

# APL/PC Version 2.1

by IBM Madrid Scientific Centre

**Programming Family**




Personal  
Computer  
Software

ZZ33-0523-0

---

# APL/PC Version 2.1

by IBM Madrid Scientific Centre



Personal  
Computer  
Software

ZZ33-0523-0

| **First Edition (November 1986)**

| This edition describes APL/PC Version 2, Release 1. Changes from the original published edition supplied with the APL/PC Version 2 product (6391329) are marked with a vertical bar in the margin.

It is possible that this material may contain references to, or information about IBM products (machines and programs), programming or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming or services in your country.

This publication could contain technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of this publication.

Requests for copies of this product and for technical information about the system should be made to your authorised IBM Personal Computer dealer.

## Preface

The emergence of the personal computer with its vast array of end-user programs has truly revolutionised the world of computing, but for the tens of thousands of people who have had - for nearly two decades, now - the opportunity to work at APL computer terminals, this change is more an evolutionary development along familiar lines. These people have learned to expect user-friendly, highly interactive, personally productive programs, and to them it is only natural that programs with such qualities should now become available on personal computers.

In the light of these observations, two questions come to mind: What is it about APL that has encouraged and supported the production of such programs, and thereby anticipated the current trend? And why, with so many end-user programs available, should there be interest now in a general purpose programming language? The answers can perhaps be found by putting APL into perspective relative to other programming languages, and suggesting where APL programming may fit as an intellectual activity.

## General Purpose Programming Languages

Early computing machines were programmed in “machine language”, the unadorned numeric codes that were directly interpreted by the electrical circuits of the machine. Since the only digits that these circuits could recognise were (and are) zero and one, numbers within the machine had to be represented in base-2, an awkward notational system at best. Furthermore, the operations represented by the machine codes were essentially very simple, having to do mostly with



movement of data and the application of “logical”, or “boolean”, operations to it. These basic building blocks then had to be used in various combinations to synthesise elementary arithmetic operations such as multiplication or division.

It is not difficult to imagine that developing a program to do any significant amount of computation under these conditions was extremely laborious and error prone, and attention was soon turned to the possibility of using the machine itself to take the pain out of programming, by developing programs to translate into machine code something more easily understood and managed by people than a language made up of sequences of zeros and ones.

The first step, which resulted in “assembler” languages, was to give mnemonic names to the machine operations, and also allow the use of arbitrary names for the memory addresses which are a part of each machine instruction. Programs written in these languages essentially maintain a one-to-one relationship with machine instructions, and so are completely “machine dependent”; that is, they cannot be moved between dissimilar machines without drastic revisions. Furthermore, programmers writing in assembler must still deal directly with matters that are not relevant to the inherent logic of the problem being solved but are nevertheless required for management of the machine.

The disadvantages of assembler languages were largely overcome by the development of languages such as FORTRAN, COBOL, and ALGOL60, which expressed the necessary computations in a manner that corresponded somewhat to ordinary mathematical notation, and used words drawn from a natural language such as English, albeit with specialised meanings pertinent to the computing process. These languages broke free from the absolute machine dependence of assembler, but they were still designed with certain detailed characteristics of machines in mind, and they carry along an intellectual overhead (for example, the idea of different numerical “types”) derived from the inner workings of machines.

Most general purpose programming languages follow the patterns established by these pioneering languages, although they may differ from each other in significant respects. Characteristically, they are implemented by means of “compilers”, translators that produce a complete machine language version (“object code”) of the programmed application before the actual computation starts; they generally deal with single numbers, and even when it is possible to refer to an entire collection of data by a single name, to process such a collection it is usually necessary to explicitly call for iteration, or “looping”; and to a large extent they require explicit management of storage facilities.

## **The Nature of APL**

APL is a general purpose programming language quite unlike those described above. It is based primarily on mathematical notation, which has been evolving over a very long time as a tool for dealing with both abstract ideas and practical calculations, with no concern, of course, for the internal workings of computers. As a consequence, APL differs most markedly from the other general purpose languages in precisely those attributes which characterise them:

- The principal implementations of APL as a working language are “interpretive”, rather than compiled. That is, programs written in APL are not translated ahead of time to machine code, but instead, as each operation called for in a program is encountered, an immediate translation takes place, and the appropriate machine code is directly executed. This provides immediate feedback in a local context, and greatly facilitates experimentation while developing a program.
- APL deals with collections of data, such as lists or tables, by the direct application of commonly useful operations to them, without the use of explicit looping. For example, it takes but a single expression in APL to obtain the sums of the columns in a table, or to multiply a set of quantities by

the unit price of each item. This not only makes programming easier and less prone to error, but gives APL the aspect of an exceedingly powerful calculator in activities ranging from casual use to intensive exploration of a problem area.

- The management of storage is not a concern for users of APL, as both the allocation of space and the internal representation of numbers are automatically determined as soon as an object is given a name and a value. This makes it possible to concentrate on the logic of the problem, rather than the requirements of the machine, and contributes substantially to the high productivity associated with the use of APL.

Another distinguishing characteristic of APL is the use of “shared variables”, the means by which APL manages input and output in order to communicate with the rest of the computing world. The concept is simple and fundamental: communication between two systems, be they computers or people, requires that they have something in common which each can alter and each can sense, as may be appropriate. In terms of APL, a shared variable is a data object, or array, that can be set and referenced by communicating partners, according to some synchronising discipline.

The influence of shared variables on the design of applications lies in both the power it provides and the simplicity it maintains:

- By means of shared variables APL programs are able to control the computing environment, including printers, files, game devices, displays, or even other APL programs, without giving up the mathematical attributes and the inherent simplicity of the APL language itself.
- The management of particular devices is readily isolated in separate programs, or driven from tables, independent of the overall logic of the application. This facilitates good design, and makes modification for new or different devices relatively simple.

- In effect, the uses of APL can be extended without limit by means of “auxiliary processors” - specialised programs for specific tasks - with which APL communicates through shared variables.

To summarise, then, the answer to the question of how APL came to anticipate current trends in personal computing can be found in the attributes that distinguish APL from other general purpose languages and enhance its productivity, in its shared-variable based communications, and in the interactive nature of APL systems. Together, these qualities have strongly facilitated trial and experimentation, which has led to the practice of developing APL-based applications incrementally, with constant reference to the needs of users as these become known and understood through actual use; a development method that has recently been popularised as “prototyping”.

## **The Rewards of Programming**

The second question, which is why should there be any interest in a general purpose programming language, specifically in APL, when so many application packages are available, has three answers. One is simply that a ready-made application package may not be completely satisfactory. Indeed, if such a package does precisely what is wanted, and all that is wanted, then the matter ends there. But if it doesn't quite match, and compromise is not acceptable, then building it in APL, taking advantage of the high productivity and ease of experimentation, is a good way to get what is required.

A second answer is that there is considerable satisfaction to be found in the successful construction of a program that performs a useful task, or makes something happen outside of the computer. Additionally, with APL - because of its mathematical nature and the consequent variety of ways in which equivalent operations can be stated - there is the possibility of realising esthetic satisfaction from the elegance of a particular formulation, or getting intellectual pleasure from

new insights gained from experimenting with variations. Properly approached, APL programming can be regarded as a medium of expression and a tool of thought, with characteristics of both natural language composition and mathematics.

Finally, the third answer has to do with the nature of this particular product. In addition to providing an advanced level of "standard" APL which is compatible with most other APL implementations, APL/PC 2.1 has a full complement of facilities for reaching every part of the IBM Personal Computer. There are auxiliary processors to put APL in control of hardware, including display devices, communication lines, laboratory experiments, printers, and games. There are others to access the software environment, including programs written in other languages, the DOS operating system and the BIOS interrupts; and there is a direct facility for accessing the computer memory, machine registers, and input/output ports.

APL/PC 2.1 on a Personal Computer is thus a very well equipped laboratory for conducting unlimited experiments in computing. It is an ideal ground for learning about all aspects of programming, and exploring the dynamic interactions between hardware, software, and user, bringing to bear on the subject the power and versatility of APL as the principal tool.

Adin Falkoff

# Contents

|  |            |
|--|------------|
| <b>Chapter 1. Introduction</b> .....                                       | <b>1-1</b> |
| The APL Package .....  | 1-8        |
| The APL Character ROM .....  | 1-8        |
| Displaying APL Characters using the Enhanced<br>Graphics Adapter .....     | 1-12       |
| Displaying APL Characters using the Professional<br>Graphics Adapter ..... | 1-12       |
| Backing Up Your Diskettes .....  | 1-13       |
| Installing APL on Your Fixed Disk .....                                    | 1-13       |
| Getting APL Started from Either Diskette or Fixed<br>Disk .....            | 1-15       |
| Including Options in the APL Command .....                                 | 1-16       |
| The APL Character Set .....  | 1-18       |
| The Keyboard .....   | 1-19       |
| Function Keys .....  | 1-19       |
| Typewriter Keyboard .....  | 1-21       |
| Numeric Keypad .....   | 1-24       |
| Special Key Combinations .....   | 1-25       |
| Keyboard Keycaps .....   | 1-26       |
| Keyboard Template .....  | 1-27       |
| The APL Input Editor .....   | 1-29       |
| APL Input Editor Special Keys .....  | 1-29       |
| How to Make Corrections on the Current Line .....                          | 1-32       |
| Use of Displays .....  | 1-33       |
| Disk(ette) Drives .....  | 1-35       |
| The Printer .....  | 1-36       |
| <br>   |            |
| <b>Chapter 2. APL Tutorial</b> .....                                       | <b>2-1</b> |
| What is APL? .....   | 2-3        |
| Arrays and Functions .....   | 2-4        |
| Interactivity .....  | 2-7        |
| The APL Calculator .....   | 2-8        |
| Order of Execution .....   | 2-10       |
| Powers and Logarithms .....  | 2-11       |
| Circular Functions .....   | 2-13       |
| Computations with Multiple Data Items .....                                | 2-14       |

|   |            |
|---|------------|
| Variations on the Compound Interest Problem . . .       | 2-17       |
| Named Data . . . . .                                    | 2-18       |
| APL Workspaces . . . . .                                | 2-20       |
| Other Functions and Operators . . . . .                 | 2-24       |
| Input/Output . . . . .                                  | 2-24       |
| Shared Variables . . . . .                              | 2-25       |
| A Simple Example . . . . .                              | 2-26       |
| Shared and Other Variables . . . . .                    | 2-27       |
| Debugging . . . . .                                     | 2-28       |
| Quad output . . . . .                                   | 2-29       |
| Tracing . . . . .                                       | 2-30       |
| Stopping . . . . .                                      | 2-31       |
| Alternate execution . . . . .                           | 2-31       |
| Speeding Up APL Code . . . . .                          | 2-32       |
| Use Timing Tools to Find Hot Spots . . . . .            | 2-33       |
| Eliminate Explicit Loops . . . . .                      | 2-33       |
| Avoid Waxing or Waning Variables . . . . .              | 2-35       |
| Avoid Large Outer Products . . . . .                    | 2-36       |
| Avoid Tiny Functions . . . . .                          | 2-36       |
| An Example - Computing a Histogram . . . . .            | 2-37       |
| Computing the Histogram with an Explicit Loop . . . . . | 2-37       |
| Computing the Histogram with an Outer Product . . . . . | 2-39       |
| Computing the Histogram with Grade Up . . . . .         | 2-41       |
| How to Write a Timing Function . . . . .                | 2-42       |
| A Simple Timing Program . . . . .                       | 2-42       |
| An Improved Timing Function . . . . .                   | 2-44       |
| An Alternative Solution . . . . .                       | 2-46       |
| Concluding Thought . . . . .                            | 2-47       |
| Further Reading . . . . .                               | 2-48       |
| <b>Chapter 3. Using APL . . . . .</b>                   | <b>3-1</b> |
| An Example of the Use of APL . . . . .                  | 3-3        |
| An Isolated Calculation . . . . .                       | 3-3        |
| Storing Functions and Data . . . . .                    | 3-4        |
| Characteristics of APL . . . . .                        | 3-5        |
| <b>Chapter 4. Fundamentals . . . . .</b>                | <b>4-1</b> |
| Character Set . . . . .                                 | 4-6        |
| Spaces . . . . .  | 4-8        |
| Function . . . . .                                      | 4-8        |
| Order of Execution . . . . .                            | 4-9        |
| Data . . . . .  | 4-10       |
| Arrays . . . . .  | 4-10       |

|   |            |
|---|------------|
| Constants .....   | 4-12       |
| Workspaces and Libraries .....                                  | 4-13       |
| Names .....   | 4-14       |
| Implementation Limits .....                                     | 4-15       |
| <b>Chapter 5. Primitive Functions and Operators .....</b>       | <b>5-1</b> |
| Scalar Functions .....  | 5-3        |
| Plus, Minus, Times, Divide, and Residue .....                   | 5-7        |
| Conjugate, Negative, Signum, Reciprocal, and<br>Magnitude ..... | 5-8        |
| Boolean and Relational Functions .....                          | 5-9        |
| Minimum and Maximum .....                                       | 5-11       |
| Floor and Ceiling .....   | 5-11       |
| Roll (Random Number Function) .....                             | 5-12       |
| Power, Exponential, General and Natural<br>Logarithm .....      | 5-12       |
| Circular, Hyperbolic, and Pythagorean Functions                 | 5-13       |
| Factorial and Binomial Functions .....                          | 5-15       |
| Operators .....   | 5-17       |
| Reduction .....   | 5-17       |
| Scan .....  | 5-19       |
| Axis .....  | 5-19       |
| Inner Product .....   | 5-21       |
| Outer Product .....   | 5-23       |
| Mixed Functions .....   | 5-25       |
| Structural Functions .....                                      | 5-30       |
| Selection Functions .....                                       | 5-37       |
| Selector Generators .....                                       | 5-42       |
| Index Generator and Index Of .....                              | 5-43       |
| Membership .....  | 5-44       |
| Grade Functions .....   | 5-44       |
| Deal .....  | 5-48       |
| Numeric Functions .....   | 5-48       |
| Matrix Inverse and Matrix Divide .....                          | 5-48       |
| Decode and Encode .....   | 5-51       |
| Data Transformations .....                                      | 5-53       |
| Execute and Format .....  | 5-54       |
| Picture Format .....  | 5-59       |
| <b>Chapter 6. System Functions and System Variables .....</b>   | <b>6-1</b> |
| System Functions .....  | 6-3        |
| Canonical Representation - $\square CR$ .....                   | 6-5        |
| Delay - $\square DL$ .....                                      | 6-6        |



|   |            |
|---|------------|
| Execute Alternate - $\square EA$ .....              | 6-6        |
| Expunge - $\square EX$ .....                        | 6-7        |
| Function Establishment - $\square FX$ .....         | 6-7        |
| Name Classification - $\square NC$ .....            | 6-8        |
| Name List - $\square NL$ .....                      | 6-9        |
| Peek/Poke - $\square PK$ .....                      | 6-9        |
| Transfer Form - $\square TF$ .....                  | 6-11       |
| System Variables .....                              | 6-14       |
| Account Information - $\square AI$ .....            | 6-16       |
| Atomic Vector - $\square AV$ .....                  | 6-16       |
| Comparison Tolerance - $\square CT$ .....           | 6-17       |
| Format Control - $\square FC$ .....                 | 6-17       |
| Index Origin - $\square IO$ .....                   | 6-18       |
| Horizontal Tabs - $\square HT$ .....                | 6-18       |
| Latent Expression - $\square LX$ .....              | 6-18       |
| Line Counter - $\square LC$ .....                   | 6-19       |
| Printing Precision - $\square PP$ .....             | 6-19       |
| Printing Width - $\square PW$ .....                 | 6-20       |
| Random Link - $\square RL$ .....                    | 6-20       |
| Terminal Control - $\square TC$ .....               | 6-20       |
| Terminal Type - $\square TT$ .....                  | 6-20       |
| Time Stamp - $\square TS$ .....                     | 6-20       |
| User Load - $\square UL$ .....                      | 6-20       |
| Workspace Available - $\square WA$ .....            | 6-20       |
| <b>Chapter 7. Shared Variables</b> .....            | <b>7-1</b> |
| Offers .....  | 7-6        |
| Access Control .....                                | 7-7        |
| Retraction .....                                    | 7-11       |
| Inquiries .....                                     | 7-12       |
| <b>Chapter 8. Function Definition</b> .....         | <b>8-1</b> |
| Canonical Representation and Function Establishment | 8-3        |
| The Function Header .....                           | 8-5        |
| Ambi-Valent Functions .....                         | 8-5        |
| Local and Global Names .....                        | 8-6        |
| Branching and Statement Numbers .....               | 8-7        |
| Labels .....  | 8-9        |
| Comments .....                                      | 8-9        |
| Function Editing - The $\nabla$ Form .....          | 8-10       |
| Adding a Statement .....                            | 8-10       |
| Inserting or Replacing a Statement .....            | 8-11       |

|   |             |
|---|-------------|
| Replacing the Header .....                      | 8-11        |
| Deleting a Statement .....                      | 8-11        |
| Modifying a Statement or Header .....           | 8-12        |
| Function Display .....                          | 8-12        |
| Leaving the ▽ Form .....                        | 8-13        |
| Quitting the ▽ Form .....                       | 8-14        |
| <b>Chapter 9. Function Execution .....</b>      | <b>9-1</b>  |
| Halted Execution .....                          | 9-4         |
| State Indicator .....                           | 9-4         |
| State Indicator Damage .....                    | 9-6         |
| Trace Control .....                             | 9-6         |
| Stop Control .....                              | 9-7         |
| Locked Functions .....                          | 9-8         |
| Recursive Functions .....                       | 9-9         |
| Input and Output .....                          | 9-10        |
| Evaluated Input .....                           | 9-11        |
| Character Input .....                           | 9-12        |
| Interrupting Execution during Input .....       | 9-12        |
| Normal Output .....                             | 9-12        |
| Bare Output .....                               | 9-13        |
| <b>Chapter 10. System Commands .....</b>        | <b>10-1</b> |
| Active Workspace - Action Commands .....        | 10-9        |
| Active Workspace - Inquiry Commands .....       | 10-13       |
| Workspace Storage and Retrieval - Action        |             |
| Commands .....                                  | 10-14       |
| Libraries of Saved Workspaces .....             | 10-14       |
| Workspace Names .....                           | 10-14       |
| Workspace Storage and Retrieval - Inquiry       |             |
| Commands .....                                  | 10-20       |
| Sign-Off .....                                  | 10-21       |
| <b>Chapter 11. Application Workspaces .....</b> | <b>11-1</b> |
| The AP2 Workspace .....                         | 11-5        |
| Example Session .....                           | 11-8        |
| The AP124 Workspace .....                       | 11-8        |
| Fundamentals .....                              | 11-9        |
| Building a Menu .....                           | 11-10       |
| The AP190 Workspace .....                       | 11-19       |
| The AP205 Workspace .....                       | 11-21       |
| The AP206 Workspace .....                       | 11-21       |
| The AP232X Workspace .....                      | 11-26       |

|   |        |
|---|--------|
| The AP488 Workspace .....   | 11-28  |
| Requirements .....  | 11-28  |
| Reference Documentation .....                                     | 11-28  |
| Hints to Avoid Trouble .....                                      | 11-28  |
| Description of AP488 Functions .....                              | 11-30  |
| The APLFILE Workspace .....                                       | 11-42  |
| The DEMO124 Workspace .....                                       | 11-47  |
| The DEMO206 Workspace .....                                       | 11-48  |
| The DOSFNS Workspace .....  | 11-48  |
| The EDIT Workspace .....  | 11-51  |
| The EXCHG Workspace .....   | 11-55  |
| The FILE Workspace .....  | 11-56  |
| Functions .....   | 11-57  |
| Terminology .....   | 11-58  |
| Examples of Use .....   | 11-71  |
| The FOIL Workspace .....  | 11-73  |
| The FORTRAN Workspace .....                                       | 11-74  |
| Restrictions on FORTRAN Programs .....                            | 11-74  |
| Generation Process .....  | 11-75  |
| Usage Protocol .....  | 11-78  |
| PFORTPAR Parameter Management Program .....                       | 11-79  |
| Sample FORTRAN Subroutines (IBM PC<br>Professional FORTRAN) ..... | 11-81  |
| The GEDIT Workspace .....   | 11-82  |
| The GRAPHPAK Workspaces .....                                     | 11-84  |
| The MUSIC Workspace .....   | 11-86  |
| The PLOT Workspace .....  | 11-87  |
| The PRINT Workspace .....   | 11-89  |
| The PROFILE Workspace .....                                       | 11-91  |
| The UTIL Workspace .....  | 11-92  |
| The VM232 Workspace .....   | 11-95  |
| Selecting a Terminal .....  | 11-96  |
| Saving Your Line Parameter Definition .....                       | 11-102 |
| Connection with the Host .....                                    | 11-103 |
| Functions .....   | 11-105 |
| Example of Connection with the Host .....                         | 11-108 |
| Auxiliary Files on the Host .....                                 | 11-112 |

|   |             |
|---|-------------|
| <b>Chapter 12. Auxiliary Processors .....</b>                   | <b>12-1</b> |
| The Non-APL Program Interface Auxiliary<br>Processor: AP2 ..... | 12-4        |
| Basic Functions .....   | 12-5        |
| Auxiliary Functions .....                                       | 12-8        |

|  |       |
|--|-------|
| Sample AP2 session .....   | 12-9  |
| Return Codes (Returned through the control variable) .....       | 12-9  |
| The Printer Auxiliary Processor: AP80 .....                      | 12-10 |
| Patching AP80 for Other Printers .....                           | 12-12 |
| The Stack and Profile Auxiliary Processor: AP101 ..              | 12-14 |
| Error Return Codes .....   | 12-17 |
| The BIOS/DOS Interrupt Auxiliary Processor:                      |       |
| AP103 .....  | 12-18 |
| BIOS/DOS Interrupt Function Call .....                           | 12-19 |
| I/O Port IN/OUT Request .....                                    | 12-22 |
| Joystick Algorithm .....   | 12-23 |
| The Full Screen Management Auxiliary Processor:                  |       |
| AP124 .....  | 12-24 |
| AP124 Operation .....  | 12-24 |
| Error Return Codes .....   | 12-33 |
| The Host Communications Auxiliary Processors:                    |       |
| AP190 and AP190I .....   | 12-34 |
| Possible uses for AP190 .....                                    | 12-35 |
| Getting Started .....  | 12-35 |
| Sending Keystrokes .....   | 12-35 |
| Setting Keyboard Translation Table .....                         | 12-36 |
| Getting Host Status .....  | 12-36 |
| Getting the Physical Screen .....                                | 12-36 |
| Get the Operator Information Area .....                          | 12-37 |
| Simulate a Power On Reset .....                                  | 12-37 |
| Get Cursor Position and Beep Indication .....                    | 12-37 |
| Get the Keyboard Translation Table .....                         | 12-37 |
| Get the Screen Format Array .....                                | 12-38 |
| The Full-Screen Auxiliary Processor: AP205 .....                 | 12-38 |
| The Graphic Auxiliary Processor: AP206 .....                     | 12-39 |
| Storage Management .....   | 12-39 |
| Parameters .....   | 12-40 |
| Use of AP206 .....   | 12-47 |
| Functions .....  | 12-48 |
| Return codes .....   | 12-53 |
| The File Auxiliary Processor: AP210 .....                        | 12-53 |
| Control Commands .....   | 12-54 |
| Control Subcommands .....  | 12-57 |
| AP210 Return Codes .....   | 12-59 |
| Examples of use .....  | 12-60 |
| The Asynchronous Communications Auxiliary Processor: AP232 ..... | 12-62 |

|  |             |
|--|-------------|
| Control Commands                                       | 12-63       |
| The Extended Asynchronous Communications               |             |
| Auxiliary Processor: AP232X                            | 12-69       |
| Hardware Notes   | 12-70       |
| AP232X Operation                                       | 12-70       |
| AP232X Return Codes                                    | 12-75       |
| The Music Auxiliary Processor: AP440                   | 12-76       |
| AP440 Command Syntax                                   | 12-77       |
| The IBM GPIB Support Auxiliary Processor: AP488        | 12-79       |
| Description of AP488 Functions                         | 12-80       |
| <b>Chapter 13. How to Build an Auxiliary Processor</b> | <b>13-1</b> |
| Access Control   | 13-4        |
| Format of Shared Data                                  | 13-5        |
| Shared Variable Processor Services and Return Codes    | 13-7        |
| Processor Sign-on: 00H                                 | 13-8        |
| Return to APL via Shared Variable Processor:           |             |
| 01H  | 13-9        |
| Share or Query the State of a Variable: 02H            | 13-10       |
| Get the Present Value of a Shared Variable: 03H        | 13-11       |
| Get a Block of Memory From the Workspace:              |             |
| 04H  | 13-12       |
| Release Storage to the Workspace: 05H                  | 13-13       |
| Pass a Variable to APL and Release the Space:          |             |
| 06H  | 13-14       |
| Pass a Scalar Integer Return Code to APL: 07H          | 13-15       |
| Convert an APL Object from Type Boolean to             |             |
| Integer: 08H   | 13-16       |
| Convert from APL Z-code to ASCII: 09H                  | 13-17       |
| Convert from ASCII to APL Z-code: 0AH                  | 13-18       |
| Share or Query the State of a Variable: 0BH            | 13-19       |
| Pre-read a Variable: 0CH                               | 13-20       |
| Read a Previously Pre-read Variable: 0DH               | 13-21       |
| Release a Previously Pre-read Variable: 0EH            | 13-22       |
| Pass a Value to a Variable: 0FH                        | 13-23       |
| Processor Sign-off: 10H                                | 13-24       |
| SVP Reserved Function: 11H                             | 13-24       |
| Locate an Associated Variable: 12H                     | 13-25       |
| Change the Keyboard / Screen Mode: 13H                 | 13-26       |
| Get Loop Count for Delay: 14H                          | 13-27       |
| Change the Keyboard / Screen Mode Without              |             |
| Clearing Screen: 15H                                   | 13-28       |
| Notes  | 13-29       |

|  |  |            |
|--|--|------------|
|  | Return Codes (Returned in CX Register) . . . . .                                     | 13-30      |
|  | Sample Auxiliary Processors . . . . .  | 13-30      |
|  | APL Interrupt Usage . . . . .  | 13-31      |
|  | How to Debug Auxiliary Processors . . . . .  | 13-32      |
|  | Exchange Assembly Programs . . . . .   | 13-33      |
|  | <b>Appendix A. Backing up Diskettes . . . . .</b>                                    | <b>A-1</b> |
|  | Before You Begin . . . . .   | A-1        |
|  | Protecting Your Original Diskette . . . . .  | A-1        |
|  | Backing Up Diskette with One Drive . . . . .   | A-2        |
|  | Backing Up Diskette with Two Drives . . . . .  | A-4        |
|  | <b>Appendix B. APL/PC 1.0 Workspace Migration . . . . .</b>                          | <b>B-1</b> |
|  | Workspaces in APL format . . . . .   | B-1        |
|  | Workspaces in AIO format . . . . .   | B-2        |
|  | <b>Appendix C. The APL Character Set and <math>\square AV</math> . . . . .</b>       | <b>C-1</b> |
|  | <b>Appendix D. Internal Representation of Displayed<br/>    Characters . . . . .</b> | <b>D-1</b> |
|  | <b>Appendix E. APL Keyboard Redefinition . . . . .</b>                               | <b>E-1</b> |
|  | <b>Appendix F. The GRAPHPAK Workspaces - Functions . . . . .</b>                     | <b>F-1</b> |
|  | GPBASE . . . . .   | F-1        |
|  | GPCHT . . . . .  | F-3        |
|  | GPCONT . . . . .   | F-4        |
|  | GPDEMO . . . . .   | F-4        |
|  | GPFIT . . . . .  | F-6        |
|  | GPGEOM . . . . .   | F-7        |
|  | GPPLOT . . . . .   | F-8        |
|  | <b>Appendix G. Hardware Modification for IBM 4860 PCjr . . . . .</b>                 | <b>G-1</b> |
|  | <b>Appendix H. Patch to Restore BIOS Keyboard Handler . . . . .</b>                  | <b>H-1</b> |
|  | <b>Index . . . . .</b>   | <b>X-1</b> |

