APL2 Programming:

Using the Supplied Routines

Version 2 Release 2

SH21-1056-01

APL2 Programming:

Using the Supplied Routines

Version 2 Release 2

SH21-1056-01

Note!

Before using this information and the product it supports, be sure to read the general information under “Notices” on page vii.

Second Edition (March 1994)

This edition replaces and makes obsolete the previous edition, SH21-1056-0. The technical changes for this edition are summarized under “Summary of Changes,” and are indicated by a vertical bar to the left of a change.

This edition applies to Version 2 Release 2 of APL2, 5688-228, and to any subsequent releases until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

A form for reader's comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, Department J58
P.O. Box 49023
San Jose, CA, 95161-9023
United States of America

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject material in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Corporation, IBM Director of Licensing, 208 Harbor Drive, Stamford, Connecticut, United States 06904.

Programming Interface Information

This book is intended to help programmers code APL2 applications in APL2. This book documents General-Use Programming Interface and Associated Guidance Information provided by APL2.

General-use programming interfaces allow the customer to write programs that obtain the services of APL2.

Trademarks

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States or other countries or both:

AIX/6000	IBM
APL2	MVS/ESA
APL2/6000	QMF
CICS	SQL/DS
DB2	System/370
GDDM	System/390

The following terms, denoted by a double asterisk (**) in this publication, are trademarks of other companies:

Sun	Sun Microsystems, Inc.
Solaris	Sun Microsystems, Inc.

About This Book

This book, *APL2/370 Programming: Using the Supplied Routines*, describes and explains how to effectively use the workspaces and external routines distributed with IBM* APL2*.

Who Should Use This Book

This book is for all APL2 users on CMS or TSO for System/370* or System/390*.

APL2 Publications

Figure 1 lists the books in the APL2 library. This table shows the books and how they can help you with specific tasks.

Figure 1. APL2 Publications

Information	Book	Publication Number
General product	*APL2 Fact Sheet*	GH21-1090
Warranty	*APL2/370 Application Environment Licensed Program Specifications*	GH21-1063
	APL2/370 Licensed Program Specifications	GH21-1070
	APL2 for AIX/6000 Licensed Program Specifications	GC23-3058
	APL2 for Sun Solaris Licensed Program Specifications	GC26-3359
Introductory language material	*APL2 Programming: An Introduction to APL2*	SH21-1073
Common reference material	*APL2 Programming: Language Reference*	SH21-1061
	APL2 Reference Summary	SX26-3999
System interface	*APL2/370 Programming: System Services Reference*	SH21-1056
	APL2/370 Programming: Using the Supplied Routines	SH21-1054
	APL2/370 Programming: Processor Interface Reference	SH21-1058
	APL2 for OS/2: User's Guide	SH21-1091
	APL2 for Sun Solaris: User's Guide	SH21-1092
	APL2 for AIX/6000: User's Guide	SC23-3051
	APL2 GRAPHPAK: User's Guide and Reference	SH21-1074
	APL2 Programming: Using Structured Query Language	SH21-1057
	APL2 Migration Guide	SH21-1069
Mainframe system programming	*APL2/370 Installation and Customization under CMS*	SH21-1062
	APL2/370 Installation and Customization under TSO	SH21-1055
	APL2/370 Messages and Codes	SH21-1059
	APL2/370 Diagnosis Guide	LY27-9601

For the titles and order numbers of other related publications, see the "Bibliography" on page 192.

Conventions Used in This Library

This section discusses the conventions used in this library.

lower Lowercase italicized words in syntax represent values you must provide.

UPPER In syntax blocks, uppercase words in an APL character set represent keywords that you must enter exactly as shown.

[ ] Usually, brackets are used to delimit optional portions of syntax; however, where APL2 function editor commands or fragments of code are shown, brackets are part of the syntax.

[*A* | *B* | *C***]** A list of options separated by | and enclosed in brackets indicates that you can select one of the listed options. Here, for example, you could specify either *A*, *B*, *C*, or none of the options.

{*A* | *B* | *C***}** Braces enclose a list of options (separated by |), one of which you must select. Here, for example, you would specify either *A*, *B*, or *C*.

... An ellipsis indicates that the preceding syntactic item can be repeated.

{}... An ellipsis following syntax that is enclosed in braces indicates that the enclosed syntactic item can be repeated.

The term *workstation* refers to all platforms where APL2 is implemented except those based on System/370 and System/390 architecture.

Throughout this book, the following product names apply:

Product Name	Platform
APL2/2	OS/2*
APL2 for Sun Solaris	Sun** Solaris**
APL2/6000*	AIX/6000*
APL2/370	MVS or VM
APL2/PC	DOS

Summary of Changes

Product

APL2/370, Version 2 Release 2

Date of Publication: March 1994

Form of Publication: Revision, SH21-1056-01

Document Changes

- Updated the SUPPLIED workspace example
- Added DISPLAYC information to the DISPLAY workspace
- Updated the EXAMPLES workspace
- Added fast fourier transfer to the MATHFNS workspace
- Updated the introduction for the UTILITIES workspace
- Added GPSVP to the UTILITIES workspace
- Updated external routines table
- Added DISPLAYC, EDITORX, EDITOR2, IN, OUT, and PIN

Part 1. Workspaces

Chapter 1. Introduction

The *workspace* is the common organizational unit in an APL system. It is a place in which to store programs and data. Part 1 of this manual describes a set of workspaces supplied with APL2. These workspaces provide ready-made utilities and common building blocks for your use. Some workspaces also illustrate various APL2 techniques.

The workspaces vary greatly in complexity, importance, objective, and age. Some workspaces go back to the earliest APL systems and others use current licensed programs, for example, the Graphical Data Display Manager (GDDM*). Some workspaces have restricted objectives. For example, the *TRANSFER* workspace is intended to assist in migrating VS APL and APL2 IUP workspaces to APL2. Still others are merely starter sets, providing APL functions for possible use and as examples to improve upon. Some workspaces have their own manuals that either describe them (for example, *APL2 GRAPHPAK: User's Guide and Reference*) or provide a great deal of information with which you must be familiar if you are to use them effectively (for example, the workspaces that use GDDM). As a result, the treatment of each workspace is individual.

Workspace Libraries

The workspaces discussed in this manual are usually stored in libraries available for general use, for example, *public* libraries. The recommended libraries are:

- Library 1 for general purpose workspaces
- Library 2 for workspaces that aid in the use of auxiliary processors

If the recommended library numbers are used, the workspaces discussed in this manual can be found in the following libraries:

```
    )LIB 1
DISPLAY  EXAMPLES MATHFNS  MEDIT    SUPPLIED UTILITY  WSINFO
    )LIB 2
APLDATA  CHARTX   CMS      FILESERV FSC126   FSM      GDMX
GRAPHPAK PRINTWS  SQL      TRANSFER TSO      VAPLFILE VSAMDATA
```

Note: Not all workspaces are provided for all environments.

If your organization uses other public library numbers, you must find out what they are from your system administrator.

How To Use Library Workspaces

To use all of a workspace stored in your own library or a public library, load it with the system command:

)*LOAD* [*library number*] *wsname*

To use part of a workspace, or to combine part of it with your active workspace, copy the part of the workspace you want using one of the system commands:

)*COPY* [*library number*] *wsname obj1 obj2...*
or
)*PCOPY* [*library number*] *wsname obj1 obj2...*

Use these commands to copy a single object (a function, a variable, an operator), a set of objects, or an entire workspace. For example, the *PRINTWS* workspace can be copied into the active workspace to be printed, or you can load *PRINTWS* as the active workspace and copy the objects to be printed into it.

Note: Groups are not treated the same in APL2 as in VS APL. In APL2, the group is not a special kind of object, requiring special system commands for its manipulation; it is just a character matrix of names of objects.

To copy the objects named in the matrix, rather than the matrix itself, put the name of the matrix in parentheses after the)*COPY* command.

For example, to copy the matrix *GPSTRIP* from the *UTILITY* workspace, type:

```
)COPY 1 UTILITY GPSTRIP
```

But to copy the objects named in the matrix *GPSTRIP*, type:

```
)COPY 1 UTILITY (GPSTRIP)
```

Note: If the workspace or group you copy has objects that have the same names as objects in the active workspace, the copied objects replace the active objects. When you don't want this to happen, use)*PCOPY* (Protected Copy) instead of)*COPY*. Similarly named objects then are *not* copied. When objects are not copied, a *NOT COPIED* message listing their names is transmitted to your terminal.

Most of the workspaces described in this manual contain defined groups of functions serving specialized needs. These functions are described in each workspace section of the book. To conserve space (and to minimize processor usage), use only the groups or functions you need instead of the entire workspace.

The Message Facility

All of the supplied workspaces except *CHARTX*, *DISPLAY*, *EXAMPLES*, *GDMX*, *MATHFNS*, *SUPPLIED*, *UTILITY*, and *WSINFO* return their own error messages. These messages are stored in a table whose name begins with *AP2W* (for example *AP2WSQL*).

The entries in this table contain the message number and the message. Messages are printed using the *AP2WSM* function. For example:

```
'AP2WAPLDATA' AP2WSM 106 'ZENO'
```

prints message number 106, as shown in Figure 2 on page 6.

Figure 2 shows complete example of how the message facility works in *APLDATA*, a supplied workspace.

'*ZENO*' is a token that is substituted for all occurrences of '⍵' in the message. Any text in the message occurring after a '⍺' is not displayed. However, the message ID *AP2WAPLDATA* and the additional information is assigned to the variable *MORE*.

Note: When using the message facility, you must be careful to avoid name conflicts, particularly with the variable *MORE*.

```
      )LOAD 2 APLDATA

      AP2WAPLDATA
101 THE VARIABLE NAMED ω DOES NOT EXIST
102 VARIABLE fn HAS NOT BEEN SETαω REQUIRES THE FILENAME IN fn
103 END OF FILE - FILE CLOSEDαNO ATTEMPT MADE TO READ PAST EMPTY RECORD
104 LEFT ARGUMENT MUST BE SPECIFIEDα MUST BE ω
106 THE VARIABLE NAME 'ω' IS THE SAME AS A LOCAL NAMEαCHOOSE ANOTHER NAME

      MORE
NO MORE MESSAGES

      ⍝ NO ERRORS GENERATED  ...YET
      ⍝ LET'S GENERATE ERROR NO. 106 WITH A TYPICAL TOKEN

      'AP2WAPLDATA' AP2WSM 106 'ZENO'
 THE VARIABLE NAME ' ZENO ' IS THE SAME AS A LOCAL NAME

      MORE
AP2WAPLDATA106 CHOOSE ANOTHER NAME
```

Figure 2. An Example of the Message Facility

Note: Do not use the message facility to emulate IBM messages in your own applications. This makes it difficult to tell who designed (and therefore responsible for documenting) a given error message.

However, you can use *AP2WSM* for your own purposes. To do this, copy the group *GPMESSAGE* (which exists in all workspaces that use the message facility) into your active workspace. Then replace the error message tables with ones of your own. When you call the function *AP2WSM*, use two strings in the left argument, a message ID prefix followed by a table name. For example:

```
'MYAPPLICATION' 'MYTABLE' AP2WSM 42 'THE ANSWER'
```

If *'MYTABLE'* is provided only as a left argument, then it is both the message ID prefix and the table name.

Documentation within the Workspace

All supplied workspaces contain three functions: *ABSTRACT*, *DESCRIBE*, and *HOW*. Some workspaces also contain other functions whose names begin with *HOW*.

- *ABSTRACT* gives a brief description of the purpose of the workspace.
- *DESCRIBE* gives a more detailed description of the workspace.
- *HOW* tells you how to use the main functions in the workspace.

These functions also point you to any additional documentation in the workspace. They use the external function *HELP* to extract help text from the APL2 product help files.

Workspaces with Interrupted Functions

You should check the state indicator before storing a workspace in which you have been testing programs or in which you have had difficulties with programs. For example:

```
)SI
```

might cause something like the following to be displayed on your terminal:

```
FOO[6]
GOO[5]
ZOO[3]
*
FOO[4]
GOO[5]
ZOO[3]
*
```

Such a display suggests that you tried to use a program named *FOO* and that you had problems. The first time you tried, processing stopped on statement 4. The second time, you got as far as statement 6. In both cases, you were left with *FOO* suspended.

Also, in both cases, the processing of *FOO* was requested by statement 5 in *GOO*; and the processing of *GOO*, in turn, was requested by statement 3 in *ZOO*.

Thus, in both cases, *ZOO* and *GOO* are *pendent* or *suspended* on the successful processing of the suspended function, *FOO*.

Remember that each of the entries discussed here represents a drain on the work area available in your workspace, since each is a copy of a function and, possibly, of associated local variables. This can lead to puzzling difficulties when you try to query variable values or edit the suspended functions.

It is good practice to check the state indicator when your work has caused many function suspensions. You can terminate a suspended function (and the functions hanging on it) by using the right-pointing arrow (→). This also frees any resources used by the suspended function. To terminate *all* the suspended functions, you should type the system command)*RESET* or)*SIC*.

Interrupting and Debugging

Suspending a function can be intentional. Stopping one or more functions at one or more points can be extremely helpful when they're not working and you want to know why. Remember this key fact:

> Once a function has stopped, all the resources of APL are available to you to help determine why it stopped, correct it, and in some cases, resume processing.

You can examine the values of key variables, you can run specially written analysis programs, you can alter values and restart; the possibilities open to you are many. For more information about suspending and tracing functions, see *S*Δ, *T*Δ,)*SINL*, and)*SIS* in *APL2 Programming: Language Reference*.

The Auxiliary Processor Workspaces

The auxiliary processor workspaces include:

- Environment-dependent workspaces: *CMS*, *TSO*, *FILESERV*
- Screen manager workspaces: *GDMX*, *GRAPHPAK*, *FSC*126, *FSM*, *CHARTX*
- File auxiliary processor workspaces: *APLDATA*, *VAPLFILE*, *VSAMDATA*
- *PRINTWS* workspace
- *SQL* workspace

In these workspaces, the tasks of sharing variables, transmitting initializing information, checking return codes, and performing the other housekeeping chores required to use auxiliary processors are done by what are commonly called *cover* functions. These functions are described in the chapters about individual workspaces. Figure 3 lists the APL2 auxiliary processors and their associated workspaces.

Figure 3 (Page 1 of 2). APL2 Auxiliary Processors and Associated Workspaces

AP No.	Description	Associated Workspaces
AP 100	Subsystem command auxiliary processor. Processes CP/CMS or TSO commands during an APL2 session.	*CMS* and *TSO*
AP 101	Alternate-input (stack) auxiliary processor. Stacks input to the APL2 system, replacing manual entry of input.	*CMS* and *TSO*
AP 102	Main storage access auxiliary processor. Returns the contents of specified areas of virtual (CMS) or main (TSO) storage.	*CMS* and *TSO*
AP 110	CMS file auxiliary processor. Reads or writes sequentially or randomly to a disk under control of the CMS file system.	*CMS*
AP 111	QSAM auxiliary processor. Reads or writes to a device or file supported by QSAM (TSO) at QSAM simulation (CMS).	*CMS* and *TSO*
AP 119	TCP/IP socket interface auxiliary processor.	None
AP 120	Session manager command auxiliary processor. Processes a session manager command.	*FSM*
AP 121	APL2 file auxiliary processor. Reads or writes APL2 arrays (in internal form) to and from a direct or sequential file.	*APLDATA* and *VAPLFILE*
AP 123	VSAM auxiliary processor. Performs file operations on entry-sequenced, key-sequenced, or relative-record VSAM files.	*VSAMDATA*
AP 124	Full-screen auxiliary processor. Provides full-screen text capabilities.	None
AP 126	GDDM auxiliary processor. Transfers AP 126 service requests and GDDM call requests to GDDM for control of an IBM 3270-family display station with programmable symbol set (PSS) feature.	*CHARTX*, *FSM*, *FSC*126, *GDMX*, and *GRAPHPAK*
AP 127	SQL auxiliary processor. Passes SQL statements from APL2 to SQL/DS* and DB2*.	*SQL*

Figure 3 (Page 2 of 2). APL2 Auxiliary Processors and Associated Workspaces

AP No.	Description	Associated Workspaces
AP 210	TSO BDAM auxiliary processor. Provides relative record access to fixed-length, unkeyed disk data sets through BDAM.	*TSO*
AP 211	APL2 object file auxiliary processor. Reads or writes APL2 arrays (in internal form) to and from a file. Allows access to the arrays by name.	None

Chapter 2. Information Workspaces

This chapter describes the information workspaces WSINFO and SUPPLIED.

WSINFO: Information About the Library Workspaces

There is one principle function in this workspace:

```
LIST
```

This function provides a list of each workspace distributed with the APL2 system for use in your particular environment (CMS or TSO). *LIST* prompts you to enter the name of a workspace, then returns descriptive information about the workspace.

This is an example for TSO. Results could differ slightly for CMS.

```
      LIST

 Tutorial information is available for the following workspaces:
 You can enter ? at any time to redisplay the list.

  Public Library Number  1

      DISPLAY  EXAMPLES MATHFNS  MEDIT    SUPPLIED UTILITY  WSINFO

  Public Library Number  2

      APLDATA  CHARTX   FILESERV FSC126   FSM      GRAPHPAK GDMX
      PRINTWS  SQL      TRANSFER TSO      VAPLFILE VSAMDATA

 Enter a workspace name, or press ENTER to exit:
```

SUPPLIED: Information About External Functions

APL2 has a wide variety of external routines that can be accessed using ⎕*NA*.

The *SUPPLIED* workspace contains associations to all the APL2 external routines. In addition, the *SUPPLIED* workspace contains a function that can help you learn how to use the external routines. See Chapter 11, “External Routines” on page 129 for detailed explanations of each routine.

```
LIST
```

The *LIST* function lists APL2's external routines and prompts the user to enter a function name. When a function name is entered, *LIST* displays tutorial information that describes the purpose, syntax, arguments, and results for the function.

Example:

```
      LIST

 Tutorial information is available for the following functions:
 You can enter ? at any time to redisplay the list.

      ΔEXEC    ΔF       ΔFM      ΔFV      APL2PI   APL2PIE  ATP
      ATR      ATTN     BUILDRD  BUILDRL  CAN      CSRIDAC  CSRREFR
      CSRSAVE  CSRSCOT  CSRVIEW  CTK      CTN      DAN      DFMT
      DISPLAY  DISPLAYC DISPLAYG DSQCIA   EDITORX  EDITOR2  EXP
      FED      HELP     IDIOMS   IN       KTC      MSG      OPTION
      OUT      PACKAGE  PBS      PFA      PIN      PTA      QNS
      RAPL2    RTA      SAN      SERVER   SVI      TIME     TSOIVP

 Enter a function name, or press ENTER to exit:
```

Chapter 3. General Purpose Workspaces

The general purpose workspaces are:

DISPLAY
EXAMPLES
MATHFNS
UTILITY

None of these workspaces use an auxiliary processor.

DISPLAY contains functions for showing the structure of arrays.

The *EXAMPLES* workspace contains many short functions that are suitable for study and experimentation by APL2 beginners. They reflect programming practices of varying quality. If you study them with a critical attitude, you might find the exercise useful in developing good APL2 programming judgment.

The *MATHFNS* workspace contains two advanced mathematical functions and two functions for formatting complex numbers in polar form.

The *UTILITY* workspace functions are for general use. These functions are nonsuspendable. (For more information about setting the processing properties of functions and operators, see *APL2 Programming: Language Reference*.)

By convention, each unlocked defined function and operator in the *UTILITY* and *EXAMPLES* workspaces contains a description of itself in the first line.

The DISPLAY Workspace

This workspace contains *DISPLAY*, *DISPLAYC*, and *DISPLAYG*; these functions are useful in showing the structure of nested and mixed arrays.

```
Z←DISPLAY X
Z←DISPLAYC X
Z←DISPLAYG X
```

Z is a character matrix that represents the array *X*.

DISPLAY and *DISPLAYC* use characters that display on all implementations. *DISPLAYC* is identical to *DISPLAY* and is included for compatibility with the *DISPLAY* workspace distributed with the workstation APL2 implementations. *DISPLAYG* uses box characters.

The following characters are used to convey shape information:

→ or ↓	Indicates a dimension of at least one
⊖ or ϕ	Indicates an axis of length zero. If an array is empty, its *prototype* is displayed
(None of the above)	Indicates no dimension (a rank 0 array)

The following characters are used to convey type information:

~	Indicates numeric
+	Indicates mixed
∈	Indicates nested
–	Indicates a scalar character that is at the same depth as nonscalar arrays
(None of the above)	Indicates a character array that is not a simple scalar

```
      X←⍳5
      DISPLAY X
┌→────────┐
│1 2 3 4 5│
└~────────┘
      X←⊂⍳5
      DISPLAY X
┌─────────────┐
│ ┌→────────┐ │
│ │1 2 3 4 5│ │
│ └~────────┘ │
└∊────────────┘
      X←(⊂⊂⍳4)(2 2⍴'ABCD')(2 2⍴'FORTY-TWO' 'IS' 'THE' 'ANSWER')
      DISPLAY X
┌→──────────────────────────────────────────────┐
│ ┌───────────────┐ ┌→┐ ┌→──────────────────┐   │
│ │ ┌───────────┐ │ ↓AB│ ↓ ┌→────────┐ ┌→┐ │   │
│ │ │ ┌→──────┐ │ │ │CD│ │ │FORTY-TWO│ │IS│ │   │
│ │ │ │1 2 3 4│ │ │ │'─'│ │ '─────────' '─' │   │
│ │ │ '~──────' │ │ │   │ ┌→┐       ┌→────┐ │   │
│ │ '∊──────────' │ │   │ │THE│     │ANSWER│ │   │
│ '∊──────────────' │   │ '───'     '──────' │   │
│                   '∊──────────────────────' │
'∊──────────────────────────────────────────────'
      ⍴X
3
      ⍴⍤1 X
   2 2  2 2
      DISPLAY ⍴⍤1 X
┌→──────────────┐
│ ┌⊖┐ ┌→──┐ ┌→──┐ │
│ │0│ │2 2│ │2 2│ │
│ '~' '~──' '~──' │
└∊──────────────┘
```

Figure 4. DISPLAY Examples

The EXAMPLES Workspace

This section describes the EXAMPLES workspace.

Introduction

The functions in this workspace are examples of ways to use APL2 in solving problems. The functions are brief, often no more than one or two statements, but they illustrate some of the ways in which APL2, with relatively few statements, can do calculations that require many more statements in other programming languages. These functions are not necessarily the best way, or the only way, to solve the problem. Rather, they illustrate ways to use APL2 that are not always obvious. We encourage you to examine the listings of all functions and operators in the workspace. Some of them are *very* simple.

The examples fall into three categories: scientific, miscellaneous, and special examples of the new capabilities of APL2. There are also a few of interest to programmers, such as decimal-hexadecimal conversions and hexadecimal arithmetic.

Mathematic and Scientific Functions

ASSOC	Tests associativity of putative arithmetic tables
BIN	Binomial coefficients
COMB FC LFC	Combinations
GCD	Greatest common divisor
HILB	Hilbert matrix
PALL PER PERM	Permutations
PO POL POLY POLYB	Polynomials
TRUTH	Truth tables
ZERO	Roots of a function

Figure 5. Simple Scientific and Mathematical Functions

```
Z←ASSOC M     ⍝ ASSOCiativity
```

The function *ASSOC* tests any putative group multiplication table *M* (assuming group elements in `⍳⍴⍴`*M*) for associativity and yields a value `1` if it is associative, `0` otherwise.

```
      MULTTABLE←5 5⍴(6⍴1),(4⍴2),(⍳3),(2⍴3),(⍳4),4,(⍳5)

      MULTTABLE
1 1 1 1 1
1 2 2 2 2
1 2 3 3 3
1 2 3 4 4
1 2 3 4 5

      ASSOC MULTTABLE
1

      MULTTABLE[3;3]←1

      MULTTABLE
1 1 1 1 1
1 2 2 2 2
1 2 1 3 3
1 2 3 4 4
1 2 3 4 5

      ASSOC MULTTABLE
0
```

```
Z←BIN N      ⍝ BINomial
```

The function *BIN* produces all binomial coefficients up to order *N*.

```
      BIN 7
1 0  0  0  0  0 0 0
1 1  0  0  0  0 0 0
1 2  1  0  0  0 0 0
1 3  3  1  0  0 0 0
1 4  6  4  1  0 0 0
1 5 10 10  5  1 0 0
1 6 15 20 15  6 1 0
1 7 21 35 35 21 7 1
```

```
Z←COMB N
Z←FC N
Z←LFC N
```

The function *COMB* employs recursive definition to produce a $2\times N$ by 2 matrix of all possible pairs of elements from ιN.

```
      COMB 5
1 2
1 3
2 3
1 4
2 4
3 4
1 5
2 5
3 5
4 5
```

The function *FC* shows an alternate method that yields the same pairs but in a different order.

```
      FC 5

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

The function *LFC* employs *FC* to generate letter pairs.

```
      LFC 5

AB
AC
AD
AE
BC
BD
BE
CD
CE
DE
```

```
Z←L GCD R      ⍝ Greatest Common Divisor
```

The function *GCD* employs the Euclidean algorithm to produce the greatest common divisor.

```
      30 GCD 40
10

      GCD/ 30 40
10

      GCD/ 30 40 45
5

      GCD/ 30 40 39 45
1

      GCD/ 30 42 39 45
3
```

```
Z←HILB N      ⍝ HILBert matrix
```

The function *HILB* produces a Hilbert matrix of order *N*.

```
Z← PALL N
Z← PER N
Z← B PERM N
```

The function *PALL* produces the matrix of all permutations of order *N*. Its subfunction *PERM* produces the *B*-th permutation of order *N* by a method due to L.J. Woodrum.

The function *PER* employs recursive definition. It produces all permutations by a method much faster than that used in the function *PALL*. The permutations are not produced in the same order as those produced by *PALL*.

```
      PALL 3
1 2 3
1 3 2
2 1 3
2 3 1
3 1 2
3 2 1

      PER 3
1 3 2
2 3 1
1 2 3
2 1 3
3 2 1
3 1 2
```

```
Z← C POLY  X     ⍝ Scalar right argument only
Z← C POL   X     ⍝ Scalar right argument only
                 ⍝ (uses inner product)
Z← C POLYB X     ⍝ Scalar right argument only
                 ⍝ (uses base value)
Z← C PO    X     ⍝ Scalar or vector right argument
```

The functions *POLY*, *POL*, *PO*, and *POLYB* each evaluate a polynomial (or polynomials), whose coefficients are determined by the left argument, and whose point (or points) of evaluation is determined by the right argument. The coefficients are in ascending order of associated powers.

```
      ¯1 0 1 PO ¯2 ¯1 0 1 2
3 0 ¯1 0 3

      ¯1 0 1 POL 2
3

      ¯1 0 1 POLY 1
0

      ¯1 0 1 POLYB ¯1
0
```

To find the zeros of polynomials, see the *POLYZ* function in the *MATHFNS* workspace, shown in Figure 9 on page 31.

```
Z←TRUTH N
```

The function *TRUTH* produces the matrix of arguments of the truth table for *N* logical variables.

```
      TRUTH 3
0 0 0
0 0 1
0 1 0
0 1 1
1 0 0
1 0 1
1 1 0
1 1 1
```

```
Z←TOL (F ZERO) R
```

The operator *ZERO* employs the bisection method to determine, within a tolerance *TOL*, a root of the function *F* lying between the bounds *R*[1] and *R*[2]. *F*(*R*[1]) and *F*(*R*[2]) must be opposite signs. *ZERO* should only be applied to continuous functions.

```
      ⎕FX'Z←SIN X' 'Z←1○X'
SIN

      .1 SIN ZERO ¯1 1
0

      .1 SIN ZERO 1 4
3.0625

      .001 SIN ZERO 1 4
3.141601563
```

Miscellaneous Utility Functions

PACK	Illustrates the use of Base value (*L* ⊥ *R*)
UNPACK	Illustrates the use of Representation (*L* ⊤ *R*)
DEC2HEX	Converts from decimal to hexadecimal
HEX2DEC	Converts from hexadecimal to decimal
HEX	Performs hexadecimal arithmetic
SORTLIST	Sorts according to a collating sequence
TIME	Provides processor time used

Figure 6. EXAMPLES: Miscellaneous Utility Functions

```
Z←PACK X
Z←UNPACK X
```

The functions *PACK* and *UNPACK* illustrate the use of the ⊤ and ⊥ functions in changing from a four-number encoding of serial number (1 to 9999), month, day, and year to a single-number encoding of the same data.

```
      PACK 117 1 1 84
4315283
      UNPACK 4315283
117 1 1 84
```

```
Z←DEC2HEX R
Z←HEX2DEC R
Z←L(F HEX) R
```

The functions *DEC2HEX* and *HEX2DEC* work with nonnegative hexadecimal numbers represented as strings of characters selected from '0123456789*ABCDEF*'. The *HEX* operator performs an arithmetic function *F* on hexadecimal arguments, returning a (character) hexadecimal result. The arguments presented to a function derived by the *HEX* operator must have a depth no greater than two.

DEC2HEX	Converts decimal to hexadecimal
HEX2DEC	Converts hexadecimal to decimal
+ *HEX*	Performs hexadecimal addition

```
- HEX      Performs hexadecimal subtraction
...        and so on
      'FF' +HEX '1'
100
      (⍳HEX '5')∘.×HEX(⍳HEX 'C')
1  2  3  4   5   6   7   8   9   A   B   C
2  4  6  8   A   C   E  10  12  14  16  18
3  6  9  C   F  12  15  18  1B  1E  21  24
4  8  C 10  14  18  1C  20  24  28  2C  30
5  A  F 14  19  1E  23  28  2D  32  37  3C
```

```
Z←SORTLIST R
```

R is a character matrix. *Z* is *R* with its rows sorted according to the collating sequence defined in *DCS*, a global variable.

```
A←TIME
```

The function *TIME* yields the amount (in minutes, seconds, and milliseconds) of processor time used since the last time the function was run. It is useful in measuring the processing times of other functions. The global variable *TIMER* is assigned the value of the cumulative processor time each instance the function *TIME* is run.

Special Functions and Operators of APL2

The group *GPAPL2* contains various functions and operators designed to show some of the capabilities of APL2 that are not available in VS APL.

Workspace Information Functions

EXAMPLE	Demonstrates a specified program in *GPAPL2*
EXAMPLES	Demonstrates the programs in *GPAPL2*

Miscellaneous Functions

EXPAND	Function version of \
IOTAU	Find index (iota underbar in the IUP)
REP	Represents an array, function, or operator
REPLICATE	Function version of /
TYPE	Returns the type of an array
UNIQUE	Removes duplicates

Operators to Conform Arguments

CR	Conforms ranks
PAD	Conforms axes by overtake
TRUNC	Conforms axes by undertake

Operators for Debugging

TRACE	Traces function processing
TRAP	Traps error, returns error message

Operators to Handle Depth

EL	Each left
ER	Each right
PL	Pervasive on left
PR	Pervasive on right

Operators for Program Control

ELSE	Conditional processing
IF	Conditional processing

Miscellaneous Operators

AND	Applies two functions
COMMUTE	Reverses function arguments
FAROUT	All-level outer product
NOP	No operation
POWER	Applies a function monadically N times

Figure 7. GPAPL2 Main Functions and Operators

```
EXAMPLE R
```

This function processes the examples found in the leading comments of the program named in *R*.

```
EXAMPLES
```

This function processes the examples found in the leading comments of all of the programs in the workspace.

```
Z←L IOTAU R       ⍀ IOTA Underbar
```

This is the Find Index function from the APL2 Installed User Program. *R* and *L* can be any array. *Z* is an integer matrix that contains the starting positions (in row major order) where pattern *R* begins in the array *L*.

```
      ⍴'A' IOTAU 'A'
0 1
      'ABABABA' IOTAU 'AB'
1
3
5
      1 (2 3) (4 5) 2 3 4 5 IOTAU 2 3
4
      L←4 5⍴'ABCABA'
      L
ABCAB
AABCA
BAABC
ABAAB
      L IOTAU 'BA'
3 1
4 2
      L IOTAU 2 1⍴'BA'
1 2
1 5
2 3
3 1
3 4
```

```
Z←L REPLICATE R
Z←L EXPAND R
```

These functions are identical to the primitive functions Replicate and Expand, respectively represented by '/' and '\', except that the primitive versions are really operators, so you cannot apply operators to them.

The defined *REPLICATE* and *EXPAND* really *are* functions, so you *can* apply operators to them.

```
      (1 0 1)(0 3)REPLICATE¨'ABC' 'DE'
AC EEE

      REPLICATE/5 '*'
*****

      (1 0 1)(0 1 0) EXPAND¨ (2 4) 6
2 0 4  0 6 0
```

```
Z←REP R     ⍝ REPresentation
```

Z is a representation of the array, function, or operator named in *R*. Specifically, *Z* is ⍎*R* or ⎕*CR R*, whichever is appropriate. This is an example of the use of the *ELSE* operator in this group.

```
Z←TYPE R
```

Z is a scalar zero if *R* is numeric, and a scalar blank if it is character. This function is compatible with a VS APL library function of the same name. It is not meant to be applied to mixed or nested arguments.

```
Z←UNIQUE R
```

R is a vector. *Z* is a vector containing the elements of *R* with duplicates eliminated.

