File No. S370-22
Order No. SC20-1846

Systems

APL/CMS User's Manual
Programming RPQ MF2608

IBM

File No. S370-22
Order No. SC20-1846

7/74

Systems

APL/CMS User's Manual Programming RPQ MF2608

Program Number 5799-ALK

This publication describes APL/CMS. It also describes the APL/CMS auxiliary processors, which allow the APL program to perform input and output operations to disks, magnetic tapes, line printers, and other devices.

The programming RPQ described in this manual, and all licensed materials available for it, are provided by IBM on a special quotation basis only, under the terms of the License Agreement for IBM Program Products. Your local IBM branch office can advise you regarding the special quotation and ordering procedures.

IBM

First Edition (July 1974)

This edition corresponds to Release 1 of APL/CMS and to all subsequent releases until otherwise indicated in new editions or Technical Newsletters.

Changes are periodically made to the specifications herein; before using this publication in connection with the operation of IBM systems, consult the latest IBM System/360 and System/370 Bibliography, Order No. GA22-6822, for the editions that are applicable and current.

Requests for copies of IBM publications should be made to your IBM representative or to the IBM branch office serving your locality.

A form for readers' comments is provided at the back of this publication. If the form has been removed, comments concerning the contents of this publication may be addressed to IBM Scientific Center, APL/CMS Publications, 2670 Hanover Street, Palo Alto, California 94304.

This publication provides information on how to use APL/CMS, differences between APL/CMS and APL\360, and the auxiliary processors available with APL/CMS.

This manual has the following sections and two appendixes:

- Section 1 is the introduction to APL/CMS.
- Section 2 describes APL under CMS, the virtual machine concept, logging on to VM/370, related CP, CMS, and APL commands, how APL uses virtual storage, how to save and copy workspaces, and common error situations and what to do about them.
- Section 3 describes how APL/CMS differs from APL\360 in terms of keyboard I/O operations, error handling, function definition, primitive functions, and APL commands.
- Section 4 describes new functions not available with APL\360: new primitive functions, new system functions and system variables and shared variables.
- Section 5 describes CMS facilities for the APL/CMS user
- Section 6 describes the auxiliary processors that allow APL programs to request certain CMS services:

  --AP100 the Command Processor that passes commands to CMS and CP.
  --AP101 the Stack Input Processor that stores and supplies input entries for use by CMS or APL/CMS.
  --AP110 the CMS Disk I/O Processor that provides sequential and random access to CMS files.
  --AP111 the FILEDEF I/O Processor that provides sequential access, via QSAM, to any I/O device supported via FILEDEF. This includes readers, punches, printers, and real OS disks (in read/only mode).

- Appendixes that describe translation table options and auxiliary processor return codes.

The APL/CMS system combines the programming features of APL and the virtual machine facility of VM/370. Several subsets of the system can be defined. One can learn a subset of the system, use it to solve problems, and go on to learn more advanced features as the need arises. The study and the use of APL/CMS is as follows:

1. Read Sections 1 and 2 in this manual. If unfamiliar with APL, one should read the APL\360 User's Manual.
2. Read Section 3 and Section 4 through "System Functions and System Variables" of this manual. Try making use of these features.
3. Read the remainder of this manual. Try using the shared variable facility and the APL/CMS auxiliary processors.

PREREQUISITE PUBLICATIONS

APL\360 User's Manual, Order No. GH20-0906

COREQUISITE PUBLICATIONS

The following publications also provide information that may be of interest to the APL/CMS user.

APL/CMS Installation Manual, Order No. SC20-1845

APL\360 Primer, Order No. GH20-0689

IBM Virtual Machine Facility/370:

- Command Language Guide for General Users, Order No. GC20-1804
- System Messages, Order No. GC20-1808
- Terminal User's Guide, Order No. GC20-1810

APL Shared Variables (APLSV) User's Manual.

FIGURES

APL/CMS is an APL system that runs under CMS (Conversational Monitor System), a component of VM/370 (IBM Virtual Machine Facility/370). It provides the facilities of APL\360 together with language enhancements introduced by APLSV (the APL Shared Variable System).

The APL language and system commands used in APL/CMS are fundamentally the same as described in the APL\360 User's Manual. The reader must be familiar with that document. Minor differences are noted herein being mainly in the areas of system commands, primitive functions, function definition, error handling, and keyboard entry and output. Major differences are listed below.

Shared Variables: The addition of a shared variable facility which provides a simple and effective way of working with CMS disk files, magnetic tape files, and other high speed input and output devices. The facility is managed by dynamically executable system functions.

Large Workspaces: The use of virtual memory which allows the workspace size to vary from user to user up to a maximum of 16,400,000 bytes.

High Performance: A new implementation designed for rapid execution of the APL language. An APL Assist feature is available for the System/370 Model 145 to provide a further increase in execution speed.

Auxiliary Processors: Four auxiliary processors designed to give the APL user a convenient way to use the powerful facilities of VM/370 through the medium of shared variables.

Shared System: APL/CMS utilizes the feature of VM/370 which allows one copy of the system to service many independently operating APL/CMS virtual machines.

Scan: A new operator is provided for efficient representation and execution of algorithms which otherwise require iteration.

Execute and Format: Efficient conversion between character arrays and numerical arrays is provided by these new primitive functions.

Canonical Representation: System functions are provided to convert between defined functions and their representation as character matrixes.

System Variables: These specially treated shared variables communicate with the APL system to control parameters such as the index origin and to provide information such as the time of day.

Latent Expression: This system variable is automatically executed when a workspace is loaded.

SECTION 2: USING APL UNDER CMS

THE VIRTUAL MACHINE

VM/370 is a system that manages the resources of a single computer so that many different computing systems, called virtual machines, appear to exist. Each virtual machine appears to have a system console, a CPU, storage and input/output devices. It is not necessary to understand the implications of these terms, but you may want to know, for example, that the storage size of the virtual machine affects the size of an APL/CMS workspace.

To use APL/CMS, you need access to a virtual machine. Usually, your VM/370 system operations group can prepare a virtual machine for your use and tell you the name (called the userid) of the machine and a password that controls access to it.

VM/370 maintains a directory of all the virtual machines that share the resources of the real computer. The directory contains the userid, password, accounting information, normal size of virtual memory, maximum size of virtual memory, and a list of I/O devices, of varying properties, identified by numbers. Some of the properties of the machine can be changed during a terminal session (for example, increasing the size of the virtual storage up to the limit specified in the VM/370 directory).

Each user of APL/CMS must have access to a virtual machine. When the user logs on, VM/370 looks up the directory entry and supplies a virtual machine with the appropriate properties. Typically, a virtual machine has a console (for example, an IBM 2741), a CPU, storage, one or more virtual disks, a virtual card reader, punch, and line printer. A virtual disk is a part of the space on a disk device (for example, five cylinders on an IBM 3330).

ESTABLISHING A CONNECTION TO VM/370

Consult with your system operations group for the location and properties of the terminals you can use, and to find out how to establish a connection. The APL\360 User's Manual describes some of the terminals used for APL. In this manual, an IBM 2741 is assumed. The terminal has two type elements: one used for VM/370 and one used for APL. Install the VM/370 type element, and establish a connection. The terminal will type a response that will include VM/370 ONLINE or RESTART. If the keyboard is locked, press the attention key. Type LOGON followed by your userid (the name of your virtual machine) followed by M (short for MASK).

```
logon smith m
```

would be used by someone with the user identification SMITH. You may enter lowercase letters, as shown in the example. The machine will usually reply in uppercase letters. Typing the m causes VM/370 to print a mask string to conceal your password. The machine response is:

```
ENTER PASSWORD:
```
[redacted]

Enter your password on top of the printed mask string. The machine then prints the time and date, and any information that the operator wants you to know about. If you have problems logging on, contact your system administrator, or check the *VM/370: Terminal User's Guide* for guidance on what to do.

VM/370 has two major components. The first component is CP, which is the control program. It manages the real resources of the installation to provide independent virtual resources, in the form of virtual machines, for its users. CMS (the Conversational Monitor System) is an interactive monitor system that runs under CP and offers user-oriented features which control the virtual machine.

All programs running on a VM/370 system are under control of CP; they may or may not use CMS. APL/CMS is a system that provides the APL language using the facilities of CMS and CP under VM/370.

LOADING APL/CMS

After your password is accepted, the introductory message is printed, the keyboard is unlocked and CP waits for you to enter a CP command. Now enter the command which causes CP to load the APL system. This is done in one of two ways, depending on how your system operations group has set up the APL system.

The first method is to install the APL type element and type

```
IPL APL
```

IPL stands for initial program load. This command initiates the loading of the APL/CMS system. The machine responds:

```
A P L / C M S
CLEAR WS
```

When the keyboard unlocks, you can begin to enter APL statements and commands. If the machine responds

```
⌈↑⌈~ε| α*⊏     ⌊○ε⌈ ⊤○~ ε⊃ι⌈~
```

This message is SYSTEM APL DOES NOT EXIST with the VM/370 type element. You should verify that APL is the name assigned to APL/CMS at your installation.

The second method of invoking APL/CMS is to enter

```
ipl cms
apl
```

You will then be asked to install the APL type element and press the return key. The system responds as described in the previous paragraph. You can now use the system in the same way as an APL\360 system. You can log off the system by typing the APL command)OFF. The following is an example of a short session at the terminal.

```
d'x38z irvyr;      vm/370 online

l smith m
ENTER PASSWORD:
[redacted]
LOGON AT 18:01:18 PDT TUESDAY 03/19/74

ipl apl

A P L / C M S
CLEAR WS
      2+2
4
      2×3
6
      3÷2
1.5
     )OFF
CONNECT= 00:08:45 VIRTCPU= 000:00.66 TOTCPU= 000:02.20
LOGOFF AT 18:10:03 PDT TUESDAY 03/19/74
```

CP, CMS, AND APL COMMANDS

Some of the messages that you enter at the terminal will be treated as commands. The kind of command that can be used at any time depends on the current environment. There are three environments: CP, CMS, and APL. When you first log on, you are in a CP environment. If you IPL CMS, then you are in the CMS environment. If you IPL APL (or enter 'apl' when in the CMS environment), then you go to the APL environment. A summary of the way the environment changes is:

To go from	To	Enter
Not logged on	CP	logon
CP	APL*	ipl apl
CP	CMS*	ipl cms
CMS	APL*	ipl apl
CMS	APL	apl
APL	Logged off	)OFF
APL	CMS	)OFF HOLD†
APL	CP	)OFF HOLD†
CMS	Logged off	logoff
CP	Logged off	logoff
CMS	CP	CP

----
*Note that IPL APL and IPL CMS are not available in some installations.
†You return to CP if you invoked APL by an IPL.

When you are in the CP environment, all input data is treated as CP commands. When you are in the CMS environment, input data is treated as CMS commands; if they are not recognized as CMS commands, then they are treated as CP commands. When in the APL environment, all input data is treated as APL statements or commands. CMS does not accept APL commands, nor does APL obey CP or CMS commands.

If you have made a typing error while entering a CP or CMS command, then you have a chance to correct the error if you have not yet pressed the return key. Entering ¢ causes the machine to ignore everything to the left of the ¢. Entering @ causes the previous character to be ignored; entering @@ causes the two previous characters to be ignored, and so on. For example,

```
ipx a¢ ipl cms
iq@pl cms
ipl dm@@cms
```

all have the same effect as if "ipl cms" had been typed without any errors. If you make an error and fail to correct it, then you must wait until CP or CMS has processed it; typically, the result is an error message indicating an unknown command, the keyboard unlocks, and you can try again. (The @ and ¢ are the characters ¯ and ≥ of the 987 APL type element and ← and → of the 988 APL type element.) These VM/370 logical line edit characters can not be used while in the APL environment.

When a CMS command has finished execution, it responds as follows:

```
R; T=n.nn/x.xx  hh.mm.ss
```

This is called a Ready message. n.nn and x.xx are the virtual and real CPU times (in seconds and hundredths of seconds). hh.mm.ss is the time of day (hours, minutes, and seconds). If an error has occurred, then the ready message (R;) is replaced by R(nnnnn) where nnnnn is an error code; in most cases an explanatory message appears on a previous line.

USE OF VIRTUAL STORAGE

An APL program uses an area of storage called a workspace. The workspace contains user defined functions and data, some space for system tables, and a work area. As new functions and data are defined, the space they need is taken from the work area. The amount of work area can be determined by the APL system variable

⎕*WA* (I22 *ON APL*\360)

In an APL\360 system, the size of the workspace is fixed when the system is generated and all users have the same workspace size. In APL/CMS the size of the workspace is determined dynamically when you load APL. After space is allocated for the system and for input/output areas, the rest of virtual storage is allocated to the workspace. Subsequently, when loading a workspace, its work area is adjusted accordingly.

The space used by APL/CMS varies from installation to installation, but a typical figure is about 340,000 bytes, not including the workspace. The size of your virtual storage is determined by an entry in your VM/370 directory. When in the CP or CMS environment, you can find out what the size is by entering the command:

```
query storage
```

2^{10}

2^{20} The reply may be in units of K (for kilo or 1024) bytes or M (for mega or 1,048,576) bytes. If the reply is, for example, 512K then the work area in the workspace is approximately 170,000 bytes. As you become familiar with APL, you should be able to relate the workspace size to the kinds of APL programs you can run. The maximum virtual storage allowed by VM is 16M bytes. The maximum work area under APL/CMS is about 16,400,000 bytes. 2^9

SAVING AND RESTORING WORKSPACES

CMS refers to virtual disks by one of the letters A through G, S, Y, and Z. The distributed version of APL/CMS uses the following disks:

A -- User's private library
D -- Temporary space used during)COPY and)PCOPY, for example
G -- Libraries 1000 through 999999 (read/write access)
Z -- Libraries 1 through 999 (read-only access)

The A-disk and D-disk belong to your virtual machine, and normally only your machine has access to them. The other disks belong to another virtual machine which every APL/CMS user can access. The APL)SAVE command stores a workspace on the A-disk. A library number causes the public library to be accessed. A workspace cannot be saved in a read-only library. The read-only library is usually maintained by the APL system librarian. Everyone has access to the read/write public library if one exists.

Under APL\360 the user's account number identifies the user's private library to other users. Under APL/CMS, each private library is maintained on the user's A-disk. Users cannot directly access another user's private library. Numerous methods exist for transferring workspaces between users in a secure manner.

APL/CMS sets no limit on the number of workspaces that can be stored in the private library; the limit depends upon the total amount of disk space available. If insufficient space is available to perform a)SAVE, then a disk full error occurs. You can make room with the)DROP command. Section 5 contains an explanation as to how you can find the amount of disk space used by each workspace.

If the read/write public library (G-disk) is full, consult the APL/CMS systems staff at your installation. If the message DISK NOT AVAILABLE appears when attempting to save a workspace in the public libraries, try it again in a few seconds as another user was probably accessing that disk.