**Notes:**

# Figures

|       |  |       |
|-------|--|-------|
| 1-1.  | Character ROM Location on Display Adapter Cards .....        | 1-10  |
| 1-2.  | Removing the Character ROM .....                             | 1-11  |
| 1-3.  | Keyboard with APL Character Set .....                        | 1-28  |
| 4-1.  | APL Character Set .....                                      | 4-7   |
| 5-1.  | Primitive Scalar Functions .....                             | 5-5   |
| 5-2.  | Identity Elements of Primitive Scalar Dyadic Functions ..... | 5-7   |
| 5-3.  | The Pythagorean Functions .....                              | 5-15  |
| 5-4.  | Inner Product .....  | 5-22  |
| 5-5.  | Primitive Mixed Functions .....                              | 5-25  |
| 5-6.  | Scalar Vector Substitutions for Mixed Functions .....        | 5-29  |
| 6-1.  | System Functions .....                                       | 6-4   |
| 6-2.  | System Variables .....                                       | 6-15  |
| 7-1.  | System Functions .....                                       | 7-4   |
| 7-2.  | Access Control of a Shared Variable .....                    | 7-9   |
| 7-3.  | Some Useful Settings for the Access Control Vector .....     | 7-11  |
| 10-1. | System Commands .....  | 10-4  |
| 10-2. | Trouble Reports .....  | 10-7  |
| 10-3. | Symbols Used in Command Definitions .....                    | 10-8  |
| 10-4. | Environment Within a Clear Workspace .....                   | 10-9  |
| 11-1. | IEEE-488 Addresses .....                                     | 11-33 |
| 11-2. | Timeout Control Codes .....                                  | 11-39 |
| 11-3. | Mask Layout .....  | 11-40 |
| D-1.  | Internal Representation of Displayed Characters .....        | D-2   |



**Notes:**

## Part 1. APL Introduction

|   |            |
|---|------------|
| <b>Chapter 1. Introduction</b>                                    | <b>1-1</b> |
| The APL Package   | 1-8        |
| The APL Character ROM   | 1-8        |
| Displaying APL Characters using the Enhanced Graphics Adapter     | 1-12       |
| Displaying APL Characters using the Professional Graphics Adapter | 1-12       |
| Backing Up Your Diskettes   | 1-13       |
| Installing APL on Your Fixed Disk                                 | 1-13       |
| Getting APL Started from Either Diskette or Fixed Disk            | 1-15       |
| Including Options in the APL Command                              | 1-16       |
| The APL Character Set   | 1-18       |
| The Keyboard  | 1-19       |
| Function Keys   | 1-19       |
| Typewriter Keyboard   | 1-21       |
| Numeric Keypad  | 1-24       |
| Special Key Combinations  | 1-25       |
| Keyboard Keycaps  | 1-26       |
| Keyboard Template   | 1-27       |
| The APL Input Editor  | 1-29       |
| APL Input Editor Special Keys                                     | 1-29       |
| How to Make Corrections on the Current Line                       | 1-32       |
| Use of Displays   | 1-33       |
| Disk(ette) Drives   | 1-35       |
| The Printer   | 1-36       |
| <br>  |            |
| <b>Chapter 2. APL Tutorial</b>                                    | <b>2-1</b> |
| What is APL?  | 2-3        |
| Arrays and Functions  | 2-4        |
| Interactivity   | 2-7        |
| The APL Calculator  | 2-8        |
| Order of Execution  | 2-10       |
| Powers and Logarithms   | 2-11       |
| Circular Functions  | 2-13       |
| Computations with Multiple Data Items                             | 2-14       |

|   |      |
|---|------|
| Variations on the Compound Interest Problem . . | 2-17 |
| Named Data . . . . .                            | 2-18 |
| APL Workspaces . . . . .                        | 2-20 |
| Other Functions and Operators . . . . .         | 2-24 |
| Input/Output . . . . .                          | 2-24 |
| Shared Variables . . . . .                      | 2-25 |
| A Simple Example . . . . .                      | 2-26 |
| Shared and Other Variables . . . . .            | 2-27 |
| Debugging . . . . .                             | 2-28 |
| Quad output . . . . .                           | 2-29 |
| Tracing . . . . .                               | 2-30 |
| Stopping . . . . .                              | 2-31 |
| Alternate execution . . . . .                   | 2-31 |
| Speeding Up APL Code . . . . .                  | 2-32 |
| Use Timing Tools to Find Hot Spots . . . . .    | 2-33 |
| Eliminate Explicit Loops . . . . .              | 2-33 |
| Avoid Waxing or Waning Variables . . . . .      | 2-35 |
| Avoid Large Outer Products . . . . .            | 2-36 |
| Avoid Tiny Functions . . . . .                  | 2-36 |
| An Example - Computing a Histogram . . . . .    | 2-37 |
| Computing the Histogram with an Explicit Loop   | 2-37 |
| Computing the Histogram with an Outer Product   | 2-39 |
| Computing the Histogram with Grade Up . . . . . | 2-41 |
| How to Write a Timing Function . . . . .        | 2-42 |
| A Simple Timing Program . . . . .               | 2-42 |
| An Improved Timing Function . . . . .           | 2-44 |
| An Alternative Solution . . . . .               | 2-46 |
| Concluding Thought . . . . .                    | 2-47 |
| Further Reading . . . . .                       | 2-48 |

# Chapter 1. Introduction

|  |      |
|--|------|
| The APL Package .....  | 1-8  |
| The APL Character ROM .....  | 1-8  |
| Displaying APL Characters using the Enhanced<br>Graphics Adapter .....     | 1-12 |
| Displaying APL Characters using the Professional<br>Graphics Adapter ..... | 1-12 |
| Backing Up Your Diskettes .....  | 1-13 |
| Installing APL on Your Fixed Disk .....                                    | 1-13 |
| Getting APL Started from Either Diskette or Fixed<br>Disk .....            | 1-15 |
| Including Options in the APL Command .....                                 | 1-16 |
| The APL Character Set .....  | 1-18 |
| The Keyboard .....   | 1-19 |
| Function Keys .....  | 1-19 |
| Typewriter Keyboard .....  | 1-21 |
| Numeric Keypad .....   | 1-24 |
| Special Key Combinations .....   | 1-25 |
| Keyboard Keycaps .....   | 1-26 |
| Keyboard Template .....  | 1-27 |
| The APL Input Editor .....   | 1-29 |
| APL Input Editor Special Keys .....  | 1-29 |
| How to Make Corrections on the Current Line                                | 1-32 |
| Use of Displays .....  | 1-33 |
| Disk(ette) Drives .....  | 1-35 |
| The Printer .....  | 1-36 |

**Notes:**

APL is a general-purpose language that enjoys wide use in such diverse applications as commercial data processing, system design, mathematical and scientific computation, and the teaching of mathematics and other subjects. It has proved particularly useful in data-base applications, where its computational power and communication facilities combine to increase the productivity of both application programmers and end users.

When implemented as a computing system, APL is used from a typewriter-like keyboard. Statements that specify the work to be done are typed and the computer responds by displaying the result of the work at a device such as a video display or printer. In addition to work purely at the keyboard and its associated display, entries may also specify the use of printers, disk files, or other remote devices.

A programming language should be relevant. That is, you should have to write only what is logically necessary to specify the job you want done. This may seem an obvious point, but many of the earlier programming languages forced you to be concerned as much with the internal requirements of the machine as with your own statement of the problem. APL takes care of those internal considerations automatically.

A programming language needs both both *power* and *simplicity*. By power, we mean the ability to handle large or complicated tasks. By simplicity, we mean the ability to state what must be done briefly and neatly, in a way that is easy to read and easy to write. You might think that power and simplicity are competing requirements, so that if you have one, you can't have the other, but that is not necessarily so. Simplicity does not mean the computer is limited to doing simple tasks, but that the user has a simple way to write instructions to the computer. The power of APL as a programming language comes in part from its simplicity.

The letters, *APL*, originated with the initials of a book written by K. E. Iverson, *A Programming Language* (New York: Wiley, 1962). Dr. Iverson first worked on the language at Harvard University, and then continued its development at IBM with the collaboration of Adin Falkoff and others at the IBM T.J. Watson Research Centre. The term *APL* now refers

to the language that is an outgrowth of that work. *APL* is the language, and *IBM Personal Computer APL* is the “brand-name” of a particular implementation of that language, with extensions. The implementation and extensions were developed by the IBM Madrid Scientific Centre. This implementation, hereafter called *APL*, has the following features:

- Shared variables, which allow the exchange of information between independently operating processors. This allows the separate loading of only those auxiliary processors needed for a particular work session or application. It also makes possible the design of new auxiliary processors for an application that may not be currently supported by the system.
- Facilities for conversion between the internal form and transfer form of *APL* objects, including *IN*, *OUT* and *TF*, that allow workspaces to be interchanged between different systems.
- Asynchronous communications with the IBM Virtual Machine Facility/370 permits the exchange of workspaces and data files between systems, and allows devices attached to the host to be used.
- All dyadic-defined functions are ambivalent, which allows them to be used monadically without generating a syntax error. The system function, *NC*, can be applied to the left argument within a function to determine, at execution time, whether the function actually has been called as dyadic or monadic.
- Improved error recovery is made possible by the *RESET* command, which clears the state indicator, and *EA* (execute alternate), which allows the trapping of an *APL* interrupt and error message to permit programmed means of recovery.
- Event handling facilities are provided through an *APL* interface to the BIOS/DOS interrupts, thus allowing these

interrupts to be trapped or generated for more control of the system environment.

- The APL Workspace consists of two parts:
  - The *main workspace*, which has a maximum size of 64K bytes, where all APL statements are executed and small APL objects are created and modified.
  - The *elastic workspace*, which can use all additional free memory. If space is needed for an operation in the main workspace, every object not currently being referenced will be automatically relocated to the elastic workspace, and returned as needed.
- The following four data types are supported, and the system automatically performs data-type conversions whenever possible to minimise storage space:
  - *Floating-point*, with eight bytes per element
  - *Integer*, with two bytes per element
  - *Character*, with one byte per element
  - *Boolean*, with one bit per element
- The IBM Personal Computer Math Co-Processor may be used for improved performance of floating-point operations, such as the APL transcendental functions.
- The ability to start an application automatically by specifying an APL system command at load time before starting a work session. Functions that imitate some of the system commands also are provided allowing the system environment to be controlled from within a defined function.
- The execution of machine code subroutines and the PEEK and POKE memory contents is provided through the `⎕PK` system function.



- The appearance of numeric output can be improved using *picture format*, and the dyadic grades allow character data to be sorted in a specified collating sequence.
- Dynamic switching (either from the keyboard or by software control) between the APL and National character sets on the keyboard provides access to an extensive set of characters that can be entered with one keystroke. The keyboard layout may also be redefined dynamically to allow for the various keyboard arrangements used on European keyboards.
- Multiple display monitors can be used, with dynamic switching between all modes.
- A full-screen input and output capability provides a full-screen input editor, which allows corrections to be made to a previous line that can then be re-entered for execution. A full-screen, defined-function editor, and multiple line deletion under the *del* (∇) editor, increase the ease with which programs can be created and edited.
- A file management capability allows the control of either APL or DOS files, with sequential or direct access of fixed length and variable length records.
- The optional IBM Graphics Printer can produce APL characters, and can be used either as a system log to provide a record of a work session, or to selectively print a desired APL object or result.
- The speaker attached to the system unit of the IBM personal computer can be used to generate music.

To use the IBM Personal Computer APL system, you must have the following minimum configuration:

- Either: the IBM Personal Computer, the IBM Portable Personal Computer, the IBM Personal Computer XT, the IBM Personal Computer AT, the IBM 5140 PC Convertible, the IBM 3270 Personal Computer/G, or the IBM 3270 Personal Computer/GX.

The IBM 3270 Personal Computer may be used, but this requires the “All Points Addressable Adapter card” in order to be able to display the APL character set correctly.

The IBM 4860 PCjr must be modified with a small hardware change. This is given in Appendix G, “Hardware Modification for IBM 4860 PCjr”.

- 192K or more of random access memory. A minimum of 256K is preferred.
- One double-sided 5.25 inch diskette drive
- The IBM Monochrome and Printer Adapter, the IBM Colour Graphics Adapter (or any hardware that emulates it), the IBM Enhanced Graphics Adapter, or the IBM Professional Graphics Adapter.
- DOS 2.0, or later version.
- Either: the IBM Monochrome Display, the IBM Colour Display, the IBM Enhanced Colour Display, the IBM Professional Colour Display, any other monitors that attach to an appropriate adapter, or a television set and RF modulator. (Television sets and RF modulators are not sold by IBM).
- Optional IBM Personal Computer Math Co-Processor
- Optional IBM 80 CPS Graphics Printer or the IBM Proprinter, with either the Parallel Printer Adapter or the Monochrome Display and Printer Adapter.
- Optional IBM Personal Computer Expansion Unit
- Optional IBM Asynchronous Communications Adapter

## The APL Package

| APL comes to you on a set of four 5.25 inch diskettes. These are:

- APL System Diskette (Executable code, examples and demonstration workspaces)
- APL Workspaces - 1 (General distributed workspaces)
- APL Workspaces - 2 (Graphic distributed workspaces)
- | • APL Workspaces - 3 (FORTRAN distributed workspace  
| and sample programs)

The APL program must be loaded into memory before you can use it. You should read this entire chapter before trying to use the APL system.

## The APL Character ROM

The APL character ROM replaces certain lesser used symbols in the IBM PC extended ASCII character set with the special characters and symbols needed for APL.

It can be installed on either: the IBM Personal Computer Monochrome Display and Printer Adapter; or the IBM Personal Computer Colour Graphics Adapter. Under no circumstances should it be installed on any other display adapter.

It should be noted that APL will function correctly without the ROM, but that any editing of APL functions or the entry of APL expressions would be very difficult if APL characters cannot be displayed correctly. The Colour Graphics Adapter, when in graphics mode, can display most of the APL character set (including all of those actually used by APL/PC 2.1) without the use of the ROM. In addition, since the installation of the ROM may cause certain symbols used by

other programs to be replaced by APL characters, care should be taken in deciding where to install it.

- Monochrome-only systems:

The ROM must be installed on the IBM Personal Computer Monochrome Display and Printer Adapter since this is the only way that the monochrome screen can display APL characters.

- Colour-only systems:

When installed on the IBM Personal Computer Colour Graphics Adapter, the ROM allows the display of APL characters in the alphanumeric as well as the graphics modes. The use of alphanumeric mode will provide a faster display of output since graphic mode has the overhead of generating the characters in software. However, as APL characters can be displayed without the ROM, it is recommended that it is not installed if use is made of the system for non-APL text applications such as word processing.

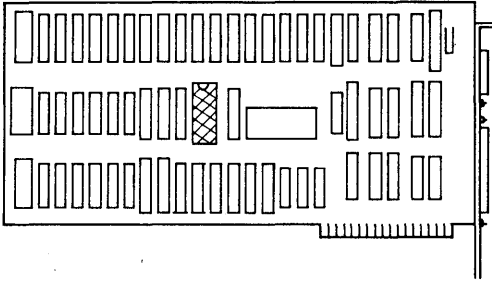
- Systems with two displays:

In this case the ROM may be installed on either adapter, but will give a clearer definition of the APL characters on the monochrome display. It is recommended that the ROM be installed in whichever adapter is least used for non-APL applications. If the system is intended solely for APL use, then the ROM should be installed in the monochrome adapter.

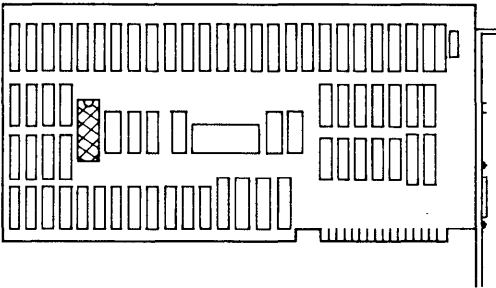
## Installing the ROM

1. Open your PC and remove the adapter onto which you intend to install the ROM (consult the appropriate *IBM Personal Computer Guide to Operations* for details of how to do this).
2. Locate the existing character ROM. This will be the only socketed (i.e. non-soldered) component on the board.

Figure 1-1 on page 1-10 shows the approximate location of this component.



**Monochrome Display Adaptor - ROM Location**



**Colour Graphics Adaptor - ROM Location**

**Figure 1-1. Character ROM Location on Display Adapter Cards**

3. Carefully insert the module puller between one end of the ROM and its connector. (A suitable tool is provided with this package). Rock the puller from side to side until the end raises slightly (see Figure 1-2 on page 1-11). Repeat this process at alternate ends until the ROM is free of its connector. Remove the ROM and set it aside.

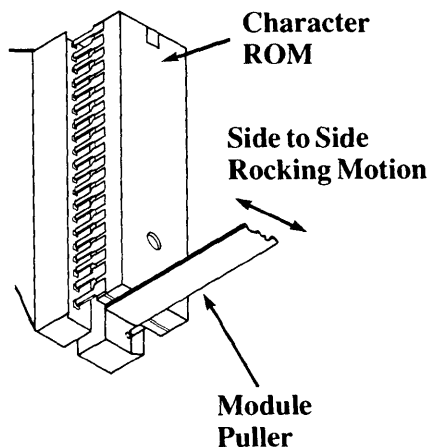


Figure 1-2. Removing the Character ROM

**Warning:**

- Incorrect placement of the APL character ROM can damage the display adapter. If you are unsure of your ability to follow these instructions, you should consult your IBM Dealer.
  - The ROM is static sensitive. Maintain personal grounding, by touching the system frame with one hand, while installing the ROM.
  - The pins of the ROM are easily bent.
4. Carefully align the pins of the APL character ROM with the connector and press firmly into place. Ensure that the notch at one end of the ROM is facing the same way as a similar notch in the connector.
  5. Return the adapter to the PC and close the case (consult the appropriate *IBM Personal Computer Guide to Operations* for details of how to do this).

The adapter will now display the APL character set.

*Note:* The characters displayed when the IBM Personal Computer Diagnostic Diskette is run will no longer correspond to those shown in the Guide to Operations.

## **Displaying APL Characters using the Enhanced Graphics Adapter**

The APL character ROM cannot be installed in the IBM Personal Computer Enhanced Graphics Adapter. In normal operations, this adapter will behave like a Colour Graphics Adapter without the ROM, and APL characters will be available in graphics modes only. However, a short program called EGAAPL.COM is provided on the APL System Disk which will load an APL font into the Enhanced Graphics Adapter, giving the same advantages of speed as the ROM.

To load the APL font, ensure that EGAAPL.COM is in the current directory and type EGAAPL. The font will remain loaded until the system is rebooted. Alternatively include EGAAPL in your AUTOEXEC.BAT file.

## **Displaying APL Characters using the Professional Graphics Adapter**

The APL character ROM cannot be installed in the IBM Personal Computer Professional Graphics Adapter. This adapter will behave like a Colour Graphics Adapter without the ROM, and APL characters will be available in graphics modes only.

## Backing Up Your Diskettes

Because you have only one copy of the APL system, you should *back up* each diskette before you begin to use APL. *Backing up a diskette* means to copy a diskette's data to another diskette. A *backup*, that is, the copy, saves you the time, trouble, and sometimes the expense, of recovering the information on a diskette that has been lost, damaged, or accidentally written over.

It is a good practice to back up your important *program diskettes* as soon as you purchase or create them. Then store your original diskettes in a safe place where they cannot be damaged. Use the backup diskettes for everyday operations.

Your *data diskettes* should be backed up every time you add or change information on them.

Full instructions on the procedure to create backup diskettes may be found in Appendix A, "Backing up Diskettes".

## Installing APL on Your Fixed Disk

If you have an IBM Personal Computer XT, an IBM Personal Computer Expansion Unit, or an IBM Personal Computer AT with fixed disk, you may wish to install APL on your fixed disk. To do this, simply:

1. Start DOS from any drive, then make sure that the prompt displayed is C> .
2. When the DOS prompt appears:
  - a. Insert your APL system diskette in drive A.
  - b. Type

```
cd \
```



and press the Enter key.

c. Type

```
copy a:fdtrans.bat c:
```

and press the Enter key.

d. Type

```
fdtrans
```

and press the Enter key.

e. Insert each of the APL workspace diskettes when requested.

When you see the message

```
APL transfer complete
```

the following will have occurred:

- A subdirectory named "APL" was created on your fixed disk.
- The files from your APL diskette were copied to your fixed disk (in subdirectory "APL").
- A batch file named "APL.BAT" was copied to the root directory on your fixed disk to make it easy for you to start APL.

When the transfer is complete, the "FDTRANS.BAT" may be erased by typing:

```
erase fdtrans.bat
```

and press the Enter key.

In addition to the APL.BAT file that is created during the fixed disk transfer procedure, a file called "RUN.BAT" is provided as an example "BAT" file to start APL with a useful set of auxiliary processors.

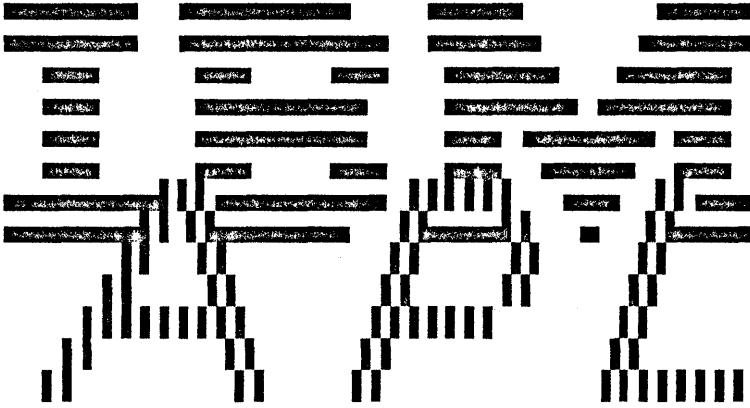
## Getting APL Started from Either Diskette or Fixed Disk

This section describes how to start APL from a diskette and from a fixed disk.

- To start APL from diskette:
  1. Insert your DOS diskette in drive A.
  2. Switch on the power to your computer.
  3. After you receive the DOS prompt, insert the APL system diskette in drive A and enter the command  
**APL**
- To start APL from your fixed disk:
  1. Ensure APL is installed on your fixed disk.
  2. Start DOS and enter the command, **APL**.

This will cause the batch file, **APL.BAT**, in the root directory to be invoked.

After the APL command is executed, the following will appear on the display screen:



*Version 2.10  
Personal Computer*

\*\*\* IBM Internal Use Only \*\*\*  
(C) Copyright IBM Corp 1985, 1986  
Produced by  
IBM Madrid Scientific Center

*CLEAR WS*

## Including Options in the APL Command

You can include options in the APL command when you bring up the system. The complete format of the APL command is:

`APL [APx] [APy] ... [)Q] [APL system command]`

where the maximum number of names given after APL, of the form APx, is fifteen.

- APx, APy, . . . , represent the filenames of auxiliary processors, which are programs that carry out special actions not included in the APL language. You can also build your own auxiliary processors (see Chapter 13, "How to Build an Auxiliary Processor"). The following auxiliary processors are included with the system:

## IBM Internal Use Only

AP2: Interface to non-APL programs  
AP80: IBM Graphics Printer control  
AP101: Stack and Profile management  
AP103: BIOS/DOS interrupt handling and I/O port control  
AP124: Full-screen display management  
AP190: IBM PC 3278/79 communications  
AP190I: IRMA<sup>1</sup> PC 3278/79 communications  
AP205: Full-screen display management  
AP206: Graphics processor  
AP210: DOS file management  
AP232: Asynchronous communications  
AP232X: Extended asynchronous communications  
AP440: Music generator  
AP488: GPIB/IEEE488 communications

- “)Q” (quiet) means that you don’t want the starting APL message to appear on the screen. No output will appear until after the first input, or the *RT* function from the UTIL workspace has been executed. (See “The UTIL Workspace” on page 11-92).
- “APL system command” means that you can enter here any APL system command to be executed at load time, thus giving you the possibility of automatically starting an APL application. (For the syntax of APL system commands, see Chapter 10, “System Commands”). This field, if given, must always be the last one in the line, and it must start with a right parenthesis. All letters must be uppercase.

If you wish to always have some auxiliary processors included, or to automatically execute an APL system command, you may create a batch file to do so (see your IBM Personal Computer DOS manual).

The APL system command, *)OFF*, is used to exit from an APL work session and transfer control to DOS. The active workspace is lost unless it was explicitly stored earlier in the work session with a *)SAVE* or *)OUT* command. Any variables

---

<sup>1</sup>IRMA is a trademark of Technical Analysis Corporation.

actively shared with an auxiliary processor will be automatically retracted upon exit from the APL system.

Examples:

```
APL AP2 AP101 )Q )LOAD PROFILE
```

This starts APL and auxiliary processors AP2 and AP101. Also, the workspace called PROFILE is loaded.

```
APL AP80
```

This will start APL and the printer auxiliary processor.

## The APL Character Set

The APL language has its own character set, which can be divided into four main classes:

- Alphabetic, which consists of the Roman alphabet in uppercase and lowercase form, and delta and delta underbar.
- Numeric, which consists of the digits 0 through 9.
- Special APL characters (see Figure 4-1 on page 4-7).
- Blank.

## The Keyboard

The APL system supports two different character-set mappings of the IBM Personal Computer keyboard: the APL character set and the National character set. The APL mapping is normally active under the APL system, and is automatically loaded with the system at the start of a work session. The National character set can be accessed under control of the APL system through the *Ctrl-Backspace* key combination, as described in the section: “Special Key Combinations” on page 1-25.

The keyboard may also be redefined to allow special key arrangements, such as those required in some European countries. See Appendix E, “APL Keyboard Redefinition”.

The keyboard consists of three general areas:

- Function keys, labelled F1 through F10, on the left side of the keyboard.
- The typewriter area in the middle, where you find the familiar letter and number keys.
- The numeric keypad, which is similar to a calculator keyboard, on the right side.

All keys are *typematic*, which means they repeat their function for as long as you press them.

### Function Keys

The only function keys automatically supported by the APL system are:

- *Alt-F1*: switch to the Monochrome Display mode, with 80 characters per line. This sets the BIOS video mode to 7. (A special APL ROM is required to see the APL special characters properly).