```
      UNIQUE 'THE ANTS WERE HERE'
THE ANSWR
      UNIQUE 'GUFFAW 17 (⍳4) 'GUFFAW'
GUFFAW 17  1 2 3 4
```

```
Z←L (F CR) R        ⍝ Conform Ranks
Z←L (F PAD) R
Z←L (F TRUNC) R     ⍝ TRUNCate
```

The *CR* operator conforms the ranks of *L* and *R* and then applies the function *F*. The *PAD* operator conforms the axes of *L* and *R* by overtake. The *TRUNC* operator conforms the axes of *L* and *R* by undertake.

```
      (4 4⍴'WE  THEYUS  OURS') ∧.(=PAD) ⍉2 3⍴'WE OUR'
1 0
0 0
0 0
0 0


      (4 4⍴'WE  THEYUS  OURS') ∧.(=TRUNC) ⍉2 3⍴'WE OUR'
1 0
0 0
0 0
0 1


      (2 3 4⍴⍳24) +PAD CR 5 6⍴100×⍳30
 101  202  303  404  500  600
 705  806  907 1008 1100 1200
1309 1410 1511 1612 1700 1800
1900 2000 2100 2200 2300 2400
2500 2600 2700 2800 2900 3000

  13   14   15   16    0    0
  17   18   19   20    0    0
  21   22   23   24    0    0
   0    0    0    0    0    0
   0    0    0    0    0    0
```

Z←L (F TRACE) R	*Z←(F TRACE) R*

TRACE traces the processing of *F*. It is most useful when the derived function is passed to another operator. Every time *F* is called, the derived function displays its argument(s) and the result.

```
      +TRACE\ 1 4 9        ←─── Expression as entered
1 4                        ←───TRACE output
5                                 .
4 9                               .
13                                .
1 13                              .
14                         ←───TRACE output
1 5 14                     ←─── Final result

      2 +TRACE/ 1 2 3 4    ←─── Expression as entered
1 2                        ←───TRACE output
3                                 .
2 3                               .
5                                 .
3 4                               .
7                          ←───TRACE output
3 5 7                      ←─── Final result
```

```
Z←L (F TRAP) R                                Z←(F TRAP) R
```

The derived function (*F TRAP*) is just like *F*, except that if an error occurs during the processing of *F*, the enclosed error message becomes the result.

```
      2 ÷TRAP 0
DOMAIN ERROR
       L F R
       ^ ^

      ⍴⊃2 ÷TRAP 0
3 12
```

```
Z←L (F EL) R                                  ⍝ Each Left
Z←L (F ER) R          Z←(F ER) R              ⍝ Each Right
```

These operators are like the Each operator (¨), except that *EL* applies Each only on the left argument and *ER* applies Each only on the right argument.

```
      (2 2 3)(4 3)(2 6) ⍴EL ⍳12
 1  2  3     1  2  3    1 2 3  4  5  6
 4  5  6     4  5  6    7 8 9 10 11 12
             7  8  9
 7  8  9    10 11 12
10 11 12

      2 3 ⍴ER 4 5 6
4 4 4   5 5 5   6 6 6
4 4 4   5 5 5   6 6 6
```

```
Z←L (F PL) R                                  ⍝ Pervasive Left
Z←L (F PR) R          Z←(F PR) R              ⍝ Pervasive Right
```

PL causes *F* to be treated as pervasive down to depth 1 (simple arrays) on its left argument, and *PR* causes *F* to be treated as pervasive down to depth 1 on its right argument.

```
      1 (2 3) ⍴PL ⍳6
1   1 2 3
    4 5 6

      3 ⍴PR 1,⊂2,⊂3 4
1 1 1   2 2 2  3 4 3

      (⍴PR 'A' 'BC' ('DEF' 'HIJK')) ⍴PL '□'
□ □□   □□□ □□□□
```

```
Z←C (F ELSE G) R
```

If *C* is 1, then *Z* is *F R*. If *C* is 0, then *Z* is *G R*.

```
Z← (F IF C) R
```

If *C* is 1, then *Z* is *F R*. Otherwise, *Z* is *R*.

```
Z←L (F AND G) R                    Z←(F and G) R
```

The *AND* operator applies two functions to the same argument(s).

```
      3 +AND× 5
8 15
      +AND- 5
5 ¯5
      (⍳4) (∘.×)AND(∘.+) (⍳4)
 1  2   3   4    2 3 4 5
 2  4   6   8    3 4 5 6
 3  6   9 12    4 5 6 7
 4  8 12 16    5 6 7 8
```

```
Z←L (F COMMUTE) R
```

The *COMMUTE* operator switches the arguments of the function to which it is applied.

```
      0.5 *COMMUTE 9
3
```

```
Z←L (F FAROUT) R      ⍝  FAr Reaching OUTer product
```

This operator applies outer product to all levels of the arrays *L* and *R*.

```
      (10 20)(30 40 50) +FAROUT (1 2)(3 4 5)
 11 12    13 14 15
 21 22    23 24 25

 31 32    33 34 35
 41 42    43 44 45
 51 52    53 54 55
```

```
Z←L (F NOP) R      Z←(F NOP) R        ⍝ No OPeration
```

The derived function (*F NOP*) is just *F*. This operator is useful for separating the array right operand of an operator from the right argument of the derived function. It sometimes eliminates one layer of parentheses.

```
      ⍝ Compare with the next example.
      ⍴POWER 2 NOP 2 3 4⍴⍳24
3
```

```
Z←(F POWER N) R
```

POWER applies *F* monadically *N* times.

```
      ⍝ Parentheses are redundant here.
      (⍴ POWER 2) 2 3 4⍴⍳24
3
```

The MATHFNS Workspace

This workspace contains the following functions, as shown in Figure 8

EIGEN	Computes eigenvalues and eigenvectors
FFT	Computes fast Fourier transform
FMTPD	Formats in polar form with angular measure in degrees
FMTPR	Formats in polar form with angular measure in radians
IFFT	Computes inverse fast Fourier transform
POLYZ	Computes the zeros of polynomials

Figure 8. Functions in the MATHFNS Workspace

Eigenvalues

```
Z← EIGEN R
```

The right argument *R* must be a simple square matrix of real numbers. *Z* is a simple real or complex matrix of shape 1 0+⍴*R* that contains the eigenvalues and the eigenvectors of *R*. If *R* has shape *N* by *N*, then *Z* has *N*+1 rows and *N* columns. The first row of *Z* contains the eigenvalues of *R*, and the remaining rows of *Z* contain the corresponding right eigenvectors of *R*. That is, each column of *Z* contains an eigenvalue, and its corresponding right eigenvector.

```
      EIGEN 2 2⍴1 0 0 2
1 2
1 0
0 1
```

The eigenvalues *X* and the right eigenvectors *V* can be obtained by:

```
      Z← EIGEN R
X←Z[1;]
V← 1 0↓Z
```

They obey the identity:

```
X×[2]V ←→ R+.×V
```

The eigenvalues *X* and the left eigenvectors *V* can be obtained by:

```
      Z←⌽ EIGEN ⌽ R
X←Z[;1]
V←0 1↓Z
```

They obey the identity:

```
X×[1]V ←→V+.×R
```

The eigenvalues and eigenvectors are computed using the implicit QL algorithm if *R* is symmetric, or the QR algorithm if *R* is not symmetric. The numerical accuracy of the result is dependent upon the condition of the matrix of eigenvectors. In particular, accuracy can be degraded if there are repeated eigenvalues.

Fast Fourier Transform

```
Z←FFT R      ⍝ Fast Fourier Transform
```

This function computes the discrete Fourier transform of a set of 2^n numbers *R*.

The right argument *R* is a simple vector of `2*N` complex or real numbers, where N is a positive integer. The result *Z* is a simple vector of `2*N` complex numbers with the discrete Fourier transform of *R*.

The result of the *FFT* function corresponds to that of the discrete Fourier transform given by:[1]

$$y_k = \sum_{j=0}^{n-1} x_j e^{2\pi\sqrt{-1}\,(\frac{k}{n})j}$$

```
Z←IFFT R     ⍝ Inverse Fast Fourier Transform
```

This function computes the inverse Fourier transform of a set of 2^n numbers *R*.

```
      R ⇔ IFFT FFT R
```

The right argument *R* is a simple vector of `2*N` complex or real numbers, where N is a positive integer. The result *Z* is a simple vector of `2*N` complex numbers with the inverse discrete Fourier transform of *R*.

The *IFFT* function differs only in scale and phase.

$$y_k = (\frac{1}{n}) \sum_{j=0}^{n-1} x_j e^{-2\pi\sqrt{-1}\,(\frac{k}{n})j}$$

For example:

```
      IFFT 2 0J1 0 0J¯1
0.5 1 0.5 0
      IFFT 2 1 0 1
1 0.5 0 0.5
```

[1] SC23-0526, *Engineering and Scientific Subroutine Library, Version 2, Guide and Reference*, page 787.

Formatting Complex Numbers

```
Z←FMTPD R                          ⍝ ForMaT Polar Degrees
```

This function formats complex numbers in the right argument *R* in polar form with angular measure in degrees. *Z* is a simple character array.

```
Z←FMTPR R                          ⍝ ForMaT Polar Radians
```

This function formats complex numbers in the right argument *R* in polar form with angular measure in radians. *Z* is a simple character array.

Roots of Polynomials

```
Z← POLYZ R                         ⍝ POLYnomial Zeros
```

The right argument *R* must be a simple nonempty vector of real or complex numbers, and must not contain leading zeros. *R* represents a polynomial with coefficients in decreasing order of powers (constant on the right). *Z* is a simple vector of shape ¯1+⍴*R*, that contains the zeros of the polynomial *R*.

If *F* is the polynomial represented by *R* and F(x) = $Ax^3 + Bx^2 + Cx + D$, then *R* is the vector (*A B C D*). If the result *Z* is the vector (*P Q R*), then F(x) = (x-P)(x-Q)(x-R). If *R* is real, and the length of *R* is even, then *Z* contains at least one real number.

```
      POLYZ ¯2 1
0.5
      POLYZ 2 0J1
0J¯0.5
      POLYZ 1 ¯2 1
1 1
      POLYZ 1 0 1
0J1 0J¯1
      POLYZ 1 ¯6 11 ¯6
1 2 3
      POLYZ 1 ¯20 154 ¯584 1153 ¯1124 420
1 2.000000033 1.999999967 3 5 7
```

Figure 9. POLYZ Example

The zeros are computed using the Jenkins and Traub algorithms. The accuracy of the solution depends on the condition of the polynomial. In particular, accuracy can be degraded if there are repeated zeros. Also, numerical roundoff can cause a pair of equal real zeros to appear as a complex conjugate pair.

| *POLYZ* uses subroutines *POLYZC* and *POLYZF*.

The UTILITY Workspace

This section describes the UTILITY workspace.

Introduction

The UTILITY workspace is made up of defined functions organized into groups of functions. The groups are listed in the next section and described in the sections that follow.

The two major ways in which you are likely to find the UTILITY workspace useful are:

- Functional
- Instructional

The functional use is relatively straightforward:

- Copy the objects you need from the UTILITY workspace into the active workspace
- Use the UTILITY functions as "pseudo-primitives" in your own defined functions.

The instructional use may not be as obvious, but may be even more important. Instructionally, you can use the UTILITY workspace to:

- Acquire familiarity with APL2 by experimenting with the functions in the UTILITY workspace, listing and reading them, trying to deduce what each statement does and why you might choose that particular way to do it.
- Develop your APL2 programming skills by modifying the functions to improve their efficiency or to add features you need.
- Extend your programming skills by adding complementary utility functions that you find useful.

This workspace is of most use to you if you try to use it for both functional and instructional purposes.

The Function Groups

GPDATACV	Data conversion
GPMISC	Miscellaneous utility functions
GPSTRIP	Removing comments from functions
GPSVP	Controlling communication though the Shared Variable Processor
GPTEXT	Text processing
GPTRACE	Managing trace and stop vectors
GPXLATE	Character translation

Figure 10. Groups in the UTILITY Workspace

A list of the main functions in each group is presented in a captioned figure at the beginning of each section. Functions that belong to more than one group are usually listed only once.

GPDATACV: Converting between External and Internal Representations

Data Conversion Functions

Conversion Type	In	Out
Boolean (Logical)	*Z← LI R*	*Z← LO R*
System/370 Integer	*Z← II R*	*Z←L IO R*
IBM PC Integer	*Z← PCII R*	*Z←L PCIO R*
System/370 Floating Point	*Z← FI R*	*Z← FO R*
Packed Decimal	*Z← PDI R*	*Z←L PDO R*
VS APL	*Z← ICI R*	*Z← ICO R*

Data Conversion Variables

zc	Used in translating between APL2 EBCDIC and VS APL Internal Characters
pds	Used in determining sign of packed-decimal number
pdd	Used in translating numerical portion of packed decimal

Figure 11. GPDATACV: Data Conversion Functions and Variables

```
Z←LI R                    ⍝ Logical In
```

R is a simple character array whose last axis contains logical data; that is, a string of bits.

Z is a numeric array consisting of zeros and ones representing the logical data in *R*. The rank of *Z* is the same as the rank of *R*, but the last axis of *Z* is 8 times as long as the last axis of *R*. A scalar value for *R* produces an 8-element vector.

```
⍴Z←→(¯1↓⍴R),8×¯1↑1,⍴R
```

```
Z←LO R                    ⍝ Logical Out
```

R is a simple numeric array consisting of only zeros and ones. The length of its last axis must be a multiple of 8.

Z is a character array whose last axis contains the representation of the logical data in the last axis of *R*. The rank of *Z* is the same as the rank of *R*, but the length of the last axis of *Z* is one-eighth of the length of the last axis of *R*.

```
⍴Z←→(¯1↓⍴R),(¯1↑⍴R)÷8
```

```
Z←II R                    ⍝ Integers In
```

R is a simple character array whose last axis must have a length of between 1 and 7 inclusive. The array must also contain the System/370 binary representations of integers.

Z is an array of integers representing the binary numbers in *R*. The rank of *Z* is one less than the rank of *R*.

```
⍴Z←→¯1↓⍴R
```

```
Z←L IO R                  ⍝ Integers Out
```

R is a simple array of integers. *L* is an integer scalar not greater than 7, which gives the number of bytes in which each integer is represented. *L* must be large enough to represent the largest magnitude of the integers in *R*.

Z is a character array whose last axis contains the System/370 binary representations of the integers in *R*. The rank of *Z* is one greater than the rank of *R*.

```
⍴Z←→(⍴R),L
```

```
Z←PCII R                  ⍝ PC Integers In
```

R is a simple character array whose last axis must have a length of 1, 2 or 4, and which contains the IBM PC (reversed) binary representations of integers.

Z is an array of integers representing the binary numbers in *R*. The rank of *Z* is one less than the rank of *R*.

```
⍴Z←→¯1↓⍴R
```

```
Z←L PCIO R                ⍝ PC Integers Out
```

R is a simple array of integers. *L* is an integer scalar with a value of 1, 2 or 4, and gives the number of bytes in which each integer is to be represented. *L* must be large enough to represent the largest magnitude of the integers in *R*.

Z is a character array whose last axis contains the IBM PC (reversed) binary representation of the integers in *R*. The rank of *Z* is one greater than the rank of *R*.

```
⍴Z←→(⍴R),L
```

```
Z←FI R                    ⍝ Floating In
```

R is a simple character array; its last axis must have a length of 4 or 8. The last axis thus represents either single or double precision System/370 floating-point numbers.

Z is an array of numbers equivalent to the floating-point representations in *R*. The rank of *Z* is one less than the rank of *R*.

```
ρZ←→¯1↓ρR
```

```
Z←FO R                    ⍝ Floating Out
```

R is a simple numeric array.

Z is a character array whose last axis has length 8, and which contains the System/370 double precision floating-point representations of the numbers in *R*. The rank of *Z* is one greater than the rank of *R*. If single precision is required, then drop the last four columns of the result.

```
ρZ←→(ρR),8
```

```
Z←PDI R                   ⍝ Packed Decimal In
```

R is a simple character array whose last axis must have a length of between 1 and 16 inclusive, and which contains valid System/370 packed decimal representations of integers.

Z is an array of integers representing the packed decimal numbers in *R*. The rank of *Z* is one less than the rank of *R*.

```
ρZ←→¯1↓ρR
```

Note that if the length of the packed decimal number is greater than 9 bytes, a loss of precision can result.

```
Z←L PDO R                 ⍝ Packed Decimal Out
```

R is a simple array of integers. *L* is an integer scalar not greater than 16; it gives the number of bytes in which each integer of *R* is represented. *L* must be large enough to represent the largest magnitude of the integers in *R*.

Z is a character array whose last axis contains the System/370 packed-decimal representations of the integers in *R*. The rank of *Z* is one greater than the rank of *R*.

```
ρZ←→(ρR),L
```

```
Z←ICI R                   ⍝ VS APL Internal Characters In
```

R is a simple character array. *Z* is a character array of the elements of *R* as they would be displayed and interpreted in VS APL.

```
Z←ICO R          ⍝ VS APL Internal Characters Out
```

R is a simple character array. *Z* is a character array whose elements are displayed and interpreted the same in VS APL as the corresponding elements of *R* are displayed and interpreted in APL2.

GPMISC: Miscellaneous Utility Functions

ANNOTATE	Add comments to lines in character matrix
ASSIGN	Specify values for a set of names
CASE	Gives case attribute of active workspace
CODECOUNT	Count commented and uncommented lines in all the defined operations in a workspace
CONCEAL	Make a function nonsuspendable
DATETIME	Give date and time in *hh:mm:ss* format
EXPAND	Function version of \
FNHEADS	List function headers for a set of functions
FRAME	Put a border around a character matrix
HEXDUMP	Produce character and hexadecimal representations of a character string
LINECOUNT	Count commented and uncommented lines in a set of defined operations
LIST	Convert an arbitrary array to vector
MASKCONV	Convert fullword integers to their component subfields
MESH	Mesh two or more vectors as prescribed by a mask
NAMEREFS	Find all names in a defined function or operator
NAMES	Find all names in a string
NHEAD	Produce character representations of index vectors
REPLICATE	Function version of /
REVEAL	Make a function suspendable
TYPE	Determine type (alphabetic or numeric) of a simple, homogeneous APL array
UNIQUE	Remove duplicates

Figure 12. GPMISC: Miscellaneous Utility Functions

```
Z←L ANNOTATE R
```

R is a simple character matrix and *L* is a numeric scalar. *Z* is *R* with rows padded or truncated to length *L* and with comments interactively appended to each row.

```
L ASSIGN R
```

L is a character matrix of names. *R* is a character matrix of valid APL2 expressions. Each row of *L* is evaluated and its value is given the name in the corresponding row of *R*.

```
Z←CASE
```

`Z` is the case attribute of the active workspace.

```
Z←CODECOUNT
```

This function counts the function and operator lines in the workspace and returns a 2-element numeric vector. `Z[1]` is the total number of lines in the workspace that contain something other than a comment; `Z[2]` is the total number of lines that consist only of a comment. *CODECOUNT* does not count its own lines. See also *LINECOUNT* on page 38.

```
CONCEAL R
```

Make the function named by `R` nonsuspendable.

```
Z←DATETIME
```

`Z` is the date and time in the form of *mm/dd/yy hh:mm:ss*.

```
      DATETIME
11/26/85    12:00:42
```

```
Z←L EXPAND R
```

`R` is any array. `L` is a Boolean vector. `Z` is `L\R`. See “Special Functions and Operators of APL2” on page 22 for a discussion of this function.

```
Z←FNHEADS R                    ⍝ FunctioN HEADerS
```

`R` is a character matrix of function or operator names. `Z` is a character matrix of corresponding function and operator headers, exclusive of explicit local variables.

```
Z←FRAME R
```

`R` is a simple character scalar, vector, or matrix. `Z` is `R` bordered by straight lines.

```
Z←HEXDUMP R
```

R is a simple character array. *Z* is a four row matrix, with one column for each element of ,*R*. The first row of *Z* is *R*; the second row is ⎕*AF* ,*R*; the third row contains the hexadecimal representations of the numbers in the second row; and the fourth row contains characters that mark off character positions by fives.

```
Z←LINECOUNT R
```

R is a character scalar, a simple vector or matrix, or a vector of vectors. *LINECOUNT* counts the lines of the functions and operators named in *R* and returns a two element numeric vector. *Z*[1] is the number of lines that contains something other than a comment; *Z*[2] is the total number of lines that consist only of a comment. This function does not count its own lines. See also *CODECOUNT* on page 37.

```
Z←LIST R
```

This function creates a vector or scalar out of *R*. *R* can be any array. If *R* is a simple scalar, then *Z* is ,*R*. If *R* is a simple vector, then *Z* is ,⊂*R*. If *R* is a nested scalar or vector, then *Z* is *R*. Otherwise, *Z* is *R* enclosed along all axes but the first, which forms a nested vector.

```
Z←L MASKCONV R          ⍝ MASK CONVert
```

MASKCONV encodes the number (or numbers) *R* to the base 2*×L*. It is primarily useful in analyzing sections of storage defined by fields of varying lengths from one bit to a full word.

```
      1 2 1 4 24 MASKCONV ¯1+2*32
1 3 1 15 16777215
```

```
Z←L MESH R
```

L is a mask and *R* is a concatenation of the vectors to be meshed. If the mask *L* consists of zeros and ones, the elements of *R* are placed, in order of occurrence, in the positions of *Z* corresponding to zeros; after these are filled, the remaining elements are placed in the positions corresponding to ones. If *R* is a concatenation of vectors of lengths equal to the number of zeros and the number of ones respectively, the result is to mesh them. This can be generalized to any number of vectors by providing masks with elements of 0, 1, 2,...

```
                                                  00122233333333
                                                  ↓↓↓↓↓↓↓↓↓↓↓↓↓↓

      0 2 2 1 3 3 3 3 3 2 3 0 3 3  MESH 'HE IS WORDSMAN'
HIS WORDS MEAN

↑↑↑↑↑↑↑↑↑↑↑↑↑↑
02213333323033
```

In the example above, 0 selects the first two characters (`'HE'`) and puts them in the first and twelfth positions of the result; 1 puts a blank in the fourth position; 2 puts `'IS  '` in positions 2, 3, 10; and 3 puts the remainder.

```
Z←NAMEREFS R
```

R is the name of a function or operator. *Z* is a character matrix that contains a list of all the names that occur in *R*.

```
Z←NAMES R
```

R is a character vector. *Z* is a matrix of all the names in *R*.

```
Z←L NHEAD R                ⍝ Numeric HEADers
```

L and *R* are integers. *Z* is a character array giving ⍳*R* in column form if *L* is 0 and row form if it is not.

```
       0 NHEAD 5
1
2
3
4
5
       1 NHEAD 5
12345
       1 NHEAD 40
          1111111111222222222233333333334
1234567890123456789012345678901234567890
```

This function is also in the group *GPTEXT*.

```
Z←L REPLICATE R
```

R is any array. *L* is a vector of integers. *Z* is *L/R*. See “Special Functions and Operators of APL2” on page 22 for a discussion of this function.

```
REVEAL R
```

If possible, make the function named by *R* suspendable.

```
Z←TYPE R
```

Z is a scalar zero if *R* is numeric, and a scalar blank if it is character. This function is compatible with a VS APL library function of the same name. It is not meant to be applied to mixed or nested arguments.

```
Z←UNIQUE R
```

R is a vector. *Z* is a vector that contains the elements of *R* with duplicates eliminated.

```
      UNIQUE 'THE ANTS WERE HERE'
THE ANSWR
      UNIQUE 'GUFFAW' 17 (⍳4) 'GUFFAW'
 GUFFAW 17  1 2 3 4
```

GPSTRIP: Removing Comments

DECOMMENT	Remove comments from all defined functions and operators
STRIP	Remove comments from all defined functions and operators named in a list
WORDS	Split a character vector into nested pieces

Figure 13. GPSTRIP: Removing Comments

```
DECOMMENT
```

This function removes comment lines from all unlocked functions and operators in the active workspace. Running decommented functions requires less storage. When using this function, you should keep a backup copy of the workspace.

```
STRIP R
```

STRIP removes comments from all unlocked functions and operators named in *R*. *R* is a simple character matrix, a nested vector of names, or a simple string of names separated by blanks.

```
Z←L WORDS R
```

WORDS is a surrogate name for the supplied external function *DAN* (See "DAN—Delete And Nest" on page 151.) *R* is a character vector. *L* is a scalar or vector of delimiter characters. *Z* is a character vector, each of whose elements is a vector of the elements of *R* lying between occurrences of the delimiters in *R*. Consecutive occurrences of delimiters in *R* are ignored.

See Figure 14 for an example using *WORDS*.

```
      Z←'And what exactly ARE the commercial'
      Z←Z,' possibilities of ovine aviation?'
      ⍴Z
68
      Z←' ' WORDS Z
      ⍴Z
10
      ⊃Z
And
what
exactly
ARE
the
commercial
possibilities
of
ovine
aviation?
      ⍴⍴Z
 3 4 7 3 3 10 13 2 5 9
```

Figure 14. WORDS: Extracting Words from Character Vectors

GPSVP: Controlling Communication through SVP

APSERVER	Server for implementing APs using a client-server protocol over a single shared variable interface
ID	Convert enclosed character processor IDs to large integers
SVOFFER	Share one or more variables with an auxiliary processor
SVOPAIR	Share control and data variables with auxiliary processors that use a two-variable protocol.
Δ*SVO*	□*SVO* extension to support enclosed character vectors as processor IDs.
Δ*SVQ*	□*SVQ* extension to support enclosed character vectors as processor IDs.

Figure 15. GPSVP: Controlling Communication through SVP

```
APSERVER R
```

APSERVER is the general AP server for implementing auxiliary processors using a client-server protocol over a single shared variable interface.

The *APSERVER* function uses a registered callback interface, where you choose to supply a minimum of zero (for the default “echo” AP) to a maximum of four callback function names. The syntax of the *APSERVER* call is:

```
APSERVER 'Init_fn' 'Wait_fn' 'Process_fn' 'Exit_fn'
```

If a callback function is not provided, the corresponding item in the 4 element general array argument should contain an empty character vector.

The first name in the argument list is the name of the initialization function that gets called by *APSERVER* when a new share offer arrives. The syntax of *Init_fn* is:

```
RC←Init_fn PID SVNAME
```

APSERVER passes to the initialization function the SVP processor number of the client and the name of the shared variable being offered. If the AP chooses to accept the share, it returns an explicit result of 1. To reject the share offer, a 0 is returned.

The initialization function can be used to open files, establish shares with other APs, or to initialize global variables. Since the AP runs as a single task, care should be taken to avoid blocking on a shared variable access within the callback functions if the AP is designed to support multiple shares or multiple clients.

The second name in the *APSERVER* argument list is the name of the wait callback function. If no wait routine is supplied the default action of the *APSERVER* is to enter a □*SVE* wait for any shared variable event, then scan for new offers, new client requests, or shared variable retractions. The *Wait_fn* function, if provided, must be a niladic function with no explicit result. You may wish to provide your own wait function to issue □*SVE* so that you can check the state of other shared variables used for your own purposes, or so that you can

provide a time-out on the □*SVE* wait (for example, to do some administrative work such as journaling). When you supply wait routine exits, the *APSERVER* performs the usual checking for client events.

The third item in the *APSERVER* argument list is the name of the process function—the meat of the AP. The syntax is:

```
RESULT←(PID SVNAME) Process_fn REQUEST
```

The right argument is the APL2 array representing the client request. The *APSERVER* provides the client processor ID and shared variable name in the left argument. Provide the necessary code in the process routine to service the client request, and then return, as the explicit result of the function, the APL2 array that is to be sent back to the client in response to the request. If the process callback is elided, the default action of the *APSERVER* is to echo the request back to the client.

The fourth item in the *APSERVER* argument list is the name of the exit callback function. The syntax is:

```
Exit_fn PID SVNAME
```

The *APSERVER* again passes the client processor ID and shared variable name in the right argument. The exit function is called when the client retracts the shared variable. The exit function is often used as the inverse to the initialization function, to close files, retract other associated shares, and expunge global variables. When the *APSERVER* gets control back from the exit routine, it completes the retraction from the server side.

Note: A current restriction of APs written in APL2 using the *APSERVER* client-server protocol is that the client must reference all return values sent by the server, prior to issuing another request. Failure to do so could result in a request being lost due to a race condition.

```
Z←L ID R
```

Convert enclosed character processor IDs to large integers and vice versa. Typically used with the SVP profile in support of cross-system SVP shares for cooperative processing. Under CMS, the processor profile is file AP2TCPIP APL2PROF. Under TSO, the processor profile is member AP2TCPIP in the data set pointed to by DDNAME APL2PROF.

```
Z←L SVOFFER R
```

Offer shared variables, named in right argument, to SVP processors identified by numbers in the left argument. Returns the final degree of coupling for each shared variable. The function delays up to 15 seconds for shares to be accepted by the partner. It sets standard access control to inhibit a double set or use.

R is a character scalar, vector, matrix, or vector of vectors containing the name or names of the shared variables to be offered to an auxiliary processor. Surrogate names for shared variables can also be used. *L* is a numeric scalar or vector containing the processor ID (the number) of the AP. *Z* is the degree of coupling

for the shared variable; a 2 indicates that the corresponding variable is fully shared with the AP.

```
      211 SVOFFER 'S1' 'S2'
2 2
```

```
Z←L SVOPAIR R
```

Offer shared variables, named in right argument, to SVP processors identified by numbers in the left argument. This function is used for auxiliary processors that support a two-variable interface, where the control variable begins with “CTL,” and the data variable begins with “DAT” (such as AP 124).

Note: The function is included in the mainframe APL2 product for compatibility with the workstation products, and is useful for writing portable code that uses AP 124.

```
Z←L ΔSVO R
```

⎕*SVO* extension to support enclosed character vectors as processor IDs. Typically used with the SVP profile in support of cross-system SVP shares for cooperative processing. Uses the *ID* function to map the character vector to a processor ID.

```
Z←L ΔSVQ R
```

⎕*SVQ* extension to support enclosed character vectors as processor IDs. Typically used with the SVP profile in support of cross-system SVP shares for cooperative processing. Uses the *ID* function to map the character vector to a processor ID.

GPTEXT: Manipulating Text

Note that many text functions also work on other kinds of data.

DOUBLE	Replace selected characters in character vector two-for-one
FIND	Search for text in all functions and operations in the active workspace
GATHER	Collect parsed fields surrounded by delimiters
GVCAT	Concatenate rows to arrays of any rank
HCAT	Concatenate matrices by columns
INBLANKS	Separate characters by blanks
LADJ	Left adjust
LINEFOLD	Fold line to specified width and indent folded portions a specified amount
MAT	Make a matrix out of any array
MATFOLD	Fold matrix to specified width and indent folded portions a specified amount
NOQUOTES	Remove quoted substrings
OBLANKS	Remove outer blanks
QREPLACE	Replace '?' occurrences by character strings
RADJ	Right adjust
RCNUM	Produce numerical headings for rows and columns
REPLACE	Replace substrings in character strings
RTBLANKS	Remove trailing blanks
VCAT	Concatenate matrices by rows
XBLANKS	Remove leading and trailing blanks and reduce all intermediate blank substrings to single blanks

Figure 16. GPTEXT: Text Processing Functions

```
Z←L DOUBLE R
```

DOUBLE replaces each occurrence of the scalar *L* in the vector *R* by a pair of scalars *L*.

```
      V←'ABC''DEFGH''IJK'
      V
ABC'DEFGH'IJK
      '''' DOUBLE V
ABC''DEFGH''IJK
```

```
[namelist] FIND 'text' ['newtext']
```

Gives a listing of all functions and operators in the active workspace that contain the indicated text.

If '*newtext*' is specified, this function replaces '*text*' in the objects listed in *namelist* with the new text.

```
Z←L GATHER R
```

L is a scalar or a one- or two-element vector, for example `'( )'`. *R* is any array. *GATHER* searches the rows of *R* for a sequence enclosed within the first and second elements of *R* and unravels them into a vector. A blank is inserted at each point where the resulting vector crosses a row boundary in *R*.

```
Z←L GVCAT R    ⍝ Generalized Vertical conCATenation
```

L and *R* are arrays of any rank. *Z* is the result of concatenating *L* to *R* along the first coordinate of the array of higher rank.

```
Z←L HCAT R           ⍝ Horizontal conCATenation
```

HCAT concatenates columns; given two matrices, it places them side-by-side. *L* and *R* should not be of rank greater than 2. *Z* is always of rank `2`.

```
Z←L INBLANKS R
```

If characters in *L* are contained in *R*, separate them with blanks.

```
Z←LADJ R             ⍝ Left ADJust
```

R can be any array. *Z* is that array with nonblank characters shifted to the left as far as possible.

```
Z←L LINEFOLD R
```

This function folds the line *R* so that it is no greater than the length specified by the first (or only) element in *L*. If *L* has a second element, then this specifies the number of blanks to be used in offsetting the second and subsequent rows in the output *Z*. *Z* is always of rank `2`.

```
Z←MAT R              ⍝ MATrix
```

Z is an array of rank `2` that contains all the elements of *R*.

```
Z←L MATFOLD R              ⍝ MATrix FOLD
```

L has one or two integer components. *R* can be any array. *Z* is a matrix with a number of columns equal to the first (or only) component of *L*. Any lines longer than this width are folded as in *LINEFOLD*.

```
Z←NOQUOTES R
```

R is a vector. *Z* is the same vector with all quoted substrings removed. This function is also in the group *GPMISC* on page 36.

```
Z←OBLANKS R                ⍝ Outer BLANKS
```

Remove outer blanks. *R* is a vector. *Z* is *R* with all leading and trailing blanks removed.