ERRORS

APL/CMS fundamentally is like APL\360 in its handling of errors. Most of the error messages are the standard APL\360 error reports. You may also get the types of errors discussed in this section.

NOT IN CP DIRECTORY

This message will be received when attempting to logon when you are not an authorized user. It may also appear if you made a typing error, or if there was line noise creating a problem.

PASSWORD INCORRECT

See the installation manager to find out your correct password. Unlike APL\360, you cannot directly respecify your password. Typing errors or line noise may cause this message to appear.

DMSACC112S DISK 'A(191)' DEVICE ERROR

This message occurs when your disk is not in the proper CMS format and must be formatted using the CMS FORMAT command. Unless this is done, APL/CMS does not have the capability to access your private library.

SYSTEM APL DOES NOT EXIST

This message is an indication that your installation has generated APL/CMS under a different name or that APL/CMS is available only from the CMS environment.

CP (keyboard unlocks)

The letters CP are typed as the result of telephone-line noise or importunate use of the attention key. Type the word BEGIN (B for short) and press the return key. If you are attempting to generate a double attention, type EXT and press the return key.

COMMAND COMPLETE ... CLEAR WS

This message (ellipsis indicates several lines of print) may be caused by an error in APL/CMS. Report the occurrence(s) to the system operations group. You can continue to work with APL, but your active workspace has been replaced by a clear workspace.

CP ENTERED, REQUEST PLEASE.

This message indicates that you are in the CP environment and must reload APL/CMS.

RESTART

-- or --

VM/370 ONLINE

Either message indicates that VM/370 has been restarted. You must logon to the system and reload APL/CMS. Your active workspace has been lost.

communication line loss

You may lose the active workspace because of an abnormal disconnect of the telephone connection to VM/370. A 15-minute time period is provided to allow you to reestablish the communication line and logon again. For the method of resuming APL execution refer to Section 3 under the heading "The CONTINUE Workspace".

SECTION 2: GENERAL SYSTEM CHANGES

This section describes how APL/CMS differs from APL\360. Differences are grouped in terms of keyboard I/O operations, error handling, function definition, primitive functions and system commands. This section is based upon the APLSV User's Manual.

CHANGES IN KEYBOARD ENTRY AND OUTPUT

DOUBLE ATTENTION

Generating a double attention under APL/CMS requires a fine touch as CP monitors attentions for possible entry to that environment. Pressing the attention key twice with deliberate speed should suffice. If the characters CP are typed, indicating entry to the CP environment, simply type EXT; the effect is the same as a properly executed double attention.

If you are in the CP environment and do not wish to generate a double attention, type BEGIN.

INPUT LINE LIMITATION

The maximum number of keystrokes which can be entered, excluding the terminal carriage return, is 130. If the entry is interrupted by the attention key, causing a caret to appear, then the keystroke count is begun again. Since input of this form appends to the line already entered, it is possible to enter more than 130 keystrokes by striking the attention on every group of 130 or less, the final group being terminated by the carriage return. The system limitation for input of this form is 762 keystrokes, including one character for each attention and carriage return. These input keystrokes must produce less than 380 APL characters.

OPEN QUOTE

All keyboard entries are terminated by a carriage return. A keyboard entry containing an open quote will invoke no special system treatment. It will produce a SYNTAX error report, as will an entry with unbalanced parentheses.

CHARACTER ERRORS

If character errors occur in an input line, a CHARACTER ERROR report is issued; the entry is then typed up to the first such error, at which point the keyboard unlocks to allow further entry as if the printed line had been entered from the keyboard. Note that the carriage return (except to terminate a keyboard entry) and the horizontal tabulate are invalid input characters. The terminal control characters backspace, horizontal tabulate, idle, linefeed, and new line are available as a five element character vector, QUADTC, in the workspace 1 SPECIAL.

COMMANDS DURING FUNCTION DEFINITION

During function definition, every keyboard entry beginning with a left bracket (supplied by the system for convenience) is treated as an EDIT or INPUT request. APL/CMS reports some errors (for example, SYNTAX and RANGE errors) on input but the entry is accepted.

A keyboard entry that does not begin with a left bracket is treated as an <u>immediate</u> <u>execution</u> entry. To produce an entry of this form, backspace and use the attention key to erase the beginning of the line provided by the system.

The following statements apply to immediate execution when in function definition mode:

- If a workspace is saved during function definition, then a subsequent loading of the workspace resumes the function definition at the point when the save command was given.
- If the function in edit is copied from a saved workspace, it will be copied in closed form.
- Any attempt to enter or leave function definition results in the message DEFN ERROR.
- Any action causing an explicit or implicit erasure of the function being edited terminates function definition mode.

ESCAPE FROM LITERAL INPUT

Overstruck O U T interrupts execution but no longer causes an exit from the function.

EXTENDED PRINT WIDTH

The maximum printing width (as set by the WIDTH command and other facilities) can be set to 254 characters.

BARE OUTPUT

Normal output includes a concluding carriage return so that the succeeding entry (either input or output) will begin at a standard position on the following line. Bare output, denoted by expressions of the form ⍞←*X*, does not include this carriage return if it is followed either by another bare output or by a character input (of the form *X*←⍞). Character input following a bare output is treated as if the user had spaced over to the position occupied at the conclusion of the bare output. For example:

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

        F
TRUE OR FALSE: THE SQUARE OF 1 IS 16 FALSE
        X
                                     FALSE
```

If the length of any single output string is more than the printing width then the carriage returns normally occasioned by the page width setting are inserted.

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

HETEROGENEOUS OUTPUT

Parentheses surrounding a heterogeneous output statement are no longer permitted. They can be systematically removed from any unlocked function by user-defined editing functions, employing the dynamic function definition capability provided by the functions ⎕CR and ⎕FX described in Section 4.

The facility for heterogeneous output does not represent a proper APL function; in particular, its result cannot be assigned a name. It was introduced in APL\360 to obviate awkward conversions of numbers to character representations. The format function described in Section 4 now provides such conversions. The user is advised to avoid the use of the heterogeneous output facility.

CHANGES IN ERROR HANDLING

DEPTH ERROR

These errors do not occur. The STACK FULL error, described below, describes a related system limitation.

STACK FULL ERROR

A portion of workspace storage, called the stack, is used to hold APL expressions during the execution of an APL statement, and to hold status information during the execution of a user-defined function. If the fixed space allocated to the stack has been used, a STACK FULL error message is typed. For corrective action, use)SI to display the state indicator. Clear the stack by repeated use of the '→' key. To change the size of this area, use the)STACK command as described in the Section 3 under the heading "Changes in System Commands"

STACK DAMAGE

The error report SI DAMAGE ENCOUNTERED is issued when execution can proceed no further because of damage to the execution stack. This may be caused by erasure of a pendent function, for example.

Generally, APL/CMS prints the report SI DAMAGED when damage occurs; however subsequent action can repair such damage. A good example is the damage which occurs when a suspended function is edited. The stack is always damaged <u>during</u> the function editing process and is normally restored to a proper condition upon termination of the editing.

ERRORS IN LOCKED FUNCTIONS

A locked function is treated essentially as primitive and its execution can invoke only a DOMAIN error, although conditions (such as WS FULL or RANGE error) arising from system limitations will also be reported. Moreover, execution of a locked function is terminated by any error occurring within it, or by a double attention.

RANGE ERROR

If an arithmetic result falls outside the range of numbers allowed by the system, then a RANGE error is given. For example, an attempt to evaluate 2*250 causes a RANGE error. An attempt to divide zero by itself also produces a RANGE error.

RESULT OF DEFINED FUNCTION

If the function header specifies that a defined function has a result, then failure to assign a value to the result will lead to a VALUE ERROR report on exit from the function.

<u>CHANGES IN FUNCTION DEFINITION</u>

IMMEDIATE MODIFICATION

An entry of the form [*N*⎕*M*] while in function definition mode now invokes the following special action for the case when *M* is zero: line *N* is displayed with the carrier resting at the end of the line, as if the line had just been entered from the keyboard. At this point, the line can be extended, or modified by backspace and attention, in the usual manner.

PRINT WIDTH LIMITATION

A line will not be extended beyond the printing width during the insertion of blanks. A line longer than the printing width cannot be directly edited. It may be possible to edit the line by setting ⎕PW to its maximum value. Alternatively, a line of this type can be changed by use of the system function ⎕*CR* to obtain a character representation of the function, modification of the resulting character matrix, and definition of the function by use of the system function ⎕*FX*.

LINE DELETION

A line of a function is deleted by entering a left bracket, a not symbol, the line number and a right bracket. For example:

```
[99]   [~4]
[4]    [~5]
[5]
```

will delete statements 4 and 5. In the above example the [99], [4], and [5] were supplied by APL and the other characters were entered by the user. The attention signal cannot be used to delete a line.

STOP AND TRACE

The stop and trace vectors associated with a function are nullified when the definition of that function is edited.

STACK DAMAGE

The stack is always damaged when a suspended or pendent function is <u>being</u> edited. This damage is normally repaired when the function definition is closed. Resumption of the suspended function will produce a SI DAMAGE ENCOUNTERED report if the damage is still present.

LINE DISPLAY

Lines of a function which have been modified or added are not put into canonical form until the function definition is closed. For example, extra spaces are not removed until the function is closed.

STOP AND TRACE IN LOCKED FUNCTIONS

Settings of stop and trace are automatically nullified when a function definition is locked.

FUNCTION HEADER

A function header which contains more than one occurrance of the same name is rejected and the report DEFN ERROR is given.

COMMENTS

Comments can be placed on the same line as executable code. Characters to the right of the leftmost lamp character (⍝), which is not in a quoted string, are saved as part of the statement but are not executed.

Comments, on lines by themselves, are exdented, like labels, during function display.

LABELS

Names used as labels within a function are active only when that function is being executed or is suspended and during those times, labels act as read-only local variables. At other times, when the function is pendent, and within any other defined function, these names are unencumbered by their use as labels.

CHANGES IN PRIMITIVE FUNCTIONS

MONADIC TRANSPOSE

The monadic transpose now reverses the order of all coordinates rather than interchanging only the last two. Formally, it is defined in terms of the dyadic transpose as follows:

```
⍉A ←→ (⌽⍳⍴⍴A)⍉A
```

With this change the identity

```
M+.×N ←→ ⍉(⍉N)+.×⍉M
```

which held for matrixes *M* and *N* now holds for higher-dimensional arrays. Indeed, the corresponding identity holds for any inner product f.g if g is commutative.

RESIDUE

The residue function was previously defined to depend only on the absolute value of its left argument. It is now defined as follows:

1. If $A=0$ then $A|B$ is equal to B.

2. If $A≠0$ then $A|B$ lies between A and zero (being permitted to equal zero but not A) and is equal to $B-N×A$ for some integer N.

For example:

```
        A←3 0 ¯3

        B←¯6 ¯5 ¯4 ¯3 ¯2 ¯1 0 1 2 3 4 5 6

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

        X←21.824
        .01|X
0.004
```

The new definition of residue can be stated formally as follows:

```
A|B  ←→  B−A×⌊B÷A+A=0
```

ENCODE

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

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

The definition of the encode function for a left argument having one or more negative elements is therefore affected by the change in the definition of residue. For example:

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

        ¯2 ¯2 ¯2⊤13              ¯2 ¯2 ¯2⊤5
¯1 ¯1 ¯1                 ¯1 ¯1 ¯1

        2 0 2⊤13                 ¯2 2 ¯2⊤5
0 6 1                    0 1 ¯1
```

GENERALIZED MATRIX PRODUCT AND MATRIX DIVIDE

The domain of the ⌹ function described in the <u>APL\360 User's Manual</u>, has been extended slightly to include vector and scalar arguments. This section defines the extensions, and also provides a more comprehensive discussion of the function and its potential applications.

The domino (⌹) represents two functions which are useful in a variety of problems including the solution of systems of linear equations, determining the projection of a vector on the subspace spanned by the columns of a matrix, and determining the coefficients of a polynomial that best fits a set of points in the least-squares sense.

When applied to a nonsingular matrix *A* the expression ⌹*A* (monadic) yields the inverse of *A*, and the expression *X*←*B*⌹*A* (dyadic) yields a value of *X* which satisfies the relation ∧/,*B*=*A*+.×*X* and is therefore the solution of the system of linear equations conventionally represented as Ax=b. In the following examples the floor function is used only to obtain a compact display:

```
          A←(⍳4)∘.≥⍳4
          A                  ⌊⌹A                ⌊A+.×⌹A
    1 0 0 0            1  0  0  0          1  0  0  0
    1 1 0 0           ¯1  1  0  0          0  1  0  0
    1 1 1 0            0 ¯1  1  0          0  0  1  0
    1 1 1 1            0  0 ¯1  1          0  0  0  1

          B←1 3 6 10
          X←B⌹A
          B              X              A+.×X          (⌹A)+.×B
    1 3 6 10       1 2 3 4        1 3 6 10       1 2 3 4

          C←4 2⍴1 2 3 5 6 9 10 14
          Y←C⌹A
          C              ⌊Y              ⌊A+.×Y          ⌊(⌹A)+.×C
     1  2           1  2            1  2            1  2
     3  5           2  3            3  5            2  3
     6  9           3  4            6  9            3  4
    10 14           4  5           10 14            4  5
```

The final example given shows that if the left argument is a matrix *C*, then *C*⌹*A* yields a solution of the system of equations for each column of *C*.

If *A* is nonsingular and if *I* is an identity matrix of the same dimension then the matrix inverse ⌹*A* is equivalent to the matrix divide *I*⌹*A*. More generally, for any matrix *P* the expression ⌹*P* is equivalent to the expression

```
    ((⍳R)∘.=⍳R)⌹P
```

where *R* is the number of rows in *P*.

The domino functions apply more generally to singular and nonsquare matrixes, and to vectors and scalars; any argument of rank greater than 2 is rejected (RANK ERROR). For matrix arguments *A* and *B* the expression *X*←*B*⌹*A* is executed only if

1. *A* and *B* have the same number of rows, and

2. the columns of *A* are linearly independent.

If the expression *X*←*B*⌹*A* is executable, then ⍴*X* is equal to

```
(1↓⍴A),1↓⍴B
```

and *X* is determined so as to minimize the value of the expression

```
+/,(B-A+.×X)*2
```

The domino functions apply to vector and scalar arguments as follows: except that the shape of the result is determined as specified above, a vector is treated as a one-column matrix (since a one-rowed matrix of more than one column would be rejected by condition 2 above) and a scalar is treated as a one-by-one matrix. In the case of scalar arguments *X* and Y, the expression *X*⌹Yis equivalent to *X*÷Y and the expression ⌹Y is equivalent to ÷Y.

The <u>APLSV</u> <u>User's</u> <u>Manual</u> contains several more examples which may be of interest to the reader familiar with problems of polynomial fitting and of geometry.