- *Alt-F3*: switch to the Graphics Display in alphanumeric mode, black and white, with 40 characters per line. This sets the BIOS video mode to 0. (A special APL ROM is required to see the APL special characters properly).
- *Alt-F4*: switch to the Graphics Display in graphics mode, colour, with 40 characters per line. This sets the BIOS video mode to 4. (No special ROM is required in this case).
- *Alt-F5*: switch to the Graphics Display in alphanumeric mode, colour, with 40 characters per line. This sets the BIOS video mode to 1. (A special APL ROM is required to see the APL special characters properly).
- *Alt-F6*: switch to the Graphics Display in graphics mode, black and white, with 40 characters per line. This sets the BIOS video mode to 5. (No special ROM is required in this case).
- *Alt-F7*: switch to the Graphics Display in alphanumeric mode, black and white, with 80 characters per line. This sets the BIOS video mode to 2. (A special APL ROM is required to see the APL special characters properly).
- *Alt-F8*: switch to the Graphics Display in graphics mode, black and white, with 80 characters per line. This sets the BIOS video mode to 6. (No special ROM is required in this case).
- *Alt-F9*: switch to the Graphics Display in alphanumeric mode, colour, with 80 characters per line. This sets the BIOS video mode to 3. (A special APL ROM is required to see the APL special characters properly).

Other function key combinations may be defined through the "Stack and Profile management auxiliary processor" AP101.

Display modes listed as requiring the APL ROM on the Graphics Display require the EGAAPL program to be executed to display APL characters on the Enhanced Graphics Adapter and Display.

## Typewriter Keyboard

The middle area of the keyboard behaves much like a standard typewriter. Under APL, the capitalised Roman alphabet and the digits 0 through 9 are generated when one of these keys is pressed. Most of the APL special characters that represent the primitive functions are encoded as upper-shift, and are generated by holding down either of the Shift keys and pressing the desired key.

*Note:* The Shift keys are in the bottom row of the typewriter area and have a wide arrow pointing upward.

When the National character set is active, the lowercase Roman alphabet and the digits 0 through 9 are generated when a key is pressed. The capital letters and some other characters are obtained by holding down either of the Shift keys and pressing the desired key.

*Enter:* This key, sometimes called the Carriage Return key, is the large key with the bent arrow symbol on the right side of the typewriter area. You usually have to press this key to enter information into the computer. The Enter key is used to pass an APL statement or a system command to the APL interpreter for execution.

*Esc (Escape):* The Esc key (also known as the *Weak Attention* key) is in the upper-left corner of the typewriter area. Pressing this key once generates a *weak interrupt* that halts execution at the end of a statement. If pressed during a request for input, the whole line is erased and the cursor moves to the beginning of the same line.

*Tab keys (Tabulation):* The key located under the Esc key is the tab key. If pressed, the cursor is moved to the next tab position to the right, in the same line. Tab positions are located in columns that are multiples of 8 from the left margin. The tab key has a circular effect, i.e. the first position in the line is the next tab position to the right of the last tab position in the same line.



If the tab key is pressed together with the Shift key, the cursor is moved to the next tab position to the left in the same line. The Shift-tab key has a circular effect, i.e. The next tab position to the left of the starting position in the line is the last tab position to the right of the same line.

*Caps Lock:* Although similar to a Shift Lock key on a typewriter, the Caps Lock key affects only those keys that produce the letters of the alphabet under the National character-set mapping. Once the Caps Lock key has been pressed, the alphabetic keys will continue to generate upper-shift characters until the Caps Lock key is pressed again.

Lower-shift characters can be obtained from the Caps Lock state by holding down one of the Shift keys and pressing the desired key. When you release the Shift key, the keyboard returns to the Caps Lock state.

The state of this key is ignored while the APL character set is in effect. However, if the key is pressed under the APL character set, its state will change, though it will not be effective until the National character set gets control.

*Backspace:* The Backspace key is in the upper-right corner of the typewriter area, and is marked with an arrow pointing to the left. With the APL system, both the APL and National mappings of the keyboard interpret the Backspace key as a movement of the cursor to the left. The character to the left of the initial position of the cursor will be erased.

*PrtSc (Print Screen):* Just below the Enter key is a key labelled with *PrtSc* and \*. In both APL and National character sets, pressing this key generates an asterisk. When this key is pressed while one of the Shift keys is being pressed, a signal is generated that causes a copy of the currently-active screen to be printed. If you are using the IBM Monochrome Display without the special ROM, non-APL characters will appear on the screen but will be translated to APL characters for the printer. This operation can be performed only if you have a suitable printer attached to your system and you loaded the printer auxiliary processor, AP80, either at the start of the APL work session, or later, by means of AP2.

## IBM Internal Use Only

*Other "Shifts"*: Besides the upper-shift key previously described, the typewriter keyboard has two other "shift" keys - the *Alt* (Alternate) and the *Ctrl* (Control) keys. Like the Shift key, these keys must be held down while a desired key is pressed.

The Alt key is used with the APL character-set mapping to produce lowercase letters, and some special APL characters along the top row. Under the National character set, the Alt key has no effect with the typewriter keyboard. The Alt key is also used with the keys on the numeric keypad to enter characters not encoded on the keys. This is done by holding down the Alt key while typing the three-digit decimal ASCII code for the desired character (see Appendix C, "The APL Character Set and  $\square V$ ", and Appendix D, "Internal Representation of Displayed Characters").

Some additional keys, duplicating other keys, are included in the keyboard layout. These provide compatibility with the IBM 3270 PC/G and IBM 3270 PC/GX APL keyboard layouts. The extra keys are entered in Alt mode and give the correct characters as engraved on the front of the keys on the APL featured keyboards available as an option on these devices.

| The APL keyboard layout of the IBM 3270 PC keyboard will  
| not be correct when running Control Program (CP) if a DOS  
| keyboard program has been loaded (e.g. KEYBUK). Similarly,  
| the APL keyboard layout of the IBM 3270 PC/G or IBM 3270  
| PC/GX keyboard will not be correct when running Graphics  
| Control Program (GCP) if a DOS keyboard program has been  
| loaded. The US keyboard format must be selected (using  
| Ctrl-Alt-F1) to ensure that the APL keyboard layout is  
| correct. The appropriate national language keyboard layout  
| may be reselected after exiting APL with Ctrl-Alt-F2.

The Ctrl key is similarly used to generate certain codes and characters not otherwise available from the keyboard.

Many of the box characters may be entered from the keyboard using the Ctrl key: single box characters are produced by Ctrl-S and the surrounding eight keys; double box characters are produced by Ctrl-( (The APL "(" key to the right of the L

key) and the surrounding eight keys. The horizontal and vertical bars are entered with Ctrl-H and Ctrl-V, and double horizontal and vertical bars are entered with Ctrl-J and Ctrl-B keys.

The Ctrl-Backspace combination is used to switch between the APL and National character-set mappings. The Alt-Backspace combination also has the same effect.

## Numeric Keypad

This area of the keyboard is normally used in conjunction with the APL Input Editor, which is described later in this chapter. (See “The APL Input Editor” on page 1-29). The numeric keypad also can be used as a calculator keypad by pressing one of the Shift keys at the same time you press the keys on the keypad, or by pressing the *Num Lock* key to enter the Num Lock state. The Num Lock key affects the keys of the numeric keypad in the same way the Caps Lock key affects the alphabetic keys of the typewriter keyboard. Pressing the Num Lock key once will cause upper-shift numeric characters to be generated. You can temporarily nullify this state by holding down a Shift key. To return the keypad to its normal mode under the APL Input Editor, press the Num Lock key a second time.

On the extreme right side of the keyboard are two keys that are normally used with the numeric keypad.

The key engraved with a minus sign (-):

- When the APL character set is active, this generates the “delta underbar” symbol in lower case and the minus sign in upper case (shifted) mode.
- When the National character set is active, this generates the minus sign in both lower and upper case modes.

The key engraved with a plus sign (+):

- In lower case mode (regardless of which character set is active), this is an additional Enter key, with a function

slightly different from the normal Enter key located in the typewriter keyboard. The normal Enter key, when pressed, copies the line to be executed to the bottom of the screen, but the “plus” key executes the line in place and does not affect the position of the cursor.

- In upper case (shifted) mode, this generates the plus sign.

## **Special Key Combinations**

You should be aware of the special functions of the following keys or combinations of keys:

- *Ctrl-Backspace*: Changes the keyboard from the National character-set mapping to APL, or from the APL character-set mapping to National. The Alt-Backspace combination also has the same effect.
- *Ctrl-Break*: Generates a strong interrupt that will cause an execution within a statement to halt as soon as the interrupt is detected. The key is also used to halt a request for literal (␣) input from a defined function. If pressed during any other request for input, the present line is erased and the cursor moves to the start of the same line.
- *Ctrl-Alt-Del*: Performs a *system reset*, which is the same as switching the computer from off to on. Hold down the Ctrl and Alt keys, and press the Del key. Doing a system reset with these keys is preferable to setting the Power switch off and on again, because the system will come up faster.
- *Ctrl-PrtSc* or *Alt-PrtSc*: This serves as an on-off switch for sending display output to the printer as well as the screen, provided you have previously loaded the printer-handling auxiliary processor, AP80 (see “Getting APL Started from Either Diskette or Fixed Disk” on page 1-15 or “The AP2 Auxiliary Processor”).

Press these keys to send display output to the printer, then press them again to stop sending to the printer. Although this action enables the printer to function as a *system log*,

it slows down some operations because the computer waits during the printing.

- *Ctrl-Num Lock*: Puts the computer into a *pause* state. This can be used to temporarily stop printing or program listing. The pause continues until any key, except the “shift” keys, the Break key, the Ctrl-Num Lock key, or the Ins key, is pressed.

## Keyboard Keycaps

To simplify the use of APL on the standard PC or PC/AT keyboard a set of APL keycaps may be purchased which will replace the existing set. These keycaps are for the 3179 colour terminal, and although they are complete and correct for the APL keyboard mode, certain upper case keys in National mode may not be correctly positioned. To correct this, you may use the DOS ANSI.SYS device driver to map these keys to their engraved characters.

These keycaps are available from IBM as part numbers:

- 1351711 - English US
- 1351712 - German
- 1351713 - French (AZERTY)
- 1351714 - Italian
- 1351715 - English UK
- 1351719 - Belgian
- 1351720 - Danish
- 1351721 - Swedish
- 1351722 - Norwegian
- 1351723 - Portuguese
- 1351724 - Swiss/German
- 1351725 - Swiss/French

## **Keyboard Template**

A template showing the APL character set for the IBM Personal Computer keyboard has been included with this book. This may be placed along the top of the keyboard as a quick reference to the positions of the APL keys.

Figure 1-3 on page 1-28 shows the keyboard with the APL character set.

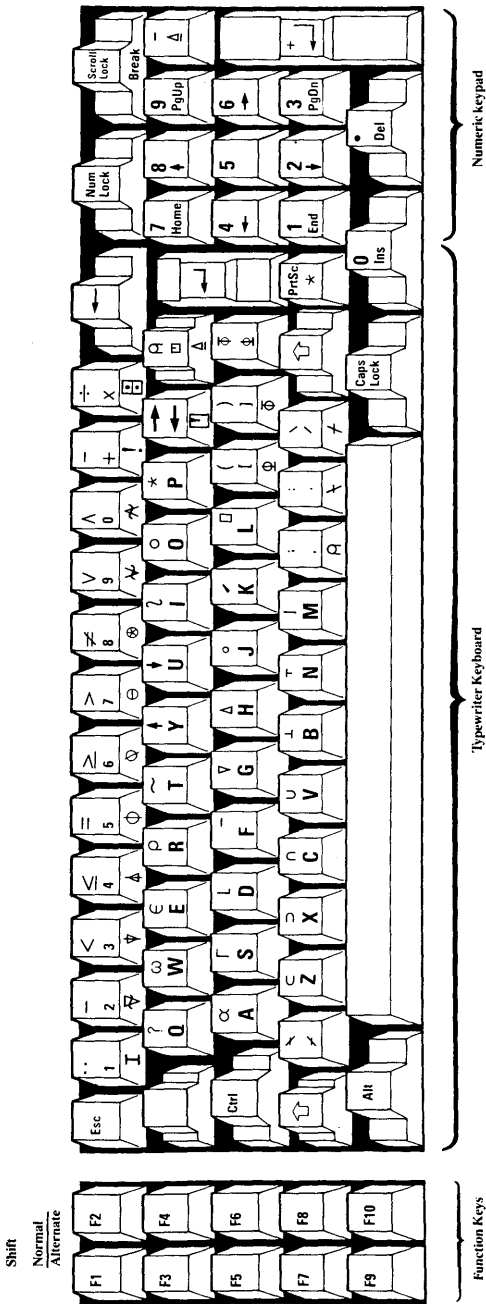


Figure 1-3. Keyboard with APL Character Set

## The APL Input Editor

The APL Input Editor is a *full-screen editor*. This means that you can enter a line (with or without a previous change) anywhere on the screen. To enter a line for execution, the *cursor* must be on that line.

The cursor is a blinking underbar or block appearing just to the right of the last character typed. You can position the cursor by using the APL Input Editor special keys, which are described in the next section. The cursor marks the position at which a character is to be typed, inserted, or deleted.

The input editor can save much time during program development by eliminating unnecessary re-typing. In execution mode, the input editor can be used to make changes to a previous line. When the changed line is entered, if the normal Enter key was pressed, the line is echoed at the bottom of the screen and executed. However, if the secondary Enter key (the key engraved with a plus sign at the right of the numeric keypad) was pressed, the line is executed in place.

The input editor also can be used within the del (∇) editor during function definition (see Chapter 8, “Function Definition”) to help create or modify programs.

A full-screen, defined-function editor is included with the EDIT workspace and is described in Chapter 11, “Application Workspaces”. This special editor provides additional features that help make function definition even easier.

## APL Input Editor Special Keys

You can use some of the keys on the numeric keypad, and the Backspace key, to move the cursor on the screen, to insert characters, or to delete characters. The keys and their functions are:

- *Up Arrow (Cursor Up - Numeric Keypad 8)*: Moves the cursor up one line. If the cursor advances beyond the



upper end of the screen, it will move off the screen and reappear at the lower end in the same column.

- *Down Arrow (Cursor Down - Numeric Keypad 2)*: Moves the cursor down one line. If the cursor advances beyond the lower end of the screen, it will move off the screen and reappear at the upper end in the same column.
- *Left Arrow (Cursor Left - Numeric Keypad 4)*: Moves the cursor one position to the left. The cursor cannot advance beyond the left edge of the screen.
- *Right Arrow (Cursor Right - Numeric Keypad 6)*: Moves the cursor one position to the right. The cursor cannot advance beyond the right edge of the screen.
- *Home (Numeric keypad 7)*: The cursor moves to the start of the current line.
- *End (Numeric Keypad 1)*: The cursor moves to the end of the current line (just after the last non-blank character in the line).
- *Ctrl-Home (Control key plus Numeric keypad 7)*: All characters to the left of the current position of the cursor are erased in the current line and replaced by blanks.
- *Ctrl-End (Control key plus Numeric keypad 1)*: Erases characters from the current cursor position to the end of the line.
- *Ins (Numeric Keypad 0)*: Sets Insert mode on or off. If Insert mode is off, pressing this key will turn it on. If Insert mode is already on, pressing this key will turn it off.

You can tell when Insert mode is on, because the cursor is displayed as a blinking block covering the character position instead of a blinking underbar. When Insert mode is on, the character at the cursor position, and characters following the cursor, are moved to the right as you type characters at the current cursor position. After each keystroke, the cursor moves one position to the right. If you try to write beyond the right edge of the screen

(regardless of the state of Insert mode), you will hear a warning *beep*.

When Insert mode is off, any characters you type will replace the existing characters on the line.

Pressing an Enter key when Insert mode is on will automatically turn Insert mode off.

- *Del* (Numeric Keypad Decimal Point (.)): Deletes the character at the current cursor position. All characters to the right of the one deleted move one position to the left to fill the empty space.
- *Backspace* (Left arrow to left of Num Lock key): Its function is the same as the Cursor-Left key followed by the Del key, because the APL backspace is destructive.
- *Esc*: When pressed anywhere in a line, Esc causes the whole line to be erased, and the cursor moves to the beginning of the line. The line is not passed to APL for processing. If you press Esc while a defined APL function is executing (see Chapter 9, “Function Execution”), the function is interrupted after the current line is executed. This is called a *weak interrupt*.
- *Tab*: The cursor advances to the next tab position (a multiple of eight positions from the left margin). The key has a circular effect (wrap around on the same line).
- *Back Tab* (Shift-Tab): The cursor returns to the preceding tab position (a multiple of eight positions from the left margin). The key has a circular effect (wrap around on the same line).

## How to Make Corrections on the Current Line

Any line of text typed while APL is in the input state will be processed by the line editor, so you can use any of the keys described in the previous section. APL is in the input state whenever the cursor is visible. When one Enter key is finally pressed, the entire line in which the cursor lies is passed to APL for processing. The cursor is not visible during processing time. When the cursor appears again, APL has returned to the input state.

*Changing Characters:* If you are typing a line and discover you typed something incorrectly, use the Cursor-Left or Cursor-Right keys to move the cursor to where the mistake was made, then type the correct characters over the incorrect ones. You can then move the cursor back to the end of the line, using the End key, and continue typing.

*Erasing Characters:* If you notice you have typed an extra character in the line, you can erase (delete) the character using the Del key or the Backspace key. Use the Cursor-Left or other cursor-control keys to move the cursor to the character you want to erase. Then press the Del key, and the character is deleted. Or move the cursor to the right of the character to be deleted, then press the backspace key. Use the End key to move the cursor back to the end of the line and continue typing.

*Adding Characters:* If you see that you have omitted characters in the line you are typing, move the cursor to where you want to add the new characters. Press the Ins key to set Insert mode on, then type the characters you want to add. The characters you type will be inserted at the cursor position. The character that was at the cursor position, and those following the cursor, will be pushed to the right. When you are ready to resume typing where you left off, press the Ins key again to set Insert mode off (the cursor will return to its ordinary form), and use the End key to get back to your place in the line. Then continue typing. If you forget to press the Ins key to set Insert mode off, it will automatically be turned off when you press an Enter key. You don't have to move the

cursor to the end of a line before pressing the Enter key. The whole line will be executed anyway.

*Erasing Part of a Line:* To end a line at the current cursor position, press the Ctrl-End key. Then you can continue typing. Similarly, the line up to (but not including) the current cursor position may be erased with the Ctrl-Home key.

*Cancelling a Line:* To cancel a line that you are typing, press the Esc key anywhere in the line. (You do not have to press Enter). The line is completely erased, and the cursor moves to the line beginning.

## **Use of Displays**

APL enables you to work, sequentially, in the following eight modes during the same working session:

- Monochrome Display
- Colour Graphics Adapter in alphanumeric modes:
  1. Black and white, 40 characters per line.
  2. Colour, 40 characters per line.
  3. Black and white, 80 characters per line.
  4. Colour, 80 characters per line.
- Colour Graphics Adapter in graphic modes:
  1. Colour, 40 characters per line.
  2. Black and white, 40 characters per line.
  3. Black and white, 80 characters per line.

Only the Colour Graphics Adapter graphic modes directly support APL characters. You may use the Monochrome Display mode or the alphanumeric modes of the Colour Graphics Adapter. However, some APL characters will not be displayed unless a special APL ROM is installed in the corresponding Adapter.

*Note:* At any time during the work session, you can change modes without leaving APL.

At load time, if you have both a Monochrome and Printer Adapter, and a Colour Graphics Adapter, the Monochrome mode is activated.

If you want to change to the Colour Display mode, press one of the Alt F-key combinations, according to the following table:

| <u>Key-Pressed</u> | <u>Changes to Mode</u>          |
|--------------------|---------------------------------|
| Alt-F1             | Monochrome                      |
| Alt-F3             | Alphanumeric Black and White 40 |
| Alt-F5             | Alphanumeric Colour 40          |
| Alt-F7             | Alphanumeric Black and White 80 |
| Alt-F9             | Alphanumeric Colour 80          |
| Alt-F4             | Graphic Colour 40               |
| Alt-F6             | Graphic Black and White 40      |
| Alt-F8             | Graphic Black and White 80      |

| For further information on these F-key settings, see "Function  
| Keys" on page 1-19.

APL does not allow you to switch to a display that is not available.

When you switch from one display to another, for example, from display A to display B, the screen on display B clears; however, the screen on display A does not. Thus you can keep part of the session displayed on display A (graphics, listing of APL objects, etc.) and continue working with display B.

To clear a screen you are working with, switch to that display by pressing the appropriate Alt-F key combination. If you try to switch to a display that is not physically connected or

## IBM Internal Use Only

switched on, you can return to the original display by pressing the appropriate Alt-F key combination.

*Note:* The APL system detects the IBM Personal Computer configuration reflected in the switch settings (see the PC Technical Reference manual). If your actual configuration is different (for example, you forgot to switch on your Colour Display), the system may switch to a display that is not operating, and you will not be able to see anything you type, although it can be executed if you press an Enter key (more about this later). This condition can easily be confused with a system hang. If this happens, the best action is to return to the active display by pressing the appropriate Alt-F key combination.

## Disk(ette) Drives

APL workspaces are collected into *libraries*, which are identified by an integer number. Each disk drive of the IBM Personal Computer represents an APL library, with the following default identification number:

| <u>Device</u>         | <u>DOS Drive Spec.</u> | <u>APL Library</u> |
|-----------------------|------------------------|--------------------|
| First diskette drive  | A                      | 1                  |
| Second diskette drive | B                      | 2                  |
| First fixed disk      | C                      | 3                  |
| Second fixed disk     | D                      | 4                  |

Disk drives are usually controlled under APL by system commands (see Chapter 10, "System Commands") relating to workspace storage and retrieval. If no library number is specified for these commands, the device that is the current DOS default drive will be used. Specifying an invalid library number that corresponds to a non-existent drive should be avoided, because the system may perform an unintended action.

The disk drives also can be controlled with the DOS file management auxiliary processor, AP210, which is discussed in

Chapter 12, “Auxiliary Processors”, and the FILE workspace, which is discussed in Chapter 11, “Application Workspaces”. The drives that each library number refers to may be redefined with the AP101 Profile and Stack Auxiliary processor, and this may also be used to specify a path in addition to the drive for each library.

## The Printer

The optional IBM 5152 Graphics Printer or IBM 4201 Proprinter can be used to produce both APL and non-APL characters, if the printer auxiliary processor, AP80, is specified as a parameter to the APL command at load time, before the start of a work session, or has been loaded by means of AP2. Certain other printers may be used, but may require the use of a patched version of AP80 in order to operate correctly. See “Patching AP80 for Other Printers” on page 12-12 for further information.

*Note:* Switch 6 on the IBM 4201 Proprinter must be set to *ON* for correct operation with APL.

As described in a previous section about the keyboard, the following key combinations can be used to control the printer:

- *Shift-PrtSc*: A printed copy is made of the currently-active screen.
- *Ctrl-PrtSc* or *Alt-PrtSc*: Acts as an On/Off switch for sending display output to the printer, as well as to the screen. This allows the printer to be used as a system log to provide a record of the work session.

The AP80 auxiliary processor also allows selective printing of desired APL objects or results. Chapter 12, “Auxiliary Processors” discusses, in detail, the use of AP80 to control the printer with a shared variable, and Chapter 11, “Application Workspaces” explains the use of the PRINT workspace.

## **BM Internal Use Only**

Control codes can be sent to the printer, but they will not affect the APL special characters.



**Notes:**

## Chapter 2. APL Tutorial

|  |      |
|--|------|
| What is APL? .....                             | 2-3  |
| Arrays and Functions .....                     | 2-4  |
| Interactivity .....                            | 2-7  |
| The APL Calculator .....                       | 2-8  |
| Order of Execution .....                       | 2-10 |
| Powers and Logarithms .....                    | 2-11 |
| Circular Functions .....                       | 2-13 |
| Computations with Multiple Data Items .....    | 2-14 |
| Variations on the Compound Interest Problem .. | 2-17 |
| Named Data .....                               | 2-18 |
| APL Workspaces .....                           | 2-20 |
| Other Functions and Operators .....            | 2-24 |
| Input/Output .....                             | 2-24 |
| Shared Variables .....                         | 2-25 |
| A Simple Example .....                         | 2-26 |
| Shared and Other Variables .....               | 2-27 |
| Debugging .....                                | 2-28 |
| Quad output .....                              | 2-29 |
| Tracing .....                                  | 2-30 |
| Stopping .....                                 | 2-31 |
| Alternate execution .....                      | 2-31 |
| Speeding Up APL Code .....                     | 2-32 |
| Use Timing Tools to Find Hot Spots .....       | 2-33 |
| Eliminate Explicit Loops .....                 | 2-33 |
| Avoid Waxing or Waning Variables .....         | 2-35 |
| Avoid Large Outer Products .....               | 2-36 |
| Avoid Tiny Functions .....                     | 2-36 |
| An Example - Computing a Histogram .....       | 2-37 |
| Computing the Histogram with an Explicit Loop  | 2-37 |
| Computing the Histogram with an Outer Product  | 2-39 |
| Computing the Histogram with Grade Up .....    | 2-41 |
| How to Write a Timing Function .....           | 2-42 |
| A Simple Timing Program .....                  | 2-42 |
| An Improved Timing Function .....              | 2-44 |
| An Alternative Solution .....                  | 2-46 |
| Concluding Thought .....                       | 2-47 |

Further Reading ..... 2-4

**Notes:**

## What is APL?

APL is an interactive, array-oriented computer language. It is a system of computation based on four cardinal ideas:

1. APL is array-oriented; that is, data items are collected and operated on in aggregates. It is as simple to operate on a one-dimensional list (a “*vector*”) or two-dimensional table (a “*matrix*”) as it is to work with a single number. Arrays can be dynamically expanded or contracted to hold any amount of data.
2. APL is function-oriented; that is, data arrays are transformed into other data arrays by the application of *functions*. A rich set of primitive functions is built-in to APL; new functions are easy to write and have compatible syntax and capabilities.
3. APL is interactive; that is, the user can readily write new APL code at a display, can test it and correct it. A *session manager* retains the most recent display of input and output lines for ready reference. The user can build packages of functions and data to perform some task; such a package (a “*workspace*”) can easily be examined or altered.
4. APL hides details of its implementation; that is, the physical details of the computer running APL are not the concern of the user. APL automatically allocates the correct amount of storage, assigns the format of numeric data, performs all input/output, and checks automatically for many kinds of errors, such as ensuring the correct type and size of data being passed to subfunctions.

Some of the other notable features of APL include:

- simple and consistent syntax
- function recursion
- function tracing, suspension and resumption

- uniform object naming within a one level store
- executability of incomplete functions
- simple input and output
- code portability
- easy library updating
- easy code updating
- direct links to other languages

A very simple example of calculation using APL:

```
TWOS←2*1 2 3 4 5 6 7 8 9 10
)SAVE TWOPOWER
```

This creates a list of powers of two and saves it on the user's disk in a workspace named TWOPOWER. Many more examples will be given below.

## Arrays and Functions

Let us look in more detail at each of the terms used above. An array can be as simple as a single number or single character, and is then called a *scalar*. If it is a simple one-dimensional list, it is called a *vector*. A two-dimensional table is called a *matrix*. No special name exists for arrays with three or more dimensions. The number of dimensions of an array is called its *rank*, and each dimension is called an *axis*.

An array must be *homogeneous*; that is, it must contain only numbers or only characters. Characters can be chosen from the 256 ASCII characters, including: letters, digits, punctuation marks, special characters, and control characters such as line-feed. Numbers can be whole integers, or may have fractional parts or exponents: in the terminology of other computer languages, APL numbers may be either fixed or floating point, at will. Boolean arrays (values of 0 and 1, which are also truth values in APL) are packed together by

PL to save storage, but for computational purposes, they are treated like any other numbers.

An array with more than one dimension must be *rectangular*: that is, all parallel axes must contain the same number of data elements. For example, it is not permissible in APL to have a matrix with a different number of items in each row.

