTN RETURN	ATTN BACKSPACE	ATTN ATTN	∅ or →	RETURN

*Sign-on characters apply only at the University of Massachusetts installation. These characters should be the first characters typed after dialing the computer. One exception to the above is that bit-pairing ASCII-APL terminals require ◊RETURN as the sign-on characters.

⎕AV Index	APL Symbol (Overstrike)	ASCII Symbol (Name)	TTY33 Symbol	BATCH Printer	B501 Printer	APL Coded Files
0	N (NU)	(NUL)	$NU	$NU	$NU	
1	H (SH)	(SOH)	$SH	$SH	$SH	
2	S (ST)	(STX)	$ST	$ST	$ST	
3	E (ET)	(ETX)	$ET	$ET	$ET	
4	B (EO)	(EOT)	$EO	$EO	$EO	
5	N (EN)	(ENQ)	$EN	$EN	$EN	
6	K (AK)	(ACK)	$AK	$AK	$AK	
7	B (BL)	(BEL)	$BL	$BL	$BL	
8	B (BJ)	(BS)	$BJ	$BJ	$BJ	
9	H (HT)	(HT)	$HT	$HT	$HT	
10	L (LF)	(LF)	$LF	$LF	$LF	
11	V (VT)	(VT)	$VT	$VT	$VT	
12	F (FD)	(FF)	$FD	$FD	$FD	
13	C (CR)	(CR)	$CR	$CR	$CR	
14	O (SO)	(SO)	$SO	$SO	$SO	
15	S (SI)	(SI)	$SI	$SI	$SI	
16	D (DE)	(DLE)	$DE	$DE	$DE	
17	D (D1)	(DC1)	$D1	$D1	$D1	
18	D (D2)	(DC2)	$D2	$D2	$D2	
19	D (D3)	(DC3)	$D3	$D3	$D3	
20	D (D4)	(DC4)	$D4	$D4	$D4	
21	N (NK)	(NAK)	$NK	$NK	$NK	
22	S (SY)	(SYN)	$SY	$SY	$SY	
23	E (EB)	(ETB)	$EB	$EB	$EB	
24	A (CA)	(CAN)	$CA	$CA	$CA	
25	M (EM)	(EM)	$EM	$EM	$EM	
26	B (SB)	(SUB)	$SB	$SB	$SB	
27	E (ES)	(ESC)	$ES	$ES	$ES	
28	F (FS)	(FS)	$FS	$FS	$FS	
29	G (GS)	(GS)	$GS	$GS	$GS	
30	R (RS)	(RS)	$RS	$RS	$RS	
31	U (US)	(US)	$US	$US	$US	
32	blank	blank	blank	blank	blank	blank
33	: ('.)	:	$EX	$EX	$EX	:
34						
35	$ (SI)	$	$DO	$DO	$DO	$
36		'	$QT	$QT	$QT	'
37						
38						
39						
40	'	'	'	'	'	'
41	)	)	)	)	)	)
42	(	(	(	(	(	(
43	'	'	'	'	'	'
44	*	*	*	*	*	*
45	-	-	-	-	-	-

□AV Index	APL Symbol (Overstrike)	ASCII Symbol (Name)	TTY33 Symbol	BATCH Printer	B501 Printer	APL Coded Files
46	.	.	.	.	.	.
47	/	/	/	/	/	/
48	0	0	0	0	0	0
49	1	1	1	1	1	1
50	2	2	2	2	2	2
51	3	3	3	3	3	3
52	4	4	4	4	4	4
53	5	5	5	5	5	5
54	6	6	6	6	6	6
55	7	7	7	7	7	7
56	8	8	8	8	8	8
57	9	9	9	9	9	9
58	:	:	$SCL	$SCL	$SCL	:
59	;	;	$SSC	$SSC	$SSC	;
60	<	<	$SLT	$SLT	$SLT	<
61	=	=	$SEQ	$SEQ	$SEQ	=
62	>	>	$SGT	$SGT	$SGT	>
63	?	?	$SQU	$SQU	$SQU	?
64		@				@
65	A	A	A	A	A	A
66	B	B	B	B	B	B
67	C	C	C	C	C	C
68	D	D	D	D	D	D
69	E	E	E	E	E	E
70	F	F	F	F	F	F
71	G	G	G	G	G	G
72	H	H	H	H	H	H
73	I	I	I	I	I	I
74	J	J	J	J	J	J
75	K	K	K	K	K	K
76	L	L	L	L	L	L
77	M	M	M	M	M	M
78	N	N	N	N	N	N
79	O	O	O	O	O	O
80	P	P	P	P	P	P
81	Q	Q	Q	Q	Q	Q
82	R	R	R	R	R	R
83	S	S	S	S	S	S
84	T	T	T	T	T	T
85	U	U	U	U	U	U
86	V	V	V	V	V	V
87	W	W	W	W	W	W
88	X	X	X	X	X	X
89	Y	Y	Y	Y	Y	Y
90	Z	Z	Z	Z	Z	Z