```
Z←L QREPLACE R             ⍝ Question mark REPLACEment
```

R is a vector that contains one or more question marks. *L* is a character vector that contains one or more subvectors to be substituted for the question marks. The first character of *L* is a delimiter used to identify the substitution vectors. This delimiter must also be the last character of *L*. *Z* is *R* with the substitutions made.

```
Z←RADJ R                   ⍝ Right ADJust
```

Z is *R* right-adjusted, so that the rightmost character of each row is not blank unless all the characters of the row are blank. *R* can be an array; the rows are right-adjusted individually.

```
Z←RCNUM R                  ⍝ Row and Column NUMbers
```

R is a matrix. *Z* is *R* with column numbers across the top and row numbers along the left side.

```
Z←L REPLACE R
```

R can be any array. *Z* is *R* with every occurrence of a seek string replaced by a replace string. *L* is a two-element vector, each of whose elements is a scalar or vector. The first element is the seek string and the second element is the replace string.

REPLACEV is a subfunction of *REPLACE*.

```
      TEXT←4 4⍴'HEREIS  SOMETEXT'
      REPLACE/' _' ('HERE' 'THERE') TEXT
THERE
IS___
SOME_
TEXT_
```

Figure 17. REPLACE: A String Replacement Function

```
Z←RTBLANKS R          ⍝ Remove Trailing BLANKS
```

R is a simple array. *Z* is *R* with trailing blanks or trailing blank columns removed.

```
Z←L VCAT R            ⍝ Vertical CATenation
```

L and *R* are arrays of rank 2 or less. *Z* is a matrix. Its width is the that of the wider of *L* or *R*. *L* is at the top of *Z* and *R* is at the bottom.

```
Z←XBLANKS R           ⍝ eXtra BLANKS
```

Remove extra blanks. *R* must be a vector. *Z* is *R* with leading and trailing blanks removed and intermediate blank sequences reduced to a single blank.

GPTRACE: Setting and Removing Trace and Stop Vectors

The functions in this group can be used in *debugging* your defined APL2 operations by establishing trace and stop vectors when you are checking the operations out and removing them when you are finished.

STOPALL	Create stops on all statements in all functions of a workspace
STOPOFF	Set all stop vectors to the empty vector
STOPONE	Create stops at first statements of all functions and operators in the active workspace
TRACEALL	Trace all statements in all functions and operators of the active workspace
TRACEBR	Trace branch lines of a given function or operator
TRACELIST	Trace all statements in functions and operators named in a list
TRACEOFF	Set all trace vectors to the empty vector
TRACEONE	Trace first statements of all functions and operators in the active workspace

Figure 18. GPTRACE Functions

```
STOPALL
```

This function creates stops on all statements in all functions in the active workspace.

```
STOPOFF
```

This function cancels all the stop vectors in the active workspace.

```
STOPONE
```

STOPONE creates stop vectors for the first statement of all the functions and operators in the active workspace.

```
TRACEALL
```

TRACEALL creates trace vectors for all the statements of all the functions and operators in the active workspace.

```
TRACEBR R          ⍝ TRACE BRanch
```

This function creates a trace vector for every branch statement of the function or operator named in *R*. *R* is a single name.

```
TRACELIST R
```

This function creates trace vectors for all the statements in the functions and operators named in *R*. *R* is a simple scalar, vector, or matrix, or a vector of vectors.

```
TRACEOFF
```

This function cancels all the trace vectors in the active workspace.

```
TRACEONE
```

TRACEONE creates trace vectors for the first statements of all the functions and operators in the active workspace.

GPXLATE: Translating from One Character Representation to Another

GPXLATE contains three functions and two global variables. The variables are used as translate tables by the functions *LCTRANS* (which converts from uppercase to lowercase), and *UCTRANS* (which converts from lowercase to uppercase).

The third function, *TRANSLATE*, is a general-purpose translate function that requires a translate table as its left argument. The functions and their syntax are shown in Figure 19.

Translation Functions

`Z← LCTRANS R`	Translate uppercase to lowercase
`Z← UCTRANS R`	Translate lowercase to uppercase
`Z←L TRANSLATE R`	Translate *R* into *Z* using translate-table *L* (a numeric vector)

Translation Group Constants

`LCTt`	Table used for translating to lowercase
`UCTt`	Table used for translating to uppercase

Figure 19. Functions for Translating Character Arrays (GPXLATE)

The use of the uppercase and lowercase translate functions is demonstrated in Figure 20.

```
      CV←'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'

      UCTRANS CV
ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ

      LCTRANS CV
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
```

Figure 20. Examples of Lowercase and Uppercase Translation

```
(1)        ⎕IO←0

(2)        LOWERINDICES←⎕AF 'abcdefghijklmnopqrstuvwxyz'

(3)        UPPERINDICES←⎕AF 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

(4)        LCTt←⍳256

(5)        LCTt[UPPERINDICES]←LOWERINDICES

(6)        LOWERINDICES
   129 130 131 132 133 134 135 136 137 145 146 147 148 149 150 151 152 153 162 163
          164 165 166 167 168 169

(7)        UPPERINDICES
   193 194 195 196 197 198 199 200 201 209 210 211 212 213 214 215 216 217 226 227
          228 229 230 231 232 233

(8)        LCTt[UPPERINDICES]
   193 194 195 196 197 198 199 200 201 209 210 211 212 213 214 215 216 217 226 227
          228 229 230 231 232 233

(9)        ⎕CR 'LCTRANS'
   A←LCTRANS B
   ⍝ A IS B WITH UPPERCASE LETTERS TRANSLATED TO LOWERCASE LETTERS
   A←LCTt[⎕IO+⎕AF B]
```

Figure 21. Constructing and Using a Lowercase Translate Table

Figure 21 shows how the lowercase translate table was constructed:

(1) Since translation requires selecting values from tables, it is important to establish a known index origin. The first action is to set the origin to `0` because `0` is more useful than `1` for translating purposes.

(2) The indexes of the lowercase letters in $\square AV$ are determined by the use of $\square AF$.

(3) The indexes of the uppercase letters in $\square AV$ are determined by the use of $\square AF$.

(4) The lowercase translate table is initialized as 256 consecutive integers from 0 to 255.

(5) The indexes of the lowercase letters replace the indexes of the uppercase letters in the translate table.

(6), (7), and (8)
Show how the index sets and the translate table are interconnected.

(9) A canonical representation of the $LCTRANS$ function, showing how it does lowercase translation.

Chapter 4. The Display Terminal Workspaces

Using the screen to communicate between a program and a terminal user and controlling all the terminal's signaling, sensing, formatting, and other features is called *full-screen management*. The auxiliary processor AP 126 provides a full-screen management capability for APL2 users.

AP 126: The GDDM/PGF Auxiliary Processor

AP 126 manages the screen by using the Graphical Data Display Manager (GDDM), including its presentation graphics feature (PGF). Information about GDDM and PGF is contained in two manuals:

GDDM Base Application Programming Reference
Presentation Graphics Feature: User's Guide

These manuals can help you use AP 126 effectively.

GDDM provides a comprehensive set of subroutines that perform a variety of screen-management functions, among them:

- Defining field size and placement.
- Defining up to nine field attributes (for example, color and highlighting).
- Defining up to three attributes for individual characters within a field.
- Using alternative symbol sets, that is, replacing the standard characters with others designed for special purposes either throughout an entire field or character-by-character.
- Drawing lines, curves, and axes using a feature usually called vector graphics.
- Drawing plots, graphs, charts, and histograms using a collection of subroutines collectively called the *presentation graphics feature* (PGF).

GDDM Workspaces: CHARTX, GDMX, GRAPHPAK, FSC126, FSM

Five workspaces use AP 126 to perform GDDM functions:

CHARTX This workspace offers a call interface to the GDDM *Interactive Chart Utility* (ICU). It also offers a facility for using predefined ICU chart formats.

GDMX This workspace contains *GDMX*, a cover function for GDDM. It can be used to call GDDM routines directly, taking advantage of APL2's general array facilities to pass multiple GDDM calls in one *GDMX* call.

GRAPHPAK This workspace contains a comprehensive set of functions for plots, graphs, charts, 3-dimensional geometry, and curve-fitting.

*FSC*126 This workspace is a replacement for the *FSC*124 and *FULLSCRX* VS APL workspaces, which used AP 124 for full-screen text input and output. *FSC*126 provides functions whose names and syntax match those in the VS APL workspaces, but use AP 126 instead. Because AP 126 uses GDDM for full-screen support, it can operate on a wider variety of displays and devices than AP 124.

FSC126's primary purpose is to help maintain older applications using full-screen "panels"—defined screen formats, text, field names, and associated information.

Note: *FSC126* is now obsolete. New applications can more fully use the potential of GDDM for combined full-screen text and graphics support by using the *GDMX* workspace.

If you have a VS APL application that uses the *FSC124* or *FULLSCRX* versions of the functions in *FSC126*, and you wish to continue to use the VS APL versions, you can migrate them along with your application by using the *)MCOPY* system command.

FSM This workspace has functions corresponding to many of the GDDM and PGF subroutines. It also includes a function that processes APL2 session manager commands.

Note: This workspace is obsolete and is not supported by facilities introduced after Version 1 Release 3 of GDDM. It is included here only for the benefit of older applications that use its facilities. New applications should use *GDMX* (described above).

An Introduction to Text and Vector Graphics

A display screen, like a sheet of paper, is used either for typing or for drawing. The following describes two uses of display screen: "text graphics" and "vector graphics".

Text Graphics

In text graphics, the screen is divided into rectangular cells. These cells contain either display characters (output or input text) or nondisplay characters (characters that delimit field boundaries and determine field attributes). A display terminal with 32 rows and 80 columns, for example, contains 2560 cells.

The smallest individual object in text graphics is a *character*. Characters can be combined into *fields*. A *format* defines a set of fields that make up a *page*. A page can contain several text fields but, at most, one graphics field. The graphics field is a rectangular area of the screen reserved (primarily) for drawing rather than typing.

Vector Graphics

To simulate line-drawing vector graphics, GDDM divides the surface of the display into *display points*: (sometimes called *pels* or *pixels*). These display points are a collection of dots, covering the entire surface of the screen, which can be individually illuminated. Drawing a line between two points illuminates the individual points that lie on the line connecting them. This is done by a beam that crosses the face of the display from side-to-side in successive lines from top to bottom. It illuminates the points on the line as it sweeps across them.

The expression vector graphics used below refers to this display point type of vector graphics.

Pages: Text and Vector Graphics

Pages are an important GDDM feature. Once a page is created by an APL2 program, it continues to exist until it is explicitly deleted or the program discontinues its use of GDDM. To replace a current page, you merely select one that you created earlier. The replaced page is retained; if needed, it can be selected again later. In selecting a page, you can accomplish in one step what might otherwise require extensive reformatting, text output, graphics output, and perhaps other activities, such as a request for input to be repeated.

Coordinate Systems: Text and Vector Graphics

The coordinate system for text graphics starts at the upper left corner of the screen in the position occupied by row 1, column 1. The coordinate system for vector graphics starts at the lower left corner of the graphics field in the position occupied by the point 0,0.

The difference in coordinate systems reflects the different conventions of text and vector graphics. GDDM uses both systems. In reserving space for a graphics field, the text graphics row/column convention is used. When you are processing graphics functions; however, the 0,0 position of the picture space is the lower left corner.

Coping with Complexity: Form and Chart Design

Designing forms and charts is a detailed, painstaking business. It requires a constant juggling to use limited space to best advantage. Multiple-copy snapout forms can contain interleaved spot carbons, shadow printing, blackout sections and other refinements known only to professional form designers.

Experienced cut-and-paste specialists design information to provide the relevant parts of a common body of information to people with different, but complementary objectives; for example, sales clerks, production planners, tally clerks, billing clerks, sales analysts, and customers.

Similarly, draftsmen who present numerical information graphically must plan to use a given space to best advantage; for example, they determine the size of the chart, and its placement on the page.

The point is that both of these activities require detailed sets of steps that are seldom explicitly listed—until you write a computer program that does them.

In writing a GDDM program for text graphics, you are specifying explicitly the activities that a form designer does implicitly, that is, without a set of specific instructions.

In writing a GDDM program for vector graphics, you are specifying explicitly what draftsmen do when preparing graphs or charts.

Although there are a large number of subroutines that make up GDDM, they are necessary. A possible strategy for learning them is:

- Learn those things that are specific to GDDM: how to start it, how to use it, and how to stop it.
- Think of what you would like to do in terms with which you are familiar. Write down a set of actions in those terms.

- To translate them into a program that uses GDDM, look for the GDDM subroutines that do what you want. Rewrite your procedure using subroutine names instead of the terms you originally wrote down.

You might find it harder to learn something as comprehensive as GDDM if you think of it as a collection of arbitrary facts all of which have to be mastered before any of them can be used. Start out thinking first not of GDDM; but, of what you have to do, and *then* looking for the functions that help you do it.

Finally, since you are using APL2, you have to learn how to translate the conventions used in GDDM publications (which are written for languages that call subroutines rather than process functions) into the conventions used in APL2.

CHARTX—an APL2/ICU Data Interface

The *CHARTX* function offers a call interface to the GDDM Interactive Chart Utility (ICU). Data can be passed to the ICU in a variety of formats. *CHARTX* offers a facility for using predefined ICU chart formats.

Tied and Free Data

The ICU allows the simultaneous graphical display of several groups of data. For example, a graph with three line plots has the data for each line plot represented as a data group. The ICU distinguishes between two types of data format modes for representing data groups either as tied data or free data. In tied data mode, all data groups have the same set of X values. In free data mode, each data group has its own set of X values or coordinates, which are independent of other groups.

CHARTX handles both ICU data format modes; the mode is determined from the structure of the arguments to *CHARTX*.

Using CHARTX for Tied Data

For tied data, *CHARTX* has the following call sequence:

```
XT CHARTX YT
```

Where:

- *XT* is the simple numeric array of X values.
- *YT* is the array of Y values.

If *XT* is not specified, *CHARTX* uses a default X-coordinate vector consisting of consecutive integers that are appropriate for *YT*, starting with ⎕*IO*.

YT is a simple numeric scalar, vector, or matrix. If *YT* is a scalar or vector, it forms one data group. If *YT* is a matrix with *M* rows and *N* columns, it forms *M* data groups.

If *YT* is a scalar or vector, then *XT* must be the same shape as *YT*. If *YT* is a matrix, then *XT* must be a vector, the length of which is the same as the number of columns of *YT*. That is:

```
⍴XT ←→ ¯1↑⍴YT.
```

Examples

Each of the following lines in Figure 22 on page 57 is a separate example. When you enter an example, the screen clears, then displays a chart.

```
      CHARTX 12 22 18 32 7

      (⍳5) CHARTX 12 22 18 32 7 ⍝ Same result as previous example

      CHARTX 1 10 ∘.× ⍳12

      (⍳12) CHARTX 1 10 ∘.× ⍳12 ⍝ Same result as previous example

      1 2 5 8 9 CHARTX 12 22 18 32 7

      (2⊗⍳12) CHARTX 1 10 ∘.×⍳12

      CHARTX 1 2 ∘.○ 0.1×⍳120
```

Figure 22. Examples of Using CHARTX for Tied Data

Using CHARTX for Free Data

For free data, *CHARTX* has the following call sequence:

```
XF CHARTX YF
```

Where:

- *XF* is the array of X values.
- *YF* is the array of Y values.

XF must have the same structure as *YF*. Items of *XF* form the X-coordinates for corresponding items of *YF*.

If *XF* is not specified, *CHARTX* uses a default X-coordinate array, each item of which consists of consecutive integers, starting with ⎕*IO*, that is appropriate for the corresponding item in *YF*.

YF is a numeric vector of depth 2, each item of which is a simple scalar or vector. Each item of *YF* forms an independent data group.

Examples

Each of the following lines in Figure 23 is a separate example. When you enter an example, the screen clears, then displays a chart.

```
      CHARTX (3 7 16) (10 14 8 3 0)

      (⍳3) (⍳5) CHARTX (3 7 16) (10 14 8 3 0) ⍝ Same as above

      (2 3 4) (⍳5) CHARTX (3 7 16) (10 14 8 3 0)

      CHARTX ?⍨⍳⍨5⍴10
```

Figure 23. Examples of Using CHARTX for Free Data

Usage Notes

- You can specify the global variable *FORMNAME* as the name of a predefined chart format. If *FORMNAME* is undefined, or if *FORMNAME* has the value '*', the default format is used. The default format is a line graph with autoscaled axes, default line colors, default axis markers and labels, and so on. If *FORMNAME* is assigned the name of an unknown chart format, then an error message is issued.
- Some facilities available in the ICU chart call are not used by *CHARTX*. These include specification of chart keys, labels, and headings. If you want to use these facilities, you must modify the *CHARTX* function.
- The main purpose of *CHARTX* is to make it easy to generate ICU charts and graphs. Once in the ICU environment, you can modify the chart type and format to suit your needs by using the ICU interactive facilities. Data transferred to the ICU can be displayed in any of the following:
  - Charts
    - Bar
    - Pie
    - Polar
    - Surface
    - Tower
  - Venn diagrams
  - Line graphs
  - Histograms
  - Scatter plots

GDMX

GDMX is a cover function for AP 126, the Graphical Data Display Manager (GDDM) auxiliary processor. *GDMX* offers an easy way to use GDDM, which takes advantage of the general array facilities of APL2 to pass multiple GDDM calls in one *GDMX* call.

The discussion here assumes some knowledge of AP 126 and GDDM. For more information on AP 126, see *APL2/370 Programming: System Services Reference*.

For a detailed discussion of GDDM, see *Graphical Data Display Manager: Base Programming Reference*.

Using GDMX

GDMX has the following calling sequences:

```
CODE GDMX ARG
```

CODE is a simple character vector that is the name of a GDDM call. *ARG* is the argument list appropriate for the GDDM call.

```
      'GSCOL' GDMX 2
```

Sets the GDDM color attribute to red.

```
      'ASCPUT' GDMX 5 6 'CATFAT'
```

or

```
      'ASCPUT' GDMX 5 (⍴T) T←'CATFAT'
```

Fills alphanumeric field 5 with the string 'CATFAT'.

```
       FM←3 5⍴ 1 2 6 1 10 2 13 8 2 14 3 18 9 5 5

       FM
1  2 6 1 10
2 13 8 2 14
3 18 9 5  5
       'ASDFMT' GDMX (⍴FM) FM
```

Formats three alphanumeric fields, using the format matrix FM.

```
CODE1 CODE2...CODEN GDMX ARG1 ARG2 ARG3...ARGN
```

*CODE*1 *CODE*2 **...** *CODEN* are simple character vectors that are the names of GDDM calls; the left argument of *GDMX* is thus a vector, each item of which is a simple character vector. *ARG*1 *ARG*2 **...** *ARGN* are argument lists appropriate for the corresponding GDDM calls.

```
'GSCOL' 'GSLT' GDMX 2 3
```

Sets the GDDM color attribute to red, and sets the line attribute to dash-dot line.

```
      'GSFLD' 'GSWIN' GDMX (1 1 5 7) (0 10 0 20)
```

Defines a graphics field 5 rows deep and 7 columns wide, starting at row 1, column 1, and defines a graphics window within that field with horizontal coordinates 0 to 10 and vertical coordinates 0 to 20.

```
(⊂CODE) GDMX ARG1 ARG2 ARG3 ... ARGN
```

CODE is a simple character vector that is the name of a GDDM call; the left argument of *GDMX* is thus a scalar, the only item of which is a simple character vector. *ARG1 ARG2 ... ARGN* are argument lists, all of which are appropriate for the GDDM call.

```
      (⊂'FSPDEL') GDMX 4 8 12
```

Deletes GDDM pages with page identifiers 4, 8, and 12.

```
GDMX 'ERRTOL' N
```

N is a simple numeric scalar. This call tells *GDMX* to report GDDM errors only if the severity code exceeds *N*.

```
      GDMX 'ERRTOL' 0      ⍝ SET ERROR TOL. TO 0

⍝ SELECT PAGE 99, WHICH DOES NOT EXIST...

      'FSPSEL' GDMX 99
GDDM ERROR. RC= 8
      'FSPSEL' GDMX 99
      ^         ^

      GDMX 'ERRTOL' 8      ⍝ SET ERROR TOL. TO 8

      'FSPDEL' GDMX 99
```

```
GDMX 'MORE'
```

Provides more information about the last GDDM error.

```
      'FSPDEL' GDMX 99
GDDM ERROR. RC= 8
      'FSPDEL' GDMX 99
      ^         ^

      GDMX 'MORE'
FSPDEL  ADM0132 E PAGE 99 DOES NOT EXIST
```

```
GDMX 'TERM'
```

Terminates the current connection with GDDM by retracting and deleting variables shared with AP 126. (See “Global Variables” below.)

```
GDMX A
```

A is a properly formed array for direct AP 126 processing. To be properly formed, the Enlist of *A* (*A*) must be a series of APL numeric call codes and their corresponding GDDM arguments. The first item of *A* must be numeric.

```
GDMX 514 2 424 5 (⍴T) T←'CATFAT'
```

The call code for *GSCOL* is 514; the code for *ASCPUT* is 424. This example sets the GDDM color attribute to red, and fills alphanumeric field 5 with the string *'CATFAT'*.

Global Variables

GDMX creates and uses the following global APL arrays:

DAT_G The AP 126 data variable. *GDMX* automatically creates and shares *DAT_G* as required.

CTL_G The AP 126 control variable. *GDMX* automatically creates and shares *CTL_G* as required.

RET_G The result of calls to AP 126. Each invocation of *GDMX* that calls AP 126 results in the assignment *RET_G←CTL_G DAT_G*.

The calls *GDMX 'MORE'* and *GDMX 'ERRTOL' N* set *RET_G* to an empty character matrix. The call *GDMX'TERM'* deletes *RET_G*.

G_CODES A 2-item vector. The first item is a vector of character vectors giving the names of GDDM calls made by *GDMX*; the second item is a simple numeric vector giving the corresponding call codes. Names and codes are added to this list the first time they are encountered by *GDMX*. Thus the list contains only the names and codes used in a particular workspace.

G_ERRTOL The current setting of the error tolerance. If *GDMX* encounters a GDDM error and *G_ERRTOL* is not defined, *G_ERRTOL* will be initialized to 0.

Usage Notes

- In applications with a single top-level user function, it is generally appropriate to make all global variables used by *GDMX* local to the top-level function, with the exception of *G_CODES*. In applications with more than one top-level function, it is generally appropriate to leave these variables global.
- When making multiple calls to GDDM with *GDMX*, the multiple calls are packaged into a single call to AP 126. So while the expression:

```
CODE1 CODE2 GDMX ARG1 ARG2
```

is equivalent to the expression:

```
CODE1 CODE2 GDMX¨ ARG1 ARG2
```

the former makes only one call to AP 126 while the latter makes two.

- Some GDDM calls require no argument. *GDMX*, however, still requires a right argument for the GDDM argument list. In these cases, code an empty vector for the right argument of *GDMX*. For example:

  ```
  'FSALRM' GDMX ''
  ```

  The terminal beeps when the screen is next updated.

- For some application workspaces, it might be appropriate to have *GDMX* return an explicit result rather than return the result in the global variable *RET_G*. To have *GDMX* return its result explicitly, modify the *GDMX* function header to include *RET_G*. Thus, the first line of *GDMX* would read:

  ```
  [0]  RET_G←CODES GDMX ARGS; ...
  ```

Example

This example lets you draw lines on your screen by moving the cursor around and pressing ENTER. The program draws a line from the current cursor point back to the last location of the cursor. Pressing PF3 quits the program and returns you to APL2.

```
      ∇ DEMO
[ 1]  ⍝ APL2 example using GDMX function
[ 2]   'GSSEG' GDMX 1                  ⍝ Open segment 1
[ 3]   'GSMOVE' GDMX 50 50             ⍝ Initialize current position
[ 4]  LOOP:'ASREAD' GDMX ''            ⍝ Wait for operator action
[ 5]   →(1 3∧.=2↑5↓1⊃RET_G)/END        ⍝ Check for PF3
[ 6]   'GSQCUR' GDMX ''                ⍝ Find where cursor is
[ 7]   'GSLINE' GDMX ¯2↑1⊃RET_G        ⍝ Draw line from previous point
[ 8]   →LOOP                           ⍝ Loop back
[ 9]  END:'GSSDEL' GDMX 1              ⍝ Delete segment 1
[10]  ∇
```

GRAPHPAK—a Vector Graphics Workspace

GRAPHPAK is a comprehensive set of defined functions for drawing pictures on display device screens. It duplicates many presentation graphics feature (PGF) services, and includes many features that PGF does not have (3-dimensional plotting of various kinds, fitting and plotting curves, and drawing organization charts).

One important point: *GRAPHPAK* uses AP 126 to get basic GDDM services, but it does so in a manner different from the *FSM* and *FSC*126 public library workspaces. This means that the *GRAPHPAK* functions that work with the screen *cannot* be combined with GDDM functions from those workspaces. The mathematical functions (those that do curve-fitting, for example) can, however, be used in any other workspace that does not contain similarly-named objects.

The *GRAPHPAK* workspace is fully described in *APL2 GRAPHPAK: User's Guide and Reference*. This manual contains examples and illustrations of how to use *GRAPHPAK* functions. It also contains a list of each user function, a detailed discussion of the function, and the type of arguments each requires.

VS APL Compatible Workspaces

Two obsolete display terminal workspaces, *FSC126* and *FSM* (full-screen manager), are provided only to maintain compatibility with older applications. These workspaces are no longer supported.

FSC126 is used for designing and using full-screen panels. For information on other ways to do this, see *GDDM Interactive Map Definition*, *Application Prototype Environment Guide and Reference*, or AP 124 in *APL2/370 Programming: System Services Reference*.

FSM is superseded by *GDMX*, which uses a much better approach to calling GDDM routines. (For every supported GDDM routine, *FSM* contains a function of the same name that calls it.) *FSM* is not supported for facilities introduced after GDDM Version 1 Release 3. For information on *GDMX* see "Using GDMX" on page 59.

FSC126 Workspace

FSC126 is a functional extension of the VS APL workspaces, *FSC124* and *FULLSCRX*. *FSC126* provides a core set of functions whose names and syntax match those in the VS APL workspaces, but that use AP 126 instead of AP 124. In addition, *FSC126* provides a small set of new functions that make limited use of extended 3270 text handling capabilities of AP 126 and GDDM, not supported by the previous AP 124.

Documentation of the *FSC126* workspace is provided online in the workspace itself.)*LOAD* the workspace and type *ABSTRACT*, *DESCRIBE*, or *HOW*.

FSM Workspace

The *FSM* workspace contains functions that facilitate the use of *Graphical Data Display Manager*, including its presentation graphics feature.

In addition, the workspace contains a function (*SM*) that can be used to process APL2 session manager commands directly or from within APL functions. Its syntax is:

SM command

Where *command* is a character string representing one of the session manager commands (COLUMN, COPY, DISPLAY, HELP, LINE, LOG, PAGE, PFK, PROFILE, and SUPPRESS) followed by a space and the text, if any, required to complete the command.

The *SM* function shares two variables with AP 120 (the session manager auxiliary processor): *CTLSM* and *DATSM*. Data resulting from processing a command is returned in *DATSM*.

The APL2 session manager is described in *APL2/370 Programming: System Services Reference*.

IBM recommends using the *GDMX* workspace instead.

Chapter 5. Environment-Dependent Workspaces

The environment-dependent workspaces are:

```
CMS
TSO
FILESERV
```

Their purpose is to make environment facilities readily available to the APL2 user. They do this by providing defined APL2 *cover functions* for that purpose.

Communicating with the world outside the APL2 workspace requires a series of detailed steps: offering to share one or more variables, checking share status, establishing initial values, checking return codes, and so on. Many of the cover functions in the *CMS* and *TSO* workspaces perform standard sequences of this type. In addition, the *CMS* and *TSO* workspaces contain comprehensive functions like *OPEN*, *CLOSE*, *GET*, *PUT*, *GETFILE*, and *PUTFILE*, for easy use of external files and data sets.

Note: APL2 Version 2 provides several new techniques for file access that are operating-system independent. Application developers should consider whether one of these techniques serves their needs better than the workspaces described here:

- Processor 12 provides direct access to files using APL2 primitives.
- AP 211 provides the ability to store APL2 objects in files by name. This processor is compatible with the AP 211 provided on workstation APL2 platforms.
- Processor 10 external functions ∆*FV* and ∆*FM* provide a simple way to read and write a file as a whole. Compatible functions for these are also available on the workstation APL2 platforms.

For information on these file access techniques, see *APL2/370 Programming: System Services Reference*.

Command, Alternate-Input, and Specialized File APs

Environment dependencies are reflected in three types of auxiliary processors:

- Command
- Alternate-input or *stack*
- Specialized file

These are each discussed in the sections describing individual workspaces. However, one of them, the alternate-input processor, has some unique characteristics that make a preliminary discussion here advisable.

The Alternate-Input Processor

The objective of the alternate-input processor is to replace input from the terminal by preplanned input from a stack.

An example suggests why this might be useful. Using the CMS command processor, you can write an APL function that sorts a CMS file before either bringing it into the workspace or performing some other operation on it. To do this,

execute the CMS SORT command, specifying the file to be sorted and the sorted file to be created.

The CMS SORT program types a message at the terminal, requesting you to specify the fields on which the file is to be sorted, and then unlocks the keyboard so you can type in the numbers identifying the first and last positions of each field. The SORT program then sorts the old file on these fields and creates a new one with the requested name.

In general, it is inconvenient to have the APL function stop execution to request the information needed by the SORT program. The user of the APL function does not, in general, know what fields to specify; it is the *writer* of the program who has this information. Without an alternate-input processor, no one could use an APL function of this kind without knowing what responses are required at various points.

The alternate-input processor eliminates both the interruptions and the need for detailed instructions. When the APL programmer knows the required user responses, they can be stacked. Then, when the executing function requests a line of terminal input, instead of requesting it from the terminal it takes it from the stack, starting with the first stacked line if the stack organization is FIFO (first in, first out) and the last if it is LIFO (last in, first out). No input from the terminal is requested until the stack is exhausted.

Obviously, there is little advantage to using the alternate-input processor except within a defined function. In immediate execution, putting a line on a stack causes it to be immediately executed and removed. This is a waste of time and effort. The line could be entered directly from the keyboard with the same effect. This practical restriction to use within a defined function is one of the ways the alternate-input auxiliary processor differs from the others; most of them can be used effectively by means of immediate-execution keyboard entries.

If you find yourself frequently entering the same sequence of operations from a keyboard, you might find it useful to create a procedure by stacking a set of alternate-input lines to be executed on demand. The *CMS* and *TSO* workspaces each have a function called *PROC* that does this. *PROC* creates a function for the purpose, giving it the name you specify. In particular, this offers a way to execute system commands under program control. The function does the necessary variable-sharing and initializing and stacks the alternate-input lines that make up the procedure. These lines are then be stacked and executed every time you type the name of the created function. The *PROC* function that creates procedures is described in the following section.

The CMS Workspace

The following sections discuss how to use the CMS workspace.

Characteristics of the CMS Environment

As a CMS user, you have your own virtual machine complete with a terminal and one or more virtual readers, punches, printers, tapes, and disks. Virtual disks are parts of real disks that are specially formatted for use by CMS. Because they are usually not complete disks but rather selections of one or more cylinders from real disks, they are sometimes called *minidisks.* Each minidisk has a virtual device address and a mode, denoted by a single alphabetic character followed by a single numeric character. You can use AP 110 to read or write CMS files on minidisks. You can use AP 111 to read and write files formatted for use by other system control programs and also for tapes or virtual readers, printers, and punches.

CMS operates under a control program called CP, which is concerned with the real machine on which several concurrent virtual machines are usually in operation. Both CMS and CP have sets of commands appropriate to the functions each performs. Many of these commands can be executed from an APL2 workspace by the use of AP 100 or the `)HOST` command.

CMS Command, Alternate-Input, and File Processors

The `CMS` workspace contains functions to help you use the command processor (AP 100), alternate-input processor (AP 101), and disk-file processor (AP 110).

The defined functions `CP` and `CMS` can be used to issue host system commands from the active workspace.

Creating APL2/CMS/CP Procedures

Using AP 101, the alternate-input auxiliary processor, is not a straightforward matter. The function `PROC` in this workspace (and in the `TSO` workspace) was written to help you create procedures made up of stacked lines of alternate input. Figure 24 on page 68 gives an example of how to use `PROC`. In the example, `PROC` accepts two lines of input from the terminal:

```
CMS 'LISTFILE * APLWSV2 (E'
WSLIST←GETFILE 'CMS EXEC'
```

and creates a function called `GETWKSPA` that shares a variable with the alternate-input processor, then stacks the two lines of input accepted from the terminal.

The input required to do this and the output message produced by `PROC` are shown in section 1 of the figure.

Section 2 of the figure lists the `GETWKSPA` function created by `PROC`.

Section 3 shows an execution of the `GETWKSPA` function and its result: a return code of 0, indicating successful completion.

Section 4 displays the variable `WSLIST` created by the alternate-input commands stacked by `GETWKSPA`. `WSLIST` is a listing of all the files of type APLWSV2 on the A disk. The file type APLWSV2 identifies CMS files that are stored, private-library APL2 workspaces.

Note: The first alternate-input line executed the CMS command LISTFILE (using the defined function *CMS* to do so). Since the option E was specified, this command caused a file to be created with a filename CMS, filetype EXEC, and filemode A.

The second alternate-input line used the defined function *GETFILE* to read in the CMS EXEC file and give it the name *WSLIST*.

1. Use *PROC* to create function with stacked input commands

```
      PROC 'GETWKSPA'                          Create GETWKSPA procedure
CMS 'LISTFILE * APLWSV2 (E'                    First line of procedure
WSLIST←GETFILE 'CMS EXEC'                      Second line of procedure
                                               Null line to terminate
   GETWKSPA PROCEDURE HAS BEEN CREATED.        Output message from PROC
```

2. List *GETWKSPA* function created by *PROC*

```
      ∇GETWKSPA[□]∇
    ∇
[0]   GETWKSPA;B;C
[1]   B←'CMS(192'
[2]   C←101 □SVO 'B'
[3]   B←'CMS ''LISTFILE * APLWSV2 (E'''
[4]   B←'WSLIST←GETFILE ''CMS EXEC'''
    ∇ 1992-03-27 12.18.01 (GMT-8)
```

3. Execute *GETWKSPA* function (which creates *WSLIST*)

```
      GETWKSPA
0
```

4. List workspaces on A disk by typing *WSLIST*

```
      WSLIST
 &1 &2 ENVWSHDS APLWSV2   A1
 &1 &2 IMFX0323 APLWSV2   A1
 &1 &2 READ121  APLWSV2   A1
 &1 &2 TEMP     APLWSV2   A1
 &1 &2 TEMP2    APLWSV2   A1
 &1 &2 WKSPCNT  APLWSV2   A1
```

Figure 24. Creating and Using AP 101 Functions in CMS

Reading and Writing CMS Disk Files

Disk File AP Functions:

The defined functions in the *CMS* workspace that can help in using AP 110 are shown in Figure 25 on page 69.

CLOSE	Close a file
CLOSEALL	Close all open files
GET	Get a record from an open file
GETFILE	Bring an entire file into the workspace
OPEN	Open a file
PUT	Put a record into an open file
PUTFILE	Write a variable out as a file
RETRACT	Retract shared variables
SHARES	Determine the names of all shared variables

Figure 25. CMS: File Auxiliary Processor Functions

These functions use subfunctions, which are listed below. Most of the supplementary functions are copied from the *UTILITY* workspace.

BOX CHECKNAME DOUBLE INBLANKS MEMBER
OBLANKS PREPARSE2 TYPE VCAT XBLANKS

Using the Functions in CMS

Command Functions

```
A←CMS B
```

B is a character string that contains a CMS or CP command. *A* is a return code.

The command *SUBSET* puts the user in CMS subset mode. In this mode, the CMS command RETURN returns the user to APL2.

The command *CP* puts the user in CP mode. The CMS command BEGIN (B for short) returns to APL2.