Functions come in several varieties. Most important are the primitive built-in functions, which perform arithmetic, relational, structural and general computing operations. For example:

- “+” does addition
- “×” does multiplication
- “\* ” does exponentiation
- “⊙” computes the logarithm
- “!” computes the factorial (Gamma) function
- “∨” computes the boolean “or”
- “∇” finds the maximum
- “∈” determines membership in a set,
- “∩” searches a vector for a given item
- “⊎” determines the alphabetic or numeric order of an array
- “?” generates random integers
- “⍒” converts to any desired number base
- “⍉” inverts a matrix

The primitive structural functions are unique to APL among the widely used programming languages. They include:

- catenation (using comma, “,”)
- determining the current shape of a data array (monadic rho, “ $\rho$ ”)
- changing the shape of an array of data (dyadic rho, “ $\rho$ ”)
- extraction of the first (or last) few columns of a matrix or items of a vector (take, “ $\uparrow$ ”)
- dropping of the first (or last) few columns or items (drop, “ $\downarrow$ ”)
- transposition (transpose, “ $\Phi$ ”)
- rotation and reflection (rotate, “ $\Phi$ ”)
- extraction of an arbitrary rectangular subarray using vector subscripts (with “[ ]”)
- extraction of an arbitrary rectangular subarray according to a boolean relationship (such as “ $(C \neq ' ')/C$ ”)

There are nearly sixty primitive functions in APL, each one symbolised by a single special character. With some exceptions, the primitive functions can operate on arrays of any size or rank. Similarly, the non-arithmetic and structural functions work on numeric or character arrays in similar ways.

Other kinds of functions may be created by the user of APL as needed. *Derived* functions are written as two or three adjacent special characters, and represent a compound of one or two primitive functions with a primitive *operator*. For example, the built-in operator “/” is called *reduction* and means “combine all items along one axis according to the indicated function”. “+/6 7 2” computes the sum 15 of the vector shown, “×/6 7 2” computes the product 84, and “ $\Upsilon$ /6 7 2” computes the maximum 7. The *inner product* operator, represented by a period, combines two functions in a particular way. For example, “ $A + . \times B$ ” is the familiar “dot product” of matrix algebra.

Lastly, *defined*, or *user-written* functions correspond to the subroutines or procedures of other languages, and may contain many lines of code. They are generally written with the aid of a *function editor*. Only one control structure is permitted in defined functions - the *branch* (or “*go-to*”), symbolised by an arrow pointing to the right (→). Loops can be written explicitly with branches, but in practice, they are required much less often in APL than in other programming languages because of the built-in array functions and operators.

*Note:* The terms *function* and *operator* refer to two very different kinds of object in APL. Functions work on data arrays to produce new data arrays, while operators work on functions to produce new functions. In other computer languages, these terms are often used interchangeably.

## **Interactivity**

Normally, all lines of APL must be entered from the keyboard. However, any line may invoke user-defined functions, so often all the user need type is a brief “trigger” expression to start a long computation running. It is possible with the aid of Auxiliary Processor 101 (AP101) to automate responses that would otherwise have to be typed in manually to a program that requests them.

To write a defined function, a function editor is needed. APL provides one built-in, called the *del-editor*, after the special character “∇”, which causes entry to it. The body of a defined function consists of APL lines to be executed in order, plus branch statements to alter the order of execution. The header line gives the name of the defined function, and the names of the dummy variables which are the input, output and local work variables. (Other function editors are available too; workspace EDIT has one written in APL, and another that can invoke a DOS editor such as Personal Editor).

Testing a defined function is as simple as typing in a line of APL code that executes it. A natural style of writing a complex program in APL is to write and test subcomponents separately. Test data for the subcomponents can be pre-computed or entered by hand, and kept together with the



function being tested in the workspace. It is never necessary to write a complete function in APL before testing it. This procedure greatly decreases the time needed to produce a working program.

When an error occurs in a line of APL code, the APL system stops immediately and displays an error message along with a copy of the offending statement and a pointer to the error position. Often, the user can immediately determine the source of the error, correct it, and resume execution from the point of error. For example, suppose that inside a defined function named *COMPUTE*, line 4 performs a division by variable *X*. Should *X* happen to be 0 at some time, then execution of that line would produce the following display:

```
DOMAIN ERROR
COMPUTE[4]
      Z←1÷X
      ^
```

The execution of function *COMPUTE* is now *suspended*, and the APL system is waiting for the user to type in a new command. The user can determine that *X* is indeed 0, and then perhaps might set *X* to another value and resume execution at this same line to see what other errors show up. Eventually, the user may abandon the current execution altogether, correct all the errors discovered in the defined functions, and restart. For more details see the section “Debugging” on page 2-28.

## The APL Calculator

For those who sit for the first time at an APL display, it can be something as simple and easy to use as a pocket calculator. In fact, the simpler operations are written with the usual mathematical notation, with no need to learn complicated keywords. You just type the operation you wish to perform, press the Enter key, and the system answers immediately with the result. For instance:

```
      2+3
5
```

*Note:* From now on, lines entered by the user will be indented six positions to the right with respect to the results produced by the machine. This notation is not arbitrary, since it corresponds to the way things appear on the screen in most APL systems.

In the same way as we have performed an addition, we can also use any other typical arithmetic operation, the symbols of which are well known.

```
      12-3
9
      3-12
-9
      4×7
28
      30÷6
5
```

Observe that, in APL, negative numbers are represented with a special sign symbol, different from the one used for the subtraction operation. This *overbar* (or *high minus*) is not a function symbol, but is part of the representation of negative numbers, just as the decimal point is used in the representation of fractions. The *overbar* character is used whenever you want to introduce a negative number, and can be obtained by pressing and holding the Shift key and then pressing the 2 key.

If the number to be input is not an integer, the integer part should be separated from the fractional part by a decimal point.

```
      2.5×4
10
```

You don't have to worry about whether the result of an operation will be an integer or not; the system takes care of that.

```
      4÷3
1.3333333333
```

## Order of Execution

Several consecutive operations may be performed in a single line:

```

      4×3÷2
6

```

However, APL has a property that, at first sight, you may find surprising: when several operations are to be performed successively, the first one to be executed is the rightmost one. In the above example, the division  $3 \div 2$  will take place first, giving a result of  $1.5$ ; then this number will be multiplied by  $4$  ( $4 \times 1.5$ ), giving the expected result of  $6$ .

Sometimes, if this rule is not taken into account, you may believe at first sight that the result given by the machine is wrong. But, with a little practice, you will soon get used to this order of evaluation.

Let us look at an example:

```

      4×3+2
20

```

If we are used to giving multiplication precedence over addition, we would tend to believe that the result of this operation should be  $14$  ( $4 \times 3 = 12$ ;  $12 + 2 = 14$ ). But, since APL evaluates from right to left, what has happened here is:  $3 + 2 = 5$ ;  $4 \times 5 = 20$ .

The reason why APL performs its operations in this way is not arbitrary. Because APL has many more functions than ordinary arithmetic, precedence rules based upon functions would be quite complex, especially since they would have to take into account the derived and user-defined functions, as well as the primitives. To avoid this complication, the simple right-to-left rule is used, which makes the understanding of any statement dependent only on its visible form, independent of the particular functions involved. In any case, it is always possible to make APL execute the calculations in any desired order. A proper use of parentheses is sufficient.

```
      (4*3)+2
14
      4*(3+2)
20
```

In this way, we tell APL that we want to perform first the operation located between the parentheses. Notice that while it is not incorrect, it is never necessary to parenthesise the last part of an APL statement because the right-to-left rule accomplishes the same result.

Obviously, you may use as many parenthesis pairs as you want. You should only remember that, in a given line, there must be as many opening parentheses as there are closing parentheses, and these should be in the proper order, for otherwise the whole line will be rejected by the system with an error message.

## **Powers and Logarithms**

Besides the operations we have seen (addition, subtraction, multiplication and division), APL contains several more, of the type usually available in pocket scientific calculators. For instance, the power function, represented here by an asterisk:

```
      4*3
64
      16*0.5
4
```

4 to the power 3 ( $4*3$ ) is equivalent to multiplying 4 times itself three times, and is thus equal to 64. Observe, however, that APL does not contain a symbol for the “square root”, since this is really a special case of the power operation, with exponent equal to 0.5 (one half).

The power function can also be mixed with other operations in the same line. For instance, let us apply the formula of compound interest to calculate the capital to be obtained, after 5 years, by £1000 at 11% cumulative interest. We would simply type the following:

```
      1000*(1+0.11)*5
1685.058155
```

Notice that as operations are performed from right to left, the power function  $(1.11)*5$  would take place before the product.

The operation inverse to the power function is the logarithm. APL has a special symbol ( $\otimes$ , obtained by pressing Alt-8) that makes it available. This function has two forms:

```

      10⊗2
0.3010299957
      ⊗10
2.302585093

```

The first case computes the decimal logarithm of 2. The second, the natural logarithm of 10. In other words: when the logarithm function is written between two values, (as in  $10\otimes 2$ ), the first value specifies the base of logarithm and the second is the number for which the logarithm will be computed. If the left value is not given, the system assumes we want to obtain the natural logarithm (base  $e$ , where  $e = 2.718281828459$ ). The same rule applies also to the power function, i.e. if the power base is omitted, the system will use in its place, the number  $e$ .

```

      *2
7.389056099
      *1
2.718281828

```

Observe that  $*1$  ( $e$  to the power 1) represents the value of number  $e$ .

From now on, we will call *dyadic functions* (from the Greek “duo”, meaning two) those applied to two different values (the “arguments”), one of which is written to the left and the other one to the right of the symbol for the function, as in  $3*4$ . On the other hand, we will call *monadic functions* (from the Greek “monos”, meaning one) those that only have a right argument, as in  $*2$ .

Sometimes the same symbol represents different functions when used in these two different ways, and we will see examples of this later.

## Circular Functions

Another set of functions frequently used in scientific computations is the trigonometric circular and hyperbolic functions (sine, cosine, tangent, and their inverses). In APL, all of these are represented by a circle, (o, obtained by pressing the Shift-O combination), as a mnemonic for the “circular functions”. The right argument of the circle is the value of the angle (in radians) to be used in the computation. The left argument is a number indicating which trigonometric function we want to calculate, according to the following table:

| <u>Left argument</u> | <u>Trigonometric function</u> |
|----------------------|-------------------------------|
| 1                    | Sine                          |
| 2                    | Cosine                        |
| 3                    | Tangent                       |
| -1                   | Arc sine                      |
| -2                   | Arc cosine                    |
| -3                   | Arc tangent                   |

As you see, the negative numbers represent the trigonometric functions inverse to those obtained with the corresponding positive numbers. Left arguments of 0, 4 to 7 and -4 to -7 are reserved for other operations, including the hyperbolic functions; see the reference section in this manual). Recall that the negative sign is obtained by pressing the Shift-2 combination.

Let us look at some examples:

```

      1o1
0.8414709848
      2o1
0.5403023059
DOMAIN ERROR
      2o2
      ^
    
```

The first two cases compute the sine and cosine of an angle of 1 radian. In the third example we find an error message, indicating that we have tried to perform an operation with an invalid argument. In fact, it is impossible to find an angle whose cosine is 2, since the cosine of every (non-complex)

angle belongs to the closed interval  $[-1, 1]$ . This error message also appears with the ordinary arithmetic operations. For instance:

```

      1÷0
DOMAIN ERROR
      1÷0
      ^
      1000*1000
DOMAIN ERROR
      1000*1000
      ^

```

In the first case, the error happens because you cannot divide an ordinary integer by zero. In the second case, 1000 to the 1000-th power is larger than the maximum number representable in this APL system, and the operation cannot be done.

Like the power and logarithm, the circular function also has a monadic form (with a single argument). Its result is the product of  $\pi$  times the value of the argument:

```

      o2
6.283185307
      o1
3.141592654

```

Observe that  $\circ 1$  represents the value of  $\pi$ .

## Computations with Multiple Data Items

In APL, any of the functions we have just seen (and several others we will see later) can be applied to a multiplicity of data elements at the same time, thus allowing the execution of many simultaneous operations in a simple way. A few examples will explain it clearly:

```

      5×1 3 5 7
5 15 25 35
      3÷1 2 3
3 1.5 1
      1 2 3÷4
0.25 0.5 0.75

```

In the first case, 5 is multiplied by the four values at the right of the "times" sign. The four results appear then in a single line. In the second case, 3 is divided by 1, 2 and 3. Finally, in the third example, each of the numbers 1, 2 and 3 is divided by 4. Notice that the multiplicity of values can be either at the right or at the left of the operation to be performed.

This reduced notation finds many applications. For instance: let us suppose we must perform the conversion to Dollars of certain quantities expressed in Pounds Sterling, such as 1532, 12288 and 23881. If the exchange rate for the Dollar on a certain date is 1.23, (1 Pound = 1.23 Dollars) the conversion will be performed in the following way:

$$\begin{array}{rcccc} & 1532 & 12288 & 23881 & \times & 1.23 \\ 1884.36 & 15114.24 & 29373.63 & & & \end{array}$$

This is not the only way to perform computations with multiple data items. Any of the seven function symbols we have seen up to now may also apply to data collections on both sides at the same time. But in this case there exists a restriction: the number of data items at the left must be the same as the number of data items at the right. The operation is now performed in a pairwise manner, namely: the first element on the left is combined with the first one on the right, the second on the left with the second on the right, and so on, until the end of both collections.

$$\begin{array}{rcccccccc} & & 2 & 3 & 5 & 7 & \times & 4 & 3 & 2 & 1 \\ 8 & 9 & 10 & 7 & & & & & & & \end{array}$$

In the above example, the result is equal to  $2 \times 4$ ,  $3 \times 3$ ,  $5 \times 2$  and  $7 \times 1$ . Instead of the times sign, you may use any other:

$$\begin{array}{rcccccccc} & & 2 & 3 & 5 & 7 & \leq & 4 & 3 & 2 & 1 \\ 1 & 1 & 0 & 0 & & & & & & & \end{array}$$

where we have obtained the results of  $2 \leq 4$ ,  $3 \leq 3$ ,  $5 \leq 2$  and  $7 \leq 1$ , respectively.

What happens if the number of data items at the left is different from the number of data items at the right? Let's try it:



```

      1 2 3 × 3 4
LENGTH ERROR
      1 2 3×3 4
      ^

```

The system recognises an anomalous condition, and answers with an error message.

There are other ways in which computation can be performed on many data elements at once, using the APL operators: *reduction*, *scan*, *inner product* and *outer product*. These operators are used with primitive functions to produce derived functions, which then operate on data collections in useful ways.

For example, in the Pound to Dollar conversion problem above, if we had been interested only in the total number of Dollars, rather than the individual converted amounts, we could have obtained this result directly, using the derived function, plus-reduction:

```

      +/1532 12288 23881 × 1.23
46372.23

```

Similarly, in the second example, to find the number of instances in which the left argument is not greater than the right, it is only necessary to write:

```

      +/2 3 5 7 ≤ 4 3 2 1
2

```

As it happens, since both arguments in this case are vectors, this form is interchangeable with the inner product, and the same result could be obtained by writing:

```

      2 3 5 7 +.≤ 4 3 2 1
2

```

Similarly, we may compute the total number of Dollars using *inner product*, by:

```

      1532 12288 23881 +.× 1.23
46372.23

```

Many other examples of inner product, using higher rank arrays, will be found in the reference section of this manual.

## Variations on the Compound Interest Problem

The execution of simultaneous operations with a series of data items may be combined with the execution of several consecutive operations in a single line. To give an example, we shall repeat the compound interest problem we solved above (£1000 over 5 years at 11%).

```
      1000*(1+0.11)*5
1685.058155
```

Results like this are more sensibly displayed with just two decimal digits, and in what follows we will do this in the simplest way, using the primitive format function ( $\Phi$ ):

```
      2Φ1000*(1+0.11)*5
1685.06
```

Let us now see what will be the effect of different interest rates on the resulting capital. Let us try 10, 10.5, and 11%. We just have to replace 0.11 in the formula by the whole series of values we now want:

```
      2Φ1000*(1+0.10 0.105 0.11)*5
1610.51 1647.45 1685.06
```

Let us suppose we now want to maintain the interest rate at 11%, but we want to test the effect of varying the time to 4, 5 and 6 years.

```
      2Φ1000*(1+0.11)*4 5 6
1518.07 1685.06 1870.41
```

We could also keep the interest of 11%, and a time of 5 years, but vary the starting capital:

```
      2Φ1000 1500 2000*(1+0.11)*5
1685.06 2527.59 3370.12
```

Commonly, compound interest is presented in terms of a table of values for a set of interest rates over a number of years, for

a unit amount of capital, say £1000. Such a table is easily constructed in APL using the *outer product*:

```

      2⊖1000×(1+0.10 0.105 0.11)∘.*1 2 3 4 5
1100.00 1210.00 1331.00 1464.10 1610.51
1105.00 1221.02 1349.23 1490.90 1647.45
1110.00 1232.10 1367.63 1518.07 1685.06

```

## Named Data

Up to now, we have operated with numeric data explicitly introduced whenever we want to make use of it. However, many of these data collections take part more than once in our calculations. If the data is a simple number (such as 3) this is not a problem, but it is not easy to remember all the time that the Dollar exchange rate is 1.23. In APL, this problem may be solved by assigning a name to a value, and this can be done by means of the following expression:

```
DOLLAR ← 1.23
```

with the meaning “the name *DOLLAR* is assigned the value 1.23”. From now on, we may use the name *DOLLAR* in any place where we would make use of its value. For instance: let us repeat the Pound Sterling to Dollar conversion we performed above.

```

      1532 12288 23881 × DOLLAR
1884.36 15114.24 29373.63

```

If the exchange rate changes the next day (for instance, going up to 1.29) and we want to repeat the above calculations, we will just have to assign to the name *DOLLAR* the new value of the rate, and repeat the same line, which thus becomes independent of the current exchange rate:

```

      DOLLAR ← 1.29
      1532 12288 23881 × DOLLAR
1976.28 15851.52 30806.49

```

In the above calculation there still exists a repetitive element. It is possible that we are always interested in converting the same amounts to their equivalent values. In this case, why not also assign a name to these data items?

## IBM Internal Use Only

```
PRICES ← 1532 12288 23881
PRICES × DOLLAR
1976.28 15851.52 30806.49
```

If the Dollar rate now changes again (going down to 1.18), the repetition of the calculation becomes trivial.

```
DOLLAR ← 1.18
PRICES × DOLLAR
1807.76 14499.84 28179.58
```

You can see that it is equally easy to assign a name to a single number or to a collection of data items.

A valid APL name should start with a letter (upper case or lower case), delta ( $\Delta$ ) or delta underbar ( $\underline{\Delta}$ ) and may continue with letters, delta ( $\Delta$ ), delta underbar ( $\underline{\Delta}$ ), numbers, the underbar character ( $\_$ ) or the overbar ( $\bar{\_}$ , the same one used in the negative numbers). The number of characters in a valid name should be at most twelve. If longer names are used, the APL system accepts them anyway, but they are truncated to twelve characters. Examples of valid names are:

```
DOLLAR PRICES values
Aa aA X1 X1a
A JOHN_SMITH JOHN¯SMITH
Very_long_name
```

though the last one will be converted by the system into “*Very\_long\_na*”. All the names in the preceding example are different, and may contain different values at the same time. Uppercase letters are distinguished from lower case, therefore *Aa*, *AA*, *aa* and *aA* may be the names of four independent data collections.

Once a name has been assigned a given value, we may want to recall the value after some time has passed. For this purpose, it is sufficient to type the name, ending the line, as always, with the Enter key. For instance, if we have forgotten the last exchange rate for the Dollar we assigned to the *DOLLAR* name, we can ask the system what the value was. We can also do the same with *PRICES*.

*DOLLAR*

1.18

*PRICES*

1532 12288 23881

Finally, we may want to keep, not the original data, but the result of an operation. We should do exactly the same thing, since we can assign to a name the result of any computation, simple or complex.

*values ← PRICES × DOLLAR**values*

1807.76 14499.84 28179.58

In the above example, we have assigned the result of the operation *PRICES × DOLLAR* the name “*values*”. Then we asked for the value of “*values*”, and obtained, of course, the same numbers as when the operation was performed directly.

In APL, the names assigned to values are called “variables”. The reason is that, as we have seen in the case of *DOLLAR*, the actual value associated with the name may vary with time, through successive assignments.

## APL Workspaces

We have just seen how it is possible to give a name to one or several data items if we intend to do many operations with them. But what shall we do to preserve the values from one day to the next? The problem is that all memory contents are lost when the machine is turned off. Thus, if we turn it on later and want to use APL again, we will find that the variables we had been working with before have lost their values, they have no value at all:

```

DOLLAR
VALUE ERROR
DOLLAR
^
PRICES
VALUE ERROR
PRICES
^

```

The new error message tells us that the name pointed to by the caret has not been assigned any value, or has lost it for some reason.

However, once we have assigned values to one or more variables, we can keep them on a diskette to be used later. The set of things that are kept there (data, names, and other objects) is called a *workspace*.

Let us suppose that we want to save the values of the data assigned to the names “*DOLLAR*”, “*PRICES*” and “*values*”. After assigning to these variables their respective values, and before we turn off the machine or get out of APL, we will insert a formatted diskette in the diskette drive and, after closing the door, we will type the following line:

```
)SAVE SAMPLE
```

The light on the diskette unit will turn on for a few seconds. Then the APL system answers with the present time and date and the name of the new workspace (SAMPLE in this case).

*Note:* The drive door should never be opened while the light is on. All the information contained on the diskette might be destroyed.

All APL instructions beginning with a right parenthesis “)” belong to a special group called “APL system commands”, and are associated with workspace management, either on disk or in memory. In our case, the **)SAVE** command, the name assigned to the workspace will be used whenever we want to recover it from the disk. This name may be the same as the name of one of the variables contained in the workspace, and may contain one to eight characters. The first must be an uppercase letter, the others can be chosen from the uppercase letters or the digits (0123456789). Workspace names with lowercase letters are not allowed.

*Note:* The actual file name of the DOS file that APL creates as a result of the **)SAVE** command is padded to eight characters with underbars. However, these underbars should not be entered when referring to the file with APL system

commands. (This also applies to files created by the *)OUT* command).

If any function in the workspace is suspended for any reason, then the *)SAVE* command will fail with a *COMMAND ERROR* message. If this occurs, clear all suspensions with a *)RESET* command and retry the *)SAVE* command.

Let us suppose we now turn off the machine, turn it on again after some time, and wish to go on working with the data we saved in workspace *SAMPLE*. It is now necessary to copy the workspace from the diskette to the memory. This can be done by means of the following command:

*)LOAD SAMPLE*

Before typing this command, the appropriate diskette (the one where we saved the workspace) should be inserted in the Personal Computer disk drive. Once the door is closed, we may execute the above command, and the drive light will turn on. After some seconds, the light will turn off again, and APL will write the time and date when the workspace was last saved on the diskette. We now have access to the data we wish to use:

```

      DOLLAR
1.18  PRICES
1532 12288 23881

```

There exists one command that gives us the list of all the variables contained in the workspace now active in memory. This command is invoked in the following way:

```

      )VARS
DOLLAR PRICES values

```

We can use it whenever we want to be reminded of the names of all the variables in our active workspace.

Recall that, if some variable values are modified and you want to keep their new values, the workspace must be saved again before the machine is turned off or APL is exited, since otherwise the changes will be lost, and the next time we

execute **)LOAD SAMPLE** we will find the old values, the last to be saved in the diskette.

To save the active workspace on the diskette again, we don't have to give its name in the **)SAVE** command, since the system remembers it. It is thus enough to type:

**)SAVE**

which will have the same effect as **)SAVE SAMPLE**. The new values of the variables will replace those on the disk, which will be lost.

Finally, it is possible to create a totally new unnamed workspace to introduce new data collections that we want to keep separated from any others we may have. The following command can be executed for this purpose:

**)CLEAR**

and the system answers with the message "**CLEAR WS**". If we now ask for the names of the variables in the active workspace, we will find there are none:

**)VARS**

We can then create the new variables we require. Some of these may have the same names as others contained in a different workspace, but the system differentiates between them, since at a given time only the variables in the active workspace are accessible.

*Note:* Be careful not to issue a **)LOAD** or a **)CLEAR** command if the current workspace has been changed since the last time it was **)SAVED**, unless it is desired to discard those changes.

We will later see how to preserve a sequence of operations as an APL function, or program, so that the same operations can be performed repeatedly on different data without the necessity of entering the operations from the keyboard each time. Once defined, such functions are kept, like named data collections, in a saved workspace, and are therefore available whenever that workspace is loaded.



## Other Functions and Operators

Many more functions exist in APL than can be touched upon in this brief chapter. In particular, no examples are given here of how to work with character data. We shall merely say that a character array, such as one containing the letters of an English sentence, can be manipulated by all the non-arithmetic functions of APL which were illustrated above for numeric arrays. (In fact, the histogram functions given below require only minimal changes if their input vectors contain characters instead of numbers).

The remaining sections of this chapter are of a less elementary nature than what has gone before. They cover more advanced topics, and are more concerned with conveying the “style” of APL than merely the “facts”. They are intended for users of APL who have taken the first steps, and feel comfortable in writing simple code.

As with any language, one of the best ways to gain familiarity with APL is to read the code of those well versed in it. The defined functions in the supplied workspaces are a good place to begin, as are the pieces of sample code given throughout this book.

## Input/Output

The simplest form of APL input and output is that used in the calculator mode described above. You enter what you wish from the keyboard, and APL presents the results, if any, on the display device. To get input to, or output from, a function, APL provides the *quote-quad* ( $\square$ ) and the *quad* ( $\square$ ). With them, a function can output numeric or character information to the user at the display in an extremely simple manner. “Quote-quad” has a shape mnemonic of its purpose: as a window to pass character data between the display and an APL function, without modification. “Quad” also passes

## IBM Internal Use Only

information, but it evaluates its input, which may be any valid APL expression.

Syntactically, the quad and quote quad are used like any ordinary APL variable. For example, we can write a simple APL statement:

$$Y \leftarrow X$$

Then we can substitute quad or quote-quad for either of the variables:

$$\begin{aligned} Y &\leftarrow \square \\ \square &\leftarrow X \end{aligned}$$

These lines mean “read a character string from the display, and store it in variable *Y*”, and “show the contents of variable *X* to the user”. If special formatting is desired for input or output, the whole power of APL is available.

A more realistic example would be:

$$\begin{aligned} \square &\leftarrow 'ENTER YOUR NAME, PLEASE' \\ N &\leftarrow \square \end{aligned}$$

This presents an invitation on the display, then reads in the line typed by the user and places it into variable *N*.

If an APL expression is not assigned to a variable at the left hand end of a line, then it is automatically assigned  $\square$ , so that its value is displayed.

## Shared Variables

The primitive concept underlying all communication with APL is the idea of a *shared variable*; that is, a variable accessible by two parties, each of whom may both set and reference its value.

In simplest terms, a shared variable is a conduit, a communications link, between an APL workspace and a second party. It looks and acts syntactically like an ordinary variable. In this APL system the second party, at the other

end of the link, is always an *auxiliary processor*, a program that runs outside of APL and logically connects to some other subsystem on your machine. In versions of APL which allow more than one simultaneous user, the second party can be another APL workspace, and in all APL systems there are permanently shared variables, called *system variables*, which are shared with the underlying APL processor itself.