SUBSCRIPTED SPECIFICATION

In an expression such as,

```
R[P]←Q

  --or--

R[S;T]←Q
```

APL/CMS requires the shape of the array subscript, ⍴P or (⍴S),⍴T to be the same as the shape of Q unless Q is a scalar or one-element array.

COMPRESS AND EXPAND

As stated in the <u>APL\360</u> <u>User's</u> <u>Manual</u>, a scalar right argument of compress and expand is not extended automatically if required by the left argument. A multiple condition branch may be coded:

```
→(∨/C1,C2, ... ,CN)/LABEL
```

<u>One</u> <u>Element</u> <u>Arrays</u>: One-element arrays are acceptable as the left argument of take, drop, and reshape, the right argument of take and drop, and as the coordinate indicator for subscripted operators and functions.

Enclosed Specification: Enclosing a specification statement within parentheses causes an explicit result to be produced. This result will be printed if it is leftmost on the line.

CHANGES IN SYSTEM COMMANDS

COMMUNICATION COMMANDS

There are occasions when a user may wish to be undisturbed by messages arriving from another terminal. The command)MSG OFF blocks all messages from other terminals and the command)MSG ON restores the acceptance of messages. The commands)OPR and)MSG do not lock the keyboard; under APL/CMS they have the same effect as)OPRN and)MSGN. The)MSG command must specify the userid of the virtual machine to receive the message.

THE ERASE COMMAND

The ERASE command now acts on any global object, and no longer distinguishes between pendent functions and others. Problems that may possibly arise from erasing a pendent function are forestalled by the response SI DAMAGE, which advises the user to take appropriate action before resuming execution. If execution is resumed and SI damage is encountered, an SI DAMAGE ENCOUNTERED report is given and the SI is reduced.

THE SYMBOLS COMMAND

The command)SYMBOLS without a number prints the current number of names accommodated. The number of symbols can be set in a clear workspace by the command)SYMBOLS N. The minimum number of symbols allowed is 50. Once a name is used, it occupies space in the symbol table even if erased. Copying a workspace into a clear workspace will minimize the number of occupied spaces in the symbol table.

THE STACK COMMAND

A portion of workspace storage, called the stack, is used to hold APL expressions during the execution of an APL statement, and to hold status indication during the execution of a user defined function. The amount of space allocated to the stack may be changed in a clear workspace by the command)STACK N. When a clear workspace is loaded, N has the value 512.

WORKSPACE IDENTIFICATION

The command)WSID can be used to set a lock as well as the workspace name, using the same form as the)SAVE command. Used as an inquiry,)WSID returns only the workspace identification.

Note: APL/CMS supports a maximum of eight characters as the workspace identification. If more than eight characters are used to save, load, or copy a workspace, a WS NAME IS TOO LONG report is given.

LOCAL FUNCTION NAMES

As a result of the introduction of the system function ⎕*FX* (defined in Section 4), local names may now refer to functions as well as variables.

THE CONTINUE WORKSPACE

If a workspace named CONTINUE exists and does not have a lock, it is loaded when APL/CMS is invoked.

If the machine loses a connection with a terminal, then the active workspace is not saved. However, VM/370 provides a grace period of about 15 minutes for the user to establish a new connection and logon on to the system. Once reconnected, type:

```
term apl on attn off linesize 130 mode vm
begin
```

to resume APL execution.

APL/CMS can be run normally in a disconnected state for long jobs which do not require a terminal. A function in 1 APFNS provides this capability. The example shown in Section 6 for auxiliary processor 101 demonstrates a way to checkpoint a workspace by use of an APL function.

This section describes the new primitive functions, system functions, and system variables introduced by the APLSV system and available in APL/CMS. This section is based upon the APLSV User's Manual.

NEW PRIMITIVE FUNCTIONS

SCAN

For any dyadic function *α* and any vector *X*, the *α*-scan of *X* (denoted by *α\X*) yields a result *R* of the same shape as *X* such that *R[I]* is equal to *α/I↑X*. For example:

```
      +\1 2 3 4 5 6
1 3 6 10 15 21
      ρ+\⍳0
0
```

The scan is extended to any array as follows: if *R←α\[I]A*, then *ρR* equals *ρA* and the vectors along the *I*th coordinate of *R* are the *α*-scans over the vectors along the *I*th coordinate of *A*; scan applied to a scalar yields the scalar unchanged.

The following examples show some interesting uses of the scan:

```
      L←0 0 1 0 1 0 1
      L
0 0 1 0 1 0 1
      ~L
1 1 0 1 0 1 0
      ∨\L
0 0 1 1 1 1 1        ALL 1'S FOLLOWING THE FIRST 1
      ∧\~L
1 1 0 0 0 0 0        ALL 0'S FOLLOWING THE FIRST 0
      <\L
0 0 1 0 0 0 0        REMOVE ALL 1'S FOLLOWING THE FIRST

      X
 2  3  5  7
 3  1  7  8
 4  7  9  2

      ∧/X=⌈\X        1'S INDICATE ROWS OF X
1 0 0                WHICH ARE IN ASCENDING ORDER
      +\⍳5
1 3 6 10 15          TRIANGULAR NUMBERS
      ×\⍳5
1 2 6 24 120         FACTORIALS
```

For any associative function,*α* the following definition of *R←α\X* is formally equivalent to the definition *R[I]=α/I↑X*:

```
   R[1]=X[1]
   R[I]=R[I-1]αX[I] FOR I∊1↓⍳ρX
```

This definition requires only ¯1+⍴X applications of α (as compared to .5×(⍴X)×¯1+⍴X), and is the one actually used for associative functions. Because of the finite precision used in machine arithmetic the results of the two definitions may differ, and differ significantly, if the elements of X differ by many orders of magnitude. For example, compare the last element of the scan with the corresponding reductions in the following case:

```
      X←1E6 ¯1E6 1E¯16
      +\X
1000000 0 1E¯16
      +/X
0
      +/⌽X
1E¯16
```

Therefore, the scan, as well as reduction, should be used with care in work requiring high precision.

EXECUTE

Any character vector can be regarded as a representation of an APL statement (which may or may not be well-formed). The monadic function denoted by ⍎ (⊥ and ∘ overstruck) takes as its argument a character vector or scalar and evaluates or <u>executes</u> the APL statement it represents. When applied to a character argument that might be construed as a system command or the opening of function definition, an error will necessarily result when evaluation is attempted, because neither of these is a well-formed APL statement.

There are several major uses of the execute function:

1. In those instances where it is desired to use the <u>name</u> of an APL object as an argument of a function, rather than its <u>value</u>, the name can be enclosed in quotes, and the argument later evaluated within the function by means of the execute function. A common example of this is in the use of a general integration function whose arguments might be the vector of grid points and the name of the function to be integrated. For example:

```
        ∇ Z←L INT X;Y
   [1]    Z←(1↓X−¯1⌽X)+.×0.5×1↓Y+¯1⌽Y←⍎L,' X'
        ∇
        ∇ Z←Q X
   [1]     Z←X*3
        ∇
          'Q' INT .1×⍳5
   0.0162
```

2. When applied to a vector of characters representing numerical constants, the execute function will convert them to numerical values. This is particularly useful in this system, in which access to data generated by alien systems is provided through the shared variable facility, and large quantities of such data may need to be converted to numerical APL arrays.

3. Where it is necessary to treat collections of data that are related but cannot be combined into a single array, the execute function

allows families of names to be used for related variable. The proper variables for each case can be generated and used under program control, either by selecting one of a set of names from a character matrix, by computing a numerical suffix to a generic name, or by other means.

4. The construction ⍎⍞ is nearly equivalent to the use of ⎕ for requesting input from the keyboard during function execution, and has certain advantages: it allows complete control over output prior to the requested input, and permits the input to be examined by the function prior to the attempted execution.

5. Conditional expressions can be constructed in which execution is applied only to the expression selected by the condition, avoiding possible error generation or unnecessary computation. For example, a recursive definition of the factorial function can be written as a single conditional statement:

```
          ∇Z←FACT N
     [1]   ⍎3 ¯12[1+N≠0]↑'Z←1   Z←N×FACT N-1'
          ∇
```

The execute function may appear anywhere in a statement, but it will successfully evaluate only valid (complete) expressions, and its result must be at least syntactically acceptable to its context. Thus, execute applied to a vector which is empty, contains only spaces, or starts with → (branch symbol) or ⍝ (comment symbol) produces no explicit result and therefore can be used only on the extreme left. For example:

```
      ⍎''
      Z←⍎''
SYNTAX ERROR
      Z←⍎''
      ^
```

The domain of ⍎ is any character array of rank less than two, and RANK and DOMAIN errors are reported in the usual way:

```
      C←'3 4'
      +/⍎C
7
      ⍎1 3⍴C              ⍎3 4
RANK ERROR           DOMAIN ERROR
      ⍎1 3⍴C              ⍎3 4
      ^                   ^
```

An error can also occur in the attempted execution of the APL expression represented by the argument of ⍎; such an indirect error is reported by the usual error message followed by a display of the argument of the execute function, a caret showing the point of difficulty, a display of the statement containing the execute function and another caret marking the point of difficulty. For example:

```
      123+⍎'4÷0'
RANGE ERROR
⍎:    4÷0
      ^
      123+⍎'4÷0'
          ^
```

The symbol ⍕ (⊤ and ∘ overstruck) denotes two format functions which convert numerical arrays to character arrays. There are several significant uses of these functions in addition to the obvious one for composing tabular output. For example, the use of format is complementary to the use of execute in treating bulk input and output (via the shared variable facility), and in the management of combined alphabetic and numeric data.

The monadic format function produces a character array identical to the printing normally produced by its argument, but makes this result explicitly available. For example:

```
      M←2=?4 4⍴2
      R←⍕M
      M                 R                 R[;¯1+2×⍳4]
0 1 0 1           0 1 0 1           0101
0 0 1 1           0 0 1 1           0011
1 0 1 1           1 0 1 1           1011
0 0 1 1           0 0 1 1           0011
      ⍴M                ⍴R
4 4               4 8
      ⍴⍕2 5
3
      ∧/,R=⍕R
1
      ⍕'ABCD'
ABCD
```

The format function applied to a character array yields the array unchanged, as illustrated by the last two examples above. For a numerical array, the shape of the result is the same as the shape of the argument except for the required expansion along the last coordinate, each number going, in general, to several characters. The format of a scalar number is always a vector.

The dyadic format function accepts only numerical arrays as its right argument, and uses variations in the left argument to provide progressively more detailed control over the result. Thus, for *F⍕A*, the argument *F* may be a single number, a pair of numbers, or a vector of length 2×¯1↑1,⍴*A*.

In general, a pair of numbers is used to control the result: the first determines the total width of a number field, and the second sets the precision. For decimal form the precision is specified as the number of digits to the right of the decimal point, and for scaled form it is specified as the number of digits in the multiplier. The form to be used is determined by the sign of the precision indicator, negative numbers indicating scaled form. Thus:

```
      ρ⎕←A
 12.34            ¯34.567
 0                 12
 ¯0.26           ¯123.45
3 2
```

```
      ρ⎕←12 3⍕A
 12.340      ¯34.567
  0.000       12.000
 ¯0.260     ¯123.450
3 24
```

```
      R←9 2⍕A
      S←9 ¯2⍕A
```

```
      ρ⎕←R
   12.34   ¯34.57
    0.00    12.00
   ¯0.26  ¯123.45
3 18
```

```
      ρ⎕←6 0⍕A
    12   ¯35
     0    12
     0  ¯123
3 12
```

```
      ρ⎕←S
  1.2E01   ¯3.5E01
  0.0E00    1.2E01
 ¯2.6E¯01 ¯1.2E02
3 18
```

```
      ρ⎕←7 ¯1⍕A
   1E01   ¯3E01
   0E00    1E01
  ¯3E¯01  ¯1E02
3 14
```

If the width indicator of the control pair is zero, a field is chosen such that at least one space will be left between adjacent numbers. If only a single control number is used, it is treated like a number pair with a width indicator of zero:

```
      ρ⎕←2⍕A
 12.34  ¯34.57
  0.00   12.00
 ¯0.26 ¯123.45
3 16
```

```
      ρ⎕←¯2⍕A
  1.2E01   ¯3.5E01
  0.0E00    1.2E01
 ¯2.6E¯01  ¯1.2E02
3 18
```

```
      ρ⎕←0 2⍕A
 12.34  ¯34.57
  0.00   12.00
 ¯0.26 ¯123.45
3 16
```

```
      ρ⎕←0 ¯2⍕A
  1.2E01   ¯3.5E01
  0.0E00    1.2E01
 ¯2.6E¯01  ¯1.2E02
3 18
```

Each column of an array can be individually composed by a left argument that has a control pair for each:

```
      ρ⎕←0 2 0 2⍕A
 12.34  ¯34.57
  0.00   12.00
 ¯0.26 ¯123.45
3 15
```

```
      ρ⎕←8 3 0 2⍕A
  12.340  ¯34.57
   0.000   12.00
  ¯0.260 ¯123.45
3 16
```

```
      ρ⎕←6 2 12 ¯3⍕A
 12.34   ¯3.46E01
  0.00    1.20E01
 ¯0.26   ¯1.23E02
3 18
```

```
      ρ⎕←8 0 0 ¯2⍕A
      12 ¯3.5E01
       0  1.2E01
       0 ¯1.2E02
3 17
```

```
      6 2 8 3 3 0 4 0 5 0 12 4⍕,A
 12.34 ¯34.567  0  12    0   ¯123.4500
```

The format function applied to an array of rank greater than two applies to each of the planes defined by the last two coordinates. For example:

```
      L←2=?2 2 5⍴2
      L                    4 1⍕L
1 1 0 0 1            1.0 1.0 0.0 0.0 1.0
1 1 1 0 1            1.0 1.0 1.0 0.0 1.0

1 0 0 1 0            1.0 0.0 0.0 1.0 0.0
0 0 0 0 0            0.0 0.0 0.0 0.0 0.0
```

Tabular displays incorporating row and column headings, or other information between columns and rows, are easily configured using the format function together with extended catenation. For example:

```
      ROWHEADS←4 3⍴'JANAPRJULOCT'
      YEARS←71+⍳5
      TABLE←.001×¯4E5+?4 5⍴8E5

      (' ',[1]ROWHEADS),(2⌽9 0⍕YEARS),[1]9 2⍕TABLE
          72        73        74        75        76
JAN   204.49     26.21   ¯362.36    143.44    ¯93.20
APR   264.77   ¯357.23    136.92    ¯93.27    ¯66.01
JUL    71.18    276.93   ¯326.43    ¯67.20    328.26
OCT  ¯190.04    188.87    106.11    392.83   ¯202.37
```

It is no longer necessary to use heterogeneous output in order to conveniently combine literal statements with numerical results. For example:

```
      X←17.34                          M←'THE VALUE OF X IS '
                                       X←25.4
      'THE VALUE OF X IS ';X           M;X
THE VALUE OF X IS 17.34          THE VALUE OF X IS 25.4
      'THE VALUE OF X IS ',⍕X          (⍕M),(⍕X)
THE VALUE OF X IS 17.34          THE VALUE OF X IS 25.4
```

There are obvious restrictions on the left argument of format, since the width of the field must be large enough to hold the requested form, and if the specified width is inadequate the result will be a DOMAIN error. However, the width need not provide open spaces between adjacent numbers. For example, boolean arrays can be tightly packed:

```
      1 0⍕2=?4 4⍴2
1111
0110
0100
0000
```

The following formal characteristics of the format function need not concern the general user, but may be of interest in certain applications.