```
A←CP B
```

B is a character string that contains a CP command. *A* is a return code. If *B* is a null vector, the user is put into CP mode. To return to APL2, type B.

Alternate Input Function

```
PROC A
```

A is a character string that contains the name of the alternate-input procedure to be created by *PROC*. *PROC* then accepts all nonnull inputs to be stacked by that procedure. When a null input is entered, a function is created and given the name in *A*. Typing this name results in execution of the given lines.

File Functions

```
A←CLOSE B
```

B is a character string that contains a file name. *CLOSE* retracts the shared variables associated with this file, deletes them from the workspace, and removes the file name and processing information from the variables *OPENFILEs* and *PROCACc*.

```
A←CLOSEALL
```

CLOSEALL retracts and deletes all shared variables and initializes the variables *OPENFILEs* and *PROCACc*.

```
A←GET B
```

B is a character string that contains the name of an open file. *A* is the next sequential record from the file. The variable *rcode* contains the return code from each *GET*. A read past the end of the file gives a null vector value for *A* and a value of `12` for *rcode*.

```
A←GETFILE B
```

B is a character string that must contain at least a file name and a file type (separated by at least one blank). These can be followed by a file mode. If a conversion other than `192` is required, this must follow the file name, type, mode specification and be separated from it by the character `'('`.

Note: If your conversion option is the default (192 or EBCDIC), you should use external function *ΔFM*. It is supplied with Processor 10 and is much faster. For more information, see *APL2/370 Programming: System Services Reference*

```
A←B MSG C
```

C is a message number, optionally followed by one or more character strings. *B* is a single number or is empty. If *B* is not supplied, empty, or less than or equal to the global variable *msgw*, then *A* is the message returned by the *AP2WSM* function, called with right argument *C* and left argument *'AP2WCMS'*. (For more information, see “The Message Facility” on page 5.) Otherwise, *A* is a 0-by-0 character matrix.

```
A←B OPEN C
```

B is a character string that contains a file name. It can be the name of a new file. *C* is a string that contains all the other information required to open the file: file type, file mode, and, if necessary, the character '(' followed by fix, access, and conversion options as needed. All *C* specifications are optional. If they are missing, the following default values are used:

SCRIPT	File type
A	File mode
FIX	Fixed length records
U	File can be read or written
192	Full APL2 EBCDIC translation

OPEN opens the file, using shared variables that are named by adding suffixes to the name in *B*. The data variable is identified by the suffix '*d*'; the control variable by the suffix '*c*'. *A* is a return code.

The file name is appended to the global variable *OPENFILEs*. The information contained in input *C* is appended to the global variable *PROCACc*.

```
A←B PUT C
```

B is a character vector. *C* is a file name. *PUT* writes the data in *B* sequentially to the file *C*. *A* is a return code. If an error occurs, then the global variable *rcodes* is also assigned the return code, and the global variable *errdata* is assigned *R*.

```
A←B PUTFILE C
```

B is a character matrix. *C* is a file name. *PUTFILE* writes the character matrix *B* into the file *C*. If the file *C* is not a new file the user is asked whether the output is to be added to the end of the file. The output is performed only if the answer is *YES* or some shortened form of it.

Note: If your conversion option is the default (192 or EBCDIC), you should use external function Δ*FM*. It is supplied with Processor 10 and is much faster. For more information, see *APL2/370 Programming: System Services Reference*

```
A←RETRACT B
```

B is a matrix of names. *RETRACT* retracts shares for all the names in *B* and returns the preexisting coupling values in *A*. In other words, *RETRACT* is a synonym for the system function □*SVR*.

```
A←SHARES
```

A is a matrix of the names of all variables in the workspace that have a nonzero share coupling value.

Input/Output from Peripheral Devices

To communicate with tapes, readers, punches, and printers, you must use the CMS FILEDEF command to specify a name by which the file is to be known, the type of device you intend to use, and any other characteristics (record length, block size, and so forth) that are relevant. You can use CMS FILEDEF before you start to use APL2 or by using the defined function *CMS*.

The TSO Workspace

As a TSO user, you can get at any part of your computing complex that you are authorized to access. In particular, you can read or modify data sets that you may or may not have created, as long as you have not been explicitly denied access to them.

You do your work in TSO by executing TSO commands or collections of commands called CLISTs. See *TSO Terminal User's Guide* for a general discussion of TSO commands and CLISTs.

TSO Command, Alternate-Input, and File Processors

The auxiliary processors you can use to interact with the TSO environment are:

AP 100	The TSO command processor
AP 101	The alternate-input processor
AP 210	The BDAM auxiliary processor
AP 111	The QSAM auxiliary processor

User functions in *TSO* are divided into three groups:

Command AP Function: *TSO*

This function is for executing TSO commands without leaving APL2.

Alternate-Input AP Function: *PROC*

This function stacks alternate-input lines for later execution when keyboard input is requested. See the discussion of Figure 24 on page 68.

File AP Functions and Auxiliary *TSO* Functions

ALLOCATE	Allocate a data set
ATTRIBS	Establish data set attributes
CLOSE	Close a file
CLOSEALL	Close all open files
GET	Get a record from an open file
GETFILE	Bring an entire file into the workspace
OPEN	Open a file
PUT	Put a record into an open file
PUTFILE	Write a variable out as a file
RECID	Specify a record number for direct-access use
RETRACT	Retract shared variables
SHARES	Determine the names of all shared variables

Figure 26. TSO: File Auxiliary Processor Functions

The file functions use subfunctions, which are listed below. Most of these are copied from the *UTILITY* workspace.

DOUBLE INBLANKS INDEX LADJ MEMBER OBLANKS PREPARSE2 TYPE VCAT XBLANKS

The functions *apAI*, *apCMD*, *apFILE*, and *apQSAM* return the numerical values that identify the alternate input, command, direct-access and QSAM auxiliary processors. They should be modified to reflect local usage if different from the default values used in the APL2 workspace.

Using the Functions in TSO

Command Function

```
A←TSO B
```

B is a character string that contains a TSO command. *A* is a return code. If it is not 0 (successful completion), it is accompanied by a message.

Alternate Input Function

```
PROC A
```

A is a character string that contains the name of the alternate-input procedure to be created by *PROC*. *PROC* then accepts all nonnull inputs to be stacked by that procedure. When a null input is entered, a function is created and given the name in *A*. Typing this name results in execution of the given lines.

File Functions

```
A←B ALLOCATE C
```

B is a file name (ddname). *C* contains a data set name which can be followed by one or more blanks and a list of allocation specifications. (See *TSO Extension Command Language Reference Manual*.) *ALLOCATE* uses the *TSO* function to do the requested allocation. *A* is the return code from *TSO*.

```
A←B ATTRIBS C
```

B is a file name. *C* is a list of data-set attributes. *ATTRIBS* uses the *TSO* function to associate the attributes of *C* with the attribute file *B*. The name *B* can then be preceded by the characters *'USING'* as part of an allocation command.

```
A←CLOSE B
```

B is a character string that contains a file name. *CLOSE* retracts the shared variables associated with this file, deletes them from the workspace and removes the file name and processing information from the variables *OPENFILEs* and *PROCACc*. *A* is a return code.

```
A←CLOSEALL
```

CLOSEALL retracts and deletes all shared variables and initializes the variables *OPENFILEs* and *PROCACc*. *A* is a return code.

```
A←GET B
```

B is a character string that contains the name of an open file. *A* is the next sequential record from the file. The variable *rcode* contains the return code from each *GET*. A read past the end of the file gives a null vector value for *A* and a value of 12 for *rcode*.

```
A←GETFILE B
```

B is a character string that contains the name of an existing OS data set. The entire data set is read in (using conversion option 192) and is returned as the value *A*.

```
A←B OPEN C
```

B is a character string that contains a name to be used as the internal name of the file being opened. In TSO terms, it is used as a FILE or ddname. This is the name that is meant when the term *file name* is used in describing the *GET*, *PUT*, *GETFILE*, *PUTFILE*, *CLOSE*, and *RECID* functions.

C is a character string that can contain, as needed, a data set name and a variety of options: disposition, keep, conversion, type of access, and auxiliary processor. The default options are, respectively: *OLD*, *KEEP*, 192, *R*, and *apQSAM*. The options must follow the data set name (if present). If the data set name is omitted, then the options must be preceded by the character '('.

OPEN opens the file, using shared variables that are named by adding suffixes to the name in *B*. The data variable is identified by the suffix '*d*'; the control variable by the suffix '*c*'.

The file name is appended to the global variable *OPENFILEs*. The auxiliary processor identification and access type are concatenated and appended to the global variable *PROCACc*.

The data set name need not be specified if it is already allocated and associated with file name *B*. This can be done:

- Before APL2 is activated
- By direct use of the *TSO* function
- By use of the *ALLOCATE* function

Specification of various data set attributes can require use of the *ATTRIBS* function (described above). The permissible options are named in global variables whose names start with the letters *opts*. These are:

Global Variable	Permissible Options
optsACC	*R*, *W*, *U*, *READ*, *WRITE*, *UPDATE*
optsALCD	*OLD*, *SHR*, *MOD*, *NEW*, *SYSOUT*
optsALCK	*KEEP*, *DELETE*, *CATALOG*, *UNCATALOG*
optsAP	*apFILE*, *apQSAM*, 111, 210
optsCONV	*BYTE*, *EBCD*, *TN*, *BCD*, *APL*, *BIT*, *VAR*, 192,

```
A←B PUT C
```

B is a character vector. *C* is a file name. *PUT* writes the data in *B* sequentially to the file *C*.

```
A←B PUTFILE C
```

B is a character matrix. *C* is a data set name. *PUTFILE* writes the character matrix *B* into the data set *C*. If *C* is already the name of a data set, the old data set is lost.

```
A←B RECID C
```

B is a character string that contains a file name. *C* is an integer. *RECID* returns the file name as the value *A*, and specifies *C* as the number of the record to be read or written by a *GET* or *PUT*. *RECID* should be used for direct-access input-output operations.

GET fn RECID no

Gets record number *no* from file *fn*. *PUT* works the same way.

```
A←RETRACT B
```

B is a matrix of names. *RETRACT* retracts shares for all the names in *B* and returns the preexisting coupling values in *A*. In other words, *RETRACT* is a synonym for the system function ⎕*SVR*.

```
A←SHARES
```

A is a matrix of the names of all variables in the workspace that have a nonzero share coupling value.

The FILESERV Workspace

FILESERV	Performs imports and exports
VCAT	Aids in constructing the *SYSIN* variable
SYSIN	Contains import and export commands
MORE	Contains error messages

Figure 27. TSO: User Functions and Variables in the FILESERV Workspace

The *FILESERV* workspace allows you to transport APL data files into and out of TSO APL data file libraries. It is distributed only with TSO. It can be used to provide backups, to interchange data with other APL2 systems, or to migrate data from VS APL systems.

APL data file libraries in a TSO system are implemented using keyed VSAM clusters. There is one VSAM cluster for each library of files. (Files are identified to AP 121 by library number and file name.) This workspace can *EXPORT* files from the VSAM cluster, creating sequential files; and can *IMPORT* sequential files into the library. These services are analogous to the *EXPORT* and *IMPORT* commands provided by MVS access method services.

The sequential files are in a form compatible with:

- AP 121 CMS files used by APL2 or VS APL
- The VS APL TSO *FILESERV* workspace
- The VS APL CICS* service program
- The VSPC service program.

The *FILESERV* function can perform a series of imports and exports in any combination and using any APL data file libraries. It is driven by either a SYSIN file or the contents of a *SYSIN* variable. The syntax of data in the SYSIN source is described in “Transporting Files in Batch Mode” on page 79.

Exporting Files Interactively

Three steps are required to export files interactively; the first is frequently done automatically during APL2 start up.

1. Allocate the file library or libraries that contains the APL data files to be exported. If this is not done by the CLIST that invokes APL2, use a TSO ALLOCATE statement:

   ALLOC F(F*libno*) DA(*cluster*)

   where:

   libno A library number, with no leading zeros.

   Note: The *FILESERV* workspace does not recognize F0 as a private library. You must allocate the library with some other number (such as F1001) to be able to export from it.

   cluster The data set name of a VSAM cluster, using TSO data set name conventions.

2. Allocate the sequential files to which the APL data files are to be copied. The format of this command varies depending on the device type and whether the

data set already exists. Consult TSO command language references for details. The following model can be used to create a new DASD data set:

```
ALLOC F(fname) DA(dsname) RECFM(V,B) -
    LRECL(4096) SPACE(prim,sec) TRACKS
```

where:

fname A ddname used in the *TO* parameter of the *EXPORT* statement. For simplicity, use the APL data file name as an *fname*. The *TO* parameter is not required in this case.

dsname The data set name to be used for the sequential file. You might want to include the *fname* as a part of this.

prim,sec Primary and secondary (overflow) space allocation on the device.

3. Set up *SYSIN* and invoke the *FILESERV* function.

   a. Invoke APL2.

   b.)*LOAD 2 FILESERV*

   c. Assign one or more commands to the *SYSIN* variable. *SYSIN* can contain a character vector or matrix. Each row of a matrix is a separate command, unless it ends with a plus (+) or minus (-) sign (as described in "Format of Commands" on page 80). The *VCAT* function can help you in constructing this matrix. For example:

```
SYSIN←'EXPORT 2361 DEPT17'
SYSIN←SYSIN VCAT 'EXPORT 2361 DEPT18'
```

   d. Invoke the *FILESERV* function. It takes no arguments.

Importing Files Interactively

To import files interactively, as many as four steps may be needed, though the first step is usually not necessary, and the second is frequently done automatically during APL2 invocation.

1. Define any new file library or libraries you are using. See the discussion on creating APL2 VSAM libraries in *APL2/370 Programming: System Services Reference* for details. You might be able to define a library having the same attributes as an existing library:

   ```
   DEFINE CL(NAME(new.lib) MODEL(old.lib) )
   ```

   where:

   new.lib The data set name of the library you are creating.

   old.lib The data set name of an existing APL2 file library.

2. Allocate the file library or libraries that you are importing APL data files into. If this is not done by the CLIST that invokes APL2, use a TSO ALLOCATE statement:

   ```
   ALLOC F(Flibno) DA(cluster)
   ```

   libno A library number, with no leading zeros.

   Note: The *FILESERV* workspace does not recognize F0 as a private library. You must allocate the library with some other number (such as F1001) in order to import into it.

cluster The data set name of a VSAM cluster, using TSO data set naming conventions.

3. Allocate the sequential files from which the APL data files are to be copied:

   ALLOC F(*fname*) DA(*dsname*) SHR

   fname A ddname used in the *FROM* parameter of the *IMPORT* statement. For simplicity, use the APL data file name as an *fname*. The *FROM* parameter is not required in this case.

   dsname The name of the data set that contains the file.

4. Set up *SYSIN* and invoke the *FILESERV* function.

   a. Invoke APL2.

   b.)*LOAD 2 FILESERV*

   c. Assign one or more commands to the *SYSIN* variable. *SYSIN* can contain a character vector or matrix. Each row of a matrix is a separate command, unless it ends with a plus (+) or minus (-) sign (as described in "Format of Commands" on page 80). The *VCAT* function can help you in constructing this matrix. For example:

   ```
   SYSIN←'IMPORT DEPT17 TYPE(DIRECT)'
   SYSIN←SYSIN VCAT 'IMPORT 1018 DEPT18 TYPE(DIRECT)'
   ```

   d. Invoke the *FILESERV* function. It takes no arguments.

Transporting Files in Batch Mode

Batch processing is essentially the same as interactive processing, except that:

- You provide MVS job control language used to start a TSO session.
- Data sets are allocated using either JCL or TSO ALLOC statements that are in a SYSIN stream.
- APL2 is invoked by a command in the SYSIN stream.
- The workspace is loaded and the *FILESERV* function is invoked by using either the INPUT parameter on the APL2 invocation or an APLIN stream.
- The *FILESERV* commands are normally provided in a SYSIN stream rather than assigned to a *SYSIN* variable.

This is an example of a *FILESERV* jobstream. The JOB control statement, PROFILE and ALLOC statements, and the *EXPORT* and *IMPORT* commands have to be modified to meet the requirements of your installation and to provide the particular services you need:

```
//APLBAT  JOB (DEPT18),APLADMIN
//BATCH  EXEC PGM=IKJEFT01,REGION=2000K,DYNAMNBR=50
//SYSPRINT DD SYSOUT=*,DCB=LRECL=133
//SYSTSPRT DD SYSOUT=*
//APLPRINT DD SYSOUT=*
//SYSTSIN  DD *
  PROFILE PREFIX(APLADMIN)
  ALLOC F(F1001)  DA(PRIVATE.APLLIB)
  ALLOC F(F2361)  DA('PROJ2361.APLLIB')
  ALLOC F(F1018)  DA('DEPT18.APLLIB')
  ALLOC F(DEPT17) DA(SAVE.DEPT17) RECFM(V,B) -
                  LRECL(4096) SPACE(5,5) TRACKS
  ALLOC F(DEPT18) DA(SAVE.DEPT18) RECFM(V,B) -
                  LRECL(4096) SPACE(5,5) TRACKS
  APL2 SH(60K) WS(512K) INPUT(')LOAD 2 FILESERV' 'FILESERV')
//SYSIN    DD *
  EXPORT 2361 DEPT17
  EXPORT 2361 DEPT18
  IMPORT      DEPT17  TYPE(DIRECT)
  IMPORT 1018 DEPT18  TYPE(DIRECT)
//
```

Format of Commands

The first field in the command is the command name; it identifies the action to be performed. The command and its operands can be typed between columns 1 and 72; columns 73 through 80 are ignored. Commands and operands are separated by blanks or commas. (An operand that ends with a right parenthesis does not need to be separated from a following operand.)

To specify that a command is to be continued to the next line or record, use either a plus (+) or minus (-) sign.

1. A minus sign, when used as the last nonblank character in a line or record, indicates that the command is continued with the first character of the next line or record:

   ```
   IM-
     PORT
   ```

   is the same as:

   ```
   IM   PORT
   ```

   The minus sign indicates that separators (blanks in the preceding example) should be included in the continuation of the line or record.

2. A plus sign as the last nonblank character in a line or record indicates that the command is to be continued with the first nonseparator character of the next line or record:

   ```
   IM+
    PORT
   ```

   is the same as

   ```
   IMPORT
   ```

   The blanks are omitted.

Comments to Commands

You can replace any of the separators (blanks or commas) with a comment. A comment must begin with the characters /* and end with the characters */. To specify that a comment be continued on the next line or record, use a plus or minus sign, or end the first line or record with */ and begin the next with /*. For example:

```
/*THIS COMMENT*/
/*HAS*/
/*THREE PARTS*/
```

Using the EXPORT and IMPORT Commands

You can use the *FILESERV* EXPORT and IMPORT commands to transport AP 121 data files to and from TSO.

The syntax of the EXPORT command is:

EXPORT [*nnnnnnn*] *file name*/[*pass*
TO(*ddname*)

where:

nnnnnnn The file library number. The default file library number is 1001.

file name The name of the APL data file to be transported from TSO.

password The VSAM password of the file library data set, if necessary.

ddname The ddname of the sequential file to which the APL data file is to be copied. If the TO keyword is omitted, *file name* is used.

To transport an APL data file with the file name WORK (in a password-protected file library with the library number 1002) to a sequential file with a ddname of WORK, issue the following command:

```
SYSIN←'EXPORT  1002 WORK/PW'
FILESERV
```

The syntax of the IMPORT command is:

IMPORT [*nnnnnnn*] *file name*/[*pass*
 FROM(*ddname*)
 [TYPE(SEQUENTIAL|DIRECT)]
 [NOREPLACE|REPLACE]

nnnnnnn	The file library number. The default library number is 1001.
file name	The name of the APL data file.
password	The VSAM password of the file library, if necessary.
ddname	The ddname of the sequential file to be transported to TSO. If the FROM keyword is omitted, *file name* is used.
SEQUENTIAL|DIRECT	Specifies the type of file to be created. SEQUENTIAL is the default.
NOREPLACE|REPLACE	Specifies that the file can or cannot replace a file with the same name. NOREPLACE is the default.

To transport a sequential file with a ddname of WORK to an APL direct data file with the same name (in a file library with the library number of 1002), issue the following command:

```
SYSIN←'IMPORT 1002 WORK/PW TYPE(DIRECT)'
FILESERV
```

Error Handling

If an error code (other than zero) is returned from an auxiliary processor, the appropriate error message is printed and the processing stack is cleared. It might be necessary to delete shared variables before calling *FILESERV* again. The *RETRACTALL* function does this.

As a debugging aid, the variable *clearsw* can be set to 0 instead of 1. Then, after printing the auxiliary processor error message, processing is terminated with a *SYNTAX ERROR* in the function *vchk*. It is recommended that *vchk* be locked in actual use so a *DOMAIN ERROR* points to *vchk*.

Special Handling of Selected Errors

An option is provided that allows applications to handle selected error return codes without the function printing any error messages, and without suspending processing.

To use this facility, insert a vector of AP return codes into the global variable *pcodes*. Special programming in the application enables it to handle the return codes. This workspace deals with 2-element return codes, but maps the two elements into one by using `100000⊥`*COD*. The values in *pcodes* use that mapping. The actual return code given by the auxiliary processor can always be found in the global variable *rcode*, which can be referenced by the application.

If any function receives a return code from the auxiliary processor contained in *pcodes*, that function terminates with the actual return code in *rcode*. For functions that give an explicit result, the result is the empty vector.

The default value of *pcodes* is the empty vector. Do not store a zero in *pcodes* under ordinary circumstances.

FILESERV Groups

The following groups exist in the workspace:

GPFILESERV	Contains functions and variables specific to this workspace. It must be augmented by other groups to be useable.
GPDESC	Contains descriptive information about the workspace.
GPMESSAGE	Contains functions for message handling.
GPUTILITY	A set of utility functions, many of them from the *UTILITY* workspace.
GPVSAM	A set of functions and variables taken from the *VSAMDATA* workspace.

Chapter 6. File Auxiliary Processor Workspaces

The file auxiliary processor workspaces are:

APLDATA APL files
VSAMDATA VSAM data sets
VAPLFILE Compatible with the *APLFILE* workspace from VS APL

They have many similarities. The most important is that they specify the file or data set to which subsequent input/output functions are to apply by means of the function *USE*.

Notes:

1. The CMS workspace and the TSO workspace also have functions for handling native (CMS or TSO) files.
2. APL2 Version 2 provides several new techniques for file access that are operating-system independent. Application developers should consider whether one of these techniques serves their needs better than the workspaces described here:
   - Processor 12 provides direct access to files using APL2 primitives.
   - AP 211 provides the ability to store APL2 objects in files by name. This processor is compatible with AP 211 provided on workstation APL2 platforms.
   - Processor 10 external functions ΔFV and ΔFM provide a simple way to read and write a file as a whole. Compatible functions for these are also available on the workstation APL2 platforms.

For information on these file access techniques, see *APL2/370 Programming: System Services Reference*.

The APLDATA Workspace

This workspace assists in the use of the AP 121 format auxiliary processor.

ACREATE	Creates an APL file
DROP	Purges or deletes a file
USE	Specifies currently-selected file
AREAD	Reads from an APL file
AWRITE	Writes to an APL file
AGET	Reads from an APL file
ASET	Writes to an APL direct file
CLOSE	Closes currently-selected file
FILESIZE	Changes file size
AT	Makes referenced file the currently-selected file
SETRECLEN	Sets record length
RETRACTALL	Terminates sharing of global variables
SIZE	Returns space required by variable
STORE	Stores variable on a file of the same name
RETRIEVE	Retrieves variable from the named file

Figure 28. Groups in the APLDATA Workspace

Reading and Writing Files of APL2 Arrays

The APL Format auxiliary processor is designed for storing APL2 arrays in their internal form, including their type and structural characteristics. It is intended that such files be read and written only by this auxiliary processor.

General Operation

Because many applications use only a single file, the file name is not generally an argument of these functions. Rather, it is stored as a character vector in the variable *fn* by the *ACREATE* or *USE* function.

Also, the dyadic function *AT* puts a file name in *fn* as in the context:

R←*AGET filename AT I*

(*filename AT I*) *ASET A*

A ***filename*** is defined as:

[*libno*] *name* [*:password*]

Where:

libno A library number. Its significance in each system is:

- CMS Filetype, in the form F*nnnnnnn* where *nnnnnnn* is a 7-digit representation of the library number.
- TSO The library number associated with a VSAM data set that contains files in a form accessible to the auxiliary processor. This library number must be the ddname of an allocated data set. It should have the form L*nnn* where *nnn* is the library number represented without leading zeros.

name A string of up to 8 characters starting with a letter and continuing with letters or numerical digits.

: A separator that is required only when a password is used.

password In some systems, has less restrictive rules of formation than does *name*, but all systems accept passwords formed in the way described above for *name*. Its significance in each system is:

CMS The link password of the disk on which the data set is stored. This can be a READ password if the processing is to be read-only; it must be a WRITE password for writing or updating.

TSO The password of the VSAM data set that contains the auxiliary processor files.

APL-Format File Functions

```
TS ACREATE filename
```

ACREATE creates an APL file. *TS* is a character vector beginning with '*S*' for a sequential file. If it begins with something else, such as '*D*', a direct file is assumed. If *TS* ends with a number, it indicates the file size in bytes (for example, *TS* might be '*S*750000'). Otherwise, the default file size for that user is assumed. *fn* is a character vector that contains the file name as defined above.

There is no explicit result. A file is created and the file name is stored in *fn*.

```
DROP filename
```

DROP purges or deletes the specified APL or EBCDIC file. *filename* is a character vector that contains a file name. There is no explicit result.

```
USE filename
```

filename is a character vector that contains a file name. There is no explicit result. The contents of *filename* are stored in the global variable *fn* and thus the referenced file becomes the currently-selected file. Applications involving multiple files can use this function between uses of file access functions.

```
R←AREAD
```

AREAD reads from an APL file. *R* is the retrieved record. The currently-selected file (as defined in *fn*) is opened (if not already open) for sequential read, and the next record is read. When the file is first opened the next record is the first record in the file. If the result is an empty vector, then the end of file has been reached and the file is closed. To close the file before reaching the end, use the function *CLOSE*.

```
AWRITE A
```

AWRITE writes to an APL file. *A* contains the value to be written. If the value of *A* is an empty vector then it is not written, and the file is closed.

There is no explicit result. The currently-selected file (as defined in *fn*) is opened (if not already opened) for sequential write, and the contents of *A* are written at the present end of the file. If writing the first record in a newly-created DIRECT file then the fixed record length is established by the contents of *rl*, which is taken as 4054 or as previously set (perhaps by the use of the function *SETRECLEN*).

```
R←AGET I
```

AGET reads from an APL direct file. *I* is the record number, assuming the first record is number 1 (that is, origin 1). *I* must be an integer. If *I* is an empty vector, then the file is closed. *R* is the retrieved record.

```
I ASET A
```

ASET writes to an APL direct file. *I* is the record number, assuming the first record is numbered 1. *I* must be an integer that identifies a previously-written record. If *I* is an empty vector, then the file is closed and the contents of *A* are ignored. *A* contains the record to be updated.

There is no explicit result. Note that this function is used for updating existing records, not for extending the file.

```
CLOSE
```

The currently-selected file is closed and its associated shared variables are deleted. There is no explicit result.

```
FILESIZE NEWSIZE
```

NEWSIZE contains the numeric or character representation of the new file size in bytes. The actual size is rounded up to the next highest multiple of 1000 bytes. There is no explicit result. The size of the currently-selected APL file is changed. If reduced below the current size of its contents, then an error message appears.

```
R←filename AT I
```

filename is a character vector that contains a file name. *I* is usually a file index or key.

R is *I*. The implicit result is that the *filename* is stored in global variable *fn*, thus making the referenced file the currently-selected file. Typical use is in a multifile application in the context: *AGET filename AT I*.

```
SETRECLEN L
```

L is a numeric scalar or single element vector giving a record length in bytes. There is no explicit result. The value of *L* is stored in the global variable *rl*. *rl* is used when writing the FIRST record into a new APL direct file.

```
RETRACTALL
```

All sharing of nonshadowed variables in the workspace is terminated. There is no explicit result.

```
R←SIZE VNAME
```

VNAME is a character vector that contains the name of a variable in the workspace. *R* is the number of bytes of space required if the variable were stored in a file. This includes an allowance for an internal header, which is stored with and describes the variable. This function is especially useful in determining the required record length when setting up an APL direct file. (This function is also used by the *STORE* function in this workspace.)

Functions to Store and Retrieve Large Variables

```
STORE VAR
```

An APL sequential file is created with the same name (up to 8 characters) as the variable in *VAR*. If such a file already exists, it is dropped and recreated with an appropriate size. The file is opened and the contents of the variable in *VAR* are written as the single record in the file. The variable named in *VAR* is then deleted from the workspace. *VAR* is a character scalar or vector that contains the name of a variable. The variable name must not contain underscored letters. There is no explicit result.

```
RETRIEVE VAR
```

The APL sequential file with the same name (up to 8 characters) as the variable in *VAR* is opened and a record is read. The contents of the record are stored in the variable named in *VAR*. *VAR* is a character scalar or vector that contains the name of a variable. The variable name must not contain underscored letters, and should be the name of a variable which was previously stored with the *STORE* function. There is no explicit result.

Using the Project, Private, and Public Libraries

The names of variables being stored and retrieved can occasionally conflict with workspace and file names in the user's library and hence cause a problem. To avoid this conflict you can establish a library that can be used by the *STORE* and *RETRIEVE* functions.

To use the project library, you should store the library number in the global variable *objlib*. To use the private library, store a zero in *objlib*. (Note that *objlib* is by default an empty vector.) If *objlib* has a public or project library number, the library number is used to access the proper data set.

Error Handling

If an error code (other than zero) is returned from an auxiliary processor, then the appropriate error message is printed, all nonshadowed shared variables are retracted, and the execution stack is cleared.

A debugging option is provided if the variable *clearsw* is set to zero instead of one; after printing the auxiliary processor error message, execution is terminated with a *SYNTAX ERROR* in the function *chk*. It is recommended that *chk* be locked in actual use so a *DOMAIN ERROR* points to *chk*.

Special Handling of Selected Errors

Another option provides for applications to handle selected error return codes by special programming without the functions printing any error messages and without suspending execution.

To use this facility, the application should insert into the global variable *pcodes* a vector of AP return codes, which the application handles by special programming. The actual return code given by the auxiliary processor can always be found in the global variable *rcode*, which can be referenced by the application.

If any function receives a return code from the auxiliary processor that is contained in *pcodes*, then that function terminates with the actual return code in *rcode*. For functions that give an explicit result, this result is the empty vector.

The default value of *pcodes* is the empty vector. Do not store a zero in *pcodes* under ordinary circumstances.

APLDATA Groups

GPAPL	The name of the group that contains objects related to the support of APL format files.
GPDESC	Contains descriptive information about the workspace.
GPMESSAGE	Contains functions for message handling; it is a subset of most other groups.
GPREADAPL	The name of the group, which is a subset of *GPAPL* and contains objects needed for read-only access to APL files.
GPSTORET	The name of the group that contains the objects related to the *STORE* and *RETRIEVE* functions.

The VSAMDATA Workspace

This workspace helps you use the auxiliary processor (usually AP 123) that provides access to VSAM (virtual storage access method) data sets. This auxiliary processor can be used to access key-sequenced, entry-sequenced, and relative-record data sets. The data set must be preallocated (and defined to the APL2 system) at initialization time, or in TSO, the command auxiliary processor can be used to do this. VSAM records are brought into the APL2 workspace as character vectors. If data conversion is necessary, the group *GPDATACV* can be copied from the *UTILITY* workspace. The variable *HOWDATACV* in the *UTILITY* workspace contains a description of the data conversion functions and instructions on how to use them.

For the types of VSAM data sets described above, this workspace provides defined functions that use the shared-variable interface to do sequential reading and writing, direct reading and writing, updating, erasing individual records or entire files, positioning a pointer for subsequent sequential operations, and retrieving the key of the last record processed.

The records read from or written to VSAM data sets are always character vectors. For externally-generated VSAM data sets,this means that records read from the data set might require translation to make them intelligible within the APL2 workspace. Similarly, updates made to such records should be translated to the external form before they are written to the data set. The functions in *GPDATACV* can be used to do this translation. See “GPDATACV: Converting between External and Internal Representations” on page 33 for a description of these functions.

Functions or variables in this workspace that have names that contains underlined characters are executed by user functions. They are not ordinarily executed or used directly by the workspace user.

Each input/output function uses *⎕SVO* to check the share status of the shared variables used by the currently-active file, issuing a shared-variable offer if they are not currently shared.

File Naming Conventions

The name of the data set or *file* in current use is stored in the global variable *fn*, whose value can be changed explicitly by the *USE* function or implicitly by the *AT* function. For more information, see pages 86 and 87.

A file name is defined as:

filename:password

filename The name of the VSAM data set

password The VSAM password needed if the data set is password-protected and the **':'** is required only when the password is required.

Functions to Access External VSAM Files

This section discusses the functions you can use to access external VSAM files.

Note: When translations are needed between external and internal representations, the functions in *GPDATACV* should be used. (These can be found in the *UTILITY* workspace.)