Quote-quad (⌈) may be considered a permanently shared variable, and its auxiliary processor is the user at the display. Its half-brother, quad (□), is not a pure shared variable, since as explained above, it modifies its input.

A good analogy to the shared variable is a telephone connection. We may dial any of several telephone numbers in order to converse with a selected party at the other end of the line. Similarly, we may “share a variable” with any of several auxiliary processors (each identified by a unique number). As with the telephone, the process of calling is the same for any party, but after the link has been opened, we use different signals to “converse” with each party.

## A Simple Example

Suppose we wish to print one line of text. We need do only the following:

```

1      80 □SVO 'XYZ'
      □SVO 'XYZ'
2      XYZ←'This is a line of output.',□TC[2]
```

The first statement is called the *shared variable offer*. It dedicates the otherwise ordinary APL variable *XYZ* to carry messages to the printer. The second statement (which may optionally also have “80” to the left of the □SVO) confirms the sharing. If it did not return 2, meaning two way sharing has been established, then communication with AP80 would not be possible. (Think of the busy signal on a telephone line). The third statement passes a string to AP80, which passes it on to the printer. Since it ends with the new-line

character (written as `TC[2]`), the line will be printed immediately.

To send further lines, we do not need to re-open communications by sharing the variable again. Continuing from above,

```
XYZ←'This begins the 2nd line,'  
XYZ←' and this ends it.',TC[2]
```

We should not expect to receive the same values as we sent out, any more than we would expect the party at the other end of the telephone line merely to echo back what we say. If we look into variable `XYZ` to see what it contains, as by

```
TC←XYZ
```

we will see 0 if AP80 has accepted the text strings for processing, or some error indication if it has not.

Variable `XYZ` will continue to pass information between the APL function and the printer until it is explicitly retracted (by "`SVR`") or erased (by "`)ERASE`" or "`EX`"), or a new current workspace is loaded, or the user signs off from the current APL session.

## **Shared and Other Variables**

Syntactically, using a shared variable is just like using an ordinary variable: you put values into it (which may be computed by an APL expression), and you take values from it and put them somewhere else. Once set, the value of a variable does not change by itself, but will be changed by the party that shares the variable.

Here is a comparison of four flavours of variables in APL, showing the naming and sharing initialisation conventions:

| <u>Type</u> | <u>Form of name</u>                                 | <u>Shared with</u> | <u>Initial-isation</u>                  | <u>Example</u>   |
|-------------|---|--------------------|---|--|
| Normal var  | Var name  | None               | None                                    | <i>DOLLAR</i>  |
| Shared var  | Var name  | APnnn              | nnn <input type="checkbox"/> <i>SVO</i> | <i>XYZ</i>   |
| I/O var     | <input type="checkbox"/> , <input type="checkbox"/> | User               | Permanent                               | <input type="checkbox"/> , <input type="checkbox"/>      |
| System var  | <input type="checkbox"/> xx                         | APL                | Permanent                               | <input type="checkbox"/> WA, <input type="checkbox"/> TS |

Most shared variables communicate with an auxiliary processor, which is a program that performs a task outside the APL workspace. A major purpose of many auxiliary processors is to convert data between the workspace format and the outside format, much as a telephone modem converts sound to electrical impulses, and vice versa. They are typically written in assembler language. For example, AP80 converts the APL form of character strings into the printer's form, which may involve the transmission of dot patterns that define the APL character shapes, and also generates the necessary printer instructions to carry out the process of printing.

Chapter 13, "How to Build an Auxiliary Processor" of this manual is available for those who would like to write their own auxiliary processor. Most users will be satisfied with using the existing auxiliary processors, as described in Chapter 12, "Auxiliary Processors". Even easier to use are the APL functions in the supplied workspaces, as described in Chapter 11, "Application Workspaces". For example, function *PRINT* in the workspace of the same name uses AP80.

## Debugging

Finding errors in user-written programs is made easier by several built-in features of APL: quad output, tracing, stopping and conditional execution.

## Quad output

The first is the simplest: to insert the clause “ $\square\leftarrow$ ” at various points in an APL statement in order to display the calculated value at that point. For example, suppose a function looks like this (displaying it with the built-in editor command  $\nabla$ *FUNC* $[\square]\nabla$ ):

```
[0]  Z←X FUNC Y
[1]  Z←X+Y
[2]  Z←Z*2+(X-Y)*2
```

This function is intended to add the squares of the sum and the difference of two inputs, but for some unknown reason, it produces the wrong answer sometimes:

```
      3 FUNC 3
36
      3 FUNC 2
125
```

The latter number is wrong, the former right.

With a little judicious editing, we change the function to:

```
[0]  Z←X FUNC Y
[1]   $\square\leftarrow Z\leftarrow(\square\leftarrow X)+\square\leftarrow Y$ 
[2]  Z←Z* $\square\leftarrow 2+\square\leftarrow(X-Y)*2$ 
```

Note that parentheses were placed around *X* in line 1 so as not to display the sum *X+Y* twice.

```
      3 FUNC 2
2
3
5
1
3
125
```

We see that the error is in line 2: 2 is added to the square of the difference before it is used as an exponent. (Recall that APL obeys a strict right to left sequence of execution unless

modified by parentheses). The error may be corrected by parenthesising  $Z*2$  in line 2.

Once the error is found, the function must be re-edited to remove the  $\square\leftarrow$  clauses.

## Tracing

An easier to use, though slightly less powerful, technique is to use APL's built-in tracing feature. Any desired lines in any user-written functions can be traced; that is, the result of those lines can be displayed, identified by the function name and line number. For example,

```

      TΔFUNC←1 2
      3 FUNC 2
FUNC[1]
5
FUNC[2]
125
125

```

Note that 125 is shown twice, once from tracing line 2, and then again as the result of the function call.

If an APL statement has several operations in it, then potentially interesting partial results in the middle of the line will not be displayed by tracing. This will not happen if the lines are written in a good APL style: they should not be overly long and should not be the artificial catenation of several otherwise independent calculations.

The line numbers to be traced can be changed at any time (even from within a user function) by giving a new value to the trace vector. For example, to turn off the tracing of all lines in a function, do

```
TΔFUNC←1 0
```

## Stopping

Very similar to the tracing of lines is stopping before designated lines. For example,

```

          SΔFUNC←1 2
          3  FUNC 2
FUNC[1]

```

Execution has stopped just before line 1 of function *FUNC* would have begun. The user is now free to interrogate (or change) the values of any variables, such as

```

          X
3         Y
2         Z
VALUE   ERROR
          Z
          ^

```

Execution is resumed by entering a right arrow followed by the line number to be executed. To resume at the next line, it is common to keep the following APL statement ready for use on an F-key:

```
→□LC
```

meaning “branch to the first line number kept in the system variable □LC”.

Stopping is frequently combined with tracing. Like the trace vector, the stop vector may be reset (or set to null) at any time.

## Alternate execution

Often, within a user-written function, it may be possible to anticipate possible errors and take preventive action. For example, when dividing by a variable, if it should happen to be zero, then an error will occur, and execution will halt.



```

      X←0
      Y←3÷X
DOMAIN ERROR
      Y←3÷X
      ^

```

Such an error halt can be disconcerting, especially to a user of another person's program. A simple way of checking for and correcting any error detected by APL is to use the alternate execution function,  $\square EA$ . Its two arguments are both character strings holding APL statements. The right hand one is executed first; if no error occurs, then the left hand string is not used. But if the right hand statement fails for any reason, then the left hand statement is executed in its place. Rewriting the above statement:

```

      'Y←999'  $\square EA$  'Y←3÷X'
      Y
999

```

Often, the left hand statement contains a branch expression, which transfers control in the case of an error. A useful technique to avoid error messages from a user written function is to invoke it in the right hand statement, for then  $\square EA$  will trigger the left hand statement if *any* error occurs during the execution of that entire function.

A useful auxiliary function is  $\Delta ET$  supplied in the UTIL workspace, which gives an indication of the cause of the most recent error.

## Speeding Up APL Code

It is often the case that a slow running APL program can be speeded up by judicious modification or rewriting. Because it is important in practice, and because it offers a good way to introduce much material important to the "style" of APL, we give here a section on what to look for in such rewriting. This section will be most helpful to someone who has used APL for a while, and can therefore be skipped or skimmed on first

reading. First, we will give some helpful maxims, then we will work through an example, and lastly we will write a simple timing tool function.

## **Use Timing Tools to Find Hot Spots**

In many programs, an adaptation of Pareto's Law on the distribution of wealth holds true: that, roughly speaking, 20% of the program takes 80% of the time. The simple timing tool developed below can assist in finding that time-consuming portion.

For example, suppose it turns out on testing that one function, *BIGGEST*, takes more than half of the total running time. If it could be rewritten to be twice as fast, then the overall program would run twenty five percent faster. It is often the case that only one particular line in a function is responsible for the largest part of the time spent in computation, and the timing methods to be described can identify such lines as candidates for revision.

## **Eliminate Explicit Loops**

Explicitly written loops often increment an integer variable, and then branch back for another iteration so long as that variable is less than some limit. Such loops are to be avoided if possible, since the interpreter will read and interpret each line within the loop on each iteration, lowering the ratio of useful computation to mere interpretation. The use of APL vector and array operations can often eliminate explicit loops, and, when they can be used, will usually result in clearer code.

For example, to sum the elements of vector *V* into scalar *S*, one could write:

*Note:* 1-origin indexing is assumed here and below.

```

[0]  S←SUMUP V;I
[1]  I←S←0
[2]  L:→((ρV)<I←I+1)/0
[3]  S←S+V[I]
[4]  →L

```

However, it is both faster and clearer to use the APL reduction operator and write:

$$S \leftarrow +/V$$

Of course, a loop must still be performed through the elements of  $V$ ; but this “implicit” loop will be in the machine code which implements “plus reduction”, and it will run very fast. The explicitly written loop above requires probably the same time for arithmetic summation, but additional time for reinterpretation of the APL code, which must be done on each pass through the loop.

Other APL primitives that can be used to replace explicit loops are the scan operator (represented by backslash (\)), the outer and inner products, and vector subscripts. In addition, many of the APL primitive functions automatically sweep through the elements of array arguments.

It is important to realise that it is not necessary to rewrite all inefficient code in a program, but only those segments of code which use the most time. For example, for casual, experimental computation, an explicit loop is not necessarily bad, even though it may not be the clearest expression of an algorithm in APL. It *is* bad, however, if it consumes considerably more time than equivalent loop-free code would, in a program that is to be used frequently, and where performance is important.

Except for user unfamiliarity with APL functions, the only reasons for using explicit loops are that the algorithm is inherently iterative (as are many techniques for solving differential equations) or that too much space would be consumed by the use of APL array operators in the formation of intermediate results. In both cases, nevertheless, one should strive on each iteration of a loop to process as much data as

possible, e.g. one entire plane of a three-dimensional array per iteration.

## Avoid Waxing or Waning Variables

Arrays or vectors that grow or shrink in size are one of the most convenient features of APL. But if the arrays involved are large, their use may consume a great deal of time because it forces the APL interpreter to constantly reallocate storage for them. If the size of a variable is known in advance, it is faster to reserve space for it initially with zeroes or blanks, and later place data within it by explicit subscripting.

For example, instead of appending a new element  $E$  to a pre-existing vector  $V$  by:

$$V \leftarrow 10$$

$$\dot{V} \leftarrow \dot{V}, E$$

it is faster to use subscripting:

$$V \leftarrow N \rho 0$$

$$I \leftarrow 1$$

$$\dot{V}[I] \leftarrow E$$

$$I \leftarrow I + 1$$

In this example,  $I$  may not exceed  $N$ , the initial size assumed for  $V$ . This case can be tested for by the user, and should it occur, vector  $V$  could have another batch of zeroes appended to the end. The point is to avoid too many *automatic* increases in the storage size allocated to  $V$ .

Moving elements into many locations in such a vector may not need an explicit loop, since a vector subscript can be used, storing or retrieving a large amount of data at once. For example, to insert three values at once into  $V$ :

$$V[1 \ 2 \ 3] \leftarrow E, F, G$$

## Avoid Large Outer Products

The outer product (written  $\circ .f$  where  $f$  is a built-in function) produces a two dimensional table from its two arguments, showing the result of every element in the left hand argument combined with every element in the right hand one. This is the equivalent of a double loop when written out with explicit subscripting. Compared to the explicit loops, much running time is generally saved.

However, if the arguments are lengthy, then the outer product table can be quite large. This can lead to **WORKSPACE FULL** conditions when the table itself is only an intermediate result and the final result will not take up nearly as much space. Furthermore, it can cause slow performance because managing the storage for a large array may cause time consuming rearrangements and movement of APL objects in the workspace.

The worked out example of a histogram calculation below shows one way that large outer product tables can be avoided by the use of an alternate algorithm.

## Avoid Tiny Functions

The choice of whether to call a separate function to perform some calculation or incorporate that same code in-line everywhere it is needed is an interesting one. For clarity, one should strive to encapsulate as much code as possible in single purpose well-named independent functions (often called *tools*). There is some overhead involved in calling such functions, however, and the clarity of the code may be diminished by the need to remember the meaning of additional names. A rule of thumb for maximising performance is that if the independent function is doing anything more than some scalar arithmetic, then it should not be coded in-line, since the calling overhead will be minor compared to the rest of the calculation.

## An Example - Computing a Histogram

Let us now solve the following problem: given a vector *INPUT*, containing, for example the values, 7 7 5 2 3 4 5 3 5 7 2 7, count how many of each different value occur. The histogram (bar chart) will be represented by two vectors: vector *ITEMS*, the different items found: 2 3 4 5 7, and vector *COUNTS*, the number of occurrences: 2 2 1 3 4. Three different methods will now be given to compute the histogram. Respectively, they will conserve space, time, and both space and time. With minor changes, as noted, the three functions presented will work for character instead of numeric input vectors.

These examples assume some familiarity with APL, and so may be skipped or skimmed on first reading.

### Computing the Histogram with an Explicit Loop

The first method is to loop through the data vector, one number at a time. Check in the *ITEMS* vector if this item has occurred already. If it has, add 1 to the appropriate count in the *COUNTS* vector. If it hasn't, append the item to the *ITEMS* vector, and 1 to the *COUNTS* vector. This is the classic technique for languages that can perform computation only on scalars.

Here is the code (all examples use 1-origin indexing):

```

[0]  RESULT←HIST1 INPUT;COUNTS;ITEMS;JC;JD
[1]  A Compute histogram of vector INPUT
[2]  A Result is a 2 row matrix
[3]  A First row is unique items
[4]  A Second row is count of each item
[5]  A Space is conserved by explicit loop
[6]  COUNTS←ITEMS←10
[7]  JD←1
[8]  LOOP:→(JD>ρINPUT)/OUT
[9]  A See if next item appears in ITEMS
[10]  JC←ITEMS1INPUT[JD]
[11]  →(JC≤ρITEMS)/HAVE
[12]  A A new item has been found in INPUT
[13]  ITEMS←ITEMS,INPUT[JD]
[14]  COUNTS←COUNTS,1
[15]  →END
[16]  A Increment count for existing item
[17]  HAVE:COUNTS[JC]←COUNTS[JC]+1
[18]  END:JD←JD+1
[19]  →LOOP
[20]  A Combine ITEMS and COUNTS as a matrix
[21]  RESULT←ITEMS,[0.5]COUNTS

```

*Note:* If *INPUT* contains a character vector, we can change the last line to:

```
[21]  RESULT←(⊠AV1ITEMS),[0.5]COUNTS
```

A few points about style in writing defined functions should be noted.

- Variables local to this function are listed in the header line, to avoid cluttering up the workspace.
- Vectors *ITEMS* and *COUNTS* increase in size dynamically, since we do not know initially how big they will become.
- The loop is tested for termination *before* using the index variable *JD*.
- The variable names are long enough to be mnemonic. In general, variables used only briefly for an unimportant purpose should have short names, while variables used in

many places or for more important purposes should have longer names.

Lastly, a reasonable number of comments have been added. It is an excellent rule that APL defined functions should *always* have at least one comment, which states the purpose of that function.

Running time for this method increases linearly with the number of data items in *INPUT*, as does storage. However, the running time will be relatively large, since the explicit loops require repetitive interpretation of the code.

## Computing the Histogram with an Outer Product

We give the APL function first, and the explanation will follow:

```
[0]  RESULT←HIST2 INPUT;ITEMS;COUNTS;
      ALLITEMS;ITEMAT;ALLCOUNTS
[1]  A Compute histogram of vector INPUT
[2]  A Result is a 2 row matrix
[3]  A First row is unique items
[4]  A Second row is count of each item
[5]  A Time is conserved by outer product
[6]  ALLITEMS←~1+(⊆/INPUT)+11+(⊆/INPUT)-⊆/
      INPUT
[7]  A Compare all possible items with INPUT
[8]  ITEMAT←ALLITEMS∘.=INPUT
[9]  A Compress out duplicate items
[10] ALLCOUNTS←+/ITEMAT
[11] ITEMS←(ALLCOUNTS≠0)/ALLITEMS
[12] COUNTS←(ALLCOUNTS≠0)/ALLCOUNTS
[13] A Combine ITEMS and COUNTS as a matrix
[14] RESULT←ITEMS,[0.5]COUNTS
```

*Note:* If *INPUT* contains a character or non-integer vector, change line 6 to:

```
[6]  ALLITEMS←INPUT
```



If *INPUT* contains a character vector, we can change the last line to:

```
[14] RESULT←(⊖AV1ITEMS),[0.5]COUNTS
```

In this method, the histogram is computed by means of several intermediate arrays. First, we define vector *ALLITEMS* to contain all integers from the minimum item in *INPUT* up to the maximum item. For the example *INPUT* = 7 7 5 2 3 4 5 3 5 7 2 7, *ALLITEMS* will be 2 3 4 5 6 7. Next, compute a boolean matrix *ITEMAT* with the outer product of equality, comparing each scalar item in vector *ALLITEMS* with each item in vector *INPUT*.

For the given *INPUT*, we find *ITEMAT* to be

```
0 0 0 1 0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 1 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0
0 0 1 0 0 0 1 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0
1 1 0 0 0 0 0 0 0 1 0 1
```

For any row *J* and column *K*, item *ITEMAT*[*J*;*K*] shows the equality value (i.e. 0 or 1) of item *ALLITEMS*[*J*] with item *INPUT*[*K*].

Next, define vector *ALLCOUNTS* as the sum across each row of *ITEMAT*: We compute *ALLCOUNTS* to be 2 2 1 3 0 4. In fact, *ALLCOUNTS* holds the counts for each number in *ALLITEMS*. Lastly, compute the vector *ITEMS* by selecting from *ALLITEMS* all items with non-zero counts. As expected, *ITEMS* is 2 3 4 5 7. Compute the vector *COUNTS* by selecting from *ALLCOUNTS* all items with non-zero counts. As expected, *COUNTS* is 2 2 1 3 4.

Running time for this method increases linearly with the number of data and linearly with the range of data, so that if the data items are evenly distributed, running time could increase quadratically with the number of items. Storage required is linear with the number of items. But in fact, if the range from the smallest to the largest item in *INPUT* is large, then vector *ALLITEMS* could contain many possible items that do not actually occur in *INPUT*. This would push up

both the running time and the temporary storage needed for matrix *ITEMAT*. When the number of items is not large, however, this function will run faster than the previous method, and will not overrun the workspace available.

## Computing the Histogram with Grade Up

Again, we give the APL function first, and explain it later.

```
[0] RESULT←HIST3 INPUT;ITEMS;COUNTS;ORDINP;
    FIRST;ITEMS;FIRSTSTAT;COUNTS
[1]  A Compute histogram of vector INPUT
[2]  A Result is a 2 row matrix
[3]  A First row is unique items
[4]  A Second row is count of each item
[5]  A Both time and space are conserved
[6]  A Order the input vector
[7]  ORDINP←INPUT[ΔINPUT]
[8]  A Find first occurrence of each item
[9]  FIRST←1,(1↓ORDINP)≠1↓ORDINP
[10] A Get item list and item count
[11] ITEMS←FIRST/ORDINP
[12] FIRSTSTAT←FIRST/∖ρORDINP
[13] COUNTS←((1↓FIRSTSTAT),1+ρORDINP)-FIRSTSTAT
[14] A Combine ITEMS and COUNTS as a matrix
[15] RESULT←ITEMS,[0.5]COUNTS
```

*Note:* If *INPUT* contains a character vector, we can change line 7 and the last line to:

```
[7]  ORDINP←INPUT[□AVΔINPUT]
[15] RESULT←(□AV∖ITEMS),[0.5]COUNTS
```

First, we order the data vector and call it *ORDINP*. For our example we find that *ORDINP* is 2 2 3 3 4 5 5 5 7 7. Compare each item in *ORDINP* with its left hand neighbour, and note which are different. We find *FIRST* to be 1 0 1 0 1 1 0 0 1 0 0 0. Select out the different items, and call it vector *ITEMS*. As expected, *ITEMS* is 2 3 4 5 7. Find *FIRSTSTAT*, the locations of these items, by compressing against the list of integers from 1 through to the length on *INPUT*. For 1 origin indexing, they are at 1 3 5 6 9. Subtract each location from its right-hand neighbour to

generate the counts, and store in *COUNTS*. As expected, *COUNTS* is 2 2 1 3 4, where the last difference is from 1 plus the length of *INPUT*.

The grade-up function ( $\Delta$ ) runs in a time more than linearly proportional to, but less than quadratically proportional to, the number of items in its argument. So this method runs in near-linear time, using a linear amount of storage. It uses the least amounts of both time and space of the three techniques presented above.

## How to Write a Timing Function

This section will discuss the construction of a simple APL function useful as a tool in timing individual statements. Along the way, features of APL useful for both timing and function editing will be demonstrated, as well as considerations of environmentally independent APL coding.

This section may be skipped or skimmed on first reading as it assumes some familiarity with APL.

### A Simple Timing Program

The APL system includes a system variable  $\square AI$  (Account Information) useful for timing. The second component of this vector is the elapsed computer execution time in milliseconds. Hence, to time an APL statement, one need only record  $\square AI$ , execute the statement, and then subtract the saved  $\square AI$  from the current one:

```
Z← $\square AI$ 
A←(11000)*3
( $\square AI$ -Z)[2]
```

1950

We now write a timing function. It requires the user to pass the APL statement to be timed as a character string; then this

A character string can be executed with the execute primitive function  $\Phi$ :

```
[0]  T←TIMEO STMT
[1]  A Time a single APL statement
[2]  T←⊖AI[2]
[3]   $\Phi$ STMT
[4]  T←⊖AI[2]-T
```

A sample call is:

```
TIMEO 'A←(11000)*3'
1970
```

## Problems

Since this function will be copied from a library workspace into the user's workspace, it must be as independent as possible of the user's workspace. Three compatibility problems can be seen.

1. The function as written assumes an index origin (system variable  $\ominus IO$ ) of 1, but the user may have 0 or 1. In fact, it is even possible that  $\ominus IO$  has been localised inside a user function, and has not yet been assigned a value at the moment that it calls the timing function. In this case, no subscript calculations can be performed at all.
2. If the APL statement alters the variables  $T$  or  $STMT$ , the function will fail in unpredictable ways. E.g.

```
TIMEO 'T←7'
40946
```

3. Because the function is itself written in APL, it takes a certain (small) time to run. Also, the interpretation of the user's statement string can take a variable amount of time.
4. By definition of the execute function, its argument string may not be an APL system command or an invocation of the function editor, but only an arithmetic statement.

## An Improved Timing Function

The three problems given can be solved as follows:

1. One might set  $\square IO$  to value 1 on entry and restore the old value on exit from the timing function. (This could be done most easily by making it a local variable, by appending “; $\square IO$ ” to the header line, and adding a new line “ $\square IO \leftarrow 1$ ”). But the APL statement might need the outside value of  $\square IO$ , so this is not a good solution. It is better to use the value of  $\square IO$  inherited from the calling environment; that is, one would write

```
[2] T←AI[ $\square IO$ +1]
```

But because of the rare but troublesome possibility of no value at all for  $\square IO$ , the best solution is to avoid subscripting entirely, and use the take and drop functions as shown below.

2. The problem of conflicting variable names can be solved for most user workspaces most of the time by choosing names in our function which are unlikely to be used in the user's statement.
3. This “Heisenberg effect” of the measuring instrument changing the measured result can be partially compensated for if we execute *TIMEO* ' ' a few times and take the average execution time of *TIMEO* on null statements as the base time to be subtracted out.
4. APL system commands can be executed from inside a user-written function by means of auxiliary processor 101. For the purposes of simplicity, we shall simply not permit them here.

The rewritten function looks like this:

## IBM Internal Use Only

```
[0] T_←TIME1 STMT_  
[1] A Determine time an APL statement uses.  
[2] A Statements must not begin with ')' or  
[3] A with '∇'  
[4] T_←1↑1↓∅AI  
[5] ∅STMT_  
[6] T_←-15+(1↑1↓∅AI)-T_
```

Note that *TIME1* is independent of  $\square IO$  (the index origin) completely. Also the function has been well commented.

Let us check that the nullification constant is correct:

```
      TIME1 ' '  
-15  TIME1 ' '  
45   TIME1 ' '  
-15  TIME1 ' '
```

The constant <sup>-</sup>15 was chosen so that over a number of trials timing a null statement on an IBM PC computer, the average time appeared to be 0.

Test this function on a typical APL statement:

```
      TIME1 'A←(11000)*3'  
1875  TIME1 'A←(11000)*3'  
2175  TIME1 'A←(11000)*3'  
1905
```

We see that the time will vary somewhat dependent on internal activities of the APL system. If accurate timings are desired, invoke the timing function in a loop and take the average.

Lastly, let us time the histogram functions given in the previous section (the numbers shown are actual timings made on a PC):

```

I←?100ρ40
TIME1 'H1←HIST1 I'
6415
TIME1 'H2←HIST2 I'
5475
TIME1 'H3←HIST3 I'
375
  ^/^/H2=H3
1
  H1←H1[;ΔH1[1;]]
  ^/^/H2=H1
1

```

The input vector is of length 100 with items chosen randomly from the integers between 1 and 40. It is checked that output matrix  $H2$  is identical to  $H3$ . Since the items of  $H1$  are not in increasing order, but those of the other two output matrices are, it is necessary to apply grade-up to  $H1$  before finding that it, too, is identical to the others.

## An Alternative Solution

*TIME1* still has a major flaw for timing simple expressions (like  $A←110$ ) because the result of  $\square AI$  is imprecise for short times (for instance, due to intermittent workspace management). A suitable solution to this problem is to execute the statement to be timed several times. The following function demonstrates a technique to allow a number of iterations to be specified, as well as the statement to be timed. Its main new idea is to build a new function (named *FUN\_*) inside the timing function, and then discard it upon exit. (Discarding is done automatically by making the name *FUN\_* a local name in the header line). Note the use of system function  $\square FX$  (FiX) to create the new local function from a character matrix.

```

[0]  T_←N_ TIME2 STMT_;FUN_
[1]  A TIME2 function to average execution
[2]  A times over several iterations.
[3]  A N_ ↔ Number of iterations
[4]  A STMT_ ↔ Statement to time
[5]  Ⓞ(0=⊠NC 'N_')/'N_←1' A Default to one
[6]  STMT_←(21⌈ρ,STMT_)↑STMT_
[7]  T_←((1,ρSTMT_)ρ(ρSTMT_)↑'T_←FUN_'),[
    ⊠IO](ρSTMT_)↑'T_←⊠AI'
[8]  T_←T_,[⊠IO] (N_,ρSTMT_)ρSTMT_
[9]  T_←⊠FX T_←T_,[⊠IO] (ρSTMT_)↑'T_←(⊠AI-
    T_)[⊠IO+1]÷N_'
[10] T_←FUN_ A Execute built function
    
```

Examples:

```

0      1 TIME2 'A←110'
50     1 TIME2 'A←110'
5      10 TIME2 'A←110'
11     5 TIME2 'A←110'
4.9    100 TIME2 'A←110'
5      100 TIME2 'A←110'
    
```

Notice that the reliability of the result improves with the number of iterations.