The least width, *W*, required to represent a column of numbers *C* for an indicated precision *P* is determined as follows:

```
W←1⌈(∨/C<0)+(~P∊0 ¯1)+(|P)+(4,⌈/0,1+⌊10⍟|C+C=0)[1+P≥0]
```

The expressions (*M*⍒*A*),*N*⍒*B* and (*M*,*N*)⍒*A*,*B* are equivalent if *M* and *N* are full control vectors, that is, if

((⍴*M*)=2×¯1↑⍴*A*)∧(⍴*N*)=2×¯1↑⍴*B*

If 2=⍴*M*, then (*M*⍒*A*),*M*⍒*B* and *M*⍒*A*,*B* are equivalent.

SYSTEM FUNCTIONS AND SYSTEM VARIABLES

INTRODUCTION

Although the primitive functions of APL deal only with abstract objects (arrays of numbers and characters), it is often desirable to bring the power of the language to bear on the management of the concrete resources or the environment of the system in which APL operates. This can be done within the language by identifying certain variables as elements of the interface between APL and its host system, and using these variables for communication between them. While still abstract objects to APL, the values of such system variables may have any required concrete significance to the host system.

In principle all necessary interaction between APL and its environment could be managed by use of a complete set of system variables, but there are situations where it is more convenient, or otherwise more desirable, to use functions based on the use of system variables which may not themselves be made explicitly available. Such functions are called, by analogy, system functions.

System variables and system functions are denoted by distinguished names that begin with a quad. The use of such names is reserved for the system and cannot be applied to user-defined objects. They cannot be copied, grouped, or erased; those that denote system variables can appear in function headers, but only to be localized. Within APL statements, distinguished names are subject to all the normal rules of syntax.

SYSTEM FUNCTIONS

Like the primitive abstract funtions of APL, the system functions are available throughout the system, and can be used in defined functions. They are monadic or dyadic, as appropriate, and have explicit results. In most cases, they also have implicit results, in that their execution causes a change in the environment. The explicit result always indicates the status of the environment relevant to the possible implicit results. Several of the system functions are used to control the shared variable facility and are described below in this section under the heading "Shared Variables"; the rest follow.

Canonical Representation

The character array printed in displaying the definition of a function F is clearly an unambiguous representation of the function F. The representation remains unambiguous if the ∇ symbols and the line numbers with their brackets are removed and the lines made flush left. If the rows are then padded with spaces on the right, where necessary to make them all of equal length, the resulting matrix is called the canonical representation of F. The canonical representation of a defined function is obtained as a result of applying the system function ⎕CR to the character scalar or vector representing the name of the function. For example:

```
        ∇BIN[⎕]∇
      ∇ Z←BIN X
[1]     Z←1
[2]    L1:Z←(0,Z)+Z,0
[3]     →(X≥⍴Z)/L1
[4]    ⍝ ILLUSTRATE ⎕CR
      ∇
        M←⎕CR 'BIN'
        M
Z←BIN X
Z←1
L1:Z←(0,Z)+Z,0
→(X≥⍴Z)/L1
⍝ ILLUSTRATE ⎕CR

        ⍴M
5 16
       BIN 4
1 4 6 4 1
```

The function ⎕CR applied to any argument which does not represent the name of an unlocked function yields a matrix of dimension 0 by 0. Possible error reports for ⎕CR are RANK error if the argument is not a vector or a scalar, or DOMAIN error if the argument is not a character array.

<u>Function Establishment</u>

The definition of a function can be established or <u>fixed</u> by applying the system function ⎕FX to its canonical representation. To continue the preceding example:

```
        M[3;11]←'-'
        ⎕FX M
BIN
        ∇BIN[⎕]∇
      ∇ Z←BIN X
[1]     Z←1
[2]    L1:Z←(0,Z)-Z,0
[3]     →(X≥⍴Z)/L1
[4]    ⍝ ILLUSTRATE ⎕CR
      ∇
       BIN 4
1 ¯4 6 ¯4 1
```

As shown in the foregoing example, the function ⎕FX produces as an explicit result the vector of characters which represents the name of the function being fixed, while replacing any existing definition of the function with the same name. The argument of ⎕FX is, of course, unaffected. The name provided by the explicit result can be conveniently used in a variety of ways. For example:

```
      ⍋(⎕FX M),' 4'
  1 ¯4 6 ¯4 1
```

The name of any function established by the function ⎕FX obeys the normal rules of localization. Thus if a function ABC is established within a function G in which the name ABC is local, the definition of ABC disappears upon termination of execution of G. Function definition mode continues to apply to global names only.

An expression of the form ⎕FX M will establish a function if the following conditions are met:

1. M is a valid representation of a function. Any matrix which differs from a canonical matrix only in the addition of nonsignificant spaces (other than a row consisting of spaces only) is a valid representation.

2. The name of the function to be established does not conflict with an existing use of the name for a pendent, suspended or the current function or for a label, group, or variable.

3. Certain system restrictions must be satisfied; for example, the number of rows of the function must be less than 2049. A NONCE error is reported in these cases.

If the expression fails to establish a function then no change occurs in the workspace and the expression returns a scalar index of the row in the matrix argument where the fault was found. If the argument of ⎕FX is not a matrix a RANK error will be reported, and if it is not a character array a DOMAIN error will result. If condition 3 above is violated, a NONCE error is reported.

Dynamic Erasure

Certain name conflicts can be avoided by using the expunge function ⎕EX to dynamically eliminate an existing use of a name. Thus ⎕EX 'PQR' will erase the object PQR unless it is a label, a group, or a pendant or suspended function. The function returns an explicit result of 1 if the name is now unencumbered, and a result of 0 if it is not, or if the argument does not represent a well-formed name. The expunge function applies to a matrix of names and then produces a logical vector result. ⎕EX will report a RANK error if its argument is of higher rank than a matrix, or a DOMAIN error if the argument is not a character array.

The expunge function is like the)ERASE command except that it applies to the active referant of a name (which may be local), and cannot expunge certain names.

Name List

The dyadic function ⎕NL yields a character matrix, each row of which represents the name of an object in the dynamic environment. The right argument is an integer scalar or vector which determines the class of names produced as follows: 1, 2, and 3, respectively, invoke the names of labels, variables, and functions. The left argument is a scalar or vector of alphabetic characters which restricts the names produced to those with an initial letter occurring in the argument. The ordering of the rows is accidental.

The monadic function ⎕NL behaves analogously with no restriction on initial letters. For example, ⎕NL 2 produces a matrix of all variable names, and either of ⎕NL 2 3 or ⎕NL 3 2 produces a matrix of all variable and function names.

The uses of ⎕NL include the following:

- In conjunction with ⎕EX, all the objects of a certain class can be dynamically erased; or a function can be readily defined that will clear a workspace of all but a preselected set of objects.
- In conjunction with ⎕CR, functions can be written to automatically display the definitions of all or certain functions in the workspace, or to analyze the interactions among functions and variables.
- The dyadic form of ⎕NL can be used as a convenient guide in the choice of names while designing or experimenting with a workspace.

Name Classification

The monadic function ⎕NC accepts a matrix of characters and returns a numerical indication of the class of the name represented by each row of the argument. A single name may also be presented as a vector or a scalar.

The result of ⎕NL is a suitable argument for ⎕NC, but other character arrays may be used, in which case the possible results are integers ranging from 0 to 4. The significance of 1, 2, and 3 are as for ⎕NL; a result of 0 signifies the corresponding name is available for any use; a result of 4 signifies that the argument is not available for use as a name. The latter case may arise because the name is in use for denoting a group, or because the argument is a distinguished name or not a valid name at all.

Delay

The delay function, denoted by ⎕DL, evokes a pause in the execution of the statement in which it appears. The argument of the function determines the duration of the pause, in seconds, but the accuracy is limited by possible contending demands on the system at the moment of release. The delay can be prematurely terminated by a single or double attention. A single attention causes execution of the statment to be resumed. A double attention causes an INTERRUPT report and the keyboard unlocks. The explicit result of ⎕DL is a scalar value equal to the actual delay. If the argument of ⎕DL is not a scalar or a vector with a single numerical value, a RANK or DOMAIN error will be reported.

SYSTEM VARIABLES

System variables are instances of shared variables. The characteristics of shared variables that are most significant here are these:

1. If a variable is shared between two processors, the value of the variable when used by one of them may well be different from what that processor last specified.

2. Each processor is free to use or not use a value specified by the other, according to its own internal workings.

System variables are shared between a workspace and the APL processor. Sharing takes place automatically each time a workspace is activated and, when a system variable is localized in a function, each time the function is used.

The system variables are listed below. Also listed are the workspace functions and I-beam functions they are intended to replace. These earlier ad hoc facilities are still available, but are expected to be supplanted by the use of system variables. The old definitions of the workspace functions will no longer work. New definitions may be copied from 1 WSFNS, or defined, as in the following example:

```
      ∇ Z←ORIGIN N
 [1]    Z←⎕IO
 [2]    ⎕IO←N
      ∇
```

The system variables are:

Name	Purpose	Value in Clear WS	Range	Replaces
⎕CT	Comparison tolerance (relative)	−/2*−43 55	0 to −/2*−24 55	SETFUZZ
⎕IO	Index origin used in indexing, ?, ⍒,⍋, and ⍳	1	0 1	ORIGIN
⎕LX	Latent expression executed on activation of workspace	''	Characters	none
⎕PP	Printing precision: affects numeric output and monadic ⍕	10	⍳16	DIGITS
⎕PW	Printing width	120	29+⍳225	WIDTH
⎕RL	Random link: used in ?	7*5	⍳¯1+2*31	SETLINK
⎕AI	Accounting information: identification (always zero), computer, connect, keying times, measured in milliseconds			⌶29 21 24 19
⎕LC	Line counter: line numbers of functions in execution	⍳0		⌶27 26
⎕TS	Time stamp: year, month, day, hour, minute, second, millisecond			⌶25 20
⎕TT	Terminal type: 0 for 1050, 1 for Selectric, 2 for PTTC/BCD, ¯1 for other devices			⌶28
⎕UL	User load (always 1)			⌶23
⎕WA	Working area available			⌶22

Two classes of system variables can be discerned:

- <u>Comparison tolerance, index origin, latent expression, printing precision, printing width, and random link</u>: In these cases the value specified by the user (or available in a clear workspace) is used by the APL processor during the execution of operations to which they relate. If the user attempts to specify an inappropriate value to a system variable, the specification is suspended and a NONCE error is reported. On entry to a function whose header specifies the local instance of a system variable, that system variable is made local and then the value extant on entry to the function is assigned to the local instance.

- Account information, line counter, time stamp, terminal type, user load and work area: In these cases localization or setting by the user are immaterial. The APL processor will always reset the variable before it can be used again. When using APL/CMS, ⎕UL is always 1 because there is only one user on a virtual machine.

Latent Expression: The APL statement represented by the latent expression is automatically executed whenever the workspace is activated. Formally, ⎕LX is used as an argument to the execute function (⍎⎕LX) and any error message will be appropriate to the use of that function. Common uses of the latent expression include the form:

    ⎕LX←'G'

used to invoke the arbitrary function G, and the form:

    ⎕LX←'''FOR NEW FEATURES IN THIS WS ENTER: NEW'''

used to print a message on activation of the workspace. The form ⎕LX←'→⎕LC' can be used to automatically restart a suspended function. The variable ⎕LX may also be localized within a function and respecified therein to furnish a different latent expression when the function is suspended.

Atomic Vector: This system variable found in APLSV is not present in APL/CMS. Users with a requirement for the terminal control characters backspace, horizontal tabulate, idle, line feed, and new line can copy 1 SPECIAL QUADTC to obtain these as a five-element character vector.

SHARED VARIABLES

INTRODUCTION

Two otherwise independent concurrently operating processors can communicate, and thereby be enabled to cooperate, if they share one or more variables. Such shared variables constitute an interface between the processors, through which information can be passed to be used by each for its own purposes. In particular, variables can be shared between an APL workspace and some other processor that is part of the overall APL system, to achieve a variety of effects including the control and utilization of devices such as printers, card readers, magnetic tape units, and magnetic disk storage units.

In an APL workspace, a shared variable may be either global or local, and is syntactically indistinguishable from ordinary variables. It may appear to the left of an assignment, in which case its value is said to be set, or elsewhere in a statement, where its value is said to be used. Either form of reference is an access.

At any instant a shared variable has only one value, that last assigned to it by one of its owners. Characteristically, however, a processor using a shared variable will find its value different from what it might have set earlier. A familiar example of this in APL is the quote quad when it is used successively for output from a function and input to it from the keyboard; ⍞ is, in fact, a variable shared between the function and the user at the terminal.

A given processor can simultaneously share variables with any number of other processors. However, each sharing is bilateral; that is, each shared variable has only two owners. This restriction does not represent a loss of generality in the systems that can be constructed, and commonly useful arrangements are easily designed.

Shared variables are used to communicate between independent processors. The kinds of processors which can be used depend on the facilities of the host operating system. APL/CMS has a single APL user on each virtual machine, running under the CMS component of VM. Processors available in APL/CMS include the single user's APL program, the APL/CMS processor and four APs (auxiliary processors) which are supplied with the system. The APs are described in detail in Section 6; for example, AP110 can read and write CMS disk files.

The full power of the shared variable concept can be gained by reading the <u>APLSV User's Manual</u>. APLSV runs under the OS/VS operating system. It allows an APL program to share a variable with an auxiliary processor. It also allows two active APL users to share a variable; in other words two APL users can communicate with each other via shared variables.

<u>Note</u>: APL/CMS does not allow active APL users to communicate <u>directly</u> with each other through the use of shared variables.

We now describe the shared variable system functions, as they are normally used under APL/CMS. We then describe some additional features present for compatibility with APLSV.