```
option USE filename
```

filename is a character vector that contains a file name. *option* is a translation option. (This argument is optional.) There is no explicit result. If *option* is not 0, 1, or 2, then a *DOMAIN ERROR* results. Otherwise, the *filename* is stored in the global variable *fn*, and thus the referenced file becomes the currently-selected file. Applications involving multiple files can use this function between uses of file access functions. If the optional left argument is supplied, then it specifies the AP 123 translation option. These options are limited. In general, you need to use the group *GPDATACV* in the *UTILITY* workspace for data conversion, as mentioned above.

The conversion option is specified by *option* as follows:

0 Byte conversion (default)
1 VS APL conversion
2 APL2 conversion

Options 0 and 2 are identical in APL2. The global variable *translate* is assigned the character vector '*T*', '*T*1', or '*T*2', depending on the value of *option*. If no translation option is selected, then *translate* is set to '*T*'. If the new translation option is different from the old, then all shared variables are retracted.

```
R←VREAD
```

R is the retrieved record, which is always an APL character vector. The currently-selected file (as defined in *fn*) is opened for reading, if not already open. The next record in sequence is read. the file is first opened, the next record is the first record. You can change the position of the next record in a key-sequenced file by using the *VPOSITION*, *VGET*, *VGETHOLD*, *VERASE*, or *VSET* functions. If the result is an empty vector, then the end of file has been reached and the file is closed. To close the file prior to reaching the end, use the function *CLOSE*.

```
R←VREADHOLD
```

This function is similar to *VREAD* except that the file is opened for update, and a HOLD is placed on the record (technically, on the VSAM control interval), thus preventing other users from issuing a READ HOLD (or READ for UPDATE). The HOLD is released by the next READ or WRITE operation or by closing the file. *VREADHOLD* is used in conjunction with *VSET* for updating existing records in a file.

```
R←VGET KEY
```

The nature of *KEY* depends upon the type of data set being accessed.

Key-sequenced data sets: The key required to access key-sequenced data sets is a character vector that contains the VSAM key for the desired record. The key must not be longer than the key length of the defined file. (In some implementations it *must* be the *same* length). If it is shorter, then only that many characters are compared with the key in the record, starting at the left. It must match the bit pattern of the key in the file.

Entry-sequenced data sets: For the purposes of keyed access, an entry-sequenced data set is treated as one long string of characters. The *KEY* of a given record is the address of its first character, starting with 0 as the address of the first record and proceeding thereafter in increments of record lengths (fixed or variable).

Relative-record data sets: A relative-record data set consists of a number of fixed-length slots. The *KEY* in this case is the number of the slot.

The numerical values required for entry-sequenced and relative-record data sets can be entered either as character strings or as single numbers.

R is the retrieved record. This is always an APL character vector. The currently-selected file is opened for reading (if not already open).

```
R←VGETHOLD KEY
```

Similar to *VGET* except that the file is opened for update and a HOLD is placed on the record (technically, on the VSAM control interval), thus preventing others from issuing a READ HOLD (or READ for UPDATE). The HOLD is released by the next READ or WRITE operation or by closing the file. *VGETHOLD* is used in conjunction with *VSET* for updating existing records in a file.

```
VSET A
```

A is a character vector and represents the record to be written into the file. For entry-sequenced and relative-record data sets, the record is written at the end of the file or into the position determined by a prior *VGETHOLD*, *VREADHOLD* or *VPOSITION*. For a key-sequenced file, the key must be appropriately imbedded in the record. No translating is performed (except as specified by the *USE* function) and *A* must be constructed using data conversion functions as appropriate.

There is no explicit result. The currently-selected file is opened for writing (if not already open for writing or updating). The contents of *A* are written into the currently-selected file. If updating an existing record, then the previous operation must have been a *VREADHOLD* or a *VGETHOLD* of the same record. In updating an entry-sequenced file, the new record must not be longer than the record being replaced.

```
VERASE KEY
```

KEY is a character vector that contains the key of the record to be erased. There is no explicit result. The currently-selected file is opened for updating (if not already open for updating). The referenced record in the currently-selected VSAM key-sequenced or relative-record data set is erased. Entry-sequenced records cannot be erased.

```
VPOSITION KEY
```

KEY is a character vector that contains the key of the chosen record. If *KEY* is an empty vector, then the first record in the file is selected.

There is no explicit result. The currently-selected file is opened for reading (if not already open). A pointer is set at the beginning of the selected record and provides a starting point for the next sequential operation.

If there is no match on the key, the pointer is positioned to the record with the next higher key and a RECORD NOT FOUND error code is returned. In its default form, *VPOSITION* ignores the return code and returns the key of the next higher record. To change this so that failure to get an exact match is treated as an error, change statement number 6 in the *VPOSITION* function so that it reads:

```
[6] GESW←0
```

This causes an error interrupt if no exact match is found for *KEY*. Changing the statement to '*GESW*←1' restores the original condition; the return code is ignored, and the next higher key is returned if there is no exact match.

```
KEY←VKF    (Key Feedback)
```

VKF is a niladic function that returns the key of the record just processed or to which the file has just been positioned. This key is a character vector that contains the type of information appropriate to the data set:

- A relative-byte address for an entry-sequenced data set
- The imbedded key for a key-sequenced data set
- The relative-record number of a relative-record data set

```
VCLEAR A
```

A must be a character string that contains a file name. This file is made the currently selected file; that is, its name is stored in the global variable *fn*. It is closed for other operations and opened for CLEAR. This means that all existing records are deleted and the file opened for writing. (The file must be specified as a reusable VSAM data set.) There is no explicit result.

```
KEY VWRITE A
```

This write-with-key command is valid only for relative-record files. *KEY* must be the relative-record number of an empty slot in the file. *A* is the record (a character vector) to be written into that slot. There is no explicit result.

```
CLOSE
```

The currently-selected file is closed and its associated shared variables deleted. There is no explicit result.

VSAMDATA Groups

GPVSAM The name of the group that contains the functions and variables used to access VSAM files.

GPREADVSAM The name of the group that is a subset of *GPVSAM* and that contains objects needed for READ ONLY access to external VSAM files.

GPDESC Contains documentation variables.

GPMESSAGE Contains functions for message handling; it is a subset of most other groups.

The VAPLFILE Workspace

This workspace is retained only for compatibility with earlier APL systems.

The following sections describe a set of functions you can use to create and use a file of simple, homogeneous APL arrays. The functions are most useful when a file contain APL arrays of arbitrary rank and dimension, when variable-length records need to be accessed randomly, or when records are longer than the maximum length otherwise permitted.

VAPLFILE uses AP 121 as supplied with APL2. It is designed to be compatible with the VAPLFILE workspace that was distributed with VS APL.

Note: This workspace does not support nested arrays. AP 121, AP 211, and Processor 12 provide similar function with support for arrays of any type.

VAPLFILE, Processor 12, and AP 121 require that arrays be accessed by an index number, while AP 211 allows them to be accessed by name.

Main Functions

```
L CREATE filename
```

filename is a character vector that contains the name to be assigned to the file. (See the discussion of file names on page 97.) *L*, which is optional, is a physical description of the file. *L* is a vector of three elements or less. The first element is the maximum number of arrays that can be written (the default is 100). The second element is the block size of the data set used to store the file (the default is 560). Each array requires an integral number of blocks. The third element is the number of blocks for data (the default is 1.1 x *L*[1]). If a negative symbol precedes the file name, then other users cannot read the file using this workspace.

```
USE filename
```

This function shares appropriately-named variables with the file processor, opens the file, and defines global variables associated with a file in use. *filename* is a character vector that contains the name of a file. (See the discussion of file names on page 97.)

```
Z←RELEASE filename
```

This function retracts and deletes the variables shared with the AP, and deletes the global variables associated with a file in use. The file is closed. *filename* is a character vector that contains the name of a file. (See the discussion of file names on page 97.) *Z* is 1 if variables are actually retracted. A result of 0 means the file was not in use or *F* is not the name of a file.

```
Z←filename AT I
```

Z is *I*. This function selects the file named as the current file. See below for contexts in which this function is very useful. You can always substitute *I* for *filename AT I* below if *filename* is already the selected file.

```
I SET A
```

This function sets *A* as the *I*th array in the file whose name is in the character vector *F*. *I SET A* can be used to set *A* as the *I*th element of the file last mentioned in use of the functions *USE* or *AT*. The meaning of *I* is dependent on the workspace origin. Note that when replacing an existing array, space is found for the new array before the old one is erased. In this way an interruption in processing cannot lose an existing array.

```
Z←GET filename AT I
```

Z is the array in the *I*th position of the file whose name is in the character vector *filename*. *GET I* can be used to get the *I*th element of the file last mentioned in use of the functions *USE* or *AT*. The meaning of *I* is dependent on the workspace origin.

```
DELETE filename
```

This function deletes the file whose name is in the character vector *filename*.

Supplementary Functions

The following optional functions are not necessary for proper use of this package but can be useful.

```
Z←GETl filename AT I
```

Z is the account number of the person who last set the *I*th element of the *filename* and the time stamp of the set.

```
Z←SIZe A
```

Z is the size of array *A* in bytes.

```
Z←RHo filename
```

Z is the number of arrays that can be written in the file specified, where *filename* is a file that is in use.

```
ERASe filename AT I
```

This function undefines the *I*th element of the file specified and releases the space used by it in the file. *I* can be an array.

```
Z←FREEBLOCKs filename
```

Each array stored on the file (with the exception noted below) requires a contiguous set of blocks. *FREEBLOCKs* returns a vector of the contiguous available blocks. This can be useful on FILE FULL to determine if a file has outgrown its space, or is merely fragmented. There is a function called *COMBINE* (which is executed automatically before a FILE FULL message is given) which attempts to minimize the fragmentation. The result of *FREEBLOCKs* can change after executing *COMBINE*.

Note: Small scalar numbers take zero blocks.

```
Z←EXISt filename AT I
```

Z is 1 if *I*th element of *filename* has been set, 0 if *I*th element of *filename* does not contain a value, or ¯1 if *I* is out of range. *I* can be any simple numeric array.

```
Z←SHVARS
```

Z is a matrix of the names of currently-shared variables.

File Names

In general, file names consist of a library number, followed by a space and an alphanumeric name beginning with a letter *A-Z*. The name can also be appended with a colon followed by a password. For the functions *CREATE* and *DELETE*, if the name includes a library number it must be the number of a library in which the user is authorized to save files. For the functions *DELETE* and *USE*, the name must include the password, if any. The functions *RELEASE*, *AT*, *RHo*, and *FREEBLOCKs* ignore any library number and any password included in the file name. All functions except *CREATE* ignore any negative symbol that precedes the file name.

VAPLFILE Groups

The following groups exist in the workspace:

GPFILEREAD Functions needed for GET access only

GPFILEWRITE Additional functions needed for SET access

GPAPLFILE All the functions in the above groups and the optional ones

GPMESSAGE Functions for message handling

GPDESC Description of the workspace

Note: The following information isn't required for proper use of the *VAPLFILE* functions, but you may find it helpful in some situations.

When an error is encountered an appropriate message is printed. Normally the single `'→'` is executed which terminates the function and any associated pending functions. However, if the variable *⍵* contains a negative number, then after any error message, execution is suspended with a normal APL2 error message. This can be useful when debugging new applications.

It is recommended that the functions *CHK* and *TRY* normally be locked so that suspension occurs in the calling function.

The following global variables are defined whenever a file is used. *FILEID* contains the name of the file last referenced in the *USE* or *AT* functions. Names beginning with *ctl* and *dat* are variables shared with AP 121 The name consisting of *fd* appended to the fileid contains the file description as follows:

1 - User number of file creator
2 - Number of arrays permitted
3 - Block size
4 - Number of data blocks
5 - Row dimension of an index array
6 - Number of index arrays
7 - Number of salvage index arrays
8 - Total number of blocks
9-15 - *⎕TS* at creation

Chapter 7. The TRANSFER Workspace

This workspace provides functions to aid migration to APL2 from VS APL, the APL2 installed user program (IUP), and older versions of PC APL. For complete information on migration, see *APL2 Migration Guide*.

To migrate from APL.SV to APL2, you must first migrate to VS APL. For information on how to do this, see:

VS APL for CMS: Installation Reference Manual
VS APL for TSO: Installation Reference Manual
VS APL for CICS/VS: Installation Reference Manual
VS APL for VSPC: Installation Reference Manual

The main functions in the *TRANSFER* workspace are *MASSMCOPY_*, *FIX_*, *FLAG_*, *IN∆* and *OUT∆*, *INPC_*, and *OUTPC_*.

MASSMCOPY_

```
MASSMCOPY_
```

This function migrates multiple workspaces from VS APL to APL2. Commands are stacked to)*MCOPY* and)*SAVE* multiple workspaces. The list of workspaces to be migrated is formed from a file similar to a copyfile created by the following steps:

1. Sign on to VS APL
2. Issue the session manager command COPY ON ID APLLIBS
3. Issue)*LIB* commands for the libraries you wish to migrate:

)*LIB* (lists workspaces in your private library)
)*LIB* 17 (lists workspaces in Library 17)
)*LIB* 42 (lists workspaces in Library 42)

   Use in any order and as many as you like.
4. Issue either the)*OFF* or the)*CONTINUE* command with no other terminal input.

The CMS file name that *MASSMCOPY_* asks for is (in this case) '*file apllibs a*', as created by the COPY session manager command.

You can create the file by any other means as long as you can account for the following:

- All records before the first)*LIB* are ignored.
- All records after the first)*LIB* are interpreted to be either a)*LIB* command (containing the library designation), the results of a)*LIB* command (a list of workspaces), an)*OFF*, or)*CONTINUE*. Anything else is mistaken for one of these, and causes an error.

Once the list is built from the file, it is presented and you are given a chance to exclude individual workspaces. If you are saving into a public library, your LIBTAB must allow it.

Rather than stack all commands for the entire list of workspaces, you are prompted for the number of workspaces to be done in a batch. This gives you a chance to limit how much is stacked at one time, and time to bail out if things go awry. The default is set at 10 (workspaces).

The command sequence for each workspace is:

```
)CLEAR
)MCOPY [libno] wsname
)SAVE [libno] wsname
```

Which means:

- The)*MCOPY* can fail due to *WS NOT FOUND*, *WS FULL*, and so on
- The)*SAVE* can fail if the workspace already exists, or the library is full

Thorough checking of the results is recommended to determine if there were any problems.

You can escape at any prompt with → (right arrow).

You have the option of making any run a dry run, in which case the commands that would have been stacked are displayed, but not stacked and processed. You can then decide whether to make the run real.

The calling syntax is the function name without any arguments, that is, *MASSMCOPY_*.

The function is conversational from that point.

FLAG_ and FIX_

The functions *FLAG_* and *FIX_* help you migrate VS APL and APL2 IUP applications to APL2. *FLAG_* allows you to search all functions and operators in a workspace to identify possible problem areas in the code. *FIX_* allows you to change all functions and operators in the workspace when you know the exact code string replacements. The variable *FLAGMVSAPL_* is provided to find and fix the known problems that are expected in all VS APL applications. The variables *FLAGMIUP_* and *FIXMIUP_* help you find and fix known problems expected in all APL2 IUP applications. Migration consists of making these related changes, plus others that the application programmer identifies because of specific knowledge of the application.

```
SA FLAG_  LIST
```

*FLAG_* identifies language differences between VS APL, the APL2 IUP, and APL2. It searches through functions and operators named in *LIST* looking for specific code strings that are known or suspected to be problem areas in migrating applications from VS APL or the APL2 IUP to APL2. A default list of problem areas is provided as a starting point for items that should be flagged and inspected. An application programmer can then use *FLAG_* to find other specific code strings to inspect.

The function returns a matrix of rows where problems were found. If the left argument is elided, then *FLAG_* interactively prompts for search arguments (SAs), 1 line per argument, no quotes required.

`R←FLAGMIUP_ FLAG_ ALL_`	No prompting
`R←'/¨' ')¨' FLAG_ 'FUN1' 'FUN2'`	No prompting, 2 SAs
`R←(⊂'ONE') FLAG_ 'FUNNAME'`	No prompting, single SA (note the enclose on a single argument)
`FLAG_ 'FUN1' 'FUN2' 'FUNN'`	Prompts for SAs
`FLAG_ ALL_`	Prompts for SAs

*FLAGMIUP_* is a prepared list of items whose flagging is recommended for all IUP workspaces being migrated. Similarly, *FLAGMVSAPL_* is a prepared list of items whose flagging is recommended for all VS APL workspaces being migrated. Use the *DISPLAY* function (in the *DISPLAY* workspace) to see how they are constructed. If you want to edit them, use Editor 2 or the named editor on a variable formed by:

```
TEMP←2 ⎕TF 'FLAGMIUP_'
```

Reassign the changes by: `2 ⎕TF TEMP`. Do the same for *FIXMIUP_* (described below).

```
SA FIX_ LIST
```

This function searches a named list of functions for a list of character strings. The left argument is a set of old/new pairs (nested together). The function results in modified functions in the workspace.

*FIXMIUP_* is a prepared list of old/new pairs in this workspace that are the known required code string changes for migrating from the APL2 IUP to APL2.

`NAMES←FIXMIUP_ FIX_ ALL_`	Apply a fix list to all FNS in WS; returns names of modified functions.
`(⊂'¨/' '/') FIX_ 'F1' 'F2'`	Replace ¨/ with / where found in *F1* and *F2* (functions or operators).
`('¨/' '/') ('¨)' ')') FIX_ ALL_`	Make two corrections to entire WS.
`FIX_ ALL_`	Change all functions in WS according to old/new pairs, for which you are prompted.

You can add the *DOWN* function in front to display the resulting name list in a column format; that is, *DOWN FIX_ ALL_*.

The name list provided by *ALL_* is `⎕NL 3 4` without the functions involved in the transfer workspace.

The following caution prompts for a confirmation to proceed, and provides an escape if you don't want to complete the modification.

```
*** CAUTION - THIS WILL MODIFY YOUR WORKSPACE ***
```

Atomic Vectors

AV_VSAPL - the VS APL atomic vector

AV_APLSV - the APLSV atomic vector

You can use the above variables to replace references to ⎕*AV* in migrated code.

Differences

Some of the differences between VS APL, the APL2 IUP, and APL2 are potential problem areas, but are not simple code string replacements. Where potential problems are identified, they should be flagged to determine their extent, examined individually to verify the existence of a real problem, and then corrected with code changes. There might be other instances where mass corrections throughout the workspace are possible. Remember to keep a backup copy of the workspace.

Two functions are provided here to assist you to migrate character data whose application requires the same ⎕*AV* positions. They should only be used where that need has been determined. Both functions take a list of variable names and modify those variables in the workspace, so exercise caution. The first, *CHARIND*, operates in VS APL and translates each variable named in the list (in matrix form) to the ⎕*AV* indices. The second, *INDCHAR*, is run in APL2 to rebuild the variables encoded with *CHARIND*.

Note: Running *INDCHAR* against a variable not encoded by *CHARIND* destroys the variable.

These functions are coded so that they run in VS APL, although it is intended that *INDCHAR* be used in APL2. Executing:

```
CHARIND VAR
INDCHAR VAR
```

Translates *VAR* to numeric ⎕*AV* indexes, and back again to character data, restoring its original shape and rank.

Do not run either function on VS APL numeric variables.

VS APL Differences:

- Residue (|) uses ⎕*CT* as an implicit argument in APL2. Flagging and inspection of its use is advisable.
- The ordering of ⎕*AV* is different, so ⎕*AV* is included in the *FLAGMVSAPL_* list of items. Indexing ⎕*AV* should be discontinued and replaced by ⎕*AF*.
- Monadic format (⍕) of a simple numeric matrix does not contain a leading column of blanks, and its columns are formatted independently. Depending on how much the application uses ⍕, the programmer can flag it, or merely inspect the generated reports for possible alignment problems.

  In cases where code processing depends upon the formatted results, flagging and inspection is warranted.

- Function arguments should not be localized in function headers. APL2 provides a function called *CHKHDRS_* to identify these cases. Either corrective action by the programmer is required to change them, or the workspace must be migrated with)*MCOPY*. This deletes locals with the same name as arguments or results.

```
CORRECT←CHKHDRS_ ALL_
CORRECT←CHKHDRS_ ⎕NL 3
```

  CORRECT contains the names of functions that need corrective action, with the duplicated names.

- Referencing ⍞ always produces a vector. Flagging and inspection of ⍞ is recommended if the application depends upon its shape.
- The result of ¯4○*R* is the negative square root for negative *R*. Flagging ¯4○ is recommended, or just ○ if the left argument is generated by an expression.
- An odd root of a negative number (such as ¯8*÷3) is a complex number. If this is found to have an adverse effect on the application, the only effective protection is to replace all occurrences of power (*) with an ambivalent power function that inspects arguments and results to detect this effect. Event simulation (⎕*ES*) is a good way to post an exception.
- A one-item vector left argument to ⎕*SVO* does not extend. If this is going to cause a problem, then flag ⎕*SVO*, inspect, and correct.
- Groups are replaced by indirect copy and indirect erase. If group, copy, or erase commands are known to be coded (for use by the stack processor), then the following items should be flagged for inspection and correction:

```
)GR  search argument to find all group commands
)COPY
)PCOPY
)ERASE
```

- ⎕*NC* (name class) returns a different value for an invalid name. An invalid name was indicated by a ⎕*NC* value of 4 in VS APL. APL2 indicates the same with a value of ¯1. Any program logic that depends on finding invalid names must be changed. Problems can be identified by flagging occurrences of ⎕*NC*.

APL2 IUP Differences:

- Reverse (⌽) accepts a single axis only, that is, ⌽[1 2 3] is not valid. Flagging all occurrences of ⌽[is recommended, followed by inspection of each.
- Bracket Axis has been removed. This means that:

  F[*X*1;*X*2]

  and

  F[*X*1;*X*2;*X*3]

  are not allowed if *F* is a primitive symbol or a derived function (except for a niladic derived function returning an explicit result).

  This is particularly difficult to detect because of variations in *F* and the expressions that are possible inside the brackets, and the other prevalent and legal uses of brackets and semicolons. Specific knowledge of the application is useful here.

- Nested indexing is not allowed. This is also difficult to detect. It has the form: `[(____) ... (__)]`. Flagging and inspecting each occurrence of `[(` and `)]` is recommended.
- Encode (⍴) does not use ⎕*CT* as an implicit argument.
- First was monadic ⊃, but First is now monadic ↑. Because not all occurrences of monadic ⊃ are necessarily First, it might be necessary to flag and inspect all uses of ⊃. All cases of the symbols `;+-×÷([/⌿\⍀` to the immediate left of ⊃ are First, and are therefore included in *FLAGMIUP_* and *FIXMIUP_*.

  As a result of moving First to monadic ↑, monadic ⊃ with no axis discloses all, putting new axes at the right in the result.
- Unite (now called Enlist) has been changed from ∪ to ∈. This can be read "the scalar elements of." It is in the default *FLAGMIUP_*.
- The arguments to Find (⍷) have been reversed. It is in the default *FLAGMIUP_*.
- You cannot apply monadic grade (⍋ or ⍒) to character arrays. Use dyadic grade instead. Flagging and inspection might be necessary. For the IUP default collating sequence, use the *DCS* array in the *UTILITY* and *EXAMPLES* workspaces.
- The following items have been deleted and are in the default *FLAGMIUP_*:

  System names ⎕*MD* and ⎕*IR*
  Eigenvalue (⌹)
  Zeros of polynomials (⍜)
  Monadic squad (⌷)
  Find Index (⍸)
  Unique (∩)
- The following have been deleted, but the monadic forms are difficult to detect. Flagging and inspection might be necessary.

  monadic Type (∈) Use ↑0⍴⊂*R* instead of ∈*R*

  monadic ⎕*TF* Use (2 ⎕*TF R*) instead of ⎕*TF R*
- The following have a different definition in the IUP:

  Dyadic squad (⌷)
  Inner product
  Outer product

IN∆ and OUT∆

IN∆ and *OUT*∆ are functions for transferring APL objects between mainframe APL systems (APLSV, VS APL, and APL2).

The *GPMIGRATE* matrix contains the names of the required functions and variables. These objects must be reconstructed in the system where they are used. One way to do this is to display them in APL2, move to the other system (VS APL or APLSV), and reenter them using the session manager.

The functions automatically determine which system they are running on, and make the appropriate conversions. In VS APL and APL2 under VM/VMS, auxiliary processors 100 and 110 are used. Under TSO, AP 111 is used. In APLSV, auxiliary processor 370 is used.

Note: In APL2 Version 2 Release 2, new external functions (*IN*, *PIN*, and *OUT*) are available as programming interfaces to the)*IN*,)*PIN*, and)*OUT* system commands. These functions should be used instead of *IN*Δ and *OUT*Δ when running on an APL2 system.

The main functions are:

```
LIST IN∆ FILE
```

This function reads a transfer file that contains the transfer forms of APL objects, and defines those objects in the active workspace. The file was created by the *OUT*Δ function, or)*OUT* in APL2. The *FILE* parameter must be the *DDNAME* specified in a previously-issued allocate command.

```
LIST OUT∆ FILE
```

This function writes the transfer forms of APL objects in the active workspace to a transfer file; the file is suitable for reading by the *IN*Δ function, or)*IN* in APL2.

FILE is a nonempty character vector that indicates the name of the transfer file. It can consist of a single name or multiple names separated by dots. Multiple names are interpreted as qualifiers appropriate to the operating system in use. If only one name is provided, then a qualifier of *APLTF* is assumed. This is the file type in CMS and a prefix in TSO.

In both functions, *LIST* is a (possibly empty) character vector or matrix containing the names of objects to be transferred. It defaults to all objects in the file for *IN*Δ, or all objects in the workspace for *OUT*Δ. The most local meaning of each object is used. The name list can contain certain system variables: ⎕*CT*, ⎕*HT*, ⎕*IO*, ⎕*LX*, ⎕*PP*, ⎕*PW*, and ⎕*RL*.

On systems that have ambivalent functions, *LIST* can be included in the right argument rather than as the left, in the following form:

```
'FILENAME OBJ1 OBJ2... OBJN'
```

Names in this group end in Δ to avoid name conflicts. The global variable *NAME*Δ is a matrix containing the names of the global functions and variables in the *MIGRATE* workspace. It can be used as the argument to ⎕*EX* to delete them.

The transfer form and the format of the transfer file are described in *APL2 Programming: Language Reference*. The functions *IN*Δ and *OUT*Δ are similar to)*IN* and)*OUT* in APL2, but are more flexible with respect to the auxiliary processors they use.

The following example assumes that the functions in this group are contained in a VS APL workspace named '*MIGRATE*'

Transferring a workspace from VS APL to APL2:

```
APL
          )LOAD IT
          NL←⎕NL 2 3
          )COPY MIGRATE
          NL OUT∆ 'IT'
          )OFF HOLD
APL2
          )IN IT
          )WSID IT
          )SAVE
```

Transferring a workspace from APL2 to VS APL:

```
APL2
          )LOAD IT
          )OUT IT
          )OFF HOLD
APL
          )LOAD MIGRATE
          '' IN∆ 'IT'
          ⎕EX NAME∆
          )WSID IT
          )SAVE
```

In some unusual circumstances, you can get an error while running these functions. If that happens, the recovery is to →0.

INPC_ and OUTPC_

The functions *INPC_* and *OUTPC_* let you translate APL/PC transfer files with APL2. Uploading and downloading PC files can be performed by any PC/HOST file transfer program that does not translate the file in any way. Often such communication programs provide a binary mode to prevent data from being modified during the transfer operation.

Note: These utilities are necessary only for the older versions of PC APL. You can transfer files between mainframe and PC versions of APL2 by using)*IN* and)*OUT* directly.

APL/PC to Host

From within APL/PC, produce a transfer file of the objects to be uploaded using)*OUT*. The PC DOS file produced has an extension of AIO. Upload the file to the host system, making sure that no data is translated. Invoke APL2.

1.)*LOAD* the destination workspace or)*CLEAR* for a clear ws.
2.)*COPY* from the *TRANSFER* workspace the group *GPPC_*. If the *TRANSFER* workspace is in *library 2*, type)*COPY 2 TRANSFER* (*GPPC_*).
3. Set the translation table variable *APLPC_* to either *APLPC1_* for APL/PC 1.0 or *APLPC2_* (the default) for APL/PC 2.0. That is, do either *APLPC_*←*APLPC1_* or *APLPC_*←*APLPC2_*. The *APLPC_* variable is used by *INPC_* to determine how to translate the APL/PC transfer file from ASCII to EBCDIC.

4. Invoke the function *INPC_* with a right argument of a character string representing the name of the transfer file to read on the host. For example, *INPC_* *'MYFILE AIOBIN'*. The result of *INPC_* are the names of the objects established.
5. Erase the *TRANSFER* utility with *)ERASE (GPPC_)* and *)SAVE* the workspace.

   APL/PC uses uppercase and lowercase letters in names. APL2 mainframe allows either underscored letters or lowercase letters in names. As *INPC_* establishes objects in the workspace, APL2 can convert lowercase letters found in names to underscored letters. This is controlled by the *CASE* attribute associated with the current active workspace. See *APL2 Migration Guide* for details.

Host to APL/PC

The function *OUTPC_* writes a file suitable for downloading to APL/PC. Since APL/PC is a subset of APL2, not all APL2 objects are appropriate for downloading to APL/PC. For example, defined operators, nested arrays, and complex numbers are not supported by APL/PC. Also, APL/PC has different implementation limits than APL2. Typically, APL/PC is more restrictive than APL2. *OUTPC_* makes no attempt to determine if the APL2 objects written are appropriate for APL/PC. Refer to *APL2 for the IBM PC: User's Guide* for details on implementation limits.

1. *)LOAD* the workspace to be transferred.
2. *)COPY* from the transfer workspace the group *GPPC_*. If the *TRANSFER* workspace is in *library 2*, you would type *)COPY 2 TRANSFER (GPPC_)*.
3. Set the translation table variable *APLPC_* to either *APLPC1_* for APL/PC 1.0 or *APLPC2_* (the default) for APL/PC 2.0. That is, do either *APLPC_←APLPC1_* or *APLPC_←APLPC2_*. The *APLPC_* variable is used by *OUTPC_* to determine how to translate the transfer file from EBCDIC to ASCII.
4. Call the function *OUTPC_* with a right argument of a character string representing the name of the transfer file to create on the host. If the file already exists, it is overwritten. The left argument is optional. If used, it must be a character vector or matrix containing the names of the objects to be written with at least one blank between names or with names on separate rows. For example, *'⎕IO MYFUN DATA' OUTPC_ 'MYFILE AIOBIN'*. If no left argument is given, all user variables, defined unlocked functions and the system variables *⎕CT*, *⎕IO*, *⎕LX*, *⎕PP*, and *⎕RL* are written.

Since APL/PC does not support the underscored alphabet, all underscored characters in both names and character data are converted to lowercase.

You can download the file written by *OUTPC_* to the PC using a communication program that allows the file to be transferred without modifying its contents.

Note: Ignore the various system messages produced by TSO during the execution of *INPC_* or *OUTPC_*.

Chapter 8. The PRINTWS Workspace

This workspace is used to print APL objects; that is, APL arrays, functions, and operators. It can print:

- Selected APL objects
- The entire workspace
- A set of workspaces

You can:

- Control the format of the printed output
- Select the type of printer to be used
- Direct the formatted output to a terminal
- Store printer-formatted output (including carriage control characters) in a data set for later printing

The functions in this workspace use the command, alternate input (stack), disk file, and QSAM auxiliary processors.

Most of the functions and variables in the *PRINTWS* workspace have names with a Δ in the third position. This minimizes the likelihood of name conflicts with other workspaces. To use the *PRINTWS* workspace, you usually have to combine it with another workspace containing the functions and variables you want to print. If any of the objects (functions or variables) you want to print have the same names as *PRINTWS* workspace objects, you cannot print them without doing some relatively complicated renaming or redefining.

The main user functions do *not* include Δ in their names, so conflicts occur if the workspace to be printed contains similarly-named objects.

The PRINTWS primary user functions are:

CLEANPRINTWS	Erase all objects not in the *PRINTWS* workspace
LIST	Display a set of APL2 objects at the terminal
MULTIPRINT	Print a set of workspaces as specified interactively
PRINTWS	Print the contents of an entire workspace
PRINTFV	Print the objects named in one or two lists

Figure 29. PRINTWS: Primary User Functions

Primary User Functions

CLEANPRINTWS

This function erases all objects in the active workspace that are not in *GPPRINTWS*. It is used by *MULTIPRINT* between the printing of each workspace.