□AV Index	APL Symbol (Overstrike)	ASCII Symbol (Name)	TTY33 Symbol	BATCH Printer	B501 Printer	APL Coded Files
91	[	[	$OB	$OB	$OB	[
92	\	\	$BS	$BS	$BS	\
93	]	]	$CB	$CB	$CB	]
94	—	—	$UL	$UL	$UL	—
95						
96						
97	A (A_)	a	$AA	$AA	$AA	A
98	B (B_)	b	$BB	$BB	$BB	B
99	C (C_)	c	$CC	$CC	$CC	C
100	D (D_)	d	$DD	$DD	$DD	D
101	E (E_)	e	$EE	$EE	$EE	E
102	F (F_)	f	$FF	$FF	$FF	F
103	G (G_)	g	$GG	$GG	$GG	G
104	H (H_)	h	$HH	$HH	$HH	H
105	I (I_)	i	$II	$II	$II	I
106	J (J_)	j	$JJ	$JJ	$JJ	J
107	K (K_)	k	$KK	$KK	$KK	K
108	L (L_)	l	$LL	$LL	$LL	L
109	M (M_)	m	$MM	$MM	$MM	M
110	N (N_)	n	$NN	$NN	$NN	N
111	O (O_)	o	$OO	$OO	$OO	O
112	P (P_)	p	$PP	$PP	$PP	P
113	Q (Q_)	q	$QQ	$QQ	$QQ	Q
114	R (R_)	r	$RR	$RR	$RR	R
115	S (S_)	s	$SS	$SS	$SS	S
116	T (T_)	t	$TT	$TT	$TT	T
117	U (U_)	u	$UU	$UU	$UU	U
118	V (V_)	v	$VV	$VV	$VV	V
119	W (W_)	w	$WW	$WW	$WW	W
120	X (X_)	x	$XX	$XX	$XX	X
121	Y (Y_)	y	$YY	$YY	$YY	Y
122	Z (Z_)	z	$ZZ	$ZZ	$ZZ	Z
123	{ ([∘)		$LB	$LB	$LB	
124			$MD	$MD	$MD	
125	} (]∘)		$RB	$RB	$RB	
126	~		$TL	$TL	$TL	
127	⌐ (DZ)	(DEL)	$DZ	⟩	⌐	⟩
128		(K0)				
129		(K1)				
130		(K2)				
131		(K3)				
132		(K4)				
133		(K5)				
134		(K6)				
135		(K7)				

Table (□AV Index 136–180):

□AV Index	APL Symbol (Overstrike)	ASCII Symbol (Name)	TTY33 Symbol	BATCH Printer	B501 Printer	APL Coded Files
136		(K8)				
137		(K9)				
138		(K10)				
139		(K11)				
140		(K12)				
141		(K13)				
142		(K14)				
143		(K15)				
144		(K16)				
145		(K17)				
146		(K18)				
147		(K19)				
148		(K20)				
149		(K21)				
150		(K22)				
151		(K23)				
152		(K24)				
153		(K25)				
154		(K26)				
155		(K27)				
156		(K28)				
157		(K29)				
158		(K30)				
159		(K31)				
160	>	(K0)	$AN	$AN	$AN	
161	<	(N1)	$OR	$OR	$OR	
162	≤ (∨~)	(N2)	$ND	$ND	$ND	
163	≥	(N3)	$NR	$NR	$NR	
164	⍀	(N4)	$LE	$LE	$LE	
165	⍋	(N5)	$NE	$NE	$NE	
166	≠ (∇|)	(N6)	$GE	$GE	$GE	
167	⍒ (∆|)	(N7)	$UG	$UG	$UG	
168	⍔	(N8)	$DG	$DG	$DG	
169	↓	(N9)	$TA	$TA	$TA	
170	→	(N10)	$DR	$DR	$DR	
171	←	(N11)	$GO	$GO	$GO	
172	↑	(N12)	$IS	$IS	$IS	
173	⍫ (∇~)	(N13)	$LP	$LP	$LP	
174	⍙ (∆°)	(N14)	$LD	$LD	$LD	
175	⊽	(N15)	$DL	$DL	$DL	
176	∇	(N16)	$MN	$MN	$MN	
177	⌐	(N17)	$ML	$ML	$ML	
178	¬	(N18)	$MX &	$MX &		
179	×	(N19)	$DV	$DV	$DV	
180	÷	(N20)	$DQ	$DQ	$DQ	