OFFERS

A single offer to share is of the form P ⎕SVO N, where P is the identification of another processor and N is a character scalar or vector which represents a name. The result is a number which gives the degree of coupling: <u>zero</u> if no offer has been made, <u>one</u> if an offer has been made but not matched, <u>two</u> if sharing is completed. An offer can never decrease the degree of coupling. For example:

```
      110 ⎕SVO'99'
0
      110 ⎕SVO'ALPHA'
2
      999 ⎕SVO'ALPHA'
0
      999 ⎕SVO'BETA'
1
```

The first offer is rejected because 99 is not a name. The result 0 indicates no coupling. The second offer is offered and accepted resulting in a degree of coupling of 2. The third offer is rejected because ALPHA is already shared with auxiliary processor 110. The fourth offer produces a degree of coupling of 1 because it is offered but not accepted by a matching counter offer.

If a user attempts to share more variables than the limit determined when the system is installed the error reported is INTERFACE QUOTA EXHAUSTED. This limit is 45 in APL/CMS as distributed. Other system limits may cause the report SVP MEMORY FULL or SVP NAME TABLE FULL to be issued; these limits are described in the <u>APL/CMS Installation Manual</u> and may be changed when the system is installed.

A set of offers can be made by using a vector left argument (or a scalar or one-element vector which is automatically extended) and a matrix right argument, each of whose rows represents a name (or a name pair: see below for surrogate names). The offers are then treated in sequence and the explicit result is the vector of the resulting degrees of coupling. For example:

```
        110 ⎕SVO 3 5⍴'DELTA12345RHO  '
  2 0 2
```

If the quota of shared variables is exhausted in the course of such a multiple offer, none of the offers will be tendered.

The monadic function ⎕SVO accepts an argument N which is a scalar, vector or matrix of characters. It does not change the degree of coupling but does have an explicit result which gives the degree of coupling of the name or names in N. If the degree of coupling of a name is 1 or 2, dyadic offer operates, in effect, as a query the same as the monadic form. The expression

```
  (0≠⎕SVO ⎕NL 2)⌿ ⎕NL 2
```

produces a character matrix of shared variable names, one per row.

RETRACTION

Sharing offers can be retracted by the monadic function ⎕SVR applied to a name or a matrix of names. The explicit result is the degree (or degrees) of coupling prior to the retraction. The implicit result is to reduce the degree of coupling to zero.

Retraction of sharing is automatic if the user signs off or loads a new workspace. Sharing of a variable is also retracted by its erasure or, if it is a local variable, upon completion of the function in which it appeared, or if an object with the same name is copied into the workspace.

USING THE APL/CMS AUXILIARY PROCESSORS

The APL/CMS auxiliary processors use the initial value of the shared variable to indicate the source and/or destination of the processors input/output and to select certain options. For example, the following sequence will write two records to a CMS file called "EXAMPLE SCRIPT". The "370" conversion option is selected. The first record is "THIS GOES IN THE FIRST RECORD" and the second record is "AND THIS GOES IN THE SECOND".

```
        S←'EXAMPLE SCRIPT(370'
        110 ⎕SVO'S'
  2
        S←'THIS GOES IN THE FIRST RECORD'
        S←'AND THIS GOES IN THE SECOND'
        ⎕SVR'S'
  2
```

In this case, S is the name of the shared variable. Notice that the argument to ⎕SVO and ⎕SVR is 'S' not S. See Section 6 for further details of the auxiliary processors.

COMPATIBILITY WITH APLSV

APL/CMS has one user on one virtual machine. In addition, CMS is a synchronous system, that is, one which completes each request before continuing with the next. APLSV has many users on one real machine and it is possible for asynchronous processes to communicate with each other. APL/CMS provides certain features which have little value in the CMS environment but which may be useful for maintaining compatibility with APLSV. We give a brief description of the APL/CMS feature here; full details of the APLSV feature will be found in the <u>APLSV User's Manual</u>.

<u>Surrogate Names</u>

The description of ⎕*SVO* implied that two processors which share a variable must agree on a common name for the variable. Since it may be difficult for two processors to agree on a common name, ⎕*SVO* will accept a single name, such as 'ALPHA', or a pair of names such as 'ALPHA BETA'. The second name of the pair is called a <u>surrogate</u> name. In the case of a single name, the name is its own surrogate. The process of sharing causes variables with the same surrogate names to matched. APL/CMS allows the use of surrogate names, but since the auxiliary processors will accept any names offered to them, surrogate names need not be used.

<u>Access Control</u>

In certain practical applications of shared variables, it is important to know that a new value has been assigned since the variable was last referenced, or that the old value has been used before a new one is assigned. The shared variable facility has an <u>access control</u> mechanism which allows processors to enforce an access discipline.

The access control operates by inhibiting the setting or use of a shared variable by one owner or the other, depending on the <u>access state</u> of the variable and the value of the <u>access control matrix</u>. Access control is set by an expression of the form *B* ⎕*SVC N*, where N is a character vector representing the name of a shared variable and B is a four element vector of zeroes and ones (or a scalar or one-element vector which is extended in the usual way). See the <u>APLSV User's Manual</u> for a complete explanation.

The monadic form of ⎕*SVC* reports the current setting of the control matrix, one four element row per name in *N*. The argument *N* of ⎕SVC may specify a matrix of names, and in the dyadic case, the argument *B* may be a matrix of zeroes and ones with one row for each name.

<u>Note</u>: The APL/CMS auxiliary processors set the access control vector to 1 1 1 1, the highest degree of control possible, and are programmed to access variables shared with them in such a way as to prevent interlocks.

If a valid offer is made to a nonexistent processor then the access control vector is set to zero, for example

```
        400 ⎕SVO'X'
1
        ⎕SVC'X'
0 0 0 0
        X←99
        X
99
        1 1 1 1 ⎕SVC'X'
1 1 1 1
```

The access control vector now specifies that successive accesses by the APL program requires an intervening access by processor 400. Processor 400 does not exist so it will neither use or set X. If the APL program attempts to access X then a permanent wait would result. APL/CMS detects this situation, forces an interrupt, and prints the error report INTERRUPT: PERMANENT SV WAIT.

Inquiries

There are three monadic inquiry functions which produce information concerning the shared variable environment: ⎕SVO, ⎕SVC, and ⎕SVQ. The first two have been discussed in this section. A user who applies the shared variable query function, ⎕SVQ, to an empty vector obtains a vector result containing the identification of each other user making any sharing offer to the user.

Applying the same function to an integer scalar or one-element vector obtains a matrix of the names offered by the processor identified in the argument. This matrix includes only those names which have not been accepted by counter offers from the inquiring user.

In this part of the manual, we discuss some CMS concepts and a subset of CP and CMS commands that may be useful to the APL user. For further details, see the VM/370: Command Language Guide for General Users. CMS commands cannot be used in the APL environment directly but can be transmitted via auxiliary processor 100 described in Section 6. APL commands are not recognized by CP or CMS.

LINK AND ACCESS COMMANDS

The VM/370 directory contains the properties of individual virtual machines. One or more entries specify the virtual disk(s) that belong to a particular machine. CP identifies disks by a device address. Most machines have a primary disk at device address 191 and will have links to disks belonging to other machines. For example, the directory for an APL user may contain an entry of the form:

```
link vmaplsys 101 101 rr
```

This indicates the user has access to the 101 disk of a machine whose userid is VMAPLSYS. The last two operands define the device address as 101 and the access to the disk as read-only for this user.

As noted in the Section 2 discussion on "Saving and Restoring Workspaces," CMS refers to disks by mode letter. A correspondence between the CP device addresses and the CMS usage of the disks is established by an ACCESS command. For example,

```
access 101 z
```

allows CMS to use disk 101 as a Z disk. CMS accesses 191 as the A-disk and APL/CMS accesses 101 as the Z disk.

The CP command LINK and the CMS command ACCESS, can be used, among cooperating users, to establish a common disk for storage of workspaces and data files. Access to a user's private disk is controlled by VM/370 directory entries and passwords.

CMS FILES

Items on a virtual disk are organized into CMS files. A CMS file is known by its filename, filetype, and filemode. The filename and the filetype are each composed of one- to eight-alphameric characters. The filetype is formed like a filename, but CMS commands usually associate a particular filetype with a particular kind of file. For example, APL/CMS uses the filetype VMAPLWS for all workspaces in the private library. EXEC files usually contain a list of CMS and CP commands to be executed. The filemode is one letter followed by one number. The letter specifies the disk the file is on. The numeral 1 indicates a permanent file available for both reading and writing.

The information in CMS files is grouped into records, which are the smallest unit in the file system. Records are grouped into files. Records in each file are fixed length, meaning the length of all records is the same (and must remain so) or variable length, which imposes no such restriction. The maximum length of a record is 65,532 bytes. CMS allows its files to be accessed sequentially or by specification of the record position (1-origin).

Random accessing can be used with fixed or variable files but it is more efficient with fixed files. Records can be changed in existing files, but the length of any record of a variable file, except the last record, cannot be changed without unpredictable results. Records cannot be deleted from files but can be added to the end of an existing file.

The units of information which CMS transmits to and from disk storage are called blocks. The CMS file system collects several short records or splits up long records as it transfers information block by block to and from disk. A block is 800 bytes long. You need be aware of blocks only if you wish to compute the amount of disk space used.

LISTFILE COMMAND

At some stage of APL execution, you may encounter a DISK FULL message. You can use the APL)LIB command to list the workspaces on the disk; however, you may have other files generated by the APL auxilary processors which are not listed by)LIB. You can list all files with the CMS LISTFILE command which has some of the following variants:

```
listfile fn ft fm
```

displays information concerning the file having the given filename (fn), filetype (ft), and filemode (fm). If fm is omitted then A is assumed; if ft and fm are omitted then all files with name fn on the A-disk are listed.

```
listfile * ft fm
```

lists all files with type ft on disk fm. The command

```
listfile * vmaplws
```

lists all VMAPLWS files, and

```
listfile
```

lists all files on the A-disk. There are several options that can also be used. If the command line ends in (ALLOC), then the format (fixed or variable), the number of records, and the number of blocks are listed for each file. For example:

```
listfile alpha (alloc)
FILENAME FILETYPE FM  FORMAT    RECS BLOCK
ALPHA    VMAPLWS  A1  V  4616      3     9
ALPHA    ASSEMBLE A1  F    80    392    40
```

"V 4616" denotes variable length records, with the longest record of length 4616 bytes. "F 80" denotes fixed-length records of 80 bytes each. The number under BLOCK gives the number of 800-byte blocks occupied by the file.

QUERY COMMAND

The CMS command LISTFILE lets you find the space used on a virtual disk by each file. More generally, the CMS command

```
query disk a
```

gives a summary of the state of the A-disk (other disks may be specified by their letter). An example of the reply to a query is,

```
A (191): 3 FILES; 155 REC IN USE, 1173 LEFT (of 1328),
         12% FULL (5 CYL), 3330, R/W
```

Do not be confused by this inconsistent usage: REC refers to what LISTFILE calls blocks. The reply shows that 12 percent of the available space is used. The virtual disk address is 191 and is 5 cylinders of 3330 disk storage, which is linked in read/write mode.

As mentioned earlier, the CP command QUERY STORAGE, indicates to you the size of the user's virtual storage. The CP command QUERY NAMES lists the names of the other virtual machines currently logged on the system. You can send messages by using the APL)MSG command, or the CP MSG (or MESSAGE) command

```
cp msg userid ...any message...
```

where userid is the name of the other virtual machine. (This use of the userid corresponds to the APL\360 use of the port number.)

ERASE COMMAND

You can free the disk space used by unwanted workspaces by issuing the APL)DROP command. You can get rid of workspaces or other unwanted CMS files by using the CMS command

```
erase fn ft fm
```

If fm is omitted, then A is assumed. Replacing fn with an asterisk (*) causes all files of type ft to be erased.

WORKSPACES AND CMS FILES

An APL)SAVE command causes the APL/CMS system to store the information from the workspace into a CMS file. Only useful information is stored, not the work area, so the space on the disk is related to the APL objects in the workspace, not the gross workspace size. The A-disk is used for storage of these workspaces when no library is selected. Libraries are stored on disks apart from the APL user's virtual machine; details are supplied in the APL/CMS Installation Manual.

APL/CMS uses the D-disk, if accessed read-write, otherwise the A-disk, for storing intermediate and utility files of type VMAPLUT. These files are used for the APL stack of auxiliary processor 101, the workspace conversion utilities, and during the APL)COPY,)PCOPY, and)SAVE commands, for example.

PRINT, PUNCH, AND TYPE COMMANDS

An APL program can create files by using the auxiliary processors (see Section 6). One of the processors can send files directly to the line printer or card punch. There are some occasions when it is easier to use the auxiliary processor to create a CMS file and then use CMS commands to type or print the file. When the CMS PRINT command is issued, as follows:

```
print fn ft fm
```

it causes the contents of the specified CMS file to be printed. The graphical representation of each character is dependent upon the printer characteristics. The VMAPLWS files contain nonprintable records; however, the auxiliary processors supplied with the system can be used to write out APL variables in printable format.

The PUNCH command operates like the PRINT command except that the contents of the records are punched into cards. The punch reproduces any character. Cards may be read into the system with the CMS READCARD command. For further details, see VM/370: Command Language Guide for General Users.

The TYPE command is similar to the PRINT command except that output appears at the terminal. If the file was produced with the APL conversion option of the auxiliary processor AP110, then the commands:

```
cp term apl on
type fn ft fm
cp term apl off
```

should be used to type the file. The first command indicates that you are going to use your APL type element for typing; the third command indicates the subsequent use of the VM/370 type element.

EDIT AND SCRIPT COMMANDS

CMS has an editor, called EDIT. For additional information about the editor, see the publication VM/370: EDIT Guide. The SCRIPT processor, an Installed User Program (IUP) available through IBM for a license fee, can be used to prepare formatted and paginated output like this document from CMS files.

FILEDEF COMMAND

This CMS command is used to simulate the functions of the OS Job Control Language Data Definition (DD) in the CMS environment. Device independence is achieved by allowing the unit specification and file characteristics to be transmitted to programs that use the OS simulation macros and functions. APL/CMS includes an auxiliary processor, AP111, which supplies sequential access via QSAM to any device specified by use of the FILEDEF command.

Note: The CMS file processor, AP110, does not use FILEDEF and does not need any DD information.

COPYFILE AND MOVEFILE COMMANDS

The COPYFILE command can be used to copy part or all of a CMS file, to combine files, to change record formats and to do various transformations on data in the file. The MOVEFILE command moves data from any device supported by VM/370 to any other device supported by VM/370. The input and output devices are defined by use of the FILEDEF command.

TIME

When you log off the system, a time message is issued as follows:

```
CONNECT hh:mm:ss VIRTCPU mmm:ss:hs  TOTCPU mmm:ss:hs
```

where:

hh	is hours
mm or mmm	is minutes
ss	is seconds
hs	is hundredths of a second

The connect time is the elapsed time since you logged on. The VIRTCPU is the virtual CPU time you have used. TOTCPU is VIRTCPU plus the CPU time that was spent in CP.

The CP command QUERY TIME yields the same result.