```
LIST R
```

R is a character vector or matrix, or a vector of vectors. *LIST* displays at the terminal all the objects named in *R*. *LIST* cannot display nondisplayable functions or operators. If you use the session manager, the *LIST* function (in conjunction with the session manager COPY command) is used to write APL objects to disk files.

```
MULTIPRINT
```

This function asks you to identify the APL2 or VS APL workspaces or the transfer form files you wish to print by asking you to complete a)*PCOPY* system command. You can change the command to)*MCOPY* to identify VS APL workspaces, or)*IN* to identify transfer form files. A null response to the)*PCOPY* prompt causes each of the previously-named workspaces or transfer form files to be formatted and printed one after the other. The page headings are the commands used to identify the workspaces or transfer form files.

The example in Figure 30 writes to a disk file the contents of whichever workspaces (or transfer form files) you specify during the execution of *MULTIPRINT*. This sequence is the preferred method for printing APL objects or writing them to disk files.

)*LOAD* 2 *PRINTWS*	Substitute the correct library number for the *PRINTWS* workspace if other than 2.
*P*1400	Select format and printer.
PFILE	Indicate disk data set output.
MULTIPRINT	Identify workspaces and print them in the selected format.

Figure 30. PRINTWS Example: Printing Several Workspaces

```
PRINTWS
```

This function requires no arguments. Before it is processed, the active workspace must contain the contents of both the *PRINTWS* workspace and the workspace to be printed. This means that one must be loaded and the other must be copied.

To remove all functions and variables other than those of the *PRINTWS* workspace, run the utility function *CLEANPRINTWS*. Another workspace can then be copied into the active workspace and the *PRINTWS* function executed.

The example in Figure 31 on page 110 is one way to print the contents of workspaces *WS*1 and *WS*2.

```
)LOAD 2 PRINTWS        (substitute the correct library number for the
)PCOPY WS1               PRINTWS workspace if other than 2)
PRINTWS
CLEANPRINTWS
)PCOPY WS2
PRINTWS
```

Figure 31. PRINTWS Example: Printing Workspace Contents

PRINTWS and *PRINTFV* ask you to type in the information to be printed in the header of each output page.

```
L PRINTFV R            PRINTFV R
```

L and *R* are character matrices containing the names of the functions and variables to be printed. All the objects named in *L* are printed before the objects named in *R*. The first object in *R* is printed at the top of a page. A numerical value (0, for instance) entered as a list name is treated as a null list.

Note: If the name lists for workspaces other than the *PRINTWS* workspace itself are constructed by use of the ⎕*NL* system function, this should be done before the two workspaces are combined in the active workspace.

Printer Selection Functions

Figure 32 lists the printer selection functions and describes them briefly.

PTERMINAL	Format output for the user's terminal
P1400	Format output for an impact printer
P3800	Format output for a nonimpact printer
PFILE	Store formatted output on a disk data set

Figure 32. PRINTWS: Printer Selection Functions

These functions, if required, should be processed before any of the primary user functions. They require no arguments. If the active workspace is stored after one of them is run, it does not need to be run again when the modified workspace is reloaded. For example, an installation that uses nothing but impact printers can load the *PRINTWS* library workspace, run the function *P1400* and save the workspace. All printing is formatted for impact printers on subsequent loadings of the *PRINTWS* workspace.

Environment System Command Functions

The *PRINTWS* workspace has a defined function *CM∆D* for processing environment system commands. When the *PRINTWS* workspace is active, you can use this function to run system commands from your terminal.

Environment Dependencies

This section discusses the environment dependencies for CMS and TSO.

CMS

In CMS, the *PRINTWS* workspace uses the QSAM auxiliary processor to direct output to a virtual printer file, using the *CM∆D* function. The syntax of this function is:

```
Z←CM∆D R
```

R is a character string containing a CMS command. *Z* is a return code. 0 indicates successful execution of the command.

The meaning of nonzero codes is found in *VM/SP Diagnosis: System Messages and Codes*, SC19-6204.

Use the *CM∆D* command to specify print class, forms, line spacing, character fonts, and number of copies. You can store standard setups as character strings to be used as inputs to *CM∆D*.

TSO

In TSO, the *PRINTWS* workspace gets its printing done either by using a SYSOUT queue dedicated to the printing of APL objects, or by submitting a batch print job. The first method is the preferred one; the second should only be used when a dedicated SYSOUT queue is unavailable. For more information, see the *HOWTSO* variable in the workspace.

The JCL required for a print job varies so much from one location to another that no general-purpose function can be provided. The function that generates the JCL is called *PJ∆CL*. Figure 33 on page 112 shows how it is used. The first part of the figure shows *PJ∆CL* prompts and user responses. The second part shows the character array, *JCLDECK*, that *PJ∆CL* produced. Notice how it incorporated user responses.

(The lines containing the colon are prompt-response pairs, the prompt is to the left of the colon, the response to the right.)

1. Using *PJ∆CL* Function to Create *JCLDECK* Character Array

```
      PJ∆CL
NAME: BOGLE, C. J.
CHARGE NUMBER: XY10
DEPARTMENT NUMBER: 10XY
BUILDING: X10Y
OUTPUT BIN: 1XY0
ROOM NUMBER: 435211
```

2. Character Array *JCLDECK* Created by User Responses to *PJ∆CL* Prompts

```
      JCLDECK
//B7220001 JOB (B722166,'A=XY10,D=10XY'),
//   'BOGLE, C. J.',NOTIFY=B722166,CLASS=V,MSGCLASS=Z,
//   TIME=(0,15),USER=B722166
/*OUTPUT OUTP X=fo∆nt,C=fc∆b,F=fo∆rm,N=co∆pies
/*JOBPARM B=X10Y,D=10XY,O=1XY0,R=435211,K=0
//TSOAPL EXEC PGM=IEBGENER,COND=EVEN
//SYSPRINT DD DUMMY
//SYSIN DD DUMMY
//SYSUT2 DD SYSOUT=(A,,OUTP)
//SYSUT1    DD  DSN=B722166.wsn.LIST,
//  DISP=(OLD,DELETE,KEEP)
//JCL   DD  DSN=B722166.wsn.CNTL,
//  DISP=(OLD,DELETE,KEEP)
//
```

Figure 33. PRINTWS: Processing and Results of PJ∆CL Function

The first time you try to print a workspace under TSO, you are asked to provide the information needed to prepare JCL for a print job in your name. If the workspace is not modified, the questions you are asked are those illustrated in Figure 33. Due to local variations, however, you probably need to modify the *PJ∆CL* function to get the correct JCL.

Once you answer the questions needed to prepare the job control file required to submit your print jobs, a PRINT2.JCL file with your responses is created. The next time you sign on, this file is used to submit a print job.

Chapter 9. The SQL Workspace

You can use the facilities of the SQL workspace to pass requests to AP 127, the Structured Query Language Auxiliary Processor. SQL is a high-level language for manipulating data in relational databases.

For more information about the APL2/SQL interface, the SQL language, and the SQL workspace, see *APL2 Programming: Using Structured Query Language*.

Chapter 10. The MEDIT Workspace

The *MEDIT* workspace is used primarily for display devices without the APL feature.

If you have an APL display terminal, it is usually better to edit with the general-purpose, full-screen editors (XEDIT, for example) that are available in your environment.

Use *MEDIT* to edit APL2 programs (operators and functions) when you do not have access to the APL character set.

Editing APL Variables and Defined Functions

The *MEDIT* workspace provides a collection of defined functions that you can use to:

- Edit simple character-array variables
- Edit functions and operators without using the standard line-by-line APL function editor
- Create new objects
- Edit APL objects using display terminals *without* the APL feature

The Basic Edit Procedure

To editing an existing object:

1. Convert the object into the form required by the edit functions
2. Add, insert, delete lines of text; modify individual lines or groups of lines; display intermediate results before completing the editing process
3. Reconvert the edit text into an operator, function, or variable

Creating New APL2 Functions or Character Arrays

To create a new APL2 object:

1. Initialize the text for editing
2. Add, insert, delete lines of text; modify individual lines or groups of lines; display intermediate results before completing the editing process
3. Reconvert the edit text into an operator, function, or variable

Display Terminals without the APL feature

To use a graphic display terminal without the APL feature, the basic edit procedure must be:

- Preceded by translating each APL character to a unique set of non-APL characters
- Followed by retranslating to APL characters

Using the MEDIT Functions

This section discusses the functions provided with the MEDIT workspace.

Converting APL Objects for Editing

You must run either *APLFIN* or *APLVIN* before you can edit an existing APL2 object. *APLVIN* puts a character array variable into a form suitable for editing. *APLFIN* does the same for a defined function or operator.

They are used as in the following examples:

```
APLVIN JCLDECK
APLFIN 'REPLACE'
```

When using *APLFIN*, put the function name in quotation marks. When using *APLVIN*, do not enclose the variable name in quotation marks.

After editing is complete:

JCLDECK←*LIST ALL*	Redefines the variable *JCLDECK* to its edited value
⍞*FX LIST ALL*	Redefines the function *REPLACE* to its edited value

JCLDECK (shown in Figure 36 on page 120) and *REPLACE* (shown in Figure 34 on page 117) are the names of a variable and function that are used in the illustrative examples below.

Pre- and Post-Editing Functions

This section discusses the pre- and post-editing functions supplied with the MEDIT workspace.

Terminals without the APL Feature

If you are editing an APL function *FN* on a display terminal that cannot enter or display APL characters, you must first translate the function by using *QCR*. This produces a character array for processing by *APLVIN*, as follows:

```
APLVIN QCR 'FN'
```

Use *APLVIN* rather than *APLFIN* to edit a function that has been translated by *QCR*. This is because the output of *QCR* is a character array, which is the type of input processed by *APLVIN*, and not a function name, which is the type of input required by *APLFIN*.

After editing is complete, you can turn the edit text into a function by retranslating and *fixing* as follows:

```
QFX LIST ALL
```

LIST and *ALL* are edit functions that are described later.

You can make assignments on non-APL terminals with the *ASSIGN* function. Its left argument is a character scalar or vector that contains a name. Its right argument is any array. There is no explicit result.

For example:

```
      'NAME' ASSIGN ⍳4
      NAME
1 2 3 4
```

Editing

The edit functions can be grouped into the following categories:

CLEAR, *START*	Initialization
AFTER, *ADD*, *BEFORE*	Input
C, *CHANGE*, *DELETE*, *REPLACE*	Change
ALL, *AT*, *BOT*, *D*, *FIND*, *FROM*, *THRU*, *TOP*, *U*	Select
LIST, *NUMBER*	Output
TABS	Set tabs

The edit functions work on simple two-dimensional character arrays only. The rows in these arrays are assigned line numbers; the first line is numbered 0. At any point in the edit procedure the value of the current line number is stored in the variable *LN*. This value is established and changed by the edit operations described below.

The values of two variables *CCOL* and *LRECS* ("Cutoff Column" and "Record Length") give, respectively, the number of columns in the array and the number of the last column that can be modified by the edit functions. For example, if *CCOL* is 71 and *LRECS* is 80, you can't make any modifications past column 71; with these settings the continuation column and serial number field in a card deck are protected.

The Initialization Functions

The function *START* (with no arguments) initializes the edit text and opens the keyboard for adding lines of data to an 80-character wide matrix. It starts the function *CLEAR*, which establishes an array width of 80 and an input cutoff column of 71.

The Input Functions

The input functions (*ADD*, *AFTER*, and *BEFORE*) are similar in several ways:

- None take an argument
- They are entered as single words
- They open the keyboard for input
- Input is terminated by entering a null line (pressing the ENTER or carriage-return key with no other input)

The various input functions (including the input requested by *REPLACE*, which is discussed in the next section) are different only in where the input lines are placed.

- *ADD* appends the input lines to the bottom of the current array
- *AFTER* inserts the input lines after the current line
- *BEFORE* inserts the input lines before the current line
- *REPLACE* deletes *N* lines after the current line (where *N* is the argument) and inserts replacement lines

U: identifies user entry
E: identifies edit program response
A: identifies APL system response

```
      APLFIN 'REPLACE'                  U: Put function REPLACE in editing form
      LIST ALL                          U: Request list of all lines in function
REPLACE n                               E:
BEFORE                                  E: Edit program listing of all lines
LN←LN+1                                 E: in object being edited.
DELETE n                                E:
LN←LN-1                                 E:

      NUMBER LIST ALL                   U: Request numbered list of all lines
0 REPLACE n                             E:
1 BEFORE                                E: Edit program listing of line
2 LN←LN+1                               E: numbers and lines in object
3 DELETE n                              E: being edited.
4 LN←LN-1                               E:

      FIND ALL                          U: Request search through all lines
ENTER TEXT :                            E: Request entry of search text
LN                                      U: Entry of search text
LN←LN+1                                 E: Copy of first line found

      CHANGE ALL                        U: Request change to apply to all lines in function
OLD                                     E: Request entry of text to be changed
LN                                      U: Entry of text to be changed
NEW                                     E: Request entry of substitute text
LINENO                                  U: Entry of substitute text

      LIST 2 THRU 4                     U: Request listing of lines 2 3 4
LINENO←LINENO+1                         E: Edit program listing of
DELETE n                                E: requested lines
LINENO←LINENO-1                         E:

      AT 2                              U: Position current line at 2
1                                       E: Edit program output of value 1

      LN                                U: Request current value of LN
2                                       A: APL display of value
```

Figure 34 (Part 1 of 2). MEDIT: How to Use the Edit Functions

```
      C                                  U: Request single-line change
OLD                                      E: Request input of text to be changed
LINENO                                   U: Entry of text to be changed
NEW                                      E: Request input of substitute text
LN                                       U: Entry of substitute text
LN←LN+1                                  E: Copy of changed line

      D 2                                U: Move current line down 2
LINENO←LINENO-1                          E: Copy of current line

      C                                  U: Request single-line change
OLD                                      E: (See above)
LINENO
NEW
LN
LN←LN-1

      ⎕FX LIST ALL                       U: Convert edit object to function
REPLACE                                  A: APL output of name of function
                                            fixed by ⎕FX system function
```

Figure 34 (Part 2 of 2). MEDIT: How to Use the Edit Functions

```
      ∇DELETE [⎕]∇                          U: Request listing of DELETE
                                               on an APL terminal

    ∇
[0]   DELETE n;⎕IO                          A:
[1]   ⎕IO←0                                 A: Function
[2]   n←((0⌈LN)ρ0),(0⌈⌊n)ρ1                 A: Listing
[3]   TDS←(~(1↑ρTDS)↑n)⌿TDS                 A:
    ∇ 1984-08-31  4.00.00 (GMT-8)

      QCR DELETE                                          U: Translate DELETE function to
                                                             non-APL characters
DELETE &NUN ;&QUA IO                                      E: Output
&QUA IO&LAR 0                                             E: of
&NUN &LAR ((0&UST LN)&RHO 0),(0&UST &DST &NUN )&RHO 1     E: QCR
TDS&LAR (&TIL (1&UAR &RHO TDS)&UAR &NUN )SHB TDS          E:

      APLVIN QCR 'DELETE'                 U: Translate DELETE and put into
                                             editing form
      NUMBER LIST 1 THRU 3                U: Request numbered list of
                                             statements in DELETE
1 &QUA IO&LAR 0
2 &NUN &LAR ((0&UST LN)&RHO 0),(0&UST &DST &NUN )&RHO 1
3 TDS&LAR (&TIL (1&UAR &RHO TDS)&UAR &NUN )SHB TDS

      AT 1                                *** The remaining entries in this
1                                         *** figure show how to make a change
                                          *** in the QCR version of the
      C                                   *** DELETE function, how to
OLD                                       *** reconvert it to the APL
0                                         *** character version, and what
NEW                                       *** the function looks like on an APL
1                                         *** terminal after the change.
&QUA IO&LAR 1

      QFX LIST ALL
DELETE
      ∇DELETE[⎕]∇
    ∇
[0]   DELETE n;⎕IO
[1]   ⎕IO←1
[2]   n←((0⌈LN)ρ0),(0⌈⌊n)ρ1
[3]   TDS←(~(1↑ρTDS)↑n)⌿TDS
    ∇ 1984-08-31 13.50.12 (GMT-8)
```

Figure 35. MEDIT: Editing Functions on Non-APL Display Terminals

```
     APLVIN JCLDECK                              U: Put variable JCLDECK in
                                                    editing form

     NUMBER LIST ALL                             U: Request output of numbered
                                                    list of lines
 0 //IMPT JOB (S667221,'A=SH44,B=090,D=M46,O=X256,R=D256'),
 1 //  'STEINWAY A. J.',NOTIFY=S667221,USER=S667221,
 2 //  MSGCLASS=F,
 3 //  PASSWORD=
 4 //TIME=(1,0),MSGLEVEL=(1,1)                   E: MEDIT
 5 //BLDDS EXEC PGM=IEBGENER                     E: output
 6 //SYSPRINT DD SYSOUT=A                        E: of numbered list
 7 //SYSUT2 DD DSN=M166722.@W.PRTGP,             E: of all lines in
 8 //  UNIT=SYSDA,SPACE=(TRK,(20,5)),            E: current edit
 9 //  DCB=(RECFM=FB,LRECL=80,BLKSIZE=800),      E: variable
10 //  DISP=(,CATLG)
11 //SYSIN  DD  DUMMY
12 //SYSUT1  DD  DATA

     FIND ALL                                    U: Search all lines
ENTERTEXT :                                      E: Request for search text
DATA                                             U: Entry of search text
//SYSUT1  DD  DATA                               E: Copy of first line found

     C                                           U: Request single-line change
OLD                                              E: Request for old text
DATA                                             U: Entry of old text
NEW                                              E: Request for new text
*                                                U: Entry of new text
//SYSUT1  DD  *                                  E: Copy of changed line

      JCLDECK←LIST ALL                           E: Assign new value to JCLDECK

      JCLDECK                                    E: Check new value
//IMPT JOB (S667221,'A=SH44,B=090,D=M46,O=X256,R=D256'),
//  'STEINWAY A. J.',NOTIFY=S667221,USER=S667221,
//  MSGCLASS=F,
//  PASSWORD=                                    A: APL system output
//  TIME=(1,0),MSGLEVEL=(1,1)                    A: of value of
//BLDDS EXEC PGM=IEBGENER                        A: JCLDECK
//SYSPRINT DD SYSOUT=A
//
SYSUT2 DD DSN=M166722.@W.PRTGP,
//  UNIT=SYSDA,SPACE=(TRK,(20,5)),
//  DCB=(RECFM=FB,LRECL=80,BLKSIZE=800),
//  DISP=(,CATLG)
//SYSIN  DD  DUMMY
//SYSUT1  DD  *
```

Figure 36. MEDIT: Editing a JCL Deck

The Change Functions

One change function (*C*) applies changes only to the current line. The others (*CHANGE*, *DELETE*, and *REPLACE*) modify *N* lines starting with the current line, where *N* is the argument to each function.

C and *CHANGE* request old and new character strings and replace the old with the new. *C* does this wherever the old string occurs in the current line only. *CHANGE* does it starting with the current line and its *N*-1 successors.

REPLACE inserts new lines and deletes old ones in the manner described in the preceding section. If there is no input, the effect is the same as *DELETE*, which deletes *N* lines starting with the current one.

The syntax of each function is shown in Figure 37

R←C	*LN* is unchanged. *R* is a copy of the modified line.
CHANGE N	*LN* is the number of the last line changed.
DELETE N	*LN* is unchanged.
REPLACE N	*LN* is the number of the last line inserted.

Figure 37. MEDIT: The Change Functions

The Select Functions

The select functions all return a numerical output value except the functions *D* (Down), *U* (Up), and *FIND*. These functions change the value of *LN* and return a copy of the line numbered *LN* (except that *FIND* returns a null line if its search is unsuccessful).

The purpose of returning numerical values is to provide input values to other edit functions. The function *ALL*, for example, sets *LN* to 0 and returns the number of lines in the edit array. You can use this value as input to a function like *LIST*, and the combination produces a list of all the lines in the edit array. Experimenting with combinations of select and nonselect functions helps you develop a set of useful patterns. Some examples are presented in a later section.

The syntax of each function is given in Figure 38 on page 122.

R←AT L	Set *LN* to *L* and return the value 1.
R←ALL	Set *LN* to 0 and return the number of lines in the text.
R←BOT	Set *LN* to the number of the last line and return this value.
R←D N	Set *LN* to *LN+N* and return a copy of the new current line.
R←FIND N	Request input of search string and search *N* lines for this string. If the string is found, set *LN* to the number of the first line where it is found and return this line as output. If the string is not found, set *LN* to a number one greater than that of the last line searched and return a null vector as output.
R←A FROM B	Set *LN* to *B* and return the value *A*.
R←A THRU B	Set *LN* to *A* and return the value *1+B-A*.
R←TOP	Set *LN* to 0 and return the value 0.
R←U N	Set *LN* to *LN-N* and return a copy of the new current line.

Figure 38. MEDIT: The Select Functions

The Output Functions

The output functions produce character arrays consisting of all or part of the array being edited.

LIST	Produces a set of lines exactly as they occur in the array being edited.
NUMBER	Produces a set of lines prefixed by their respective line numbers.
VAR←LIST ALL	Stores the edited object as a variable called *VAR*.
⎕FX LIST ALL	Converts the edited object into a function. The function name is output by the *⎕FX* system function. This is the name found on the first line of the edit object (if the lines as a whole form a valid function definition).

Figure 39 on page 123 shows the syntax of both functions.

R←LIST N	Output *N* lines starting with the current one. Set *LN* to one plus the number of the last line listed.
R←NUMBER A	Print the character matrix *A* with a line number to the left of each of its lines. The lines are numbered consecutively and the number of the last line is *LN*-1. As a result of this convention: *NUMBER LIST ALL* Produces a numbered listing of the entire edit array.

Figure 39. MEDIT: The Output Functions

The Set Tabs Function

The function *TABS* assists in setting tab stops on 2741-like terminals. It takes a numeric argument that contains the desired tab positions. It types the letter T at these positions and allows time for depression of a Tab Set key by the user. Thus:

```
TABS 10 20 30
```

helps you set tabs at the 10th, 20th, and 30th positions to the right of the current left margin.

Usage Notes

This section supplies notes on the use of:

- LRECS and CCOL
- QCR and QFX
- APL and non-APL translate table

LRECS and CCOL

When you use *START* (which starts the function *CLEAR*) to begin input of a new function or variable, *LRECS* is set to 80 and *CCOL* to 71.

APLVIN sets both *LRECS* and *CCOL* equal to the number of columns in the input array.

APLFIN uses ⎕*CR* to make a character array of its input function and adds 10 blank columns on the right to allow for longer lines created by the editing process.

In all cases, lines that turn out to be longer than *LRECS* are rejected. If this is likely to occur, you can use the APL ↑ primitive function to add blank columns on the right of an input array. Thus:

```
ARRAY←(0 20+⍴ARRAY)↑ARRAY
```

redefines *ARRAY* to have an additional 20 blank columns on its right. Each line has twenty additional positions to expand into as the result of an edit operation.

QCR and QFX

The mapping that translates APL characters into non-APL character strings produces arrays that can be bewildering in appearance. Before editing these arrays, you might want to spend some time memorizing the symbol names and abbreviations in Figure 40.

QFX always attempts to convert its input to a function. When this isn't possible, *QFX* produces the translated character array as output. Thus, if the original input to *QCR* was a variable, it is redefined as a function by *QFX* if it happens to be a valid canonical representation of a function. Should this happen, the unintended function can be converted to a variable by using the system function ⎕*CR*.

Conversely, if the original input was a function, and mistakes in editing make it impossible to return the edited array to function form, *QFX* returns the translated character array as output. If you apply *QCR* and *QFX* to variables, you should name the output of *QFX* with an APL assignment.

```
VARNAME LAR QFX LIST ALL
```

This is also a good practice to follow when editing functions. If the function is successfully fixed by *QFX*, *VARNAME* contains the name of the edited function. If it is not, *VARNAME* contains a character array that is a defective canonical representation of a function.

APL, Non-APL Translate Table

Certain APL characters cannot be displayed on a terminal without the APL feature. *QCR* replaces each of these with a five-character code. The first character is a delimiter (the default delimiter is &). The last is a space. The intervening three are alphabetic codes.

The *QCR* codes, their symbol, indication of overstrikes, and character names from which they are derived are listed alphabetically in Figure 40 on page 124. Becoming familiar with the names and codes helps you edit arrays in which translated characters appear. There are no mnemonics for characters that normally appear on non-APL terminals.

Figure 40 (Page 1 of 3). MEDIT: Mnemonics for Non-APL Terminals

Mnemonic	Symbol	Overstrike	Symbol Name
ALP	⍺		Alpha
AUN	A	A_	A Underbar
	_		Bar
BUN	B	B_	B Underbar
CBA	⊖	○-	Circle Bar
CIR	○		Circle
CSL	⍉	○\	Circle Slope
CSR	⍟	○*	Circle Star
CST	⌽	○|	Circle Stile
	:		Colon
	,		Comma

Figure 40 (Page 1 of 3). MEDIT: Mnemonics for Non-APL Terminals

Mnemonic	Symbol	Overstrike	Symbol Name
CUN	C	C_	C Underbar
	.		Dot
DAR	↓		Down Arrow
DCA	∨		Down Caret
DCT	⍒	∨~	Down Caret Tilde
DDO	⍠	¨.	Dieresis Dot
DEL	∇		Del
DIE	¨		Dieresis
DIV	÷		Divide
DLS	⍫	∇|	Del Stile

Figure 40 (Page 2 of 3). MEDIT: Mnemonics for Non-APL Terminals

Mnemonic	Symbol	Overstrike	Symbol Name
DLT	⍢	∇~	Del Tilde
DSH	∪		Down Shoe (Cup)
DST	⌊		Down Stile
DTA	Δ		Delta
DTJ	⍤	⊥∘	Down Tack Jot
DTK	⊥		Down Tack
DTS	⍙	Δ|	Delta Stile
DTU	Δ̲	Δ_	Delta Underbar
DUN	D̲	D_	D Underbar
DUT	⍦	⊥⊤	Down Tack Up Tack (I-Beam)
EPU	∈̲	∈_	Epsilon Underbar
EPS	∈		Epsilon
	=		Equal
EQU	≡	=_	Equal Underbar
EUN	E̲	E_	E Underbar
FUN	F̲	F_	F Underbar
	>		Greater
GUN	G̲	G_	G Underbar
HUN	H̲	H_	H Underbar
IOT	⍳		Iota
IOU	⍳̲	⍳_	Iota Underbar
IUN	I̲	I_	I Underbar
JOT	∘		Jot
JUN	J̲	J_	J Underbar
KUN	K̲	K_	K Underbar
LAR	←		Left Arrow
LBR	[		Left Bracket
	(		Left Paren
LRB	[]	[ ]	Left Right Bracket
LSH	⊂		Left Shoe
	<		Less

Figure 40 (Page 2 of 3). MEDIT: Mnemonics for Non-APL Terminals

Mnemonic	Symbol	Overstrike	Symbol Name
LUN	L̲	L_	L Underbar
MUN	M̲	M_	M Underbar
NE	≠		Not Equal
NG			Not Greater
NL	≥		Not Less
NUN	N̲	N_	N Underbar
OBA	¯		Overbar
OME	⍵		Omega
OUN	O̲	O_	O Underbar
	+		Plus
PUN	P̲	P_	P Underbar
QDI	⍰	□÷	Quad Divide
QDO	!	'.	Quote Dot
QJO	⍜	□∘	Quad Jot
QQU	⍞	□'	Quad Quote
QSL	⍁	□\	Quad Slope
QUA	□		Quad
	?		Query
	'		Quote
QUN	Q̲	Q_	Q Underbar
RAR	→		Right Arrow
RBR	]		Right Bracket
RHO	⍴		Rho
	)		Right Paren
RSH	⊃		Right Shoe
RUN	R̲	R_	R Underbar
	;		Semicolon
SHB	⌿	/-	Slash Bar
SLB	⍀	\-	Slope Bar
SLO	\		Slope
			Space
	*		Star
STI	|		Stile
SUN	S̲	S_	S Underbar
TIL	~		Tilde
TIM	×		Times
TUN	T̲	T_	T Underbar

Figure 40 (Page 3 of 3). MEDIT: Mnemonics for Non-APL Terminals

Mnemonic	Symbol	Overstrike	Symbol Name
	_		Underbar
UAR	↑		Up Arrow
UCA	∧		Up Caret
UCT	⍲	∧~	Up Caret Tilde
USH	∩		Up Shoe (Cap)
USJ	⍦	∩∘	Up Shoe Jot
UST	⌈		Up Stile
UTJ	⍭	⊤∘	Up Tack Jot
UTK	⊤		Up Tack
UUN	U	U_	U Underbar
VUN	V	V_	V Underbar
WUN	W	W_	W Underbar
XUN	X	X_	X Underbar
YUN	Y	Y_	Y Underbar
ZUN	Z	Z_	Z Underbar

Part 2. External Routines

Chapter 11. External Routines

A number of external routines are distributed with APL2. These routines provide useful functions, such as installation verification and serviceability aids.

The external routines discussed in this chapter are called through Processor 10 and Processor 11. They are accessed through ⎕*NA* and depend upon availability of a NAMES file distributed with APL2 (AP2VN011 NAMES in VM/CMS and APL2.SAP2NICK in MVS/TSO). The NAMES file must be made available to the user, either on an accessible minidisk in VM/CMS or by allocating the NAMES file as ddname AP2TN011 in MVS/TSO. To access a specific routine, ⎕*NA* is used. For example:

```
3 11 ⎕NA 'TIME'
```

The external routines are also available in the *SUPPLIED* workspace. See "SUPPLIED: Information About External Functions" on page 11 for further information.

Once accessed, the routines perform like normal, locked APL functions.

This chapter describes each of the external routines distributed with APL2. Detailed descriptions of the external routines follow Figure 41 in alphabetical order by routine name.

Figure 41 (Page 1 of 3). APL2/370 External Routines

External Routine	Function	
Data Conversion		
ATR	Convert an APL array to a record with mixed data types	136
CTK	Convert extended character data to mixed DBCS data	149
CTN	Convert character data to numeric data	150
DFMT	Format an array of extended character data	152
KTC	Convert mixed DBCS data to extended character data	168
PFA	Generate a pattern for *ATR* or *RTA*	175
RTA	Convert a record to an APL array	181
CAN[1]	Compress and Nest	140
DAN[1].	Delete and Nest	151
SAN[1]	Slice and Nest	182

Figure 41 (Page 2 of 3). APL2/370 External Routines

External Routine	Function	
External Routine Support		
APL2PI	A niladic form of *APL2PIE*	132
APL2PIE	Interface with non-APL programs that call APL2.	133
ATP	Update parameters passed by a non-APL program	135
BUILDRD	Build a routine description for an external routine	138
BUILDRL	Build a routine list for a module containing external routines	139
EXP	Request APL evaluation in the previous namescope	160
PACKAGE	Convert a workspace to a namespace	173
PTA	Extract parameters passed by a non-APL program	177
QNS	Query the current namescope	178
APL Object Access		
EDITOR2[2]	A program interface to Editor 2	159
EDITORX[2]	A program interface to a named system editor	158
IN[2]	Program access to system command)*IN*	167
OUT[2]	Program access to system command)*OUT*	172
PIN[2]	Program access to system command)*PIN*	176
REXX Access (Processor 10)		
Δ*EXEC*	Execute a REXX program	188
Δ*F*	Obtain information about a CMS or MVS file	189
Δ*FM*	Read or write a file as a matrix	190
Δ*FV*	Read or write a file as a vector of vectors	191
System Data Access		
CSRIDAC	Access an MVS/ESA* virtual data object	142
CSRREFR	Refresh an MVS/ESA virtual data object	144
CSRSAVE	Save changes to a permanent MVS/ESA virtual data object	145
CSRSCOT	Save MVS/ESA virtual data object changes in a scroll area	146
CSRVIEW	Define a view on an MVS/ESA virtual data object	147
DSQCIA	Interact with the database Query facility	156

Figure 41 (Page 3 of 3). APL2/370 External Routines

External Routine	Function	
Environment Control		
ATTN	Query or reset the attention flag	137
MSG	Use APL2 message facilities from an application	169
OPTION	Query or set APL2 invocation options	170
PBS	Query or set the)*PBS* state	174
RAPL2[2]	Run the remote-session manager	179
SERVER	Start a TCP/IP port server	183
SVI	Determine shared variable processor numbers or user IDs.	184
Usage and Debugging Aids		
CMSIVP	Installation verification under CMS	141
DISPLAY	Display an array in a form that shows nesting and data types	153
DISPLAYC	The same as *DISPLAY*.	154
DISPLAYG	The same as *DISPLAY*, but using box characters	155
FED	Diagnostic tool for IBM service usage	163
HELP	Obtain information from APL2HELP files	164
IDIOMS[2]	Search the APL2 phrase collection	166
TIME	Performance monitoring within a workspace	185
TSOIVP	Installation verification under TSO	187

Notes:

1. The Partition primitive (⊂) should be used instead of these three functions.
2. Not available in Application Environment.

APL2PI—APL2 Program Interface

This function facilitates communication from APL2 to non-APL applications.