### Concluding Thought

Often, an APL algorithm may be speeded up at the cost of increased working storage, or vice versa. Sometimes, the minimisation of both time and space can only be achieved by expending much mental energy!



## Further Reading

There have been a number of good introductions to APL published, often as a method of computation for a special field, such as statistics.

The following books are all general introductions, and are widely available:

- *APL: An Interactive Approach*, by Leonard Gilman & Allan Rose, 3rd Edition, 1983.
- *APL: The Language and Its Usage*, by Raymond Polivka & Sandra Pakin, 1975.
- *APL\360 Programming and Applications*, by Herbert Hellerman & Ira Smith, 1976.
- *Introduction to APL2*, [by Jon McGrew,] IBM Form No. SH20-9229, 1983.
- *APL - An Introduction*, Independent Study Program, IBM Form No. SR20-7183, 1982.
- *APL Programming Guide*, IBM Form No. G320-6735, 1983.

## Part 2. APL Language

|   |            |
|---|------------|
| <b>Chapter 3. Using APL</b> .....                               | <b>3-1</b> |
| An Example of the Use of APL .....                              | 3-3        |
| An Isolated Calculation .....                                   | 3-3        |
| Storing Functions and Data .....                                | 3-4        |
| Characteristics of APL .....                                    | 3-5        |
| <b>Chapter 4. Fundamentals</b> .....                            | <b>4-1</b> |
| Character Set .....   | 4-6        |
| Spaces .....  | 4-8        |
| Function .....  | 4-8        |
| Order of Execution .....  | 4-9        |
| Data .....  | 4-10       |
| Arrays .....  | 4-10       |
| Constants .....   | 4-12       |
| Workspaces and Libraries .....                                  | 4-13       |
| Names .....   | 4-14       |
| Implementation Limits .....                                     | 4-15       |
| <b>Chapter 5. Primitive Functions and Operators</b> .....       | <b>5-1</b> |
| Scalar Functions .....  | 5-3        |
| Plus, Minus, Times, Divide, and Residue .....                   | 5-7        |
| Conjugate, Negative, Signum, Reciprocal, and<br>Magnitude ..... | 5-8        |
| Boolean and Relational Functions .....                          | 5-9        |
| Minimum and Maximum .....                                       | 5-11       |
| Floor and Ceiling .....   | 5-11       |
| Roll (Random Number Function) .....                             | 5-12       |
| Power, Exponential, General and Natural<br>Logarithm .....      | 5-12       |
| Circular, Hyperbolic, and Pythagorean Functions .....           | 5-13       |
| Factorial and Binomial Functions .....                          | 5-15       |
| Operators .....   | 5-17       |
| Reduction .....   | 5-17       |
| Scan .....  | 5-19       |
| Axis .....  | 5-19       |
| Inner Product .....   | 5-21       |

|   |            |
|---|------------|
| Outer Product .....   | 5-23       |
| Mixed Functions .....   | 5-25       |
| Structural Functions .....                                    | 5-30       |
| Selection Functions .....                                     | 5-37       |
| Selector Generators .....                                     | 5-42       |
| Index Generator and Index Of .....                            | 5-43       |
| Membership .....  | 5-44       |
| Grade Functions .....   | 5-44       |
| Deal .....  | 5-48       |
| Numeric Functions .....                                       | 5-48       |
| Matrix Inverse and Matrix Divide .....                        | 5-48       |
| Decode and Encode .....                                       | 5-51       |
| Data Transformations .....                                    | 5-53       |
| Execute and Format .....                                      | 5-54       |
| Picture Format .....  | 5-59       |
| <b>Chapter 6. System Functions and System Variables .....</b> | <b>6-1</b> |
| System Functions .....  | 6-3        |
| Canonical Representation - $\square CR$ .....                 | 6-5        |
| Delay - $\square DL$ .....                                    | 6-6        |
| Execute Alternate - $\square EA$ .....                        | 6-6        |
| Expunge - $\square EX$ .....                                  | 6-7        |
| Function Establishment - $\square FX$ .....                   | 6-7        |
| Name Classification - $\square NC$ .....                      | 6-8        |
| Name List - $\square NL$ .....                                | 6-9        |
| Peek/Poke - $\square PK$ .....                                | 6-9        |
| Transfer Form - $\square TF$ .....                            | 6-11       |
| System Variables .....  | 6-14       |
| Account Information - $\square AI$ .....                      | 6-16       |
| Atomic Vector - $\square AV$ .....                            | 6-16       |
| Comparison Tolerance - $\square CT$ .....                     | 6-17       |
| Format Control - $\square FC$ .....                           | 6-17       |
| Index Origin - $\square IO$ .....                             | 6-18       |
| Horizontal Tabs - $\square HT$ .....                          | 6-18       |
| Latent Expression - $\square LX$ .....                        | 6-18       |
| Line Counter - $\square LC$ .....                             | 6-19       |
| Printing Precision - $\square PP$ .....                       | 6-19       |
| Printing Width - $\square PW$ .....                           | 6-20       |
| Random Link - $\square RL$ .....                              | 6-20       |
| Terminal Control - $\square TC$ .....                         | 6-20       |
| Terminal Type - $\square TT$ .....                            | 6-20       |
| Time Stamp - $\square TS$ .....                               | 6-20       |

|   |             |
|---|-------------|
| User Load - □UL                                     | 6-20        |
| Workspace Available - □WA                           | 6-20        |
| <b>Chapter 7. Shared Variables</b>                  | <b>7-1</b>  |
| Offers  | 7-6         |
| Access Control                                      | 7-7         |
| Retraction  | 7-11        |
| Inquiries   | 7-12        |
| <b>Chapter 8. Function Definition</b>               | <b>8-1</b>  |
| Canonical Representation and Function Establishment | 8-3         |
| The Function Header                                 | 8-5         |
| Ambi-Valent Functions                               | 8-5         |
| Local and Global Names                              | 8-6         |
| Branching and Statement Numbers                     | 8-7         |
| Labels  | 8-9         |
| Comments  | 8-9         |
| Function Editing - The ∇ Form                       | 8-10        |
| Adding a Statement                                  | 8-10        |
| Inserting or Replacing a Statement                  | 8-11        |
| Replacing the Header                                | 8-11        |
| Deleting a Statement                                | 8-11        |
| Modifying a Statement or Header                     | 8-12        |
| Function Display                                    | 8-12        |
| Leaving the ∇ Form                                  | 8-13        |
| Quitting the ∇ Form                                 | 8-14        |
| <b>Chapter 9. Function Execution</b>                | <b>9-1</b>  |
| Halted Execution                                    | 9-4         |
| State Indicator                                     | 9-4         |
| State Indicator Damage                              | 9-6         |
| Trace Control                                       | 9-6         |
| Stop Control  | 9-7         |
| Locked Functions                                    | 9-8         |
| Recursive Functions                                 | 9-9         |
| Input and Output                                    | 9-10        |
| Evaluated Input                                     | 9-11        |
| Character Input                                     | 9-12        |
| Interrupting Execution during Input                 | 9-12        |
| Normal Output                                       | 9-12        |
| Bare Output   | 9-13        |
| <b>Chapter 10. System Commands</b>                  | <b>10-1</b> |

|   |       |
|---|-------|
| Active Workspace - Action Commands .....                    | 10-9  |
| Active Workspace - Inquiry Commands .....                   | 10-13 |
| Workspace Storage and Retrieval - Action<br>Commands .....  | 10-14 |
| Libraries of Saved Workspaces .....                         | 10-14 |
| Workspace Names .....                                       | 10-14 |
| Workspace Storage and Retrieval - Inquiry<br>Commands ..... | 10-20 |
| Sign-Off .....  | 10-21 |

# Chapter 3. Using APL

|                                    |     |
|------------------------------------|-----|
| An Example of the Use of APL ..... | 3-3 |
| An Isolated Calculation .....      | 3-3 |
| Storing Functions and Data .....   | 3-4 |
| Characteristics of APL .....       | 3-5 |

**Notes:**

APL takes one APL statement at a time, converts it to *machine instructions* (the computer's internal language), executes it, then proceeds to the next line. In contrast to program compilers that convert complete programs to machine language before executing any statements, APL allows you a high degree of interaction with the computer. If something you enter is invalid, you will get quick feedback on the problem before you go any further, which also yields high productivity gains.

## **An Example of the Use of APL**

A statement entered at the keyboard may contain numbers or symbols, such as + - × ÷, or names formed from letters of the alphabet. The numbers and special symbols stand for the primitive objects and functions of APL - primitive in the sense that their meanings are permanently fixed, and therefore understood by the APL system without further definition. A name, however, has no significance until a meaning has been assigned to it.

Names are used for two major categories of objects. There are names for collections of data that is composed of numbers or characters. Such a named collection is called a *variable*. Names may also be used for programs made up of sequences of APL statements. Such programs are called *defined functions*. Once they have been established, names of variables and defined functions can be used in statements by themselves or in conjunction with the primitive functions and objects.

## **An Isolated Calculation**

If the work to be done can be adequately specified simply by typing a statement made up of numbers and symbols, names will not be required; entering the expression to be evaluated causes the result to be displayed. For example, suppose you want to compare the rates of return on money at a fixed interest rate but with different compounding intervals. For



1000 units at 6% compounded annually, quarterly, monthly, or daily for 10 years, the entry and response for the transaction (assuming a printing precision ( $\square PP$ ) equal to 6) would look like this:

$$\square PP \leftarrow 6$$

$$1000 \times (1 + 0.06 \div 1 \ 4 \ 12 \ 365) * 10 \times 1 \ 4 \ 12 \ 365$$

1790.85 1814.02 1819.4 1822.03

(The largest gain is apparently obtained in going from annually to quarterly; after that the differences are relatively insignificant.)

Several characteristic features of APL are illustrated in this example: familiar symbols such as + - \*  $\div$  are used where possible; symbols are introduced where necessary (as the \* for the power function); and a group of numbers can be worked on together.

## Storing Functions and Data

Although many problems can be solved by typing the appropriate numbers and symbols, the greatest benefits of using APL occur when named functions and data are used. Because a single name may refer to a large array of data, using the name is far simpler than typing all of its numbers. Similarly, a defined function, specified by entering its name, may be composed of many individual APL statements that would be burdensome to type again and again.

Once a function has been defined, or data collected under a name, it is usually desirable to retain the significance of the names for some period of time - perhaps for just a few minutes - but more often for much longer, possibly months or years. For this reason APL systems are organised around the idea of a *workspace*, which might be thought of as a notebook in which all the data items needed during some piece of work are recorded together. An APL workspace will thus contain defined functions, data structures, and a state indicator.

## Characteristics of APL

The remaining chapters of this part of the book describe APL in detail, giving the meaning of each symbol and discussing the various features of APL for the IBM Personal Computer. These details should be considered in light of the major characteristics of APL, which may be summarised as follows:

- The primitive objects of the language are arrays (lists, tables, lists of tables, etc.). For example,  $A + B$  is meaningful for any conformable arrays  $A$  and  $B$ , the size of an array ( $\rho A$ ) is a primitive function, and arrays may be indexed by arrays, as in  $A[3\ 1\ 4\ 2]$ .
- The syntax is simple: there are only three statement types (name assignment, branch, or neither), there is no function precedence hierarchy, functions have either one, two, or no arguments, and primitive functions and defined functions (programs) are treated alike.
- The semantic rules are few: the definitions of primitive functions are independent of the representations of data to which they apply, all scalar functions are extended to other arrays in the same way (that is, item-by-item), and primitive functions have no hidden effects (so-called *side-effects*).
- The sequence control is simple: one statement type embraces all types of branches (conditional, unconditional, computed, etc.), and the completion of the execution of any function always returns control to the point of use.
- External communications are established by means of variables, which are shared between APL and other systems or subsystems (such as auxiliary processors). These *shared variables* are treated both syntactically and semantically like other variables. A subclass of shared variables - *system variables* - provides convenient communications between APL programs and their environment.

The usefulness of the primitive functions is vastly expanded by *operators*, which modify their behaviour in a systematic manner. For example, *reduction* (denoted by  $/$ ) modifies a function to apply over all elements of a list, as in  $+/L$  for summation of the items of  $L$ . The remaining operators are *scan* (running totals, running maxima, etc.), the *axis* operator which, for example, allows reduction and scan to be applied over a specified axis (rows or columns) of a table, the *outer product*, which produces tables of values as in  $RATE \circ * YEARS$  for an interest table, and the *inner product*, a simple generalisation of matrix product that is very useful in data processing and other non-mathematical applications.

The number of primitive functions is few enough that each is represented by a single, easily-read and easily-written symbol, yet the set of primitives embraces operations from simple addition to grading (sorting) and formatting. The complete set can be classified as follows:

- Arithmetic:  $+ - \times \div * \otimes \circ | \lfloor \lceil ! \boxplus$
- Boolean and Relational:  $\vee \wedge \approx \neq \sim < \leq = \geq > \neq$
- Selection and Structural:  $/ \setminus \neq \backslash \lceil \rfloor \uparrow \downarrow \rho , \phi \boxtimes \ominus$
- General:  $\epsilon \iota ? \perp \top \Psi \Delta \Phi \bar{\Phi}$

# Chapter 4. Fundamentals

|                                |      |
|--------------------------------|------|
| Character Set .....            | 4-6  |
| Spaces .....                   | 4-8  |
| Function .....                 | 4-8  |
| Order of Execution .....       | 4-9  |
| Data .....                     | 4-10 |
| Arrays .....                   | 4-10 |
| Constants .....                | 4-12 |
| Workspaces and Libraries ..... | 4-13 |
| Names .....                    | 4-14 |
| Implementation Limits .....    | 4-15 |

**Notes:**

A typical statement in APL is of the form:

$AREA \leftarrow 3 \times 4$

The effect of the statement is to assign to the name *AREA*, the value of the expression  $3 \times 4$  to the right of the specification arrow  $\leftarrow$ . The statement may be read informally as “AREA is three times four”.

The statement is the normal unit of execution. Two primitive types occur: the *specification* shown above, and the *branch*, which serves to control the sequence in which the statements in a defined function (see Chapter 8, “Function Definition”) are executed. There is also a third type of statement that may specify the use of a defined function without either a specification or a branch.

A variant of the specification statement produces a display of a result. If the leftmost part of a statement is not a name followed by a specification, the result of the expression is displayed. For example:

```

12      3×4
        PERIMETER←2×(3+4)
14      PERIMETER

```

The result of any part of a statement can be displayed by including the characters  $\square\leftarrow$  at the appropriate point in the statement. Moreover, any number of specification arrows may occur in a statement. For example:

```

12      X←2+□←3×Y←4
        X
14      Y
4

```

Entry of a statement that cannot be executed will cause an *error message* to be displayed, which indicates the nature of the error and the point at which execution stopped. For example:

```

      X←5
      3+(Z×X)
VALUE ERROR
      3+(Z×X)
      ^

```

Following is a list of error messages, with information about the cause and suggested corrective action.

- DEFN                    Misuse of ∇ or □ symbols:
1. Invalid function header.
  2. Use of other than a name alone in reopening a function.
  3. Improper request for a line edit or display.
- DOMAIN                Argument is not valid.
- IMPLICIT        The system variable □-- (for example, □IO) has been set to an inappropriate value, or has been localised and not been assigned a value.
- INDEX                Index value out of range.
- INTERRUPT            Execution interrupted:
1. The input line being typed is ignored. Begin typing again.
  2. Execution was suspended within an APL statement.
- TO RESUME EXECUTION, ENTER  
A BRANCH TO THE STATEMENT  
INTERRUPTED
- LENGTH                Shapes not conformable.

- RANK** Ranks not conformable.
- SI DAMAGE** The state indicator (an internal list of suspended and pendent functions) has been damaged in editing a function or in carrying out an **)ERASE**.
- STACK FULL** Too many nested functions called. Definition of a very large function with **∇**, **□FX**, **□TF** or **)IN**.
- SYMBOL TABLE FULL** Too many names used. This error can be corrected by executing the following sequences of commands:
- )OUT, )CLEAR, )IN**  
or **)OUT, )CLEAR, )SYMBOLS nnn, )IN**  
or **)ERASE, )OUT, )CLEAR, )IN**
- SYNTAX** Invalid syntax; for example, two variables adjoining; function used without an appropriate number of arguments; unmatched parentheses.
- SYSTEM** Fault in internal operation of the system.
- COMPLETE READER'S COMMENT  
FORM AT THE BACK OF THE BOOK  
AND SEND TO IBM.**
- SYSTEM LIMIT** An implementation limit has been reached.
- VALUE** Use of name that does not have a value, or an attempt to use a numeric constant whose magnitude is too large or too small for internal representation.
- ASSIGN A VALUE TO THE VARIABLE,  
DEFINE THE FUNCTION, OR  
CHANGE THE VALUE OF THE  
CONSTANT**



**WORKSPACE FULL** Workspace is filled (perhaps by temporary values produced in evaluating a compound expression, or by values of shared variables).

**CLEAR STATE INDICATOR, ERASE  
NEEDLESS OBJECTS, OR REVISE  
CALCULATIONS TO USE LESS SPACE.**

## Character Set

The characters that may occur in a statement fall into four main classes: alphabetic, numeric, special, and blank. The alphabetic characters comprise the Roman alphabet in uppercase, the same alphabet in lowercase, delta ( $\Delta$ ), and delta underbar ( $\underline{\Delta}$ ). The complete set is shown in Figure 4-1 on page 4-7 with suggested names.

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z | Δ |
| a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z | Δ |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | - |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

|    |             |   |                   |   |                |
|----|-------------|---|-------------------|---|----------------|
| ** | dieresis    | α | alpha             | ∅ | nor            |
| -  | overbar     | ┌ | upstile           | ∅ | nand           |
| <  | less        | └ | downstile         | ∇ | del stile      |
| ≤  | not greater | — | underbar          | Δ | delta stile    |
| =  | equal       | ∇ | del               | ⊙ | circle stile   |
| ≥  | not less    | Δ | delta             | ⊘ | circle slope   |
| >  | greater     | ∘ | null              | ⊖ | circle bar     |
| ≠  | not equal   | ' | quote             | ⊗ | log            |
| ∨  | or          | □ | quad              | ⊥ | I-beam         |
| ∧  | and         | ( | left parenthesis  | ∇ | del tilde      |
| -  | bar         | ) | right parenthesis | ∅ | base null      |
| ÷  | divide      | [ | left bracket      | ∅ | top null       |
| +  | plus        | ] | right bracket     | ∖ | slope bar      |
| ×  | times       | ⊂ | left shoe         | / | slash bar      |
| ?  | query       | ⊃ | right shoe        | ∅ | cap null       |
| ω  | omega       | ∩ | cap               | □ | quote quad     |
| ε  | epsilon     | ∪ | cup               | ! | quote dot      |
| ρ  | rho         | ⊥ | base              | ⊠ | domino         |
| ~  | tilde       | ⊤ | top               |   | stile          |
| ↑  | up arrow    | ; | semicolon         | * | star           |
| ↓  | down arrow  | : | colon             | ι | iota           |
| →  | right arrow | , | comma             | \ | slope          |
| ←  | left arrow  | . | dot               | / | slash          |
| ○  | circle      |   | space             | Δ | delta underbar |

Figure 4-1. APL Character Set

The names suggested are for the symbols themselves and not necessarily for the functions they represent. For example, the downstile (L) represents both the *minimum*, a function of two arguments, and the *floor* (or *integer part*), a function of one argument. In general, most of the special characters (such as +, -, ×, and ÷) are used to denote primitive functions that are assigned fixed meanings, and the alphabetic characters are used to form names that may be assigned and re-assigned significance as variables, defined functions, and other objects.

## Spaces

The blank character is used primarily as a separator. The spaces that one or more blank characters produce are needed to separate names of adjacent defined functions, constants, and variables. For example, if  $F$  is a defined function, then the expression  $3 F 4$  must be entered with the indicated spaces. The exact number of spaces used in succession is not important, and extra spaces may be used freely. Spaces are not required between primitive functions and constants or variables, or between a succession of primitive functions, but they may be used if desired. For example, the expression  $3+4$  may be entered with no spaces.

## Function

The word *function* derives from a word that means to execute or perform. A function executes some action on an array (or arrays), called its *argument(s)*, to produce an array as a result. The result may serve as an argument to another function. For example:

```

      3×4
12
      2+(3×4)
14
      (-6)÷3
-2

```

A function (such as the negation used above) that takes one argument is said to be *monadic*, and a function (such as *times*) that takes two arguments is said to be *dyadic*. All APL functions are either monadic or dyadic or, in the case of defined functions only, *niladic* (taking no argument). The argument of a monadic function always appears to the right of the function. The arguments of a dyadic function appear on each side of the function, and are called the *left argument* and *right argument*. Certain of the special symbols are used to

denote two different functions, one monadic and the other dyadic. For example,  $X-Y$  denotes subtraction of  $Y$  from  $X$  (a dyadic function), and  $-Y$  denotes negation of  $Y$  (a monadic function).

Each of the primitive functions is denoted by a single character or by an *operator* applied to such a character (see Chapter 5, "Primitive Functions and Operators"). For example,  $+$  and  $\times$  are primitive functions as are  $+/$  and  $\times/$  (since  $/$  denotes an operator).

## Order of Execution

Parentheses are used in the usual way to control the order of execution in a statement. Any expression within matching parentheses is evaluated before applying to the result, any function outside the matching pair.

In conventional notation, the order of execution of an unparenthesised sequence of monadic functions may be stated as follows: the (right-hand) argument of any function is the value of the entire expression to the right. For example, *LOG SIN ARCTAN X* means the Log of Sin of Arctan  $X$ . In APL, the same rule applies to dyadic functions as well. Moreover, all functions, both primitive and defined, are treated alike; there is no hierarchy among functions, such as multiplication being done before addition or subtraction.

An equivalent statement of this rule is that an unparenthesised expression is evaluated in order from right to left. For example, the expression  $3 \times 8 \uparrow 3 * | 5 - 7$  is equivalent to  $3 \times (8 \uparrow (3 * (| (5 - 7))))$ . Their result is 27. A consequence of the rule is that the only concrete use of parentheses is to form the left argument of a function. For example,  $(12 \div 3) \times 2$  is 8 and  $12 \div 3 \times 2$  is 2. However, redundant pairs of parentheses can be used to help improve readability. Thus, the expressions  $12 \div 3 \times 2$  and  $12 \div (3 \times 2)$  are evaluated identically, with a result of 2.

# Data

Data used in APL is one of two types - *numeric* or *character*.

Data is produced by: (1) explicit entry at the keyboard, (2) execution of APL functions or operators, and (3) use of shared variables and system functions or variables.

## Arrays

Data is organised in ordered collections called *arrays*. Arrays are characterised by their content (character or numeric), their number of axes or dimensions (*rank*), and the number of elements along each axis (*shape*). All elements of an array must be of the same type - character or numeric. Arrays range from scalars, which are dimensionless, to multi-dimensional arrays of arbitrary rank and shape. These arrays are referred to by the following terms:

- A *scalar* is an array having no dimensions.
- A *vector* is an array having one dimension.
- A *matrix* (or table) is an array having two dimensions.

Arrays having more than two dimensions can also be created.

An *empty array* is an array with one or more of its dimensions equal to 0. Such an array is either character or numeric, but contains no elements.

A vector can be formed by listing its elements as described in the discussion of constants. For example:

```
V←2 3 5 7 11 13 17 19
A←'ABCDEFGH'
```

The elements of a vector may be selected by *indexing*. For example:

**IBM Internal Use Only**

```

      V[3 1 5]
5 2 11 A[8 5 1 4]
HEAD

```

Arrays of more complex structure may be formed with the *reshape* dyadic function denoted by  $\rho$ .

```

      M←2 4ρV          B←2 4ρA
      M              B
  2   3   5   7      ABCD
11  13  17  19      EFGH

```

These results have two dimensions or axes and are called *tables* or *matrices*. A matrix has two axes and is said to be of *rank 2*; a vector has one axis and is of *rank 1*. The left argument 2 4 in the preceding examples specifies the *shape* of the resulting array. Arrays of arbitrary shape and rank may be produced by the same scheme. For example:

```

      T←2 3 4ρ'ABCDEFGH IJKLMNOPQRSTUVWXYZ'
      T
  ABCD
  EFGH
  IJKL

  MNOP
  QRST
  UVWX

```

The shape of an array can be determined by the monadic function denoted by  $\rho$ .

```

8      ρV          2 4      ρM          2 3 4      ρT

```

Elements may be selected from any array (other than a scalar) by indexing in the manner shown for vectors, except that indexes must be given for each axis:

```

      M[2;3]          P          T[2;1;4]
17      M[2 1;2 3 4]      P          T[2;1 2 3;1 2 3 4]
13 17 19      MNOP
  3 5 7      QRST
            UVWX

```

The indexing used in the preceding examples is called *1-origin*, because the first element along each axis is selected by the index 1. One may also use *0-origin* indexing by setting the *index origin* to 0. The index origin is a *system variable* denoted by  $\square IO$  (see Chapter 6, "System Functions and System Variables"). Thus:

|   |   |
|---|---|
| $\begin{array}{r} \square IO \leftarrow 1 \\ V[1 \ 2 \ 3] \\ 2 \ 3 \ 5 \\ G \quad B[2;3] \end{array}$ | $\begin{array}{r} \square IO \leftarrow 0 \\ V[0 \ 1 \ 2] \\ 2 \ 3 \ 5 \\ G \quad B[1;2] \end{array}$ |
|---|---|

All remaining examples assume 1-origin unless otherwise stated.

## Constants

A *constant* is a scalar or vector, either character or numeric, that appears explicitly in an APL statement.

All numbers entered or displayed are in decimal, either in conventional form (including a decimal point if appropriate) or in *scaled form*. The scaled form consists of an integer or decimal fraction called the *multiplier* followed immediately by the symbol *E* then an integer (which must not include a decimal point) called the *scale*. The scale specifies the power of 10 by which the multiplier is to be multiplied. Thus  $1.44E2$  is equivalent to 144.

Negative numbers are represented by an *overbar* ( $\bar{\quad}$ ) immediately preceding the number; for example,  $\bar{1.44}$  and  $\bar{144E}^{-2}$  are equivalent negative numbers. The overbar can be used only as part of a constant and is to be distinguished from the bar that denotes negation, as in  $-X$ .

A *scalar numeric constant* is a number entered by itself. A *vector numeric constant* is entered by listing the component numbers in order, separated by one or more spaces.

A *scalar character constant* may be entered by placing the character between quotation marks; a *vector character constant*

may be entered by listing no characters, or two or more characters, between quotation marks. The system displays such a vector as the sequence of characters, with no enclosing quotes and with no separation of the successive elements. The quote character itself must be entered as a pair of quotes. Thus, the abbreviation of *CANNOT* is entered as '*CAN*' '*T*' and prints as *CAN*' *T*.