SAVING WORKSPACES ON MAGNETIC TAPE

TAPE Command

The TAPE command can be used to save and restore CMS files on magnetic tape. It is useful for saving infrequently used workspaces or sending workspaces to other APL/CMS system installations. To save one or more files on a tape:

1. Ask the operator for a tape.

2. Log on to your machine.

3. IPL CMS

4. Ask the operator to ready the tape and attach it as 181 to your machine by issuing the following message:

   ```
   cp msg op please attach tape PASC123 as 181 with ring in
   ```

   (This sample message is based on the assumption that the tape has "PASC123" as a label.) The phrase "ring in" tells him to put a file protect ring in the tape; if the ring is out then it is impossible to write on the tape. In general, the operator will notify you when the tape is ready.

5. To dump the workspaces ALPHA and BETA, issue the following commands:

```
tape rew
tape dump alpha vmaplws
tape dump beta  vmaplws
tape wtm
tape rew
tape scan
```

Note that each command starts with the word TAPE followed by a space. REW causes the tape to be rewound, but note that it must be spelled REW, not REWIND. TAPE DUMP writes the named CMS file onto the tape; you may repeat this command for as many files as you wish to dump. The TAPE WTM command writes a tape mark on the tape. A tape mark is conveniently used to separate groups of workspaces. If you wish to dump all workspaces, then you can use:

```
tape dump * vmaplws
```

TAPE SCAN Command

The TAPE SCAN command reads the tape, verifies the tape contents, and types out a list of the CMS files on the tape from its current position to a tape mark.

If the tape appears to be satisfactory then use the following CP commands:

```
cp detach 181
cp msg op remove ring, and save tape
```

to unload and save the tape.

To retrieve a workspace at a later date, then log on, IPL CMS, and issue the command:

```
cp msg op please attach PASC123 as 181 with ring out
```

When the tape is attached, you can then issue the commands:

```
tape rew
tape load beta vmaplws
```

to transfer the CMS file BETA VMAPLWS to your A-disk, which will overwrite an existing file of the same name. The example shows just one file being loaded. Using the command TAPE LOAD loads all the workspaces up to a tape mark. (You should detach unit 181 when you no longer need it.)

If there is insufficient space on your disk, then you may get a DISK FULL message. You can, if you wish, make some space available on the disk, rewind the tape, and try again.

SPOOL Command

This CP command has many options to control the disposition of files associated with your virtual card reader, card punch, and line printer. For example:

```
spool printer copy n
```

specifies that n copies be made of your line printer output. The command:

```
spool punch to userid
```

causes your virtual punch output to be directed to the virtual card reader of the machine identified by userid.

```
spool punch off
```

directs future output to the real punch.

DISK DUMP and DISK LOAD Commands

The CMS command, DISK, can be used to move files from disk to card format using the DUMP option, or the reverse using the LOAD option. It is generally not used to punch real cards (the TAPE command is normally used to move files from the system); rather, it is used in combination with the SPOOL command to direct files to, or receive files from, another user. For example, the commands

```
cp spool punch to johndoe
disk dump alpha vmaplws
cp spool punch off
```

transfer the APL/CMS workspace called ALPHA to the card reader of the machine, JOHNDOE. At that machine, the command

```
disk load
```

creates the file ALPHA VMAPLWS on the A-disk.

Note: Use the SPOOL command before issuing the DISK DUMP command. If the output is not spooled to another user, it is punched on real cards. If you discover you have made such a mistake, immediately send the following message:

```
cp msg operator  please flush punch output
```

USING APL\360 OR APLSV WORKSPACES

To transfer APL\360 workspaces to an APL/CMS system requires several steps. Get the APL\360 system staff to dump your workspaces onto magnetic tape. Workspaces from an APLSV system must be dumped in APL\360 compatible form using the level 0 option of the APLSV utility program. This tape is not in the format of the CMS dump tape, so your system staff must convert the tape to an APL/CMS workspace on your disk.

If you are familiar with CMS, then you may wish to do the conversion yourself. The process is described in the APL/CMS Installation Manual.

The conversion goes in two steps. Suppose the tape contains a workspace from APL\360 called ALPHA. The first step reads the tape and produces a CMS file with the name of ALPHA and filetype of APLWS. The second step accepts ALPHA APLWS as input, converts this workspace to the form used by APL/CMS and produces the file called ALPHA VMAPLWS. The same conversion procedure can be used for APLSV workspaces. APL(CMS) IUP workspaces already exist as APLWS files, have the same format as APL\360, and require only the second conversion step.

After conversion, the workspace may contain functions with the names BADHEADERn, where n goes from 1 to the number of such names. The original header of these functions contains at least one duplicate name and is stored as a comment in the first line of the function. Edit the function, correct the original header, delete the lamp and del symbols, change the line number to zero, and close the function.

The transfer of workspace information to APLSV (or other foreign APL systems with I/O capability) can be effected by writing the canonical representation of the functions in the workspace along with the variables in the workspace onto magnetic tape. Read this tape with APL programs to create a workspace on APLSV.

SENDING WORKSPACES TO OTHER APL/CMS USERS

The magnetic tape produced by the TAPE DUMP command can be used to send workspaces to other APL/CMS users. If the other user is on the same physical machine as you, then it is more efficient to do a direct disk-to-disk transfer using the APL/CMS public library. Although an installation option, you can normally)LOAD and)SAVE into libraries 1000 through 999999. Loading the workspace you wish to transfer and saving it in the public library makes it available to another user, who can drop it from the public library after the transfer.

If you get the message

DISK NOT AVAILABLE

while attempting the)SAVE in the public library, it means that the disk is temporarily unavailable for writing because some other machine is writing on it. Retry the command in a few seconds. There is one drawback to this method, namely the lack of privacy during the transfer.

A more secure transfer is via the previously described DISK command.

Once a workspace has been transferred, it can be used by entering the APL environment and using the)LOAD command. The load command adjusts the size of the workspace to the available storage. If the virtual storage on your machine is too small to accommodate the workspace, the message

WS TOO LARGE

results and loading does not take place.

If you need to accommodate a workspace larger than your virtual storage, you may return to the CP environment and issue a DEFINE STORAGE command to expand the size of your virtual storage (within limits set by your installation manager). Reload APL/CMS and try again.

The APL/CMS system includes the following auxiliary processors:

```
AP100   COMMAND
AP101   STACK INPUT
AP110   CMS DISK I/O
AP111   FILEDEF I/O
```

The COMMAND processor enables an APL/CMS user to issue CP and CMS commands. The STACK INPUT processor stores data for input at the first opportunity to APL/CMS. The CMS DISK I/O processor provides sequential and random access using the CMS file system. The FILEDEF I/O processor provides sequential access to devices supported by the OS simulation facilities of CMS available through the FILEDEF command using QSAM.

The example introduced in Section 4 is given in more detail below; refer to it as you read the following sections.

```
      S←'EXAMPLE SCRIPT(370' ⍝  SET INITIAL VALUE
      S ⍝ DISPLAY VALUE
EXAMPLE SCRIPT(370
      110 ⎕SVO'S' ⍝ SHARE THE VARIABLE
2
      S ⍝ REFERENCE AND DISPLAY VALUE
0 1 1
      S←'THIS GOES IN THE FIRST RECORD' ⍝ SET A VALUE
      S←'AND THIS GOES IN THE SECOND' ⍝ SET ANOTHER
      S ⍝ REFERENCE AND DISPLAY VALUE
THIS GOES IN THE FIRST RECORD
      S ⍝ REFERENCE AGAIN
AND THIS GOES IN THE SECOND
      ⎕SVR'S' ⍝ RETRACT VARIABLE
2
```

INITIAL VALUE

When an offer to share with an APL/CMS auxiliary processor is made, the value of the variable being offered should be a character vector specifying the argument and options (if any) required by that auxiliary processor.

The general format for all initial values is:

'argument (options'

The auxiliary processors supplied with APL/CMS assist the APL user in the transmission of data to a destination and the receipt of data from a source. The argument passed in the initial value is used to determine this source and/or destination.

Following the source or destination argument is a left parenthesis used to indicate "options follow". It should be present only when options are specified by the user. The APL/CMS auxiliary processors ignore extra left parentheses and disallow right parentheses. The interpretation of the initial value is covered later for each auxiliary processor.

OFFER PROTOCOL

The APL/CMS auxiliary processors match all shared variable offers by the user with matching counteroffers. The initial value of the shared variable is interpreted by the auxiliary processor as a user request. After inspecting the operating environment, the auxiliary processor specifies a new value for the shared variable. This value, when referenced by the user, will be a scalar 1 if the request is rejected. A new value of the variable can then be specified by the user after which the auxiliary processor (AP) repeats this procedure.

When an acceptable initial value is specified, the AP sets the shared variable to a scalar zero or a vector with the first element zero. The effect of all subsequent references and specifications of this shared variable is to move data and control information between the APL program and the auxiliary processor, as described for each processor. To specify a different argument or options for a shared variable, that variable must be retracted and reoffered with the required initial value.

OPTIONS FOR DATA CONVERSION

Data transmitted between the APL program and the auxiliary processors can be in three distinct forms:

1. APL variables, complete with size, shape, and type information. This is the most convenient and efficient form for most applications. The conversion option for this type of data is VAR.

2. Character vectors. This form is used primarily for interchange with other non-APL processors. Two conversion options, APL and 370, are described below.

3. Bit vectors (that is, zeros and ones) that provide the most general form of data transmission and interchange. The conversion option for this type of data is BIT.

Characters outside of the workspace (for example, data file records, punched cards, and printer lines) are transferrable, as characters, to and from the workspace in two ways:

1. Characters accepted by APL as input (processed as though entered from a keyboard) and produced by APL as output (as though for typing on a terminal). This option is called APL.

2. EBCDIC codes used by the System/370. This option is called 370.

You must have some knowledge of these two forms in order to transmit character data to and from external media. Some general information follows, with details in "Appendix A: Auxiliary Processor Conversion Options."

The APL script conversion option (APL) produces characters in the workspace as though the input data were entered from the keyboard; output data is created as though the characters in the workspace were typed at the terminal. For example, the character "A" in the workspace is converted to an "A", and the character "A" in the workspace is converted to the three characters: "A", "backspace", and "_".

The System/370 EBCDIC conversion option (370) provides a direct mapping between some APL characters and some EBCDIC characters. For example, the character "A" in APL converts to the "A", and the APL character "A̲" converts to "a".

INPUT/OUTPUT PROCESSING

Introduction

The APL/CMS system includes two auxiliary processors that provide file input and output capabilities. AP110, CMS DISK I/O, supports the CMS file system which allows both sequential and random accessing. AP110 creates files with fixed or variable length unblocked records. Random access of variable length records is inefficient compared to random access of fixed length records. AP110 processes blocked or unblocked fixed length records and unblocked variable length records.

AP111, FILEDEF I/O, may be used for sequential access to CMS disk files and other types of devices such as magnetic tape, card readers, line printers, and terminals. This device independence is achieved by using the OS simulation facilities in CMS. The AP111 user must issue an appropriate FILEDEF command to CMS (using AP100) before the specified data set can be opened. AP111 supports QSAM so that blocked or unblocked and fixed or variable length records can be sequentially processed.

CMS disk files with fixed length records can be processed by either processor regardless of blocking. Both I/O processors open files for reading and writing although switching between read and write is time-consuming.

Record Variables

An APL variable used to transmit data records is called a record variable. Except for the initial reference after properly specifying an initial value, all references of the record variable yield records from the file being processed. All specifications into the record variable cause records to be written into the file. The previous example used s̲ as a record variable. r.v. i.v.

Control Variables

An APL variable used to control or monitor data transmission is called a control variable. It is paired with the most recently offered, but unpaired, record variable specifying the same file or ddname. A control variable may be required to query certain status information. For AP110, it is required to achieve indexed selection of records from the file. Except for the first reference after a proper initial value has been specified, the reference of a control variable returns a scalar (AP111) or vector (AP110) whose first element indicates the status of the previous specification or reference of its paired record variable. A zero indicates successful completion. For return codes and status indicators, see "Appendix B: Auxiliary Processor Return Codes."

Record Pointers

AP110 maintains a read pointer and a write pointer that indicate the position of the record to be processed by a reference or specification of the record variable. The value of the control variable is the status indicator followed by the read and write pointers. AP110 initializes these record pointers to 1 and N+1, where N is the number of records in the file, when the file is opened. The initial reference of the record variable returns a zero followed by these pointers.

AP110 increments the read or write pointer by one after each successful read or write of the file. Record pointers can be reset by the user at any time by specifying an integer into the control variable. A scalar sets the read pointer; a two-element vector sets both pointers. An integer of less than one does not change the pointer.

Continuing our previous example:

```
        F←'EXAMPLE SCRIPT (370'
        110 ⎕SVO'F'
  2
        F
  0 1 3
        A←F
        F←Y
        F←Z
        B←F
```

Now A and B contain the first two records from EXAMPLE SCRIPT. Records 3 and 4 contain the variables Y and Z.

AP111 processes files sequentially and does not support record pointers. The FILEDEF command, as an option, sets both read and write pointers to the top of a file (the default) or to the bottom of a file (DISP MOD). This setting of the record pointers is in effect every time a file is opened for read, write, or switching between read and write. AP111 does not explicitly alter the position of a file (magnetic tape, for example) when the record variable is retracted.

End File and Error Conditions

Whenever you reference a record variable and an end of file is read, the I/O processor assigns a null vector to the variable. This is also done if a read error occurs. These cases can be differentiated by inspecting the return code available via a control variable. Notice that null variables can be written only with the VAR option.

Space Used by Auxiliary Processors

INPUT/OUTPUT BUFFERS: AP110 and AP111 need virtual storage space for input/output buffers. This space is located outside the workspace in an area whose size is fixed immediately after APL is invoked. The standard size is 8192 bytes. If the auxiliary processors are used to transmit long records or to access many files simultaneously, they may fail to find buffer space and will post an error in the control variable.

The size of the buffer space can be set by supplying an argument when APL is invoked, thus:

```
ipl apl parm x

   -- or --

ipl cms
apl x
```

where:

x — is the number of bytes (for example, 4096) or the number of kilobytes (for example, 12K) or the number of megabytes (for example, 2M).

parm — is a necessary part of the IPL command.

The amount of space allocated is that requested, rounded up to the next page boundary. If the space is not available, a)OFF HOLD is issued.

PROGRAM STORAGE: Many CMS commands require some program storage space in which to operate. The size of this area varies greatly among commands. No such space is required for CP commands.

AP100 is used to invoke CP and CMS commands. If the APL/CMS system, as distributed, is invoked by an IPL command, then about 57,000 bytes of program storage are available for CMS commands called by AP100. In general, commands which require more program storage than is available will fail because the CMS storage management system should not allocate storage used by APL/CMS to the command; no harm should befall the APL/CMS system.