```
result←APL2PI
```

result Is a two- or three-element vector. The first two elements are an integer code that indicate the success or failure of the request. The third element, if present, is a return code issued by the non-APL application when it terminates, or a message if the non-APL application used the *APLX* service.

APL2PI is a niladic form of *APL2PIE* used to return control to the non-APL application after APL2 initialization or after an APLX call from the non-APL application. It is particularly useful because you can specified it as RUN(APL2PI) when starting APL2.

Using *APL2PI* is the same as using *APL2PIE* 0 ''.

For more information about using *APL2PI*, see the discussion of calling APL2 in *APL2/370 Programming: System Services Reference*.

Note: When establishing an association to *APL2PI* using ⎕*NA*, you must code a zero as the name class of the object. For example:

```
0 11 ⎕NA 'APL2PI'
```

This is required for proper operation of the *APL2PI* interface.

APL2PIE—APL2 Program Interface Extended

An APL2 external function to make it easier to communicate between APL2 and non-APL applications. APL2PIE is an ambi-valent function that provides several functions:

- Return control from the APL2 environment to the currently-active non-APL application
- Start a non-APL application from the APL2 environment
- Request termination of the currently-active non-APL application
- Issue a service request to a non-APL application

```
result←APL2PIE 0 ''
or
result←value APL2PIE 0 ''
```

These functions return control to the non-APL application:

value An array to be returned to the non-APL application.

result A two- or three-element vector. The first two elements are an integer code that indicates whether the request was successful. The third element, if present, is a return code issued by the non-APL application when it terminates, or a message if the non-APL application used the *APLX* service.

```
result←command APL2PIE 1 name
```

Start a non-APL application where:

name The name assigned to the application

command The command to start the application

result A two- or three-element vector. The first two elements make up an integer code that indicates whether the request was successful. The third element, if present, is a return code issued by the non-APL application when it terminates, or a message if the non-APL application used the *APLX* service.

```
result←APL2PIE 2 ''
```

Request termination of the currently-active non-APL application.

result A three-element vector. The first two elements are an integer code that indicates whether the request was successful. The third element is a return code issued by the non-APL application when it terminated.

```
result←value APL2PIE 3 name
```

Make a service request to the named non-APL application:

name The name of the application to direct the request to

value A value to pass to the application

result Either a two- or three-element array built by *APL2PIE*, or an arbitrary array built by the non-APL application.

If the array was built by *APL2PIE*, the first two elements are an integer code that indicates whether the request was successful. The third element, if present, is a return code issued by the non-APL application when it terminates, or a message if the non-APL application used the *APLX* service.

For more information about using *APL2PIE*, see *APL2/370 Programming: Processor Interface Reference*.

Note: When establishing an association to *APL2PIE* using ⎕*NA*, you must code a zero as the name class of the object. For example:

```
0 11 ⎕NA 'APL2PIE'
```

This is required for proper operation of the *APL2PI* interface.

ATP—Array to Pointer

The external function *ATP* allows pointer arguments passed from non-APL routines to be replaced (i.e. updated) with an APL2 array. You can use *ATP* with *PTA* to retrieve, then update arguments passed from non-APL programs.

```
pattern ATP array pointers
```

pattern A pattern (similar to the pattern used with ATR) that describes the data in the desired format.

array The source array.

pointers An address or list of addresses of the data to be updated.

Note: This function does not produce an explicit result. Further, it doesn't check to make sure the result fields are large enough to hold the source values.

The *ATP* function assumes a one-to-one correspondence among the data descriptors in the left argument, the data items in the array specified in the right argument, and the set of pointers in the right argument.

ATR—Array To Record

Use this routine to convert an APL array right argument to a character vector based on a pattern left argument. It is useful for converting APL objects to records that are written to a file.

```
record←pattern ATR array
```

pattern A character vector that describes the format of the right argument. For more information about *pattern*, see *APL2/370 Programming: System Services Reference*.

array Any APL array of depth 181 or less.

record A character vector created from the *ARRAY* according to the *PATTERN*.

Note: The *RTA* external routine (see “RTA—Record to Array” on page 181) is the inverse of *ATR*. The *PFA* external routine (see “PFA—Pattern from Array” on page 175) can also be used to generate patterns.

ATTN—Handling Attentions

This routine allows applications to signal an attention or detect whether the user has signaled an attention.

Frequently, you need to protect applications from interruption during critical processing. You can do this by setting the ignore attention execution attribute during function fixing. When you need to signal an application, the *ATTN* function tells the application that a signal has been received, but doesn't interrupt processing.

The *ATTN* function can signal an attention, query whether an attention has been signaled, or remove an attention that has been signaled. An application can use *ATTN* to detect whether an attention has been signaled during □*DL*, □*SVE*, or a long-running process. Note that signaling attention does not halt a shared variable interlock, or cause a shared variable event.

```
rc←ATTN arg
```

arg One of the following:

- `⍳0` Query the current attention state
- `0` Set the current attention state to off
- `1` Set the current attention state to on

rc The attention state before the call to *ATTN*.

BUILDRD—Build a Routine Description

Use *BUILDRD* to create an object file containing a routine description that Processor 11 can use to determine how to use the routine. The object file produced by *BUILDRD* is link-edited with the user's routine to make it self-describing.

For additional information on this function and Processor 11, see *APL2/370 Programming: System Services Reference*.

```
rc←file BUILDRD rdname rname rd
```

file The name of the file to be written, for example, 'XXXX TEXT'

rdname The name to be assigned to the routine description in the generated object file. Specify this name as the routine's entry point name when the routine and its description are link-edited.

rname The name of the non-APL routine.

rd A character vector containing Processor 11 tags that describe the routine.

Any Processor 11 tags can be used. The :LINK tag is required. If the :NICK is used, Processor 11 requires that its name match the name specified in the right argument of ⎕*NA*. The :LOAD, :MEMB, :ENTRY, and :DESC tags are ignored.

rc The normal result of *BUILRD* is zero, which indicates success. *BUILRD* uses Δ*FM* to write the file. Nonzero return codes are generated by Δ*FM*. Refer to the Δ*FM* documentation for information about these codes.

BUILDRL—Build a Routine List

Use this routine to build an object file containing a routine list used by Processor 11 to locate routines within a module. The object file produced by *BUILDRL* is link-edited with the user's routines.

For additional information on this function and Processor 11, see *APL2/370 Programming: System Services Reference*.

```
rc←file BUILDRL rlname rtndef [rtndef] ...
```

file The name of the file to be written, for example, 'XXXX TEXT'

rlname The name to be assigned to the routine list in the generated object file. Specified this name as the module's entry point name when the routine list and the non-APL routines are link-edited.

rtndef A character vector that defines a routine's entry in the routine list. It can take any of the following forms:

```
qnaname
qnaname rname
qnaname rname intname
```

qnaname The name in the right argument of ⎕*NA* when an association is established.

rname The name of a non-APL routine or routine description

intname The name of an interface management routine, non-APL routine, or routine description.

rc The normal result of *BUILRD* is zero, which indicates success. *BUILRD* uses Δ*FM* to write the file. Nonzero return codes are generated by Δ*FM*. Refer to the Δ*FM* documentation for information regarding these codes.

CAN—Compress and Nest

Use this routine to compress and partition a character vector based on a Boolean mask.

```
result←mask CAN characters
```

mask Is a Boolean mask. Zeros in the mask correspond to characters to be deleted from the right argument and to the beginning of the second and subsequent elements of the *RESULT*.

characters Is a simple character vector

result Is a vector of character vectors.

Note that `⍴result←→,1++/~mask` and that if *mask* begins with a zero, *result* begins with a null vector. `⍴mask←→⍴characters`. If the arguments are empty, `result←→,⊂''`.

Note: This function has been superseded by the partition primitive.

CMSIVP—Installation Verification under CMS

Use this routine to verify the installation of APL2 in the VM/CMS environment.

```
CMSIVP
```

There are no arguments and no result is returned.

Start the procedure by invoking the function *CMSIVP*, which verifies and tests various parts of the installed APL2 system. As this happens, it displays information on your terminal. You should check this information against the APL2 system you believe you have installed, and investigate or correct any discrepancies.

For information on the installation procedure, see *APL2/370 Installation and Customization under CMS*.

CSRIDAC—Request or Terminate Access to a Data Object

This routine provides the first (and last) step in using the data window callable services supported by MVS/ESA. The interface provides access to temporary hiperspaces as well as page formatted permanent files that can be viewed through a window. See also *CSRVIEW*, *CSRSAVE*, *CSRSCOT*, and *CSRREFR*. For additional information see *MVS/ESA Callable Services for High Level Languages*.

```
(rc rs)←CSRIDAC beginstr size idname [offsetname]
```

This form of *CSRIDAC* requests access to a data object.

beginstr A character vector containing one of the following forms:

```
BEGIN TEMP SCROLL
BEGIN DD ddname [SCROLL] [access]
BEGIN DS dsname [SCROLL] [access]
```

Note: The bracketed fields are optional. The brackets themselves are never part of the character vector.

TEMP Access a temporary object, which is deleted when *CSRIDAC END* is done.

DD ddname Use an existing ddname to locate a linear VSAM cluster.

DS dsname Allocate the specified linear VSAM cluster. The dsname must be fully qualified, and is given without quotation marks.

SCROLL A scroll area to be maintained while the data object is accessed. This is optional except for TEMP.

access Can be either of the following or defaulted:

READ An existing object is to be accessed as read only.

UPDATE Exclusive control with read/write authority is requested.

UPDATE is ignored for a new object (*size* nonzero), and is the default for an existing object.

size For a new object, the limit size of the object to be created, specified in pages (blocks of 4096 bytes). For an existing object, 0 must be specified. Note that a nonzero size must be given if a new permanent object is created or if a *TEMP* object is being accessed.

idname The name of the variable where the object identifier token is returned as an eight-element character vector. The data returned is not displayable and should not be manipulated.

offsetname An optional name of a variable where the current size of the object is returned as a scalar integer representing the number of 4K blocks.

(rc rs) Return code from the operation. This includes the return_code and reason_code parameters defined in *MVS/ESA Callable Services for High Level Languages*. A brief list of *(rc rs)* values is included here for convenience, but it is not necessarily complete.

0	0	Operation successful.
8	280	No hiperspace available for temporary object or scroll area.
8	282	Unable to create a linear VSAM data set.
12	28	The object is currently unavailable.
12	55	Warning: accessed with irregular SHAREOPTIONS.
12	62	Object in use. (n readers or 1 updater permitted)
16	*nnnn*	Unable to allocate as requested. Many possible reasons, including attempt to create an existing data set.
44	4	Window services not available.

```
(rc rs)←CSRIDAC 'END' id
```

This form of *CSRIDAC* terminates a data object.

id The eight-element character token returned when access was requested.

(rc rs) Return code from the operation. This includes the return_code and reason_code parameters defined in the manual referenced above. A brief list of *(rc rs)* values is included here for convenience, but it is not necessarily complete.

0	0	Operation successful.
12	10	Another service is currently using the access ID.
44	4	Window Services not available.

CSRREFR—Refresh an Object

This routine is part of the data window callable services supported by MVS/ESA. It applies to a data object and optional scroll area previously defined by *CSRIDAC*, and to windows on that data created by *CSRVIEW*. The service discards any changes made to data within specified parts of the windows or scroll area, and replaces the data with either:

- binary zeros if it is a TEMP object, or
- a current copy of data from the object if it is permanent.

For more information see *MVS/ESA Callable Services for High Level Languages*.

```
(rc rs)←CSRREFR id offset span
```

id The eight-element character token returned when access was requested.

offset An integer that is an origin-0 4K block number within the object, which identifies the location where replacement should begin. For example, if 2 is specified, the replacement begins with the data at offset 8192 in bytes from the beginning of the permanent object. Replacement is always made into the corresponding part of the scroll area and (as appropriate) into any part of the window associated with that data.

span The integral number of 4096 byte blocks to be refreshed.

(*rc rs*) Return code from the operation. This includes the return_code and reason_code parameters defined in *MVS/ESA Callable Services for High Level Languages*. A brief list of (*rc rs*) values is included here for convenience, but it is not necessarily complete.

0 0	Operation successful.
12 10	Another service is currently using the access ID.
12 23	An I/O error occurred.
12 26	Specified range does not include any mapped blocks.
44 4	Window services not available.

CSRSAVE—Save Changes Made to a Permanent Object

This routine is part of the data window callable services supported by MVS/ESA. It applies to a permanent data object and optional scroll area previously defined by *CSRIDAC*, and to windows on that data created by *CSRVIEW*. Any changes made to data within specified parts of the windows or scroll area are copied to the permanent object. It is not supported for TEMP objects. For more information see *MVS/ESA Callable Services for High Level Languages*.

```
(rc rs)←CSRSAVE id offset span [highname]
```

id The eight-element character token returned when access was requested.

offset An integer that is an origin-0 4K block number within the object identifying the location where the data should be stored. Data is taken from corresponding parts of the window, or from the scroll area.

span The integral number of 4096 byte blocks to be saved.

high An optional name of a variable where the new size of the object is returned as a scalar integer representing the number of 4K blocks.

(*rc rs*) Return code from the operation. This includes the return_code and reason_code parameters defined in *MVS/ESA Callable Services for High Level Languages*. A brief list of (*rc rs*) values is included here for convenience, but it is not necessarily complete.

0	0	Operation successful.
4	2055	Part of data set is damaged, but this operation is successful.
8	323	Cannot issue *CSRSAVE* for temporary object.
12	10	Another service is currently using the access ID.
12	23	An I/O error occurred.
12	26	Specified range does not include any mapped blocks.
44	4	Window services not available.

CSRSCOT—Save Object Changes in a Scroll Area

This routine is part of the data window callable services supported by MVS/ESA. It applies to a data object and scroll area previously defined by *CSRIDAC*, and to windows on that data created by *CSRVIEW*. The operation copies data from specified parts of the windows into the corresponding portion of the scroll area. It does not make any change to a permanent object. For additional information see *MVS/ESA Callable Services for High Level Languages*.

```
(rc rs)←CSRSCOT id offset span
```

id The eight-element character token returned when access was requested.

offset An integer that is an origin-0 4K block number within the scroll area, which identifies the location where the data should be stored. Data is taken from corresponding parts of the window.

span The integral number of 4096 byte blocks to be copied.

(*rc rs*) Return code from the operation. This includes the return_code and reason_code parameters defined in *MVS/ESA Callable Services for High Level Languages*. A brief list of (*rc rs*) values is included here for convenience, but it is not necessarily complete.

0	0	Operation successful.
4	2055	Part of data set is damaged, but this operation is successful.
12	10	Another service is currently using the access ID.
12	23	An I/O error occurred.
12	26	Specified range does not include any mapped blocks.
44	4	Window services not available.

CSRVIEW—Start or Terminate a View of an Object

This routine is part of the data window callable services supported by MVS/ESA. It controls a window that can be used to view a data object and/or scroll area previously defined by *CSRIDAC*. See also *CSRREFR*, *CSRSAVE*, and *CSRSCOT*, which you can use to transfer data between the window, scroll area, and permanent data object. For additional information see *MVS/ESA Callable Services for High Level Languages*.

```
(rc rs)←CSRVIEW 'BEGIN' id offset span wname [usage] [dispos]
```

id The eight-element character token returned when access was requested.

offset An integer that is an origin-0 4K block number within the scroll area or object, associated with the beginning of the window.

Note: Multiple concurrent views of an object are permitted, but they cannot overlap within one *id*. For permanent objects you can define overlapping views by using *CSRIDAC* to create multiple object definitions of a single stored object.

span The number of 4K blocks to reserve for the window.

wname The name of a variable that is established as a window. Subsequent APL statements accessing this variable reference or modify the window. The variable is established as a character vector.

usage An optional character vector that contains either *SEQ* or *RANDOM*. This is used as an optimizing hint to the operating system. The default is *RANDOM*.

dispos An optional character vector that contains either *REPLACE* or *RETAIN*. The default is *REPLACE*. If *RETAIN* is specified, the existing content of the *wname* variable is retained, but

- the variable must exist at the current function level,
- it must be a simple character vector, and
- the expression (⍴*wname*)=4096×*span* must be true.

VALUE ERROR, *DOMAIN ERROR*, or *LENGTH ERROR* respectively are signaled if these conditions are not met.

(*rc rs*) Return code from the operation. This includes the return_code and reason_code parameters defined in the manual referenced above. A brief list of (*rc rs*) values is included here for convenience, but it is not necessarily complete.

0 0	Operation successful.
12 10	Another service is currently using the access ID.
12 23	An I/O error occurred.
12 26	Specified range does not include any mapped blocks.
12 28	The object cannot be accessed at this time.
12 64	The request exceeds your data space limit.
44 4	Window services not available.

```
(rc rs)←CSRVIEW 'END' id wname [dispos]
```

id The eight-element character token returned when access was requested.

wname The same variable name specified by *CSRVIEW 'BEGIN'*. This defines which view of the object is terminated.

dispos An optional character vector containing either *REPLACE* or *RETAIN*.

- If *RETAIN* is specified, the *wname* variable is retained with its existing content after disassociating it from the object.
- If *REPLACE* is specified, the variable is retained, but with its existing content replaced by the data to which the window is mapped. (This is different if the variable was modified and the window is terminated without invoking *CSRSAVE* or *CSRSCOT*.)
- If neither is specified, the variable is deleted.

(rc rs) Return code from the operation. This includes the return_code and reason_code parameters defined in *MVS/ESA Callable Services for High Level Languages*. A brief list of *(rc rs)* values is included here for convenience, but it is not necessarily complete.

0	0	Operation successful.
12	10	Another service is currently using the access ID.
44	4	Window services not available.

CTK—Character to DBCS Conversion

Use this routine to convert an APL character vector to a vector of mixed EBCDIC/DBCS data. It is the inverse of *KTC*, and can be used in both monadic and dyadic form.

```
result←CTK data
```

data A simple character vector

result A vector of mixed EBCDIC/DBCS data. Characters in *DATA* for which 256>⎕AF *DATA* are placed unchanged in the result. Character strings in *DATA* for which 256 ⎕AF *DATA* are prefixed with SO (X'0E') and suffixed with SI (X'0F') are stored as 2-byte elements in *result*.

If the *DBCS*(*nnn*) invocation option is used, then the leftmost halfword in each extended character in *DATA* is checked for *nnn*. If any check fails, then *DOMAIN ERROR* results.

```
result←cid CTK data
```

cid The character set ID for the data. It is in the range 0 - 32767. If the leftmost halfword in any extended character in *DATA* is not equal to *CID*, then a *DOMAIN ERROR* results.

data An APL character vector.

result A vector of mixed EBCDIC/DBCS data. Characters in *DATA* for which 256>⎕AF *DATA* are placed unchanged in the result. Character strings in *DATA* for which 256 ⎕AF *DATA* are prefixed with SO (X'0E') and suffixed with SI (X'0F') are stored as 2-byte elements in *result*.

CTN—Character to Number

Use this routine to convert a character vector or matrix to a numeric vector or matrix. It yields a null vector or matrix if the argument does not contain valid numeric representations.

```
numbers←CTN characters
```

characters A character vector or matrix that contains the formatted representation of one or more numbers. Only numeric formats produced by monadic ⍕ are acceptable.

numbers A numeric vector or matrix that is formed by executing the *characters* argument.

DAN—Delete And Nest

Use this routine to partition a character vector based on a list of separator characters. Separator characters are not included in the result. See *WORDS* on page 40.

```
result←separators DAN characters
```

separators A list of separator characters.

characters A simple character vector.

result A vector of character vectors.

Note: This function is a subset of the partition primitive.

Strings of one or more separator characters are deleted from the right argument and mark the separation between elements of the result. The result does not contain empty items unless the entire right argument consists of separator characters.

If `0=⍴`*separators*`, `*result*`←→,⊂`*characters*.
If `0=⍴`*characters*`, `*result*`←→,⊂''`.
`  ⍴∈`*result*`←→⍴`*characters*`~`*separators*

DFMT—Format Arrays Containing DBCS Data

Use this routine to format APL arrays for display on a device, such as the IBM 5550 Multistation*, that supports double-byte character set (DBCS) data.

If the array to be formatted contains DBCS data, the columns of the formatted result are expanded so that DBCS and single-byte character set (SBCS) data is aligned correctly within the columns.

```
result←mask DFMT array
```

array The array to be formatted.

mask An optional argument that specifies how columns of the array are grouped into logical columns and whether the logical columns are left or right aligned.

If the mask is scalar, it is replicated to match the number of columns in the array.

If the mask argument is elided, a mask of 1s and ¯1s is used. ¯1s correspond to columns of the array that contain numeric scalars, 1s correspond to other columns.

ρ*MASK* ↔ ¯1↑ρ*ARRAY* after scalar extension.

1	A logical column begins at the corresponding column in the array and the logical column should be left aligned.
¯1	A logical column begins at the corresponding column in the array and the logical column should be right aligned.
0	The corresponding column in the array is part of the same logical column as the one to its immediate left.
1s and ¯1s	The beginning of logical columns in the array. Indicates whether the column is padded on the right or left with blanks if the column must be expanded to accommodate DBCS data. Actual alignment of data within the columns is the same as that produced by the ⍕ primitive; that is, if the column contains simple numeric scalars, it is right aligned, otherwise it is left aligned.

result A one-column matrix of formatted rows of the array.

↑ρ*RESULT* ↔ ×/¯1↓ρ⍕*ARRAY*

If the result is not modified, it is displayed with a leading blank column because it is nested. ⊃,*RESULT* causes the result to be displayed without the leading blank column, but trailing blanks may be appended because of the ⊃.

DISPLAY—Display Array Structure

DISPLAY produces a character array that pictorially represents the structure of its argument. Use *DISPLAY* on terminals that do not have box drawing characters available.

```
z←DISPLAY array
```

The following characters are used to convey shape information:

→ and ↓	Indicate a dimension of at least one
⊖ and ϕ	Indicate a dimension of zero
(None of the above)	Indicates no dimension

The following characters are used to convey type information:

~	Indicates numeric
+	Indicates mixed
∈	Indicates nested
–	Indicates a scalar character that is at the same depth as nonscalar arrays
(None of the above)	Indicates a character array that is not a simple scalar

DISPLAYC—Display Array Structure

DISPLAYC produces a character array that pictorially represents the structure of its argument. Use *DISPLAYC* on terminals that do not have box drawing characters available.

```
z←DISPLAYC array
```

The following characters are used to convey shape information:

→ or ↓	Indicates a dimension of at least one
⊖ or ⌽	Indicates a dimension of zero
(None of the above)	Indicates no dimension

The following characters are used to convey type information:

~	Indicates numeric
+	Indicates mixed
∈	Indicates nested
_	Indicates a scalar character that is at the same depth as nonscalar arrays
(none of the above)	Indicates a character array that is not a simple scalar

Note: *DISPLAYC* is equivalent to *DISPLAY*, and is supplied for compatibility with other APL2 platforms.

DISPLAYG—Display Array Structure

DISPLAYG produces a character array that pictorially represents the structure of its argument. Use *DISPLAYG* on terminals that have box drawing characters available.

```
z←DISPLAYG array
```

The following characters are used to convey shape information:

→ and ↓	Indicate a dimension of at least one
⊖ and ϕ	Indicate a dimension of zero
(None of the above)	Indicates no dimension

The following characters are used to convey type information:

~	Indicates numeric
+	Indicates mixed
∈	Indicates nested
–	Indicates a scalar character that is at the same depth as nonscalar arrays.
(None of the above)	Indicates a character array that is not a simple scalar

DSQCIA—QMF Callable Interface

Use this function to access the QMF* callable interface. This new interface to QMF allows a program to start QMF and issue QMF commands without the QMF environment and ISPF present. In addition to regular QMF commands, three additional commands in this interface start QMF (START) and allow the program to set and retrieve global QMF variables (SET GLOBAL and GET GLOBAL.)

```
(rc handle data)←DSQCIA handle cmdstr [names vals]
```

handle An integer that identifies which instance of QMF a call refers to. This parameter is used in the DSQ_INSTANCE_ID field of the QMF communications area block, DSQCOMM. It must be 0 if the QMF command is START, and must contain a valid handle for all other commands.

On return from *DSQCIA*, *handle* contains the handle of the instance of QMF for which the command was issued. At completion of a START command you must retain this value so that it can be passed on subsequent commands.

cmdstr A character vector that contains the QMF command to be processed.

names A vector of character vectors or scalars that are QMF keywords or variable names. The shape of the array passed must be equal to the number of names. (If only one name is passed, it must be enclosed.)

This parameter is required only for the SET GLOBAL and GET GLOBAL commands. It is optional for the START command.

vals A vector of variable values. This can be a vector of character vectors or scalars, or it can be a vector of numbers. It cannot contain a mixture of numeric and character data.

For character values, the shape of the array passed must be equal to the number of values. (If only one value is to be passed it must be enclosed.)

Since integers are the only numeric type supported by the QMF interface, you must be able to represent numbers as fullword integers. The APL2 external function issues a *DOMAIN ERROR* if the array passed does not meet those requirements.

This parameter is required only if the *names* parameter has also been specified. If the command is GET GLOBAL, the array is used to determine the type of the variable and the amount of storage to allocate for the result. The values are not modified or replaced on the workspace.

rc A numeric return code. The value is 0, 4, 8, 12, or 16 as defined by the QMF callable interface.

data A value whose meaning is dependent on the value of *rc* and *cmdstr*.

- If *rc* is 0 and *cmdstr* contains the string *GET GLOBAL*, *data* contains the values of the QMF variables requested.
- In all other cases, *data* is a character vector that contains the QMF communications area DSQCOMM, as documented in the QMF manuals.

Note: QMF Version 3 Release 1 or later is required for use of the *DSQCIA* function. See *QMF Application Development Guide*, SC26-4722, for more information on the QMF callable interface.

EDITORX—System Editor Access

This function is a program interface to a named system editor.

```
[editorname] EDITORX objectname
```

The character string right argument is the same expression that would be typed to enter the editor from APL2. The ∇ or ⍛ can be included or omitted. If omitted ∇ is assumed.

The optional character string left argument is the name of the system editor, CLIST, or EXEC to call. If omitted the editor name is taken from the most recent setting of *)EDITOR xxxx*. If *)EDITOR xxxx* was never issued, DOMAIN ERROR is reported.

Examples:

```
'MYEDITOR' EDITORX 'MYFUNCTION'

)EDITOR MYEDITOR
EDITORX '⍛MYFUNCTION'
```

Note: This function is not available when running under APL2 Application Environment.

EDITOR2—Full-Screen APL2 Editor

This function is a program interface to Editor 2, the APL2 full-screen editor.

```
EDITOR2 objectname
```

The character string right argument is the same expression that would be typed to enter the editor from APL2. The ∇ or ⍛ can be included or omitted. If omitted ∇ is assumed.

Examples:

```
EDITOR2 'MYFUNCTION'
EDITOR2 '⍛MYFUNCTION'
EDITOR2 '∇MYFUNCTION[⎕17]'
```

Note: This function is not available when running under APL2 Application Environment.

EXP—Execute in the Previous Namescope

Use this routine to process named functions, refer to variables, and specify variables in the previous namescope.

The *EXP* routine is designed to be used in namespaces and provides access to names in the namescope of the function or operator that caused entry into the current namescope.

If the *EXP* routine is run in a namescope where there is no previous namescope (for instance, in your active workspace rather than in a name space), it operates in the current workspace.

Note: Processing a function or operator declared with □*NA* causes an explicit change to the namescope of the function or operator. Processing the *EXP* function or an operand to an external operator causes an implicit namescope switch. If the *EXP* function is run in a namescope that was entered implicitly, the namescope switches to the one that originally caused explicit entry into the current namescope.

Functions processed under control of *EXP* operate the same as those processed under control of □*EC*, and exhibit the following behavior:

- Requests for quad input are handled the same as quad input under □*EC*.
- Errors generated during processing do not cause suspension of the function being processed and are reported against *EXP*.
- Stop control vectors (*S*Δ) are ignored.
- An attention signal does not cause suspension; an interrupt signal causes the *EXP* function to be interrupted.
- Branch escape (→) causes the *EXP* function to run, but its callers are not abandoned.

EXP can perform four different actions, depending on how the right argument is constructed:

- Process an expression in the previous namescope.
- Specify a variable in the previous namescope using a value from the current namescope.
- Process a named monadic function in the previous namescope with an argument from the current namescope.
- Process a named dyadic function in the previous namescope with arguments from the current namescope.

The right argument of *EXP* can contain up to three items. The number of items and their content determine what happens.

A single item on the right (must be enclosed if there is more than one item) is an expression to be processed in the previous namescope. If the expression is nothing more than the name of a variable or niladic function in the previous namescope, then you are referencing the named item.

A two-item right argument indicates a monadic function. The first item is the name (see note) of the function and the second item is the right argument (supplied from the current namescope).

A three-item right argument indicates either a specification or a dyadic function.

- If the second item of the right argument is the left arrow, then it's a specification. The first item is the name of the variable to be set and the third item is the value (from the current namescope) that it receives.
- If the second item is not a right arrow, then it must contain the name of a dyadic function in the previous namescope. In this case, the first and third items are the left and right arguments respectively (from the current namescope).

Note: System functions (such as ⎕*FX*) and system variables (such as ⎕*IO*) are included as named objects.

You can use *EXP* to process an expression in the previous namescope:

```
result←EXP ⊂expr
```

result	The result of processing *expr*
expr	A character scalar or vector that contains the expression to be processed in the previous namescope.

Example:

```
PRE_IO←EXP ⊂'⎕IO'     ⍝ GET VALUE OF ⎕IO FROM
                      ⍝ PREVIOUS NAMESCOPE
```

Example:

```
IOTA2←EXP ⊂'⍳2'       ⍝ 0 1  OR  1 2 DEPENDING
                      ⍝ ON ⎕IO IN PREV
```

The following example processes a monadic function in the previous namescope using an argument from the current namescope.

```
result←EXP fn_name value
```

result	The result of processing the named monadic function in the previous namescope.
fn_name	A character scalar or vector that contains the name of a monadic function in the previous namescope.
value	The right argument (from the current namescope) to be supplied to the monadic function

To create function in previous namescope:

```
Z←EXP '⎕FX' ('R←NEWFN RA' 'R←RA')
```

To process a dyadic function in the previous namescope using arguments from the current namescope:

```
result←EXP lvalue fn_name rvalue
```

result The result of executing the named dyadic function in the previous namescope.

fn_name A character scalar or vector containing the name of a dyadic function in the previous namescope.

lvalue The left argument (from the current namescope) to be supplied to the dyadic function.

rvalue The right argument (from the current namescope) to be supplied to the dyadic function.

This assigns a value from the current namescope to a variable in the previous namescope:

```
result←EXP vname '←' value
```

result The same as *value*.

vname A character scalar or vector that contains the name of the variable in the previous namescope.

value The value from the current namescope that is assigned to *vname*.

Example:

```
  T←EXP '⎕IO' '←' PREV_IO    ⍝ RESTORE ⎕IO IN
                             ⍝ PREVIOUS NAMESCOPE
```

FED—Diagnostic Information

Use this routine to obtain a list of recently queued or displayed messages.