## **Workspaces and Libraries**

The common organisational unit in an APL system is the *workspace*. When in use, a workspace is said to be *active*, and is located in main storage. Part of each workspace is set aside to serve the internal workings of the system, and the remainder is used, as required, to store items of information and to hold transient information generated during a computation.

The names of variables (data items) and defined functions (programs) used in calculations always refer to objects known by those names in the active workspace; information about the progress of program execution is maintained in the *state indicator* of the active workspace, and control information affecting the form of output is held within the active workspace.

Inactive workspaces are stored in *libraries*, where they are identified by arbitrary names. They occupy space on disk and cannot be worked with directly. When required, copies of stored workspaces can be made active, or (if stored in an appropriate form) selected information may be transferred from them into an active workspace.

Workspaces and libraries are managed by *system commands*, as described in Chapter 10, "System Commands".



# Names

Names of workspaces, functions, and variables may be formed from any sequence of alphabetic and numeric characters that starts with an alphabetic and contains no blank. (For the purpose of this definition, the overbar ( $\bar{\quad}$ ) and underbar ( $\underline{\quad}$ ) are regarded as numeric characters. That is, they may be used to form a name, but not to start it.)

Some additional restrictions on names exist for APL on the IBM Personal Computer:

- The number of significant characters in the name of an APL object is 12.
- Workspace names are subject to IBM Personal Computer DOS file-naming restrictions, with a maximum length of 8 alphanumeric characters, beginning with an alphabetic character.
- Lowercase letters, delta, delta underbar, overbar and underbar are not allowed as part of a workspace name.

The environment in which APL operations take place is limited by the active workspace. Hence, the same name may be used to designate different objects (that is, functions or variables) in different workspaces, without interference. Also, because workspaces themselves are never the subject of APL operations, but only of system commands, a workspace can have the same name as an object it holds.

## Implementation Limits

The APL interpreter for the IBM Personal Computer has the following implementation limits:

- The maximum value of any dimension of an APL object is 65520.
- The maximum number of elements in a variable is 65520.
- The maximum size of a boolean APL object is 8190 bytes.
- The maximum size of an integer APL object is 131040 bytes.
- The maximum size of a literal APL object is 65520 bytes.
- The maximum size of a floating-point APL object is 524160 bytes.
- The maximum number of lines in a function is 999.
- The maximum size of the symbol table is 32766 bytes.
- The maximum size of the stack is 4096 elements.

An APL workspace consists of two parts:

- The *main workspace*. It occupies a maximum of 64K bytes. It is in this part where APL statements are executed, and where APL objects smaller than 8192 bytes may be created and modified.
- The *elastic workspace*. It occupies all memory that is still available. Its size has no limit other than the physical size of the memory. APL objects that are not actually in use during an execution sequence may be moved to the elastic workspace if the space they occupy in the main workspace is needed for other purposes. APL objects larger than 8192 bytes are created and modified in the elastic workspace.

**Notes:**

# Chapter 5. Primitive Functions and Operators

- Scalar Functions ..... 5-3
  - Plus, Minus, Times, Divide, and Residue ..... 5-7
  - Conjugate, Negative, Signum, Reciprocal, and Magnitude ..... 5-8
  - Boolean and Relational Functions ..... 5-9
  - Minimum and Maximum ..... 5-11
  - Floor and Ceiling ..... 5-11
  - Roll (Random Number Function) ..... 5-12
  - Power, Exponential, General and Natural Logarithm ..... 5-12
  - Circular, Hyperbolic, and Pythagorean Functions ..... 5-13
  - Factorial and Binomial Functions ..... 5-15
- Operators ..... 5-17
  - Reduction ..... 5-17
  - Scan ..... 5-19
  - Axis ..... 5-19
  - Inner Product ..... 5-21
  - Outer Product ..... 5-23
- Mixed Functions ..... 5-25
  - Structural Functions ..... 5-30
  - Selection Functions ..... 5-37
- Selector Generators ..... 5-42
  - Index Generator and Index Of ..... 5-43
  - Membership ..... 5-44
  - Grade Functions ..... 5-44
  - Deal ..... 5-48
- Numeric Functions ..... 5-48
  - Matrix Inverse and Matrix Divide ..... 5-48
  - Decode and Encode ..... 5-51
- Data Transformations ..... 5-53
  - Execute and Format ..... 5-54
  - Picture Format ..... 5-59

**Notes:**

## IBM Internal Use Only

The primitive functions fall into two classes - scalar and mixed. Scalar functions are defined in scalar arguments and are extended to other arrays item-by-item. Mixed functions are defined in arrays of various ranks and may give results that differ from the arguments in both rank and shape. Five primitive operators apply to scalar dyadic functions and to certain mixed functions to produce many new functions.

The definitions of certain functions depend on *system variables* whose names begin with the symbol  $\square$  (as in  $\square IO$  and  $\square CT$ ). These system variables are discussed in more detail in Chapter 6, "System Functions and System Variables".

## Scalar Functions

A monadic scalar function extends to each item of an array argument; the result is an array of the same shape as the argument, and each item of the result is obtained as the monadic function applied to the corresponding item of the argument.

A dyadic scalar function extends similarly to a pair of arguments of the same shape. To be conformable, the arguments must agree in shape, or at least one of them must be a scalar or a one-element array. If one of the arguments has only one item, that item is applied in determining each element of the result. If both arguments have one item but different ranks, the result has the higher rank. For example:

```
      1 2 3×4 5 6
4 10 18
      3+4 5 6
7 8 9
      2 3+4 5 6
LENGTH ERROR
      2 3+4 5 6
      ^
```

Each of the scalar functions is defined for all real numbers with two general exceptions: the five boolean functions are defined only on the numbers 0 and 1, and the functions = and

$\neq$  are defined on characters as well as numbers. Specific exceptions (such as  $4 \neq 0$ ) will be noted where appropriate.

The scalar functions are summarised in Figure 5-1 on page 5-5 with their symbols and brief definitions or examples, which should clarify their use. The remainder of this chapter is devoted to more detailed definitions.

| Monadic form $f B$  |  | Dyadic form $A f B$                     |   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
|---|--|---|---|--------|---------------|-----|---------------|------------|---------|------------------|-----------------|--|------------|------------|-----|-------------|-------------------|-----|---------------|-------------|-----|----------|-------------|-----|----------|-------------|-----|----------|--|--|
| Definition or Example   | Name   | $f$                                     | Name      Definition or Example   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| $+B$ is $B$<br>$-B$ is $0-B$<br>$\times B$ is $(B>0)+B<0$<br>$\div B$ is $1\div B$<br>$ ^{-}3.14$ is $3.14$   | Conjugate<br>Negative<br>Signum<br>Reciprocal<br>Magnitude | $+$<br>$-$<br>$\times$<br>$\div$<br>$ $ | Plus $2+3.2$ is $5.2$<br>Minus $2-3.2$ is $^{-}1.2$<br>Times $2\times 3.2$ is $6.4$<br>Divide $2\div 3.2$ is $0.625$<br>Residue $A B$ is $B-A\times LB\div A+A=0$ |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| <table border="1"> <tr><td><math>B</math></td><td><math>LB</math></td><td><math>\Gamma B</math></td></tr> <tr><td><math>3.14</math></td><td><math>3</math></td><td><math>4</math></td></tr> <tr><td><math>^{-}3.14</math></td><td><math>^{-}4</math></td><td><math>^{-}3</math></td></tr> </table>  | $B$  | $LB$                                    | $\Gamma B$  | $3.14$ | $3$           | $4$ | $^{-}3.14$    | $^{-}4$    | $^{-}3$ | Floor<br>Ceiling | $L$<br>$\Gamma$ | Minimum $3L7$ is $3$<br>Maximum $3\Gamma 7$ is $7$ |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| $B$   | $LB$   | $\Gamma B$                              |   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| $3.14$  | $3$  | $4$                                     |   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| $^{-}3.14$  | $^{-}4$  | $^{-}3$                                 |   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| $?B$ is Random choice from $1B$   | Roll   | $?$                                     | Deal      A mixed Function (see Figure 5-1)   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| $*B$ is $(2.71828\dots)*B$  | Exponential  | $*$                                     | Power $2*3$ is $8$  |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| $\otimes B$ is $B$ is $*\otimes B$  | Natural logarithm  | $\otimes$                               | General $A\otimes B$ is Log $B$ base $A$<br>logarithm $A\otimes B$ is $(\otimes B)\div \otimes A$   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| $OB$ is $3.14159\dots$  | Pi times   | $O$                                     | Circular, Hyperbolic, Pythagorean (see table at left)   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| $!0$ is $1$<br>$!B$ is $B\times !B-1$<br>or $!B$ is Gamma $(B+1)$   | Factorial  | $!$                                     | Binomial $!1B$ is $(!B)\div (!A)\times !B-A$<br>$2!5$ is $10$ $3!5$ is $10$   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| $\sim 1$ is $0$ $\sim 0$ is $1$   | Not  | $\sim$                                  |   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| <table border="1"> <tr><td><math>(-A)OB</math></td><td><math>A</math></td><td><math>AOB</math></td></tr> <tr><td><math>(1-B*2)*0.5</math></td><td><math>0</math></td><td><math>(1-B*2)*0.5</math></td></tr> <tr><td>Arcsin <math>B</math></td><td><math>1</math></td><td>Sine <math>B</math></td></tr> <tr><td>Arccos <math>B</math></td><td><math>2</math></td><td>Cosine <math>B</math></td></tr> <tr><td>Arctan <math>B</math></td><td><math>3</math></td><td>Tangent <math>B</math></td></tr> <tr><td><math>(^{-}1+B*2)*0.5</math></td><td><math>4</math></td><td><math>(1+B*2)*0.5</math></td></tr> <tr><td>Arcsinh <math>B</math></td><td><math>5</math></td><td>Sinh <math>B</math></td></tr> <tr><td>Arccosh <math>B</math></td><td><math>6</math></td><td>Cosh <math>B</math></td></tr> <tr><td>Arctanh <math>B</math></td><td><math>7</math></td><td>Tanh <math>B</math></td></tr> </table> |  | $(-A)OB$                                | $A$   | $AOB$  | $(1-B*2)*0.5$ | $0$ | $(1-B*2)*0.5$ | Arcsin $B$ | $1$     | Sine $B$         | Arccos $B$      | $2$  | Cosine $B$ | Arctan $B$ | $3$ | Tangent $B$ | $(^{-}1+B*2)*0.5$ | $4$ | $(1+B*2)*0.5$ | Arcsinh $B$ | $5$ | Sinh $B$ | Arccosh $B$ | $6$ | Cosh $B$ | Arctanh $B$ | $7$ | Tanh $B$ |  |  |
| $(-A)OB$  | $A$  | $AOB$                                   |   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| $(1-B*2)*0.5$   | $0$  | $(1-B*2)*0.5$                           |   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| Arcsin $B$  | $1$  | Sine $B$                                |   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| Arccos $B$  | $2$  | Cosine $B$                              |   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| Arctan $B$  | $3$  | Tangent $B$                             |   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| $(^{-}1+B*2)*0.5$   | $4$  | $(1+B*2)*0.5$                           |   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| Arcsinh $B$   | $5$  | Sinh $B$                                |   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| Arccosh $B$   | $6$  | Cosh $B$                                |   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| Arctanh $B$   | $7$  | Tanh $B$                                |   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
| Table of Dyadic $O$ Functions   |  |   |   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
|   |  | $\wedge$                                | And   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
|   |  | $\vee$                                  | Or  |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
|   |  | $\approx$                               | Nand  |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
|   |  | $\neq$                                  | Nor   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
|   |  | $<$                                     | Less  |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
|   |  | $\leq$                                  | Not greater   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
|   |  | $=$                                     | Equal   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
|   |  | $\geq$                                  | Not less  |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
|   |  | $>$                                     | Greater   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
|   |  | $\neq$                                  | Not equal   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |
|   |  |   | Relations:<br>Result is 1 if relation holds,<br>0 if it does not:<br>$3\leq 7$ is $1$<br>$7\leq 3$ is $0$   |        |               |     |               |            |         |                  |                 |  |            |            |     |             |                   |     |               |             |     |          |             |     |          |             |     |          |  |  |

Figure 5-1. Primitive Scalar Functions

A dyadic function  $F$  may possess a *left identity element*  $L$ , such that  $L F X$  equals  $X$  for any  $X$ , or a *right identity element*  $R$ , such that  $X F R$  equals  $X$ . For example, *one* is a right identity element of  $\div$ , since  $X\div 1$  is  $X$ ; *zero* is a left or right identity of  $+$ ; *one* is a left or right identity of  $\times$ , and the general logarithm function  $\otimes$  has no identity element.



Identity elements become important as the appropriate result of applying a function over an empty vector; for example, the sum over an empty vector is 0 (the identity element of  $+$ ), and the product over an empty vector is 1 (the identity element of  $\times$ ). These matters are discussed further in the treatment of the reduction operator, which concerns such applications of dyadic functions over vectors.

Figure 5-2 on page 5-7 lists the identity elements of the dyadic scalar functions. The relational functions  $<$ ,  $\leq$ ,  $=$ ,  $\geq$ ,  $>$ , and  $\neq$  have no true identity elements, except when considered as boolean functions; that is, when restricted to the domains 0 and 1. These identity elements are included in the figure.

| Dyadic Function |   | Identity Element | Left-Right                              |   |
|-----------------|---|------------------|---|---|
| Plus            | + | 0                | L                                       | R |
| Minus           | - | 0                |   | R |
| Times           | × | 1                | L                                       | R |
| Divide          | ÷ | 1                |   | R |
| Residue         |   | 0                | L                                       |   |
| Minimum         | ⌊ | (Note 1)         | L                                       | R |
| Maximum         | ⌈ | (Note 2)         | L                                       | R |
| Power           | * | 1                |   | R |
| Logarithm       | ⊗ |                  | None                                    |   |
| Circle          | ○ |                  | None                                    |   |
| Binomial        | ! | 1                | L                                       |   |
| And             | ^ | 1                | L                                       | R |
| Or              | ∨ | 0                | L                                       | R |
| Nand            | ⊘ |                  | None                                    |   |
| Nor             | ⊚ |                  | None                                    |   |
| Less            | < | 0                | These apply for boolean arguments only. |   |
| Not greater     | ≤ | 1                |   |   |
| Equal           | = | 1                | L                                       | R |
| Not less        | ≥ | 1                |   | R |
| Greater         | > | 0                |   | R |
| Not equal       | ≠ | 0                | L                                       | R |

Notes:

1. The largest representable number.
2. The greatest in magnitude of representable negative numbers.

Figure 5-2. Identity Elements of Primitive Scalar Dyadic Functions

## Plus, Minus, Times, Divide, and Residue

The definitions of the first four of these functions agree with the familiar definitions, except that the indeterminate case  $0 \div 0$  is defined to give the value 1. For  $X \neq 0$ , the expression  $X \div 0$  causes a domain error.

If  $A$  and  $B$  are positive integers, the result of the residue

function  $A|B$  is the remainder when dividing  $A$  into  $B$ . The following definition covers all values of  $A$  and  $B$ .

1. If  $A=0$ , then  $A|B$  equals  $B$ .
2. If  $A\neq 0$ , then  $A|B$  lies between  $A$  and 0 (being permitted to equal 0 but not  $A$ ), and is equal to  $B-N\times A$  for some integer  $N$ .

For example:

$$\begin{array}{r}
 0.385 \quad 1|2.385 \\
 0.385 \quad 0|5.8 \\
 5.8
 \end{array}
 \quad
 \begin{array}{r}
 \phantom{0.385} \quad \phantom{0|} \phantom{5.8} \quad 0 \quad -2 \quad -3| \quad -3 \quad -2 \quad -1 \quad 0 \quad 1 \quad 2 \quad 3 \\
 \phantom{0.385} \quad \phantom{0|} \phantom{5.8} \quad 0 \quad -2 \quad -1 \quad 0 \quad -2 \quad -1 \quad 0 \\
 \phantom{0.385} \quad \phantom{0|} \phantom{5.8} \quad 0 \quad 1 \quad 2 \quad 0 \quad 1 \quad 2 \quad 0
 \end{array}$$

## Conjugate, Negative, Signum, Reciprocal, and Magnitude

The *conjugate function*  $+X$  yields its argument unchanged, the *negative function*  $-X$  yields the argument reversed in sign, and the *reciprocal function*  $\div X$  is equivalent to  $1\div X$ . For example, if  $X\leftarrow 4^{-5}$ , then:

$$\begin{array}{r}
 4^{-5} \quad +X \\
 -4 \quad 5 \quad -X \\
 0.25 \quad -\div X
 \end{array}$$

The result of the *signum function*  $\times X$  depends on the sign of its argument ( $-1$  if  $X<0$ ,  $0$  if  $X=0$ , and  $1$  if  $X>0$ ). The *magnitude function*  $|X$  (also called *absolute value*) yields the greater of  $X$  and  $-X$ ; in terms of the signum function, it is equivalent to  $X\times X$ . For example:

$$\begin{array}{r}
 -1 \quad 0 \quad 1 \quad \times^{-3} \quad 0 \quad 4 \\
 3 \quad 0 \quad 4 \quad |^{-3} \quad 0 \quad 4
 \end{array}$$

## Boolean and Relational Functions

The boolean functions AND, OR, NAND (not-AND), and NOR (not-OR) apply only to boolean arguments; that is, 0 and 1. If 0 is interpreted as false, and 1 is true, then the definitions of these functions are evident from their names. For example,  $A \wedge B$  (read as *A and B*) equals 1 (is true) only if A equals 1 (is true) and B equals 1. All cases are covered by the following examples:

|                  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |
|------------------|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|--|
| $A \leftarrow 0$ | 0 | 1 | 1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |
| $B \leftarrow 0$ | 1 | 0 | 1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |  |
| $A \wedge B$     | 0 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |  |

The monadic function NOT yields the logical complement of its argument; that is  $\sim 0$  is 1, and  $\sim 1$  is 0.

The relational functions apply to any numbers, but yield only boolean results; that is 0 or 1. The result is 1 if the indicated relation holds, and 0 otherwise. For example:

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0 | 3 | 5 | < | 5 | 3 | 1 | 0 | 1 | 3 | 5 | 7 | ≠ | 7 | 5 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

The comparisons in determining the results of the relational functions are not absolute, but are made to a certain tolerance specified by the *comparison tolerance*  $\square CT$ . Two scalar quantities *A* and *B* are considered to be equal if the magnitude of their difference does not exceed the value of  $\square CT$  multiplied by the larger of the magnitudes of *A* and *B*; that is, if  $(|A - B|)$  is less than or equal to  $\square CT \times (|A| \vee |B|)$ . Similarly,  $A \geq B$  is considered to be true if  $(A - B)$  is greater than or equal to  $-\square CT \times (|A| \vee |B|)$ , and  $A > B$  is considered true if  $A \geq B$  is true and  $A = B$  is not.

The comparison tolerance  $\square CT$  is typically set to the value  $1E^{-13}$ . The setting  $\square CT \leftarrow 0$  is also useful, because it yields absolute comparisons, but may lead to unexpected results because of the finite precision of the representation of numbers. For example, if the maximum precision is 15 decimal digits, and all digits are displayed in printing, then:

```

      □PP←15
      □CT←0
      X←0.6666666666666667
      X
0.6666666666666667
      Y←3×X
      Y-2
2.22044604925031E-15
      Z=Y
0
      □CT←1E-13
      Z=Y
1

```

When applied to boolean arguments only, the relations are, in effect, boolean functions, and denote functions that may be familiar from the study of logic, although referred to by different names and symbols. For example,  $X \neq Y$  is the *exclusive-OR* of  $X$  and  $Y$ , and  $X \leq Y$  is *material implication*. This association should be clear from the following table, which lists in the first two columns, the four possible sets of values of two boolean arguments, and in the remaining columns the values of the 16 boolean functions, with the symbols of the boolean and relational functions of APL appended to appropriate columns.

| X | Y |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |
|   |   | ∧ | > | < | ≠ | ∨ | ≈ | = | ≥ | ≤ | ⊗ |   |   |   |   |   |   |

The 10 functions listed at the bottom of this table embrace all non-trivial boolean functions of two arguments. Consequently, any boolean expression of two arguments  $X$  and  $Y$  can be replaced by a simple APL expression as follows: evaluate the expression for the four possible cases, find the corresponding column in the table, then use the function symbol at the bottom of the column, or, if none occurs, use  $X$  or  $Y$  or  $\sim X$  or  $\sim Y$  or 0 or 1, as appropriate.

## Minimum and Maximum

The dyadic functions, *minimum* and *maximum*, denoted by  $\lfloor$  and  $\lceil$ , perform as expected from their names. For example:

```

      X←-3  -2  -1  0  1  2  3
      Y←3   2  1  0  -1  -2  -3
      X⌊Y
3 2 1 0 1 2 3
      X⌈Y
-3 -2 -1 0 -1 -2 -3
    
```

## Floor and Ceiling

The monadic function *floor*, denoted by  $\lfloor$ , yields the integer part of its argument; that is,  $\lfloor X$  yields the largest integer that does not exceed  $X$ . Similarly, the *ceiling* function denoted by  $\lceil X$ , yields the smallest integer that is not less than  $X$ . For example:

```

      X←-3.14  2.718
      ⌊X
-4 2
      -⌈-X
-4 2
      ⌈X
-3 3
      -⌊-X
-3 3
    
```

The ceiling and floor functions are affected by the comparison tolerance  $\square CT$  as follows: if there is an integer  $I$  for which  $|X-I|$  does not exceed the value of  $\square CT \times 1 \lceil I$ , then both  $\lfloor X$  and  $\lceil X$  equal  $I$ . For example, if results are represented and printed to 15 decimal digits, then:

```

      X←3×0.666666666666667
      ⌊CT←1E-13
      ⌊X
2
      ⌈X
2
      ⌊CT←0
      ⌊X
2
      ⌈X
3
    
```

## Roll (Random Number Function)

The *roll* function is a monadic function named by similarity with the roll of a die; thus ?6 yields a (pseudo-) random choice from 16 that is the first six integers beginning with either 0 or 1 according to the value of the index origin  $\square IO$ . For example:

```

       $\square IO \leftarrow 1$ 
      ?6          ?6          ?6
1
      ?6 6 6 6 6 6 6 6 6 6 6 6 6 6 6 6
4 2 1 5 5 6 3 4 5 1 1 4 5
       $\square IO \leftarrow 0$ 
0 2 0 2 4 3 5 5 3 0 3 2 4

```

The domain of the roll function is limited to positive integers.

The roll function uses an algorithm by D. H. Lehmer. The result for each scalar argument  $X$  is a function of  $X$  and of the *random link* variable  $\square RL$ . The result of the roll function is system-dependent, but typically for  $X < 2 * 31$  is equal to  $\square IO + X \lfloor X \times 16807 \times \square RL \div ^{-1} + 2 * 31$ .

## Power, Exponential, General and Natural Logarithm

For non-negative integer right arguments, the *power* function  $X * N$  is simply defined as the product over  $N$  repetitions of  $X$ . It is generalised to non-positive and non-integer arguments to preserve the relation that  $X * A + B$  shall equal  $(X * A) * (X * B)$ . Familiar consequences of this extension are that  $X * -N$  is the reciprocal of  $X * N$ , and  $X * \div N$  is the  $N$ th root of  $X$ . For example:

```

      2 * ^ -3  ^ -2  ^ -1  0  1  2  3
0.125 0.25 0.5 1 2 4 8
      64 * ÷ 1  2  3  4  5  6
64 8 4 2.828427125 2.29739671 2

```

The indeterminate case  $0 * 0$  is defined to have the value 1.

The domain of the power function  $X^*Y$  is restricted in two ways: if  $X=0$ , then  $Y$  must be non-negative; if  $X<0$ , then  $Y$  must be an integer or a (close approximation to a) rational number with an odd denominator. For example,  $^-8^*.5$  yields a domain error, but  $^-8^*1\div 3$  and  $^-8^*2\div 3$  yield  $^-2$  and  $4$ , respectively.

The *exponential* function  $*X$  is equivalent to the expression  $E^*X$ , where  $E$  is the base of the natural logarithms (approximately 2.71828). For example:

```

      *^-2 ^-1 0
0.1353352832 0.3678794412 1
      *1 2
2.718281828 7.389056099
    
```

The *natural logarithm* function  $\otimes X$  is the inverse of the exponential; that is,  $*\otimes X$  and  $\otimes *X$  both equal  $X$ . For example:

```

      \otimes 1 2 3 4
0 0.6931471806 1.098612289 1.386294361
      *\otimes 1 2 3 4
1 2 3 4
      \otimes *1 2 3 4
1 2 3 4
    
```

The domain of the natural logarithm function is limited to positive numbers.

The *general logarithm* function  $B\otimes X$  is defined as  $(\otimes X)\div \otimes B$ . It is inverse to the power function in the following sense:  $B^*B\otimes X$  and  $B\otimes B^*X$  both equal  $X$ . Limitations on the domain follow directly from the defining expression.

## **Circular, Hyperbolic, and Pythagorean Functions**

The symbol  $\circ$  denotes a monadic function whose result equals  $\pi$  times its argument. For example:

```

      \circ 1 2 0.5
3.141592654 6.283185307 1.570796327
    
```



The symbol  $\circ$  is also used dyadically to denote a family of 15 related functions as follows: the expression  $I\circ X$  is defined for integer values of  $I$  from  $\bar{7}$  to  $7$ , and is in each case equivalent to one of the circular, hyperbolic, or pythagorean functions, as indicated in Figure 5-1 on page 5-5.

The *circular* functions,  $\sin$ ,  $\cos$ , and  $\tan$  ( $1\circ X$ ,  $2\circ X$ , and  $3\circ X$ ), require an argument in radians. For example:

$$\begin{array}{l}
 PI \leftarrow 0.1 \\
 1\circ PI \div 2 \quad 3 \quad 4 \\
 1 \quad 0.8660254038 \quad 0.7071067812
 \end{array}$$

The *hyperbolic* functions,  $\sinh$  and  $\cosh$  ( $5\circ X$  and  $6\circ X$ ), are the odd and even components of the exponential function; that is,  $5\circ X$  is odd,  $6\circ X$  is even, and the sum  $(5\circ X) + (6\circ X)$  is equivalent to  $*X$ . Consequently:

$$\begin{array}{l}
 5\circ X \text{ equals } 0.5 \times (*X) - (*-X) \\
 6\circ X \text{ equals } 0.5 \times (*X) + (*-X)
 \end{array}$$

The definition of the hyperbolic tangent function,  $\tanh$  ( $7\circ X$ ), is similar to that of the tangent; that is  $7\circ X$  equals  $(5\circ X) \div 6\circ X$ .