This is not the case if APL/CMS has been invoked as a command under CMS. The CMS storage management system will allocate space used by APL/CMS to any command which needs it. This means that any CMS command invoked via AP100 needing program storage space will cause an abrupt termination of APL/CMS. CMS commands available in the subset mode do not require program memory. An expert should be consulted if you wish to explore this area. Details are supplied in the APL/CMS Installation Manual.

AP100--THE COMMAND PROCESSOR

Initial Value

env
CMS
CP

Description

The operating environment available to the user of APL/CMS includes the environments of CMS and CP. Commands can be processed by CMS and CP to dynamically change the characteristics of these two environments. Character vectors, or one-element arrays, when specified into the shared variable, are immediately processed by the selected environment. Notice that the CMS environment includes a command, CP, that passes the rest of the arguments to CP. Subsequent references of the shared variable yield the return code set by CP or CMS. For details on return codes, see "Appendix B: Auxiliary Processor Return Codes". CP and CMS commands are described in the *VM/370: Command Language Guide for General Users*.

The commands destined for CMS are broken into "tokens." A token is a parenthesis or a series of nonblank characters. Only the first eight characters of each token are used. For both CP and CMS, all characters are converted via the 370 option immediately prior to transmission (see "Appendix A: Auxiliary Processor Conversion Options").

Warning: Some commands may cause abrupt failure of APL/CMS and loss of the active workspace; refer to "Space Used by Auxiliary Processors". The CP command to define the size of virtual machine storage or to IPL is an example of a disasterous command under all circumstances.

Argument

env
  specifies the command environment and defaults to CMS.

Example:

The function below requests multiple copies of any printed output.

```
     ∇ COPIES N;X
[1]    X←'CP'
[2]    100 ⎕SVO'X'
[3]    X←'SPOOL PRT COPY ',⍕N
     ∇
```

100/

AP101--THE STACK INPUT PROCESSOR

Initial Value

```
stk ( cvt ord
CMS   370 END
      APL BEG
          LIFO
          FIFO

   --or--

stk
APL
```

Description

Character vectors can be stored for subsequent input at the _first_ opportunity to CMS and APL/CMS. Two areas are available. The first, in storage and used by CMS _and_ APL, is called the CMS input stack; the second, a disk file used _only_ by APL, is called the APL input stack. The CMS stack is efficient to use but limited by available storage; the APL stack is limited only by disk storage. Only character vectors (or one-element arrays) can be stored by this processor. A reference of the shared variable obtains the return code set by the last specification; zero indicates success. For other return codes, see "Appendix B: Auxiliary Processor Return Codes."

When CMS or APL/CMS issues a request for keyboard input, a value from the beginning of the stack is used and no entry is requested from the user. This entry is made as though the user backspaces to the left margin, strikes the attention, and enters the value. The CMS stack has priority over the APL stack. If the user generates an APL interrupt or if a character error is detected by APL/CMS when using stacked input, both input stack areas are flushed and the keyboard unlocks for input.

Warning: Certain values such as HT and RT cause immediate action by CMS (and are not actually stacked) when placed in the CMS stack. Refer to the section on "Immediate Commands" in the _VM/370: Command Language Guide for General Users_.

Argument and Options

stk
  specifies the input stack to be used. The default is CMS.

cvt
  is the standard option for character translation and defaults to 370. This option may be used only with the CMS stack. The APL stack is maintained in an internal code which requires no conversion.

ord
  indicates whether the processor places data at the beginning of the stack (BEG or LIFO) or at the end (END or FIFO). The default is END. Entries into the APL stack always use the END option.

Example:

This function will save the active workspace and return. The method used is to place a)SAVE CONTINUE command at the beginning of the CMS stack. The CMS stack has priority over the APL stack and the BEG option will place the command in front of anything already in that stack. Execution is suspended by setting the stop vector for a stop on statement LABEL. The stack entry is read at this point which saves the workspace. The stack is read again whereupon the branch causes execution to be resumed. The special CMS immediate commands, HT and RT, are used to prevent the normal terminal output.

```
      ∇ CHECKPOINT;S
[1]     S←'CMS (APL BEG' ⍝ USE CMS STACK, LAST IN, FIRST OUT
[2]     S←101 ⎕SVO'S' ⍝ SHARE S AND IGNORE RESULT
[3]     S←'HT' ⍝ HALT TERMINAL OUTPUT
[4]     S←'→LABEL' ⍝ THIS WILL RESUME EXECUTION
[5]     S←')SAVE CONTINUE' ⍝ FIRST OUT OF STACK
[6]     S∆CHECKPOINT←LABEL ⍝ SET STOP VECTOR
[7]    LABEL:S←'RT' ⍝ RESUME TYPING WHEN RESTARTED
      ∇
```

AP110--THE CMS DISK I/O PROCESSOR

Initial Value

```
nam ( cvt fmt
      VAR FIX
      APL
      370
      BIT

   --or--

nam ( typ
      CTL
```

Description

This processor provides sequential and random access to disk files under control of the CMS file system. The operation of this processor is described in the section on input/output file processing. The CMS disk file system is described in the VM/370: Command Language Guide for General Users.

Argument and Options

nam
specifies the name of the CMS disk file to be accessed. It has one of the forms:

```
filename
filename    filetype
filename    filetype    filemode
```

p4

The default filemode is A1. The default filetype is VMAPLcF, where c is the first character of the conversion method used.

cvt
specifies the standard option for conversion which defaults to VAR except when the user gives a filetype of VMAPLcF. In this case, the default will be the cvt option with matching first letter. For example, if the user gives VMAPL3F as the filetype, then the default conversion is 370.

Note: All combinations of filetype and conversion can be explicitly specified.

fmt
is ignored except when a new file is created. This is a variable length file unless FIX is specified. In this case, the file is fixed with a record length set to the length of the first record written into the file. Each subsequent record must have this same length or an error is reported (in the control variable).

typ
establishes a control variable for the file, if an unpaired record variable already exists for the same file. The cvt and fmt options are ignored if the CTL option is present.

AP111--THE FILEDEF I/O PROCESSOR

Initial Value

```
ddn ( cvt
      VAR
      APL
      370
      BIT

   --or--

ddn ( typ
      CTL
```

Description

This processor provides sequential access, via QSAM, to any device supported by VM/370. The device and its characteristics are specified by use of the CMS command, FILEDEF. The operation of this processor is described in the section on input/output file processing. The FILEDEF command is described in the _VM/370: Command Language Guide for General Users_.

Argument and Options

ddn
: specifies the ddname to be accessed. It must be the ddname defined by a FILEDEF command that the APL user has already issued to CMS.

cvt
: specifies the standard option for conversion and generally defaults to VAR. If the FILEDEF specifies a CMS disk file, the default conversion option is determined in the same way as with AP110.

typ
: establishes a control variable for the file, if an unpaired record variable already exists for the same ddname. The cvt option is ignored for a control variable.

EXAMPLES USING THE INPUT/OUTPUT PROCESSORS

Examples for AP110

Example 1 uses CMS disk files for a sequential update function. SUP is given the name of a CMS file as an argument. It updates "name VMAPLAF A1" using "name CHANGES A1" and creating "name WORKFIL A1" as a temporary new file. When the update is complete, SUP erases the old file and renames the new file. The files SUP processes contain personnel records that are identified by the person's social security number as the first nine characters. The changes file consists of complete replacement records or, if the record is to be deleted, merely the social security number. SUP provides a return code of 0=successful completion, 1=update performed but no file erased or renamed, or 2=nothing done.

Example 1:

```
      ∇ Z←SUP FILENAME;OLDFIL;OLDREC;OLDSEQ;CNGFIL;
              CNGREC;CNGSEQ;NEWFIL;NUMS;CMS;⎕IO
[1]     Z←2
[2]     OLDFIL←FILENAME,' VMAPLAF'
[3]     CNGFIL←FILENAME,' CHANGES (APL'
[4]     NEWFIL←FILENAME,' WORKFIL (APL'
[5]     110 ⎕SVO'OLDFIL'
[6]     110 ⎕SVO'CNGFIL'
[7]     110 ⎕SVO'NEWFIL'
[8]     →(9≠ρOLDFIL,CNGFIL,NEWFIL)/⎕IO←0
[9]     CNGREC←9↑NUMS←'0123456789'
[10]    →STRT
[11]  USEO:NEWFIL←OLDREC
[12]    OLDSEQ←10⊥NUMS⍳9↑OLDREC←OLDFIL
[13]  LOOP:→(OLDSEQ<CNGSEQ)/USEO
[14]    →(OLDSEQ>CNGSEQ)/USEC
[15]    →(0=ρOLDREC,CNGREC)/ENDF
[16]  STRT:OLDSEQ←10⊥NUMS⍳9↑OLDREC←OLDFIL
[17]  USEC:→(9=ρCNGREC)/DLET
[18]    NEWFIL←CNGREC
[19]  DLET:CNGSEQ←10⊥NUMS⍳9↑CNGREC←CNGFIL
[20]    →LOOP
[21]  ENDF:⎕SVR 3 6ρ'OLDFILCNGFILNEWFIL'
[22]    100 ⎕SVO'CMS'
[23]    →(Z←CMS)/0
[24]    CMS←'ERASE ',FILENAME,' VMAPLAF'
[25]    CMS←'RENAME ',FILENAME,' WORKFIL A1 = VMAPLAF ='
      ∇
```

Example 2 illustrates the CMS disk file random access. The FIND function is given a personnel file, such as that updated by SUP, and a social security number. It is to find the location of the corresponding record (if any) in the file. In particular, if it returns an integer (n), then the nth record has the matching social security number. If it returns a real (n.5), then the given social security number is not in the file. If it were, it would occur between records n and n+1 (0 <= n <= number of records in the file). The only other possible return value is "SHARE ERROR" which indicates that the FIND command could not establish the necessary shares with AP110.

Example 2:

```
      ∇ Z←FILE FIND SS;REC;NUM;⎕IO;BOT;TOP;ID
 [1]    REC←FILE,' VMAPLAF'
 [2]    NUM←REC,'(CTL'
 [3]    110 ⎕SVO'REC'
 [4]    110 ⎕SVO'NUM'
 [5]    →(NUM∨1↑REC)/FAIL
 [6]    Z←2*⌈2⍟¯1↑NUM
 [7]    ⎕IO←BOT←0
 [8]   DROP:TOP←Z
 [9]   LOOP:Z←NUM←0.5×BOT+TOP
 [10]   →(Z≠⌈Z)/0
 [11]   ID←10⊥'0123456789'⍳9↑REC
 [12]   →(SS<ID)/DROP
 [13]   BOT←Z
 [14]   →LOOP×SS>ID
 [15]   →0
 [16] FAIL:Z←'SHARE ERROR'
      ∇
```

Example 3 assumes that a function is suspended with a domain error. The function was using AP110 to sequentially read and process INPUT DATA and the domain error occurred because the latest record read had alphabetic characters where a numeric field was supposed to be. To locate where in the file the bad record occurs, use the following sequence, in 0-origin, to determine its record number:

Example 3:

```
          X←'INPUT DATA ( CTL'
          110 ⎕SVO'X'
      2
          BAD←¯1+X[1]
```

Examples for AP111

Example 4 uses the unit record equipment for a card-to-printer function. The program, CTOP, expects as input a series of decks stacked in the card reader as a single file. These cards contain a sequence field in the last four columns and a deck identification code in the preceding four columns. The cards are to be listed on the printer and each deck should start on a fresh page.

Example 4:

```
      ∇ CTOP;CMS;SVPRINT;SVREAD;ID;CARD
[1]     100 ⎕SVO'CMS'
[2]     CMS←'FILEDEF CTOPOUT PRINTER (RECFM VA BLKSIZE 137'
[3]     CMS←'FILEDEF CTOPIN READER (RECFM F BLKSIZE 80'
[4]     SVPRINT←'CTOPOUT ( 370'
[5]     SVREAD←'CTOPIN ( 370'
[6]     111 ⎕SVO 2 7 ⍴'SVPRINTSVREAD '
[7]     →(SVPRINT∨SVREAD)/0
[8]     ID←'~'
[9]    LOOP:→(0=⍴CARD←SVREAD)/0
[10]    →(∨/ID≠4↑¯8↑CARD)/SKIP
[11]    SVPRINT←' ',CARD
[12]    →LOOP
[13]   SKIP:ID←4↑¯8↑CARD
[14]    SVPRINT←'1',CARD
[15]    →LOOP
      ∇
```

The function shown in Example 4 would take care of many card-to-printer tasks. However, because it uses characters, the input is converted from 370 to APL and the output is converted back from APL to 370. Thus, in the example, CTOP will not print cent symbols (¢).

Example 5 is a card-to-card function that has the same conversion problem. In fact, its conversion problem is even more acute since CTOC might be used to reproduce text decks. CTOC solves the problem by using BIT conversion. As input, CTOC expects a series of decks stacked in the card reader as separate files. The first card of each deck has an identification code in columns one to four. The remainder of the deck is to be reproduced except for the last eight columns of each card. The last eight output columns are to contain the deck identification code and a sequence number.

Example 5:

```
       ∇ CTOC FILES;⎕IO;BITS;PUN;RDR;NUM;ID;CARD;SEQ;F
[1]      ⎕IO←0
[2]      BITS←(10 4ρ1),⍉2 2 2 2⊤⍳10
[3]      100 ⎕SVO'F'
[4]      F←'FILEDEF FILEO PUNCH(RECFM F BLKSIZE 80'
[5]      F←'FILEDEF FILEI READER(RECFM F BLKSIZE 80'
[6]     BEGF:PUN←'FILEO(BIT'
[7]      RDR←'FILEI(BIT'
[8]      111 ⎕SVO 2 3ρ'PUNRDR'
[9]      →(PUN∨RDR)/NUM←0
[10]     ID←32↑RDR
[11]    LOOP:→(0=ρCARD←RDR)/ENDF
[12]     SEQ←,BITS[10 10 10 10⊤NUM←NUM+10;]
[13]     PUN←(¯64↓CARD),ID,SEQ
[14]     →LOOP
[15]    ENDF:⎕SVR'PUN'
[16]     ⎕SVR'RDR'
[17]     →(0<FILES←FILES-1)/BEGF
       ∇
```

MULTIPLE ACCESSING

The preceding examples have been in terms of single variables:

1. How do you read a file?

2. How do you send commands to CP via a shared variable?

Let us consider the implications of the sharing of multiple variables with an auxiliary processor.

The auxiliary processors, COMMAND and STACK INPUT, share multiple variables with different or identical destinations. Consider the following example:

Example 6:

```
      A←B←C←'CP'
      100 ⎕SVO 3 1ρ'ABC'
2 2 2
      A←'INCORRECT REQUEST'
      B←'SPOOL 00E OFF'
      C←'Q F'
FILES:  NO RDR,  NO PRT,  NO PUN
```

The variables A, B, and C in Example 6 are independent, yet they have something in common. Not one of them can affect any of the others. If you reference A, you obtain the return code that indicates an invalid CP command. Using B and C to execute successful commands has not changed this return code. In common, they share the same destination, so their assignments are "merged"; they are all sent to CP. For COMMAND and STACK INPUT this is not significant, but for other auxiliary processors it is.