```
messages←FED 1
```

messages A vector of 10 character vectors that contains the ten most recently queued or displayed messages. Each message includes the ID field whether or not DEBUG(1) is in effect for the session. The oldest message is shown first.

HELP—Retrieve Keyed Help Text for an Application

This routine allows applications to retrieve keyed text or a list of keys.

Frequently, applications need to present text to their users. This text can be too large to store in the application workspace, and can be difficult to maintain if the application also resides in the workspace. The *HELP* function allows applications to retrieve keyed text from an application-dependent help file. Help files can be national language specific.

An APL2 help file is a normal CMS file or a TSO partitioned data set member that contains GML-like tags. These tags define keys that delimit sections of free-form text. A help file can also refer to other files containing more text. You can use the *HELP* function to retrieve the list of keys available in a help file or the text associated with a particular key.

For more information on the format of help files, see the default APL2HELP file that came with the APL2 product.

Using Help to Retrieve a List of Keys

```
keys←applid HELP ''
```

applid Character vector of length 1 to 8. *HELP* uses this as a DDname on TSO or filetype on CMS.

If not supplied, a default value of APL2HELP is used. The current value of ⎕*NLT* is used as the member name on TSO or the filename on CMS. If a file or member in the current national language is not available, the ENP file is used.

keys Character matrix containing available keys in the help file.

Using Help to Retrieve Text

```
text←applid HELP key
```

key A character string of length 1 to 65. A *key* can contain imbedded blanks; trailing blanks are ignored. The application help file is searched for a record that contains a help tag, :HELP, followed by the contents of *key*. All records from the tag up to the next help tag are returned.

text A character matrix containing the text found after the key.

Using Help as an Online Help Facility

```
text←HELP key
```

key A character string that contains the name of an APL2 public workspace or external function. *HELP* uses the IBM-supplied help file to retrieve the tutorial text for the specified workspace or function.

text A character matrix that contains the text found after the key.

HELP Return Codes:

1 File not found. Either the indicated DDname was not allocated on TSO or neither a □*NLT* nor a ENP version of the requested file could be found.

2 The specified key was not found in the file.

3 Invalid tag record was found.

4 Invalid file.

5 Space unavailable to read file or build result.

6 File IO error

7 Data conversion error

If an error occurs while trying to process a file referred to in a help tag, then the result is negative.

IDIOMS—APL2 Phrases

This routine provides a full screen interface to commonly- used APL code segments that provide solutions to common application problems. There are more than 700 different phrases, divided into 24 categories.

IDIOMS lets you select phrases from different APL environments, with a user-selected index origin. You can write the selected phrases into a function in your workspace or into the explicit result as a character matrix.

The functions provided by this utility are all function-key driven:

F1 Displays the help information for the current screen.

F2 Pops up a window of all previous searches, which lets you reselect any previous search.

F3 Returns to APL2.

F4 Saves the APL2 idiom identified by the cursor as a function called *IDIOM_LIST*. If *IDIOM_LIST* already exists, the selected idiom is appended to the end of the function. This can be a prototype for a new program using the selected expressions to accomplish the desired tasks.

F5 Limits the search to the displayed idioms. This helps you narrow the searches without losing intermediate results.

F6 Displays a screen of the 24 categories. Place the cursor on a category and press F6 again to display the idioms within that category, or press F3 to exit without selecting anything.

F7 Scrolls one screen toward the top.

F8 Scrolls one screen toward the bottom.

F9 Appends the APL2 expressions identified by the cursor to the output of *IDIOMS*. This places them on the session-manager screen, making it easy to experiment with the phrases. *IDIOMS* places the output into a variable if invoked through *RESULT←IDIOMS*.

F10 Pops up a window to select environment(s) for idioms that should be displayed.

```
result←IDIOMS
```

result A character matrix that contains phrases selected with PF9 while in the application

Note: This function is not available when running under APL2 Application Environment.

IN—Read a Transfer File into the Active Workspace

This function is a program interface to the)*IN* system command.

```
IN 'file [names]'
```

file Is the name of a transfer file, following the naming conventions and defaults of the host system. Default filetype and filemode (CMS) or qualifiers (TSO) are added if a simple name is used.

On CMS: NAME becomes NAME.APLTF.A

On TSO: NAME becomes PREFIX.APLTF.NAME

names Are names of objects to be read and defined in the active workspace. Names can include system variables if they are present in the transfer file. If the name list is omitted, all objects in the transfer file are copied.

If a name conflict occurs, the object from the transfer file replaces the one currently in the active workspace.

Examples:

```
IN 'FILE'
IN 'FILE FUN1 FUN1 VAR3'
IN 'ABC.APLTF.A'
```

Note: This function is not available when running under APL2 Application Environment.

KTC—DBCS to Character Conversion

Use this routine to convert a character vector of mixed EBCDIC/DBCS data to an APL character vector. KTC is the inverse of *CTK* and can be used in both monadic and dyadic form.

```
result←KTC data
```

data A simple vector of mixed EBCDIC/DBCS data where DBCS strings are delimited by SO/SI (X'0E', X'0F') characters.

result An APL character vector. EBCDIC characters from *DATA* are placed in the result unchanged. DBCS characters are taken 2 bytes at a time to produce scalar elements of the *RESULT*. SO/SI characters in *DATA* are not placed in the *RESULT*.

If the *DBCS*(*nnn*) invocation option is used, then *nnn* is placed in the leftmost halfword of each extended character in *RESULT* unless the character in *DATA* is EBCDIC. (*nnn* is in the range 0 - 32767).

```
result←cid KTC data
```

cid The character set ID in the range 0 - 32767, which is put in the leftmost halfword in each extended character of the *RESULT* unless the character from *DATA* is EBCDIC.

data A vector of mixed EBCDIC/DBCS data where DBCS strings are delimited by SO/SI (X'0E', X'0F') characters.

result An APL character vector. EBCDIC characters from *DATA* are placed in the result unchanged. DBCS characters are taken 2 bytes at a time to produce scalar elements of the *RESULT*. SO/SI characters in *DATA* are not placed in the *RESULT*. DBCS characters have *CID* placed in the high order halfword of each extended character in the *RESULT* as discussed above.

MSG—Message Services Request

This routine allows APL2 messages to be displayed, queued, retrieved, or checked. See *APL2/370 Messages and Codes* for definitions of APL2 messages.

```
result←req MSG msg_data
```

req = 0 Requests display of an APL message

req = 1 Requests queuing of an APL message

req = 2 Requests an APL message be returned

req = 3 Requests a check for an APL message

msg_data A nested array that contains the message number followed by the fill-in fields. The message number must be a positive integer scalar. The fill-in fields must be character scalars or vectors.

The number of fill-in fields required depends on the message number you specify, but it must never exceed 10. (See *APL2/370 Messages and Codes* for descriptions of specific messages.) If too many fill-in fields are specified for a specific message, the extra fields are ignored. If too few are specified, the remaining fields are filled with asterisks.

The total length of the message (fixed text plus fill-in fields) should not exceed 240 bytes. If the total length exceeds 240 bytes, fill-in fields are truncated or a return code 2 (message too large) results.

result The numeric return code for the request or the message text when *REQ* is 2. When *REQ* is 2, if the message returned contains DBCS data, it is returned in mixed SBCS/DBCS format and can be converted to extended characters using the *KTC* function.

0 = success
1 = message does not exist
2 = message too long
3 = insufficient free space

OPTION—Query or Set APL2 Invocation Options

You can use this function monadically to query the current state of certain processing options, or dyadically to change those options and return their prior values.

```
result←[value] OPTION key
```

key The name (as a character vector) of an invocation option. It can have leading and/or trailing blanks, but must not be abbreviated. The following options are supported:

- CASE
- DBCS
- DEBUG
- QUIET
- SYSDEBUG
- TRACE

value An optional value to be assigned to the invocation option. The value must be specified as character scalar or vector, even when its value is numeric. Negative numbers can be indicated either with a leading minus (hyphen) or with a leading APL negative (overbar).

The option value must be a single integer or keyword, though it can have leading and/or trailing blanks. Keywords *cannot* be abbreviated. You can use any value that *APL2/370 Programming: System Services Reference* indicates is valid for the *KEY* you specified.

When the DEBUG, SYSDEBUG, or TRACE options are specified at startup, or through the)*CHECK SYSTEM* command, you can express the option values as multiple integers or as a sum. For the *OPTION* function, though, you must specify the values as a sum or use separate calls. Separate calls are required to turn some switches on and others off.

result A character vector that contains the value of the invocation option. In the dyadic form, the value returned is the one in effect *before* the *VALUE* argument was applied.

Note: As an invocation option, QUIET is a keyword with no value, or with the value ON or OFF. With the *OPTION* function, the keyword is always specified and returned as ON or OFF. Allowing programs to set and reset QUIET gives applications significantly more control over what information is displayed on the terminal. As in the past, an implicit QUIET(OFF) action occurs any time a terminal input request is issued.

Examples

To copy objects from a library workspace, suppressing all messages:

```
    ∇ COPY PARMS;OPTION;STK
[1] →(2≠101 ⎕SVO 'STK')/0                ⍝ SHARE WITH STACK PROCESSOR
[2] →(1≠3 11 ⎕NA 'OPTION')/0             ⍝ ACCESS OPTION FUNCTION
[3] STK←')COPY ',PARMS                   ⍝ STACK THE )COPY COMMAND
[4] STK←'''',('ON' OPTION 'QUIET'),''''  ⍝ SET QUIET, STACK OLD VALUE
[5] →0↑⍬ OPTION 'QUIET'                  ⍝ PROCESS STACK, RESTORE QUIET
[6] ∇
```

Note: This is a *completely* quiet function. It ignores all errors in the library system, the shared variable system, and the name association system. Its callers have to verify that the objects they needed were actually copied.

To turn DEBUG flags 1 and 32 on, and turn flag 4 off, you can use either of these forms:

```
'1' '-4' '32' OPTION¨⊂'DEBUG'
'33' '¯4' OPTION ¨⊂'DEBUG'
```

Notice that the Each operator (¨) is used to provide the effect of multiple calls to *OPTION* in a single line. The right argument must be enclosed (⊂) so that Each does not try to associate its individual characters with the items of the left argument.

Note: Installations can force certain option values through an `OVERIDE` list in the installation options module. If this has been done, the *OPTION* function completes successfully, but no change to the value occurs. You can detect this condition by a subsequent monadic use of the *OPTION* function.

OUT—Write Objects to a Transfer File

This function is a program interface to the)*OUT* system command.

```
OUT 'file [names]'
```

file Is the name of a transfer file, following the naming conventions and defaults of the host system. Default filetype and filemode (CMS) or qualifiers (TSO) are added if a simple name is used.

On CMS: NAME becomes NAME.APLTF.A

On TSO: NAME becomes PREFIX.APLTF.NAME

names Are names of objects whose transfer forms are to be written to the file. If the name list is omitted all unshared variables, defined functions, defined operators, and the system variables □*CT*, □*FC*, □*IO*, □*LC*, □*PP*, □*PR*, and □*RL* are written to the file.

Examples:

```
OUT 'FILE'
OUT 'FILE FUN1 FUN1 VAR3 □PW'
OUT 'ABC.APLTF.A'
```

Note: This function is not available when running under APL2 Application Environment.

PACKAGE—Creating a Namespace

You can use the *PACKAGE* routine to create a namespace from a saved workspace. More information on using this routine and on using namespaces can be found in *APL2/370 Programming: System Services Reference*.

Workspaces to be converted to namespaces must be saved using APL2 Version 1 Release 3 or later. If you've saved a workspace under an earlier release of APL2, you must reload and resave the workspace using APL2 Version 1 Release 3 or later before you can successfully process it using the *PACKAGE* routine.

Under MVS/TSO, saved workspaces to be converted must exist in a SAM library; if you've saved a workspace in VSAM library you must resave it in a SAM library before it can be processed. Under MVS/TSO, before the *PACKAGE* routine is processed the ddname SYSPUNCH must be allocated to the sequential data set where the resulting namespace is placed.

```
result←[name_list] PACKAGE ws_name
```

ws_name The data set name of the saved workspace to be converted. Under VM/CMS this takes the form 'fn ft fm' and must exist on an accessible minidisk. Under MVS/TSO, it has the form 'userid.Vnnnnnnn.WSNAME' and must be a cataloged sequential data set.

name_list A list of names of APL objects in the resulting namespace that are accessible by using □*NA*. If the *NAME_LIST* argument is not specified, all APL objects in the resulting namespace are accessible by using □*NA*.

NAME_LIST
can be a simple character scalar or vector representing one name,
or a matrix or vector of vectors representing a list of names.

result The name of the data set that contains the resulting namespace. In VM/CMS, this is a file with the name 'fn TEXT A' where 'fn' is the file name of the workspace data set name provided in the *WS_NAME* argument. In MVS/TSO, it is the name of the data set allocated to ddname SYSPUNCH.

If the *PACKAGE* routine is not successful in converting the saved workspace, *RESULT* is an empty vector.

PBS—Handling Printable Backspaces

The *PBS* routine allows applications to query and modify your current)*PBS* setting.

Users need to tell APL2 whether their terminals support the new APL2 characters or whether printable backspaces are required in order to enter them. You can indicate whether you can enter the new characters by using the)*PBS* system command. Applications also need to determine whether a user can enter the APL2 characters. Using the *PBS* function, applications can query and modify the user's setting. It is provided as an external function available through Processor 11 and □*NA*.

```
rc←PBS arg
```

arg One of the following:

- `⍳0` Queries the current PBS setting.
- `0` Sets the current PBS setting off.
- `1` Sets the current PBS setting on.

rc The PBS setting before the call to *PBS*.

PFA—Pattern from Array

This routine creates a CDR pattern from an APL array.

```
pattern←PFA array
```

array Any APL array.

pattern A pattern that describes the argument *ARRAY*. You can use *PATTERN* as the left argument of *RTA* or *ATR* or as the left or right argument pattern in a Processor 11 NAMES file. The *PATTERN* is described in detail in *APL2/370 Programming: System Services Reference*.

PIN—Protected Read of a Transfer File into the Active Workspace

This function is a program interface to the *)PIN* system command.

```
PIN 'file [names]'
```

file Is the name of a transfer file, following the naming conventions and defaults of the host system. Default filetype and filemode (CMS) or qualifiers (TSO) are added if a simple name is used.

On CMS: NAME becomes NAME.APLTF.A

On TSO: NAME becomes PREFIX.APLTF.NAME

names Are names of objects to be read and defined in the active workspace. Names can include system variables if they are present in the transfer file. If the name list is omitted, all objects in the transfer file are copied.

If a name conflict occurs, the object from the transfer file is not copied. Objects not copied are listed in a 'NOT COPIED:' message.

Examples:

```
PIN 'FILE'
PIN 'FILE FUN1 FUN1 VAR3'
PIN 'ABC.APLTF.A'
```

Note: This function is not available when running under APL2 Application Environment.

PTA—Pointers to Array

The external function *PTA* allows you to access arguments passed to functions as pointers by *APL2PI*. *PTA* is usually used with *ATP* to retrieve and then update arguments passed from non-APL programs.

```
array←pattern PTA pointers
```

pointers	An address or list of addresses passed from *APL2PI*.
pattern	A pattern that describes the result arrays. Its format is similar to the patterns used with *RTA*.
array	The data at the pointers arranged according to the pattern(s) in the left argument.

The *PTA* function assumes a one-to-one correspondence among the data descriptors in the left argument; among the set of pointers in the right argument.

QNS—Query the Current Namescope

Use this routine to query the current namescope.

```
result←QNS 0
```

result The left argument to ⎕*NA* for the function or operator that caused entry into the current namescope.

If the *QNS* function is processed in the user's active workspace, it returns `'' 11`.

RAPL2—Remote-Session Manager

The remote-session manager is an APL2 function that allows you to carry on an interactive session with a remote APL2 interpreter running under another user ID, perhaps on another system. It uses the shared variable interpreter interface to control the remote interpreter.

RAPL2 establishes and manages a shared variable communication link with a remote APL2 interpreter. Once the link is established, you can enter APL2 expressions and system commands and signal attention, but all input is passed to the remote interpreter.

```
rc←[time] RAPL2 proc_id
```

proc_id The processor ID of the remote interpreter. This value is used as the left argument to □*SVO* in *RAPL2*'s offer to share a variable with the remote interpreter.

time The number of seconds *RAPL2* should wait for the remote interpreter to match *RAPL2*'s share offer. If the remote interpreter does not match the offer within *time* seconds, *RAPL2* issues an appropriate message and terminates. *time* is optional; the default is 30 seconds.

rc An explicit result indicating whether connection was established (1), or was not established (0).

All □, ⍞,)*EDITOR* 1, immediate execution input prompts, and array and message output are passed back to be displayed locally by *RAPL2*. All other input and output generated by system commands, auxiliary processors, or external routines occur at the remote interpreter's location.

When the user signals an interrupt, *RAPL2* prompts the user for one of the following:

- The interrupt should be sent to the remote interpreter.
- *RAPL2* should switch to the local interpreter to process the user's input.
- A shutdown signal should be sent to the remote interpreter, causing a *CONTINUE* workspace to be saved.

When *RAPL2* uses the local interpreter to process the your input, *RAPL2* issues □ input requests. Expressions and commands entered by you are processed by the local interpreter in the namescope from which *RAPL2* was called. To resume use of the remote interpreter for executing input, signal interrupt again.

RAPL2 relinquishes control of the terminal when the remote interpreter retracts its shared variable. This typically occurs when the remote interpreter receives an)*OFF* or)*CONTINUE* system command.

Note: *RAPL2* tries to share the local variable *APL2* with the remote interpreter. If the variable *APL2* is already shared with the remote interpreter, *RAPL2* cannot establish the communication link.

For information about starting a remote interpreter, see the Description of Invocation Options section of *APL2/370 Programming: System Services Reference*. For

more about the shared variable interpreter interface, see *APL2/370 Programming: System Services Reference*.

Note: This function is not available when running under APL2 Application Environment.

RTA—Record to Array

Use this routine to convert a character vector right argument to an APL array based on a pattern-left argument. It is useful for converting records read from a file into APL objects.

```
array←pattern RTA record
```

pattern A character vector that describes the format of the right argument. *PATTERN* is described in detail in *APL2/370 Programming: System Services Reference*.

For this routine, * cannot be specified for ×/⍴, ⍴⍴ or ⍴ pattern. In addition to the representations described in the manual, *RT/RL* = *X*0 can also be specified to request that the corresponding bytes of data be skipped. For example:

```
(G0 1 2) (I4 1 4) (X0 1 10) (E8 1 5)
```

specifies that the record contains a vector of 4 fullword integers, 10 bytes to be skipped, and a vector of 5 double precision floating point numbers.

record A character vector to be converted. No check is made for a record of incorrect length. If the record is longer than the structure described by *PATTERN*, the result is unpredictable.

array An APL array constructed from the *RECORD* according to the *PATTERN*.

Note: The *ATR* external routine (see “ATR—Array To Record” on page 136) is the inverse of *RTA*. You can also use the *PFA* external routine (see “PFA—Pattern from Array” on page 175) to generate patterns.

SAN—Slice and Nest

Use this routine to partition a character vector based upon a Boolean mask.

```
result←mask SAN characters
```

mask	A Boolean mask. Zeros in the mask correspond to the beginning of the second and subsequent elements of the *RESULT*.
characters	A simple character vector.
result	A vector of character vectors.
	Note that `⍴result←→,1++/0~mask`, `⍴∈result←→⍴characters`. If *mask* begins with a zero, *result* begins with a null vector. `⍴mask←→⍴characters`. If the arguments are null, `result←→,⊂''`.

Note: This function is a subset of the partition primitive.

SERVER—TCP/IP Port Server

The external function *SERVER* manages the communication of TCP/IP port numbers between users. It is typically run in a disconnected VM machine or a TSO-started task.

```
SERVER
```

Warning: *SERVER* does not return control. It should not be used from an interactive APL session.

The function takes no arguments; it prompts for a TCP/IP port number and the authorization password to use when users make administration requests.

Further information about the TCP/IP port server can be found in the discussion of AP 119 in *APL2/370 Programming: System Services Reference*.

SVI—Shared Variable Processor Information

This routine returns information on active users of the shared variable processor.

```
accounts←SVI userid
```

SVI userid returns a list of the APL account numbers or auxiliary processor numbers for the specified *USERID*.

```
userid←SVI account
```

SVI account returns the *USERID* for the specified APL account number or auxiliary processor number.

account A single APL account number (↑⎕*AI*) or auxiliary processor number.

accounts A vector of 1 or more APL account numbers or auxiliary processor numbers.

userid A character vector that contains a VM/CMS or MVS/TSO user id. In the batch under MVS/TSO, this is the MVS job name.

Only users or accounts currently signed onto the SVP are included in the result. For an APL user, this means that some SVP operation was requested after the user's current workspace was made active.

Use this routine to determine if an auxiliary processor is active, or to determine the user id of an APL user who offered a variable:

```
      ⎕SVQ ⍳0
9625
      SVI 9625
WHEATLEY
```

SVI 0 returns the *USERID* of the global shared-variable processor, or a null vector if the global shared variable processor is not available.

The SVP does not retain user information for cross-system shared variables. When *SVA account* is issued for a cross-system account number, the issuers own user id is returned. *SVI userid* does not return any processor numbers used for cross-system shared variables.

TIME—Performance Monitoring

The performance monitoring facility measures a running application and determines the processor time used.

The facility works by associating with each line of each defined program a pair of counters to record the number of times the line is processed and the total processor time used.

Typically, timing information is obtained for an application as follows:

```
)LOAD workspace
3 11 ⎕NA 'TIME'  ⍝ To gain access to the facility.

TIME 0           ⍝ To enable and zero counters.
⍝ (run application here)
TIME 1           ⍝ To see times for program run.
⍝ (analyze timing information here)
TIME 2           ⍝ To see times for each line.
⍝ (analyze timing information here)
)CLEAR           ⍝ When time analysis is complete.
```

Using the timing facility requires space in the workspace for the counters and also increases running time by some small amount. Thus, in general you should not *)SAVE* after doing a time analysis.

```
result←[nl] TIME n
```

result Varies depending on the value of *n*. Normally, this is a 5 column matrix. Column 1 is the number of times the given line or program was actually processed. Column 2 is the accumulated processing time (in seconds) of the given line or program. Column 3 is the percentage of the total time used by the given line or program. Column 4 is the name of the program. Column 5 is the line itself, preceded by the line number.

TIME 1 returns only the first four columns. *TIME* 2 and 3 return all five columns. *TIME* 0, ¯1, ¯2, and ¯3 all return an empty (0 0⍴0) matrix.

nl An optional list of program names. It limits the scope of the current operation to the names listed; otherwise, the operation applies to all programs currently defined in the name space. The list can be a character vector containing a single name (no imbedded blanks) or a vector of character vectors each containing one name. All names listed must currently exist in the name space. *TIME* ¯1 and ¯2 do *not* accept the optional left argument.

[*nl*] *TIME* 0 Enable timing and create counters for all lines in the specified programs. The counters are set to zero. See *TIME* ¯3 and the notes to learn how to disable timing for a program and destroy its counters.

`[nl] TIME 1`

Fetch times for all specified programs that have accumulated timing information. Note that the result does not have the fifth column and is in descending order on the second column (processing time). If you want column headings, use the following:

```
HD1←'COUNT' 'TIME' '%' 'PROGRAM'
HD1,[⎕IO] TIME 1
```

`[nl] TIME 2`

Fetch times for the lines of specified programs that have accumulated timing information. The result is in descending order on the second column. If you want column headings, use the following:

```
HD2←'COUNT' 'TIME' '%' 'PROGRAM' 'STMT'
HD2,[⎕IO] TIME 2
```

`[nl] TIME 3`

Fetch times for all lines of specified programs where timing information has been enabled (even if none has been accumulated). The result is sequenced by line within function or operator. If you want column headings, use the following:

```
HD3←'COUNT' 'TIME' '%' 'PROGRAM' 'STMT'
HD3,[⎕IO] TIME 3
```

`TIME ¯1` Enable timing. If timing was disabled, timing is resumed.

`TIME ¯2` Disable timing. Stops the accumulation of timing data.

`[nl] TIME ¯3`

Delete space used by the counters for specified programs. A name list left argument is allowed and can be used to delete the timing data for selected functions and operators.

Notes:

1. Using the timing facility increases space utilization and processing time. Reported timings are approximate and should be used for relative comparisons, not absolute times.
2. Programs that do not have timing information do not appear in the results of subsequent uses of `TIME 1, 2` or `3`. A program might not have timing information for any of the following reasons:
   - It was not enabled by `nl TIME 0`.
   - It was modified (or created) after being enabled.
   - Its timing information was deleted by `nl TIME ¯3`.
3. Programs without timing information have their time accumulated by the first program (working back through the calling tree) with timing information.

TSOIVP—Installation Verification under TSO

Use this routine to verify the installation of APL2 in the MVS/TSO environment.

```
TSOIVP
```

There are no arguments and no result is returned.

Start the procedure by calling the function *TSOIVP*, which verifies and tests various parts of the APL2 system you have installed. As this happens, it displays information on your terminal. You should check this information against the APL2 system you believe you have installed, and investigate and correct any discrepancy.

For information on the installation procedure, see *APL2/370 Installation and Customization under TSO*.

∆EXEC—Execute an APL Array as a REXX Program

```
α ∆EXEC ω
```

The function ∆*EXEC* processes the REXX program contained in '*α*' with the items of '*ω*' as the argument strings.

α A character vector, matrix, or vector of vectors that contain the REXX program.

ω A vector of one to ten character vectors that are the strings passed as arguments to the REXX program.

The right argument of ∆*EXEC* does not include the name of the REXX program. The temporary program contained in the left argument array is called in the same way as described in *APL2/370 Programming: System Services Reference*.

The ∆*F* and ∆*EXEC* built-in functions provide the basis for building maintenance and test tools for REXX EXECs.

∆F—Query File Status

The function ∆*F* returns file status information.

```
result←∆F 'file'
```

'file' A character string that contains the file to be queried.

Under VM/CMS this takes the form 'fn ft fm' and must exist on an accessible minidisk. 'ft' and 'fm' default to '*'.

Under MVS/TSO, it has the form 'userid.Vnnnnnnn.WSNAME'. If the data set name is not enclosed in quotation marks, it is prefixed with the current TSO PROFILE PREFIX setting. An optional member name enclosed in parentheses can follow the data set name.

result A nested vector that contains the results of the query. It contains nine elements:

1 File identification or data set name

2 Record Format

3 Record Length

4 Number of Records

5 Number of Data Blocks ('?' on TSO)

6 When Last Written ('?' on TSO)

7 Disk Label ('?' on TSO)

8 Disk Mode (and parent) ('?' on TSO)

9 Block Size

If a file system error occurs while processing ∆*F*, a numeric return code is returned instead of the character result. These return codes are operating system dependent. See *APL2/370 Programming: System Services Reference* for more information.

∆FM—Read or Write a Fixed Record Length File

```
result←∆FM 'file'
```

'*file*' A character string that contains the name of the file to be read.

Under VM/CMS this takes the form 'fn ft fm' and must exist on an accessible minidisk. 'ft' and 'fm' default to '*' for reading. 'fm' defaults to 'A' for writing.

Under MVS/TSO, it has the form 'userid.Vnnnnnnn.WSNAME'. If the data set name is not enclosed in quotation marks, it is prefixed with the current TSO PROFILE PREFIX setting. An optional member name enclosed in parentheses can follow the data set name.

result A character matrix. For files with variable length records, records are padded on the right with blanks to the length of the longest record.

```
result←array ∆FM 'file'
```

'*file*' Has the same form as the monadic call.

array Is a character matrix or vectors of vectors. If necessary, a file with fixed length records is created by padding the records with blanks.

result A numeric return code. 0 indicates success.

If a file system error occurs during processing of ∆*FM*, a numeric return code is returned. These return codes are operating system dependent. See *APL2/370 Programming: System Services Reference* for more information.

∆FV—Read or Write a Variable Record Length File

```
result←∆FV 'file'
```

'file' A character string that contains the name of the file to be read.

Under VM/CMS this takes the form 'fn ft fm' and must exist on an accessible minidisk. 'ft' and 'fm' default to '*' for reading. 'fm' defaults to 'A' for writing.

Under MVS/TSO, it has the form 'userid.Vnnnnnnn.WSNAME'. If the data set name is not enclosed in quotation marks, it is prefixed with the current TSO PROFILE PREFIX setting. An optional member name enclosed in parentheses may follow the data set name.

result Is a vector of character vectors. Trailing blanks in any record are deleted.

```
result←array ∆FV 'file'
```

'file' Has the same form as the monadic call.

array A character matrix or vector of vectors. A file with variable length records is created if necessary. Trailing blanks in each record are deleted.

result A numeric return code. Zero indicates success.

If a file system error occurs while *∆FV* is processing, a numeric return code is returned. These return codes are operating system dependent. See *APL2/370 Programming: System Services Reference* for further information.

Bibliography

APL2 Publications

- *APL2 Fact Sheet,* GH21-1090
- *APL2/370 Application Environment Licensed Program Specifications,* GH21-1063
- *APL2/370 Licensed Program Specifications,* GH21-1070
- *APL2 for AIX/6000 Licensed Program Specifications,* GC23-3058
- *APL2 for Sun Solaris Licensed Program Specifications,* GC26-3359
- *APL2/370 Installation and Customization under CMS,* SH21-1062
- *APL2/370 Installation and Customization under TSO,* SH21-1055
- *APL2 Migration Guide,* SH21-1069
- *APL2 Programming: Language Reference,* SH21-1061
- *APL2/370 Programming: Processor Interface Reference,* SH21-1058
- *APL2 Reference Summary,* SX26-3999
- *APL2 Programming: An Introduction to APL2,* SH21-1073
- *APL2 for AIX/6000: User's Guide,* SC23-3051
- *APL2 for OS/2: User's Guide,* SH21-1091
- *APL2 for Sun Solaris: User's Guide,* SH21-1092
- *APL2 for the IBM PC: User's Guide,* SC33-0600
- *APL2 GRAPHPAK: User's Guide and Reference,* SH21-1074
- *APL2 Programming: Using Structured Query Language,* SH21-1057
- *APL2/370 Programming: Using the Supplied Routines,* SH21-1056
- *APL2/370 Programming: System Services Reference,* SH21-1054
- *APL2/370 Diagnosis Guide,* LY27-9601
- *APL2/370 Messages and Codes,* SH21-1059

Other Books You Might Need

The following books might also be of use, and can be ordered from IBM.

Application Prototype Environment

- *Application Prototype Environment Guide and Reference*, SH19-6388

GDDM

- *GDDM Application Programming Guide* SC33-0867
- *GDDM Base Application Programming Reference*, SC33-0868
- *GDDM General Information*, GC33-0866
- *GDDM Messages*, SC33-0869
- *GDDM System Customization and Administration*, SC33-0871
- *GDDM User's Guide*, SC33-0875
- *GDDM Interactive Map Definition*, SC33-0338
- *GDDM-PGF Interactive Chart Utility*, SC33-0328
- *GDDM-PGF Programming Reference*, SC33-0333

MVS/ESA

- *MVS/ESA Callable Services for High Level Languages*, Version 4, GC28-1639
- *MVS/ESA Callable Services for High Level Languages*, Version 3, GC28-1834

PGF

- *Presentation Graphics Feature: User's Guide*, SC33-0102

QMF

- *QMF Application Development Guide for MVS*, SC26-4237
- *QMF Application Development Guide for VM/SP*, SC26-4238
- *QMF Version 2 Release 4 Callable Interface Usage*, GG24-3505

TSO

- *TSO Extensions Command Language Reference Manual*, SC28-1881

VS APL

- *VS APL for CMS: Installation Reference Manual*, SH20-9182
- *VS APL for TSO: Installation Reference Manual*, SH20-9183
- *VS APL for CICS/VS: Installation Reference Manual*, SH20-9181
- *VS APL for VSPC: Installation Reference Manual*, SH20-9184

Index

Special Characters

A

B

C

D

E

F

G

H

I

O

P

Q

R

S

T

U

V

W

We'd Like to Hear from You

APL2 Programming:
Using the Supplied Routines
Version 2 Release 2

Publication No. SH21-1056-01

Please use one of the following ways to send us your comments about this book:

- Mail—Use the Readers' Comments form on the next page. If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Fax—Use the Readers' Comments form on the next page and fax it to this U.S. number: (408) 463-4488.
- Electronic mail—Use one of the following network IDs:
  - IBMMail: USIB6JN8
  - Internet: apl2@vnet.ibm.com

  Be sure to include the following with your comments:
  - Title and publication number of this book
  - Your name, address, and telephone number if you would like a reply

Your comments should pertain only to the information in this book and the way the information is presented. To request additional publications, or to comment on other IBM information or the function of IBM products, please give your comments to your IBM representative or to your IBM authorized remarketer.

IBM may use or distribute your comments without obligation.

Readers' Comments

APL2 Programming:
Using the Supplied Routines
Version 2 Release 2

Publication No. SH21-1056-01

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐
Grammatically correct and consistent	☐	☐	☐	☐	☐
Graphically well designed	☐	☐	☐	☐	☐
Overall satisfaction	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

May we contact you to discuss your comments? ☐ Yes ☐ No

Name

Address

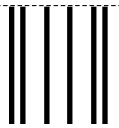
Company or Organization

Phone No.

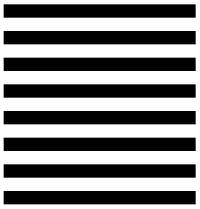
Readers' Comments
SH21-1056-01

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department M46/D12
PO Box 49023
San Jose, CA 95161-9023

Fold and Tape | **Please do not staple** | Fold and Tape

SH21-1056-01

Cut or Fold
Along Line

File Number: S370-40
Program Number: 5688-228

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

The APL2 Library

GH21-1090 APL2 Family of Products (fact sheet)
SH21-1073 APL2 Programming: An Introduction to APL2
SH21-1061 APL2 Programming: Language Reference
SX26-3999 APL2 Reference Summary
SH21-1074 APL2 GRAPHPAK: User's Guide and Reference
SH21-1057 APL2 Programming: Using Structured Query Language
SH21-1069 APL2 Migration Guide
SC33-0600 APL2 for the IBM PC: User's Guide
SC33-0601 APL2 for the IBM PC: Reference Summary
SC33-0851 APL2 for the IBM PC: Reference Card
SH21-1091 APL2 for OS/2: User's Guide
GC23-3058 APL2 for AIX/6000 Licensed Program Specifications
SC23-3051 APL2 for AIX/6000: User's Guide
GC26-3359 APL2 for Sun Solaris Licensed Program Specifications
SH21-1092 APL2 for Sun Solaris: User's Guide
GH21-1063 APL2/370 Application Environment Licensed Program Specifications
GH21-1070 APL2/370 Licensed Program Specifications
SH21-1062 APL2/370 Installation and Customization under CMS
SH21-1055 APL2/370 Installation and Customization under TSO
SH21-1054 APL2/370 Programming: System Services Reference
SH21-1056 APL2/370 Programming: Using the Supplied Routines
SH21-1058 APL2/370 Programming: Processor Interface Reference
LY27-9601 APL2/370 Diagnosis Guide
SH21-1059 APL2/370 Messages and Codes

SH21-1056-01