Table (□AV Index 181–225):

□AV Index	APL Symbol (Overstrike)	ASCII Symbol (Name)	TTY33 Symbol	BATCH Printer	B501 Printer	APL Coded Files
181	⌶	(N21)	$NG	$NG	$NG	#
182	∆	(N22)	$DT	$DT	$DT	=
183		(N23)	$RK	$RK	$RK	
184	⊤ ⊥ (↑|)	(N24)	$LK	$LK	$LK	
185	○ (↓|)	(N25)	$CI	$CI	$CI	@
186	⍟ (○⋆)	(N26)	$LG	$LG	$LG	
187	⍉ (○\)	(N27)	$RU	$RU	$RU	
188	⌽ (○|)	(N28)	$RT	$RT	$RT	
189	⍝ (∇∧)	(N29)	$TP	$TP	$TP	
190	⍋ (∆_)	(N30)	$DM	$DM	$DM	
191	⍎	(N31)	$DU	$DU	$DU	
192	⌿	(N32)	$IO	$IO	$IO	
193	⌹	(N33)	$RO	$RO	$RO	
194	⍕	(N34)	$BV	$BV	$BV	
195	⍕	(N35)	$RP	$RP	$RP	
196	⍳ (⊥T)	(N36)	$IB	$IB	$IB	
197	⍴	(N37)	$IN	$IN	$IN	
198	⍵	(N38)	$IX	$IX	$IX	
199	⍺	(N39)	$ID	$ID	$ID	
200	⍷	(N40)	$UN	$UN	$UN	
201	∪ (□')	(N41)	$NL	$NL	$NL	
202	∩	(N42)	$QD	$QD	$QD	
203	○ (\-)	(N43)	$QP	$QP	$QP	
204	□ (/-)	(N44)	$BT	$BT	$BT	
205	⌷	(N45)	$SM	$SM	$SM	
206	⌹ (□÷)	(N46)	$XD	$XD	$XD	
207	⍱	(N47)	$OM	$OM	$OM	
208	⍲	(N48)	$AL	$AL	$AL	
209	⍵	(N49)				
210	⍺ (−,)	(N50)				
211	⍳ (T∘)	(N51)				
212	⍴ (↓∘)	(N52)				
213	⍷ (⊥∘)	(N53)				
214	ε	(N54)	$EP	$EP	$EP	
215	∊	(N55)	$EV	$EV	$EV	
216	⍕	(N56)	$FM	$FM	$FM	
217		(N57)	$CN	$CN	$CN	
218		(N58)				
219	⍺ (∇∆)	(N59)				
220	⍕	(N60)	$BC	$BC	$BC	
221	⊠	(N61)				
222		(N62)				
223		(N63)				
224	⍐ (OUT)	(G0)	$G.	$G.	$G.	
225	⍌ (OU)	(G1)	$OU	$OU	$OU	@

CR Index	APL Symbol (Overstrike)	ASCII Symbol (Name)	TTY33 Symbol	BATCH Printer	B501 Printer	APL Coded Files
226		(G2)				
227		(G3)				
228		(G4)				
229		(G5)				
230		(G6)				
231		(G7)				
232		(G8)				
233		(G9)				
234		(G10)				
235		(G11)				
236		(G12)				
237		(G13)				
238		(G14)				
239		(G15)				
240		(G16)				
241		(G17)				
242		(G18)				
243		(G19)				
244		(G20)				
245		(G21)				
246		(G22)				
247		(G23)				
248		(G24)				
249		(G25)				
250		(G26)				
251		(G27)				
252		(G28)				
253		(G29)				
254		(G30)				
255		(E0)				

Appendix D. APLUM Control Card

The optional parameters on the APLUM timesharing command (or batch control card) allow specification of the type of terminal (or batch options) to be used, the workspace to be used (thus avoiding a subsequent $LOAD$ command), and the constraints on the field length to be used. The general form for the control card is:

APLUM,option=value,option=value ... option=value

Indicating <u>terminal type</u>. When no terminal type is specified, APLUM assumes that an ASCII-APL terminal is being used if the job was entered from timesharing. If the job is a batch or remote batch job, APL assumes that 63 ASCII characters must be used for output (e.g., ⍳ becomes $IO, ρ becomes $RO). Other terminal types can be specified as follows:

TT=COR   Correspondence (or EBCDIC) Selectric terminal (⎕TT=1).