The I/O auxiliary processors accept multiple record variables; thus, they allow simultaneous access to several files. Example 1 for AP110 is an update function that reads both the old file and the changes file. The I/O auxiliary processors also accept multiple record variables with identical data sources or destinations. Access to the same file with multiple variables can be very useful, although it may be confusing. The following examples explore this situation.

If A and B are both shared variables using the card reader as their source and you reference A, B, and finally A again, then the second reference of A does not read the second card. Rather, its value is for the third card since the second card was read by the reference to B. Independent variables causing a merged effect can allow, for example, any APL function to print information without knowing whether or not some higher calling function is also printing.

Now assume that A and B are both shared with the APL/CMS DISK I/O processor and that both had the initial value "SOMEFILE". If one references A, then B, and finally A again, then the second reference of A reads the second record. The first B reference and the first A reference obtain the first record. These A and B accesses are independent.

Assume that A and B are both shared with the FILEDEF I/O processor and that both are using the same CMS disk file. If they are using different ddnames (for example, different FILEDEF commands were issued each specifying the same CMS disk file) then A and B read records independently. The APL user should not attempt multiple access while the file is being created by AP111.

The I/O processors also allow multiple control variables. Because a control variable is accepted only if there is an unpaired record variable for the corresponding file, the meaning of these multiple control variables is clear. However, one should be aware of the method used for matching control variables with record variables. The control variable will be paired with the most recently shared unpaired record variable for the corresponding file (see Example 7).

<u>Example 7</u>:

```
      ∇ PROG;DATA;NUMB
[1]     NUMB←(DATA←'FT70 FILE'),'(CTL'
[2]     110 ⎕SVO'DATA'
[3]     110 ⎕SVO'NUMB'
         .
         .
         .
```

Example 7 shows that NUMB is paired with DATA. There is no problem if some higher calling function is also reading FT70 FILE without a control variable. Now consider the data in Example 8.

Example 8:

```
        ∇ F1 COMPAR F2;R1;I1;R2;I2
    [1]   I1←(R1←F1),'(CTL'
    [2]   I2←(R2←F2),'(CTL'
    [3]   110 ⎕SVO'R1'
    [4]   110 ⎕SVO'R2'
    [5]   110 ⎕SVO'I1'
    [6]   110 ⎕SVO'I2'
              .
              .
              .
```

In most cases, the COMPAR function will work. However, if the fileid in F2 is the same as that in F1, then I1 is paired with R2 and I2 is paired with R1. To avoid malfunction, statements 4 and 5 should be reversed.

OTHER AUXILIARY PROCESSOR DETAILS

All initial values are converted using the 370 option before they are inspected, thus allowing you to refer to filenames that include 370 characters such as the $ (see "Appendix A: Auxiliary Processor Conversion Options" for details).

Options can occur in any order. If conflicting options occur (for example, 370 and BIT), then the option selected depends on the auxiliary processor. Blanks can be used freely: the initial value can use or omit leading or trailing blanks. The 'options follow' left parenthesis can occur with or without a preceeding or following blank. Any blank can be replaced by multiple blanks.

In some cases, records must be changed in length. When made longer, the process is known as padding; the elements added as a result are called pad characters. The APL/CMS auxiliary processors pad records on occasion. When this is necessary, character records are padded with blanks and bit records are padded with zeros. Records using VAR conversion are never padded. BIT records may require padding, even if they are of variable format, due to hardware limitations. On the IBM System/370, for example, all BIT records must have a length that is some multiple of eight. There is no case in which character records must be padded.

The APL/CMS system includes a workspace called 1 APFNS. This workspace contains functions that facilitate usage of the auxiliary processors. For example, one function issues appropriate FILEDEF commands and offers shared variables to AP111. For details, load this workspace and type DESCRIBE.

The CMS auxiliary processors provide conversion to and from the workspace. The details of the conversion are given below.

THE 370 CONVERSION OPTION

Many characters are common to both the APL and EBCDIC character sets. The conversion preserves most of these characters. These characters are the same in both sets:

A THROUGH Z   0 *THROUGH* 9   *SPACE*
< = > + − *
' . : ; , ? !
( ) \ | / _

These characters have different graphics:

APL:	A̲ THROUGH Z̲	~	∧	¨	α	÷	≠	∆	⊖	⌽
370:	a THROUGH z	¬	&	"	@	%	$	# −0	+0

(Note that +0 and −0 are the EBCDIC left and right braces.) For example, "A̲" is converted to "a" on output and "a" is converted to "A̲" on input.

The following conversion occurs only when going from APL to 370.

APL:	←	¯
370:	=	−

The terminal control characters backspace, horizontal tabulate, idle, line feed, and new line, are translated one for one. All other EBCDIC codes are converted to "o" when translated to APL.

All other APL characters are converted to a space when going to 370. Those with graphics are:

⌶ ⊥ ⊤ ⍝ ⍕
[ ] ⌈ ⌊ ⍀ ⌿
○ ⍉ ⍟ ∘
⊂ ⊃ ∩ ∪ ⍎
∊ ⍳ ⍴ ⍵
∨ ⍱ ⍲ ≤ ≥ ×
→ ↑ ↓ ⍋ ⍒
⌹ ⍞ ⎕ ∇ ⍫ ⍙

THE APL CONVERSION OPTION

Figure 1 shows the EBCDIC code as decimal integers, with corresponding APL graphics. This table is indicative of the conversion done by VM/370 when the APL type element is specified. The full APL character set is formed by use of the backspace (BS) terminal control code in conjunction with the other characters. For example, "$\underline{A}$" is converted to "A", backspace, "_" on output and "_", backspace, "A" is converted to $\underline{A}$ on input.

Output to and input from files are both converted by the APL/CMS system as if going to or coming from the normal APL terminal. For input, all characters not in the APL/CMS input character set below, such as invalid codes and invalid compound characters (that is, those producing a character error on keyboard entry) are converted to one unique internal APL code with no graphic which CP normally prints as a space.

For output, all characters other than the APL/CMS output character set below, are converted to the EBCDIC code 0.

The Input Character Set of APL/CMS:

The input character set comprises the following:

ABCDEFGHIJKLMNOPQRSTUVWXYZ∆
$\underline{\mathit{ABCDEFGHIJKLMNOPQRSTUVWXYZ}\Delta}$
0123456789()[]⌈⌊⍀\|/⌿_⍝⊥⌶⊤⍕
⍺∊⍳⍴⍵¯-+÷×*∨∧⍱⍲<≤=≥>≠←→↑↓⍋⍒
⍟⌽⍉⊖○∘⍎∩∪⊂⊃∇⍫~⎕⍞⌹'!.:¨;,?

and SPACE.

The Output Character Set of APL/CMS:

The output character set comprises the input character set, above, plus the terminal control characters:

BS - backspace
HT - horizontal tabulate
IL - idle
LF - line feed
NL - new line

370	APL	370	APL	370	APL	370	APL	370	APL	370	APL	370	APL	370	APL
0		32		64	*SPACE*	96	−	128		160		192		224	
1		33		65	¨	97	/	129		161		193	*A*	225	
2		34		66	¯	98	→	130		162		194	*B*	226	*S*
3		35		67	≤	99	α	131		163		195	*C*	227	*T*
4		36		68	≥	100	⌈	132		164		196	*D*	228	*U*
5		37		69	≠	101	⌊	133		165		197	*E*	229	*V*
6		38		70	∨	102	∇	134		166		198	*F*	230	*W*
7		39		71	∧	103	∆	135		167		199	*G*	231	*X*
8		40		72	÷	104	∘	136		168		200	*H*	232	*Y*
9		41		73	×	105	⎕	137		169		201	*I*	233	*Z*
10		42		74		106	[	138		170		202		234	
11		43		75	.	107	,	139		171		203		235	
12		44		76	<	108		140		172		204		236	
13		45		77	(	109	_	141		173		205		237	
14		46		78	+	110	>	142		174		206		238	
15		47		79	|	111	?	143		175		207		239	
16		48		80		112	]	144		176		208		240	0
17		49		81	ω	113	⊂	145		177		209	*J*	241	1
18		50		82	ε	114	⊃	146		178		210	*K*	242	2
19		51		83	ρ	115	∩	147		179		211	*L*	243	3
20		52		84	∼	116	∪	148		180		212	*M*	244	4
21		53		85	↑	117	⊥	149		181		213	*N*	245	5
22	(*BS*)	54		86	↓	118	⊤	150		182		214	*O*	246	6
23		55		87	⍳	119	\	151		183		215	*P*	247	7
24		56		88	○	120		152		184		216	*Q*	248	8
25		57		89	←	121		153		185		217	*R*	249	9
26		58		90		122	:	154		186		218		250	
27		59		91		123		155		187		219		251	
28		60		92	*	124		156		188		220		252	
29		61		93	)	125	'	157		189		221		253	
30		62		94	;	126	=	158		190		222		254	
31		63		95		127		159		191		223		255	

Figure 1. EBCDIC Codes (Integers) to APL/CMS Characters (Graphics) via APL Conversion Option

Section 6 indicates that the APL/CMS auxiliary processors provide return codes describing the results of a previous operation. With the COMMAND and STACK INPUT processors, you reference the shared variable to obtain the return code. The I/O processors provide return codes via a control-type shared variable.

The return codes from the COMMAND processor are generally the return code from the command previously assigned to the shared variable. Since you can issue any number of commands, including commands written by yourself, it is impossible to list all possible COMMAND return codes. A few errors are intercepted and result in return codes that have been generated by the auxiliary processor itself. If an I/O problem causes the CMS OS simulation routines to take a SYNAD exit, AP111 generates a decimal return code which, when converted to hexadecimal, has the four-byte representation N0,N1,A0,A1. The first two values are the two sense bytes and the latter two values are the two status bytes. All other return codes are generated by the auxiliary processors or result from some known CMS macro instruction. All specific return codes are given below.

Numerical Listing

¯3 Unknown CMS command.

0 No error exists.

1 Attempt to read a nonexistent file or unknown CP command.

3 Permanent read error.

5 Attempt to read a file with too many records for CMS.†

6 Attempt to write too many records in a CMS file.†

7 Attempt to read a record with invalid format or attempt to write past the end of a variable-length file. You can always write at the end of file (that is, you may append a record to the file). With a fixed-length file you may also write past this point and one or more blank records are inserted into the file; this is not possible with variable length files.

8 Attempt to read a record with incorrect record length from a file with fixed format.

10 Attempt to create a file when you already have the maximum allowed by CMS.†

12 End-of-file read or attempt to write on a read-only disk.

13 Attempt to write on a full disk.

---

†The VM/370: Command Language Guide for General Users gives these limits.

14 Attempt to write on an unformatted disk.

15 Attempt to write a record with incorrect length into a file with fixed format.

17 Attempt to write a record that is too large into a variable length file.†

19 Attempt to write in a file already containing as many data blocks as CMS will allow.†

440 Data set cannot be opened for output.

441 Data set cannot be opened for input.

442 ABEND from the CMS OS simulation routines.

443* Insufficient free storage for the CMS OS simulation routines.

444* You assigned an invalid value to a shared variable. This cannot happen with VAR conversion. The value is invalid because it is null or is an array, has the wrong type, or is too big. An error occurs, for example, if BIT conversion is being used and the value is 1 2 3. An error occurs with the APL and 370 conversion options if the value to be converted is numeric instead of character.

445 You referenced a shared variable that is reading a file using VAR conversion and the resulting record is not a valid APL variable in internal form. For example, it may not have a descriptor, the element count may not equal the times reduction of the shape vector, etc.

----

†The <u>VM/370</u>: <u>Command Language Guide for General Users</u> gives these limits.

*If these errors occur, you can restart APL/CMS with more free storage and try again. For details, see Section 6, heading "Space Used by Auxiliary Processors."

Return Codes by Processor

This information is included for the experienced VM/370 user of APL/CMS. Each auxiliary processor is listed with an indication of the origin of its return codes. Each processor can, in addition, return the 44x codes.

AP100

CP: Result Register after Diagnose.
CMS: Register 15 after SVC 202.

AP101

APL: Register 15 after FSWRITE
CMS: Register 15 after SVC 202 for the ATTN function.

AP110

Register 15 after FSREAD or FSWRITE

AP111

Sense/Status Information on SYNAD exit.
Errors 12, 15, 17

Trim Along This Line

READER'S COMMENTS

Title: APL/CMS User's Manual
Programming RPQ MF2608
Program Number 5799-ALK

Order No. SC20-1846

Please check or fill in the items; adding explanations/comments in the space provided.

Which of the following terms best describes your job?

☐ Customer Engineer	☐ Manager	☐ Programmer	☐ Systems Analyst
☐ Engineer	☐ Mathematician	☐ Sales Representative	☐ Systems Engineer
☐ Instructor	☐ Operator	☐ Student/Trainee	☐ Other (explain below)

How did you use this publication?

☐ Introductory text ☐ Reference manual ☐ Student/ ☐ Instructor text

☐ Other (explain) ____________________

Did you find the material easy to read and understand? ☐ Yes ☐ No (explain below)

Did you find the material organized for convenient use? ☐ Yes ☐ No (explain below)

Specific criticisms (explain below)

Clarifications on pages ____________________

Additions on pages ____________________

Deletions on pages ____________________

Errors on pages ____________________

Explanations and other comments:

Thank you for your cooperation. No postage necessary if mailed in the U.S.A.

SC20-1846

Trim Along This Line

YOUR COMMENTS PLEASE . . .

This manual is one of a series which serves as a reference source for systems analysts, programmers, and operators of IBM systems. Your comments on the back of this form will be carefully reviewed by the persons responsible for writing and publishing this material. All comments and suggestions become the property of IBM.

Please note: Requests for copies of publications and for assistance in utilizing your IBM system should be directed to your IBM representative or to the IBM sales office serving your locality.

FOLD FOLD

FIRST CLASS
PERMIT NO. 38
PALO ALTO, CA.

BUSINESS REPLY MAIL

NO POSTAGE STAMP NECESSARY IF MAILED IN U.S.A.

POSTAGE WILL BE PAID BY

IBM SCIENTIFIC CENTER
APL/CMS Publications
2670 Hanover Street
Palo Alto, California 94304

FOLD FOLD

APL/CMS User's Manual Programming RPQ MF2608 Printed in U.S.A. SC20-1846

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)

SC20-1846

APL/CMS User's Manual Programming RPQ MF2608 Printed in U.S.A. SC20-1846

IBM

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, New York 10604
(U.S.A. only)

IBM World Trade Corporation
821 United Nations Plaza, New York, New York 10017
(International)