TT=TYPE   Type-pairing terminals (⎕TT=2).

TT=BIT   Bit pairing terminal (⎕TT=3).

TT=ASCAPL For ASCII-APL terminals (including Model 38 Teletype) equipped to print the APL character set. This type is normally assumed for timesharing users (⎕TT=4).

TT=TTY33   For Teletype Model 33 terminal or similar devices (⎕TT=5).

TT=ASCII   For full ASCII terminals not equipped to print the APL character set (⎕TT=6).

TT=BATCH   For devices that support the 63 ASCII character set ($\Box TT=7$). Usually used for batch or remote batch ASCII printers.

TT=B501   For batch 501 printer ($\Box TT=8$).

TT=TTY383 Teletype Model 38, keyboard arrangement 3 ($\Box TT=9$).

At the University of Massachusetts (and at a few other installations), the TT= option should not be specified for any terminals that print the APL character set because the operating system compensates for any differences from ASCII-APL.

Indicating batch output options.   The following options are intended primarily for batch users of APL. If the APLUM control card does not specify output options, it is assumed that timesharing users do not wish these options and that batch users do want them.

LO=EPS   Any or all of the options E, P, or S may be specified.   Any options not specified are not used.

E   Echo input.   The APL lines read as input are also sent as output.

P   Prohibit prompt.   The normal APL input prompts (6 spaces or $\Box$: plus transparent mode control bytes, a lack of which may cause the input translation for terminals to be incorrect) are not sent to the output file.

S   Shift output.   Causes a blank line to be added to the front of each output line to prevent the first character from being used for printer carriage control.

LO=0   To select none of the E, P, or S options.

Input and output file specification.   The input and output files normally used for APL are named INPUT and OUTPUT.   For timesharing jobs this causes input to come from the terminal and output to be sent to the terminal.   For batch jobs input ordinarily is from the card deck or *CSUBMIT* file, and output is to a line printer.   Other KRONOS files can be used instead. API translation of input and output is according to the TT option (or the default which depends on whether the job is batch type or timesharing type).

I=file-name   Causes input to be read from the named file.

L=file-name   Causes output to go to the named file.

D-2

L=0   No APL output is produced.   (All output is discarded.)

Initial workspace specification.   If no workspace is specified, a clear workspace is used. Some effort can be saved by specifying the initial workspace name on the APLUM control card.

WS=wsname   APL operations begin with a copy of the named workspace as the active workspace.

UN=user-number   Used to specify the user number of the initial workspace. Required only if the user number of the workspace differs from that used when signing on.

PW=passwd   If the workspace belongs to another user and has a password, the password must be provided in order to use it.

Field length specification.   If no field length is specified, the APLUM system chooses a minimum field length that depends on the current version of APLUM, and a maximum field length of 24576 words (60000 octal).   The field length is used for the APL system and the active workspace.   The actual field length used varies dynamically. If storage requirements exceed the maximum field length, a *WS FULL* message results. Note that specifying a field length greater than the user is validated to use or greater than the limit set by the computer operator may actually increase the chance of a *WS FULL* error. To check user validation, use the LIMITS control card or the LIMITS command (see Chapter 12).

MX=number   Sets the maximum field length.   The number is assumed to be in decimal form unless followed immediately by B, in which case it is interpreted as octal.   The value is actually rounded up to a multiple of 64.

MN=number   Sets the minimum field length.   The number is assumed to be in decimal form unless followed immediately by B, in which case it is interpreted as octal.   The value is actually rounded up to a multiple of 64.

D-3

Appendix E. Numerical Limits and Precision.

The CYBER computers can represent nonzero numbers having magnitudes in the approximate range $1.27E322$ to $1.27E^-322$. An operation that would ordinarily produce a number smaller in magnitude than $1.27E^-322$ actually produces zero. Most operations produce a *LIMIT ERROR* when the result would exceed $1.27E322$ in magnitude; however, simple operations such as addition and multiplication can produce $+\infty$ or $-\infty$ which are printed as $9.99E999$ and $-9.99E999$. Using these infinite values for any other operations will result in a *LIMIT ERROR*.

Numbers within this magnitude range are represented with an accuracy of about 14 decimal digits (more precisely, to within 1 part in $2*48$). The simple operations such as addition, subtraction, multiplication, and division can be expected to be accurate to within 1 part in $2*48$ except when cancellation magnifies the errors. However, operations involving numbers that are integers or powers of 2 give exact results unless the magnitudes differ greatly. For example, exact results are given by: $.5+4$, $.25-.125$, $8-3$.