The *pythagorean* functions  $0\circ X$ ,  $4\circ X$ , and  $\bar{4}\circ X$  are defined as shown in Figure 5-1 on page 5-5, and are related to the properties of a right triangle as indicated in Figure 5-3 on page 5-15. They may also be defined as follows:

$$\begin{array}{l}
 \bar{4}\circ X \text{ equals } 5\circ \bar{6}\circ X \\
 0\circ X \text{ equals } 2\circ \bar{1}\circ X \quad \text{or} \quad 1\circ \bar{2}\circ X \\
 4\circ X \text{ equals } 6\circ \bar{5}\circ X
 \end{array}$$

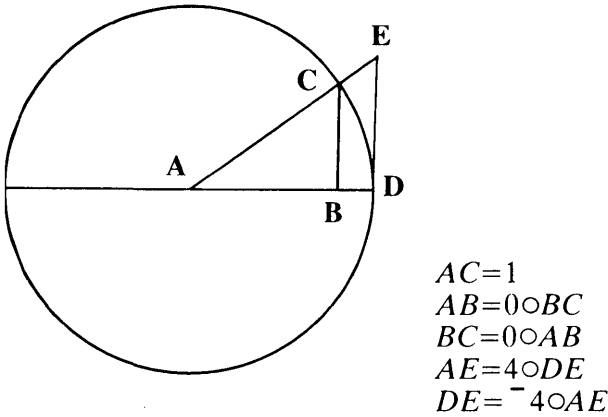


Figure 5-3. The Pythagorean Functions

Each of the family of functions,  $I \circ X$ , has an inverse in the family; that is,  $(-I) \circ X$  is the inverse of  $I \circ X$ . Certain of the functions are not *monotonic*, and their inverses are therefore many-valued. The principal values are chosen in the following intervals:

- Arcosh  $R \leftarrow \bar{6} \circ X \quad R \geq 0$   
 $R \leftarrow \bar{4} \circ X \quad R \geq 0$
- Arctan  $R \leftarrow \bar{3} \circ X \quad (|R| \leq 0.5)$
- Arccos  $R \leftarrow \bar{2} \circ X \quad (R \geq 0) \wedge (R \leq 0.1)$
- Arcsin  $R \leftarrow \bar{1} \circ X \quad (|R| \leq 0.5)$   
 $R \leftarrow 0 \circ X \quad R \geq 0$   
 $R \leftarrow 4 \circ X \quad R \geq 0$

## Factorial and Binomial Functions

The *factorial* function,  $!N$ , is defined, for positive integer arguments, as the product of all positive integers up to  $N$ . An important consequence of this definition is that  $!N$  equals  $N \times !N-1$ , or equivalently,  $!N-1$  equals  $(!N) \div N$ . This relation is used to extend the function to all arguments except negative integers. For example:

```

      N←1 2 3 4 5
      !N
1 2 6 24 120
      (!N)÷N
1 1 2 6 24
      !0 1 2 3 4
1 1 2 6 24
      F←.5 1 1.5 2 2.5
      !F
0.8862269255 1 1.329340388 2 3.32335097
      (!F)÷F
1.772453851 1 0.8862269255 1 1.329340388
      !-0.5 0 .5 1 1.5
1.772453851 1 0.8862269255 1 1.329340388

```

This extension leads to the expression  $(!0)÷0$  or  $1÷0$  for  $!^{-1}$ , and  $^{-1}$  is therefore excluded from the domain of the factorial function, as are all negative integers.

The *binomial* function,  $M!N$ , is defined, for non-negative integer arguments, as the number of distinct ways in which  $M$  things can be chosen from  $N$  things. The expression  $(!N)÷(!M)×(!N-M)$  yields an equivalent definition that is used to extend the definition to all numbers. Although the domain of factorial excludes negative integers, the domain of the binomial does not, because any implied division by 0 in the numerator  $!N$  is usually accompanied by a corresponding division by 0 in the denominator; the function, therefore, extends smoothly to all numbers, except where  $N$  is a negative integer and  $M$  is not an integer.

The result of  $I!N$  is equivalent to coefficient  $I$  in the binomial expansion  $(X+1)*N$ . For example:

```

      0 1 2 3!3
1 3 3 1

```

# Operators

An *operator* may be applied to a function to get a different function. For example, the outer product operator, denoted by the symbol  $\circ$ , may be applied to any of the primitive scalar dyadic functions to derive a corresponding "table function", as shown in the following for *times* and *power*:

|     |                  |            |      |     |
|-----|------------------|------------|------|-----|
|     | $A \leftarrow 1$ | $2$        | $3$  | $4$ |
|     | $A \circ$        | $\times A$ |      |     |
| $1$ | $2$              | $3$        | $4$  |     |
| $2$ | $4$              | $6$        | $8$  |     |
| $3$ | $6$              | $9$        | $12$ |     |
| $4$ | $8$              | $12$       | $16$ |     |

|     |           |         |          |
|-----|-----------|---------|----------|
|     | $A \circ$ | $\cdot$ | $\ast A$ |
| $1$ | $1$       | $1$     | $1$      |
| $2$ | $4$       | $8$     | $16$     |
| $3$ | $9$       | $27$    | $81$     |
| $4$ | $16$      | $64$    | $256$    |

Four of the APL operators - *reduction*, *scan*, *inner product*, and *outer product* - may apply to any primitive scalar dyadic function. The axis operator applies to functions derived from reduction and scan, and also to certain of the mixed functions.

## Reduction

*Reduction* is denoted by the symbol  $/$  and applies to the function that precedes it. For example, if  $V \leftarrow 1\ 2\ 3\ 4\ 5$ , then  $+ / V$  yields the sum of the items of  $V$ , and  $\times / V$  yields their product:

|      |         |       |              |
|------|---------|-------|--------------|
|      | $+ / V$ |       | $\times / V$ |
| $15$ |         | $120$ |              |

In general, an expression of the form  $f / V$  is equivalent to the expression obtained by placing the function symbol  $f$  between adjacent pairs of items of the vector  $V$ :

|     |                |     |   |
|-----|----------------|-----|---|
|     | $\uparrow / V$ |     | $1 \uparrow 2 \uparrow 3 \uparrow 4 \uparrow 5$ |
| $5$ |                | $5$ |   |
|     | $- / V$        |     | $1 - 2 - 3 - 4 - 5$                             |
| $3$ |                | $3$ |   |

The last example emphasises that the general rule for the order of execution from right to left is applied, and that as a consequence, the expression  $- / V$  yields the *alternating sum* of

the items of  $V$ . The alternating sum is the sum obtained after first weighting the items by multiplying alternate elements by 1 and  $-1$ . Thus:

$$\begin{array}{r}
 \phantom{1} \phantom{-2} \phantom{3} \phantom{-4} \phantom{5} \\
 \phantom{1} \phantom{-2} \phantom{3} \phantom{-4} \phantom{5} \\
 \phantom{1} \phantom{-2} \phantom{3} \phantom{-4} \phantom{5} \\
 1 \phantom{-2} \phantom{3} \phantom{-4} \phantom{5} \\
 3 \\
 3
 \end{array}
 \begin{array}{l}
 A \leftarrow 1 \quad -1 \quad 1 \quad -1 \quad 1 \\
 V \times A \\
 +/V \times A \\
 -/V
 \end{array}$$

Similarly,  $\div/V$  yields the *alternating product*:

$$\begin{array}{r}
 \phantom{1} \phantom{0.5} \phantom{3} \phantom{0.25} \phantom{5} \\
 \phantom{1} \phantom{0.5} \phantom{3} \phantom{0.25} \phantom{5} \\
 \phantom{1} \phantom{0.5} \phantom{3} \phantom{0.25} \phantom{5} \\
 1 \phantom{0.5} \phantom{3} \phantom{0.25} \phantom{5} \\
 1.875 \\
 1.875
 \end{array}
 \begin{array}{l}
 V \times A \\
 \times/V \times A \\
 \div/V
 \end{array}$$

The result of applying reduction to any scalar or vector is a scalar; the value for a scalar or one-element vector argument is the single item itself. (The application of reduction to other arrays is treated in the discussion of the axis operator).

Reduction of an empty vector by any function is the *identity element* of the function, if one exists, and a *domain error* if one does not exist. Thus if  $V$  is an empty vector,  $+/V$  equals 0, and  $\wedge/V$  equals 1.

The reason for this definition is the extension to empty vectors of an important relation between the reductions of two vectors,  $P$  and  $Q$ , and the reduction of the vector  $V \leftarrow P, Q$ , which is obtained by chaining them together. For example:

$$\begin{array}{l}
 +/V \text{ equals } (+/P) + (+/Q) \\
 \times/V \text{ equals } (\times/P) \times (\times/Q)
 \end{array}$$

If  $P$  is an empty vector, then  $+/P$  must equal 0 (the identity element of  $+$ ), and  $\times/P$  must equal 1.

## Scan

The *scan* operator is denoted by the symbol  $\backslash$  and applies to the function that precedes it. When the resulting function is applied to a vector  $V$ , it yields a vector of the same shape, the  $K$ th element of which is equal to the corresponding reduction over the first  $K$  elements of  $V$ . For example:

```

+\ $\backslash$ 1 2 3 4 5
1 3 6 10 15
x\ $\backslash$ 1 2 3 4 5
1 2 6 24 120
v\ $\backslash$ 0 0 1 0 1
0 0 1 1 1
^\ $\backslash$ 1 1 0 1 0
1 1 0 0 0
<\ $\backslash$ 0 0 1 0 1 1 0
0 0 1 0 0 0 0
    
```

The extension of *scan* to arrays other than vectors is treated in the discussion of the *axis* operator.

## Axis

A matrix can be viewed as a collection of either columns or rows, and an array of higher rank can be viewed as a collection of planes or hyperplanes. For example, a three-dimensional array of shape 2 3 4 is normally represented as two planes of 3-by-4 matrices, but it can also be viewed as three planes of 2-by-4 matrices, or as four planes of 2-by-3 matrices. For any chosen representation, the resulting (hyper)planes are orthogonal to the chosen axis, and are said to *lie along* that axis. Thus, in the preceding example, the 3-by-4 matrices lie along the first axis.

In previous sections, the reduction and scan operators were defined for a vector. This definition is extended to arrays of higher rank by applying the function argument of the operator between successive (hyper)planes. As the preceding example shows, a multi-dimensional array can be viewed as a collection of arrays of lesser rank which lie along any chosen axis. The *axis* operator is used to select the chosen axis, and determines the direction of application of the scan or reduction operators.

The axis operator is denoted by brackets immediately following a scan or reduction operator. The brackets enclose an expression yielding the index of the desired axis as a scalar or one-element vector. If a scan or reduction operator is applied to an array without the axis operator, the direction of application will be along the last axis. For example:

```

      ⍵←M←3 4p112
1  2  3  4
5  6  7  8
9 10 11 12
      +\[1]M
1  2  3  4      15 18 21 24
6  8 10 12
15 18 21 24
      +\[2]M
1  3  6 10      10 26 42
5 11 18 26
9 19 30 42
      +\M
1  3  6 10      10 26 42
5 11 18 26
9 19 30 42

```

The result of the scan operation has the same shape as the argument. The result of a reduction operation has a shape similar to the shape of the argument, but with the indicated axis of reduction removed. Indexing of axes is dependent on the current value of the index origin,  $\text{IO}$ . With  $\text{IO} \leftarrow 1$ , the leftmost or first axis has an index value of 1. The symbols  $\nabla$  and  $\backslash$  also denote reduction and scan operations, which are equivalent to the standard reduction and scan operators when used with the axis operator. When used without an axis operator however, these symbols cause the reduction or scan operation to be applied along the FIRST axis.

The axis operator is also used to specify the axis of application of the mixed functions, *reverse*, *rotate*, *catenate*, *compress*, and *expand*. The axis operator cannot be used with the inner product or outer product operators.

## Inner Product

If  $P$  and  $Q$  are vectors of the same shape, the expression  $+/P \times Q$  has a variety of useful interpretations. For example, if  $P$  is a list of prices and  $Q$  is a list of corresponding order quantities, then  $+/P \times Q$  is the total cost. Expressions of the same form using functions other than  $+$  and  $\times$  are equally useful, as suggested by the following examples (where  $B$  is used to denote a boolean vector):

$\wedge/P=Q$       Comparison of  $P$  and  $Q$

$+/P=Q$       Count of agreements between  $P$  and  $Q$

$L/P+Q$       Minimum distance for shipment to a particular destination, where  $P$  represents the distances from source to possible intermediate shipping points and  $Q$  the distances from these points to the destination.

$+/P \times B$       Sum over a subset of  $P$  specified by  $B$

$\times/P \times B$       Product over a subset of  $P$  specified by  $B$

The *inner product* operator produces functions equivalent to expressions of this form; it is denoted by a dot and applies to the two functions that surround it. Thus  $P+. \times Q$  is equivalent to  $+/P \times Q$ , and  $P \times . \times B$  is equivalent to  $\times/P \times B$  and, in general,  $Pf. gQ$  is equivalent to  $f/PgQ$ , if  $P$  and  $Q$  are vectors.

The inner product is extended to arrays other than vectors along certain fixed axes, namely the last axis of the first argument and the first axis of the last argument. The lengths of these axes must agree. The shape of the result is obtained by deleting these axes and chaining the remaining shape vectors. The consequences for matrix arguments are shown in Figure 5-4 on page 5-22.



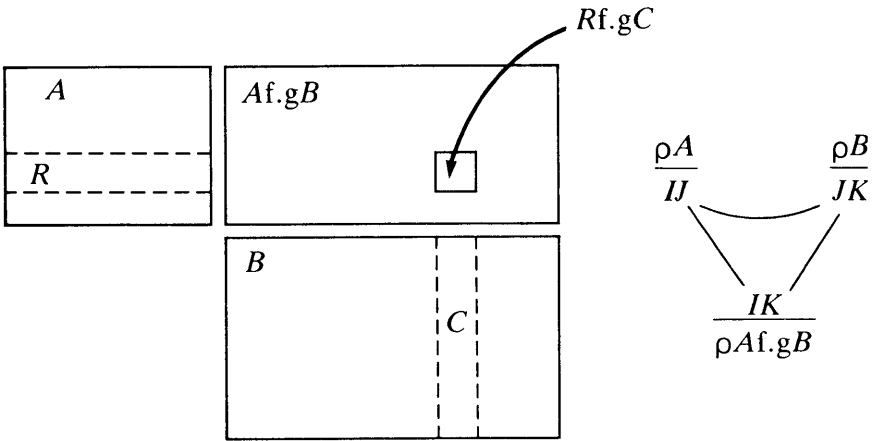
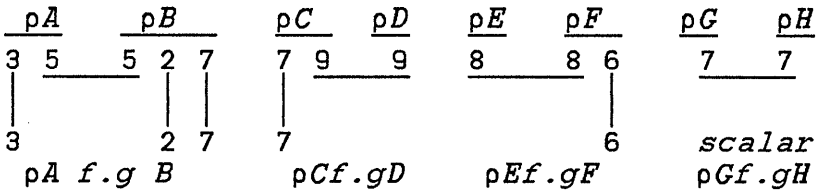


Figure 5-4. Inner Product

The consequences for the shape of inner products on some other arrays are shown in the following example:



Formally,  $\rho Af.gB$  equals  $(^{-1}\rho A), 1\rho B$ .

The inner product  $M+. \times N$  is commonly called the *matrix product*. Examples of it also are shown in the following.

$$P \leftarrow 2 \quad 3 \quad 5 \quad 7$$

$$M \leftarrow (14) \circ . \leq 14$$

|   |    |    |     |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |
|---|----|----|-----|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|----|----|----|---|---|---|----|----|--|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|----|---|----|---|---|---|---|---|---|---|----|---|---|----|-----|---|---|---|---|
| $M$ <table style="border-collapse: collapse; margin-left: 20px;"> <tr><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td></tr> </table><br>$M+ . \times M$ <table style="border-collapse: collapse; margin-left: 20px;"> <tr><td>0</td><td>1</td><td>2</td><td>3</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>2</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td></tr> </table><br>$M+ . \times P$ <table style="border-collapse: collapse; margin-left: 20px;"> <tr><td>17</td><td>15</td><td>12</td><td>7</td></tr> </table><br>$P+ . \times M$ <table style="border-collapse: collapse; margin-left: 20px;"> <tr><td>2</td><td>5</td><td>10</td><td>17</td></tr> </table> | 1  | 1  | 1   | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 2 | 3 | 0 | 0 | 1 | 2 | 0 | 0 | 0 | 1 | 17 | 15 | 12 | 7 | 2 | 5 | 10 | 17 | $M \wedge . = M$ <table style="border-collapse: collapse; margin-left: 20px;"> <tr><td>0</td><td>0</td><td>0</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>0</td></tr> </table><br>$M- . \times M$ <table style="border-collapse: collapse; margin-left: 20px;"> <tr><td>0</td><td>-1</td><td>0</td><td>-1</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>-1</td></tr> </table><br>$P \times . \times M$ <table style="border-collapse: collapse; margin-left: 20px;"> <tr><td>2</td><td>6</td><td>30</td><td>210</td></tr> </table><br>$M \wedge . = 0 \quad 0 \quad 1 \quad 1$ <table style="border-collapse: collapse; margin-left: 20px;"> <tr><td>0</td><td>0</td><td>1</td><td>0</td></tr> </table> | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | -1 | 0 | -1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | -1 | 2 | 6 | 30 | 210 | 0 | 0 | 1 | 0 |
| 1   | 1  | 1  | 1   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |
| 0   | 1  | 1  | 1   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |
| 0   | 0  | 1  | 1   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |
| 0   | 0  | 0  | 1   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |
| 0   | 1  | 2  | 3   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |
| 0   | 0  | 1  | 2   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |
| 0   | 0  | 0  | 1   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |
| 17  | 15 | 12 | 7   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |
| 2   | 5  | 10 | 17  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |
| 0   | 0  | 0  | 1   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |
| 0   | 0  | 0  | 0   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |
| 0   | 0  | 0  | 0   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |
| 0   | 0  | 0  | 0   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |
| 0   | -1 | 0  | -1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |
| 0   | 0  | 1  | 0   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |
| 0   | 0  | 0  | -1  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |
| 2   | 6  | 30 | 210 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |
| 0   | 0  | 1  | 0   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |    |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |    |   |    |   |   |   |   |   |   |   |    |   |   |    |     |   |   |   |   |

Either argument of an inner product may be a scalar or a one-element vector; it is extended in the usual way. For example,  $A+ . \times 1$  is equivalent to  $+A$ , and  $1+ . \times A$  is equivalent to  $+A$ .

## Outer Product

The *outer product* operator, denoted by the symbols  $\circ .$  preceding the function symbol, applies to any dyadic primitive scalar function, so that the function is evaluated for *each* member of the left argument paired with *each* member of the right argument. For example, if  $A \leftarrow 1 \quad 2 \quad 3$  and  $B \leftarrow 1 \quad 2 \quad 3 \quad 4 \quad 5$ , then:

|  |   |   |    |    |   |   |   |   |   |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|--|---|---|----|----|---|---|---|---|---|----|---|---|---|----|----|--|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $A \circ . \times B$ <table style="border-collapse: collapse; margin-left: 20px;"> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr> <tr><td>2</td><td>4</td><td>6</td><td>8</td><td>10</td></tr> <tr><td>3</td><td>6</td><td>9</td><td>12</td><td>15</td></tr> </table> | 1 | 2 | 3  | 4  | 5 | 2 | 4 | 6 | 8 | 10 | 3 | 6 | 9 | 12 | 15 | $A \circ . < B$ <table style="border-collapse: collapse; margin-left: 20px;"> <tr><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td></tr> <tr><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td></tr> </table> | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1  | 2 | 3 | 4  | 5  |   |   |   |   |   |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 2  | 4 | 6 | 8  | 10 |   |   |   |   |   |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 3  | 6 | 9 | 12 | 15 |   |   |   |   |   |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0  | 1 | 1 | 1  | 1  |   |   |   |   |   |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0  | 0 | 1 | 1  | 1  |   |   |   |   |   |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| 0  | 0 | 0 | 1  | 1  |   |   |   |   |   |    |   |   |   |    |    |  |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

Such tables may be better understood if they are labelled in a way that is widely used in elementary arithmetic texts: values of the arguments are placed beside and above the table, and the function whose outer product is being computed is shown at the corner. Thus:

|          |   | <i>B</i> |   |   |    |    |
|----------|---|----------|---|---|----|----|
|          |   | 1        | 2 | 3 | 4  | 5  |
| <i>A</i> | 1 | 1        | 2 | 3 | 4  | 5  |
|          | 2 | 2        | 4 | 6 | 8  | 10 |
|          | 3 | 3        | 6 | 9 | 12 | 15 |
|          |   |          |   |   |    |    |

|          |   | <i>B</i> |   |   |   |   |
|----------|---|----------|---|---|---|---|
|          |   | 1        | 2 | 3 | 4 | 5 |
| <i>A</i> | 1 | 0        | 1 | 1 | 1 | 1 |
|          | 2 | 0        | 0 | 1 | 1 | 1 |
|          | 3 | 0        | 0 | 0 | 1 | 1 |
|          |   |          |   |   |   |   |

In the preceding example, the shape of the result  $A \circ \cdot \times B$  is clearly equal to  $(\rho A)$ ,  $(\rho B)$ . This expression yields the shape for any arguments  $A$  and  $B$ . Thus, if  $R \leftarrow A \circ \cdot + B$ , and  $A$  is a matrix of shape 3 4, and  $B$  is a three-dimensional array of shape 5 6 7, then  $R$  is a five-dimensional array of shape 3 4 5 6 7. Moreover,  $R[I;J;K;L;M]$  equals  $A[I;J]+B[K;L;M]$  for all possible scalar values of the indexes.

# Mixed Functions

The mixed functions are grouped in five classes according to whether they concern the structure of arrays, selection from arrays, generation of selector information for use by selection functions, numeric calculations, or transformations of data, such as that between characters and numbers. All are listed in Figure 5-5, with brief definitions or examples.

Those functions that may be changed by an axis operator may also be used without an axis operator, in which case the axis is the last or, for the functions denoted by  $\Theta$  and  $\nearrow$ , the first axis.

Figure 5-5 summarises the restrictions on the ranks of arguments that may be used with each mixed function.

| Name  | Sign(1)   | Definition or Example (2)  |
|---|-----------|--|
| <b>Functions Concerning the Structure of Arrays</b> |           |  |
| Shape   | $\rho A$  | $\rho P$ is 4<br>$\rho E$ is 3 4<br>$\rho 5$ is 1 0  |
| Reshape   | $V\rho A$ | Reshape $A$ to dimension $V$<br>3 4 $\rho 12$ is $E$<br>12 $\rho E$ is 1 12<br>0 $\rho E$ is 1 0               |
| Ravel   | $,A$      | $,A$ is $(x/\rho A)\rho A$<br>$,E$ is 1 12<br>$\rho, 5$ is 1   |
| Reverse<br>(3)                                      | $\Phi A$  | DCBA<br>$\Phi X$ is HGFE<br>LKJI<br><br>IJKL<br>$\Phi[1]X$ is $\Theta X$ is EFGH<br>ABCD                       |
| Rotate<br>(3)                                       | $A\Phi A$ | $\Phi P$ is 7 5 3 2<br>3 $\Phi P$ is 7 2 3 5 is $\bar{1}\Phi P$<br>BCDA<br>1 0 $\bar{1}\Phi X$ is EFGH<br>LIJK |

Figure 5-5 (Part 1 of 4). Primitive Mixed Functions

| Name   | Sign(1) | Definition or Example (2)  |
|--|---------|--|
| <b>Functions Concerning the Structure of Arrays (cont)</b> |         |  |
| Catenate,<br>Laminate                                      | A,A     | P, <sub>1</sub> 2 is 2 3 5 7 1 2<br>'T','HIS' is 'THIS'<br>P,[.5]P is 2 3 5 7<br>2 3 5 7   |
| Transpose<br>(4)   | VQ A    | Coordinate I of A becomes<br>coordinate V[I] of result<br>AEI<br>2 1Q X is BFJ<br>CGK<br>DHL<br>1 1QE is 1 6 11                  |
|  | QA      | Reverse order of coordinates<br>QE is 2 1QE  |
| <b>Functions Concerning Selection from Arrays</b>          |         |  |
| Take   | V^A     | 2 3^X is ABC           ^-2^P is 5 7<br>EFG<br>Take or drop  V[I]  first (V[I]≥0)<br>or last (V[I]<0) elements of<br>coordinate I |
| Drop   | VvA     | 2 3vX is L           ^-2vP is 2 3  |
| Compress<br>(3)  | V/A     | 1 0 1 0/P is 2 5<br>1 3<br>1 0 1 0/E is 5 7<br>9 11<br>1 0 1/[1]E is 1 2 3 4 is 1 0 1^E<br>9 10 11 12                            |
| Expand<br>(3)  | V\A     | 1 0 1\1^2 is 1 0 2<br>A BCD<br>1 0 1 1 1\X is E FGH<br>I JKL   |
| Indexing<br>(4, 5)   | V[A]    | P[2] is 3<br>P[4 3 2 1] is 7 5 3 2   |
|  | M[A;A]  | E[1 3;3 2 1] is 3 2 1<br>11 10 9   |
|  | A[A;..] | E[1;] is 1 2 3 4   |
|  | ..;A]   | E[;1] is 1 5 9   |
|  |         | ABC D<br>'ABCDEFGH IJKL'[E] is EFGH<br>IJKL  |

Figure 5-5 (Part 2 of 4). Primitive Mixed Functions

| Name  | Sign(1)       | Definition or Example (2)   |
|---|---------------|---|
| <b>Functions That Generate Selector Information</b> |               |   |
| Index Generator<br>(4)                              | $\imath S$    | First $S$ integers<br>$\imath 4$ is 1 2 3 4   |
| Index of<br>(4)                                     | $V\imath A$   | Least index of $A$ in $V$ , or $1+\rho V$<br>$P\imath 3$ is 2<br>5 1 2 5<br>$P\imath E$ is 3 5 4 5<br>5 5 5 5<br>4 4 4 is 1   |
| Membership<br>(4)                                   | $A\epsilon A$ | $\rho W\epsilon Y$ is $\rho W$<br>$P\epsilon\imath 4$ is 1 1 0 0<br>0 1 1 0<br>$E\epsilon P$ is 1 0 1 0<br>0 0 0 0            |
| Grade up<br>(4)                                     | $\Delta V$    | $\Delta 3 5 3 2$ is 4 1 3 2<br>The permutation that would order $V$ (ascending or descending)                                 |
| Grade down<br>(4)                                   | $\nabla V$    | $\nabla 3 5 3 2$ is 2 1 3 4   |
| Grade up<br>(dyadic)<br>(4)                         | $A\Delta A$   | 'ABCDE' $\Delta$ 'DEAL' is 3 1 2 4  |
| Grade down<br>(dyadic)<br>(4)                       | $A\nabla A$   | 'ABCDE' $\nabla$ 'DEAL' is 4 2 1 3  |
| Deal<br>(4)   | $S?S$         | $W?Y$ is random deal of $W$ elements from $\imath Y$  |
| <b>Functions That Involve Numeric Calculations</b>  |               |   |
| Matrix inverse                                      | $\boxplus M$  | $\boxplus 2 2\rho 1 1 0 1$ is $\begin{matrix} 1 & -1 \\ 0 & 1 \end{matrix}$<br>Arguments may be scalars, vectors, or matrices |
| Matrix division                                     | $M\boxplus M$ | $(2 2\rho P)\boxplus 2 2\rho 1 1 0 1$ is $\begin{matrix} -3 & -4 \\ 5 & 7 \end{matrix}$                                       |
| Decode  | $A\perp A$    | $10\perp 1 7 7 6$ is 1776<br>$24 60 60\perp 1 2 3$ is 3723  |
| Encode  | $A\top A$     | $24 60 60\top 3723$ is 1 2 3<br>$60 60\top 3723$ is 2 3   |

Figure 5-5 (Part 3 of 4). Primitive Mixed Functions

| Name  | Sign(1)      | Definition or Example (2)  |
|---|--------------|--|
| <b>Functions That Involve Data Transformation</b> |              |  |
| Execute   | $\oplus V$   | $\oplus '1+2'$ is 3<br>$\oplus 'P'$ is 2 3 5 7   |
| Format<br>(Monadic)                               | $\ominus A$  | $\ominus '1.5'$ is $\ominus 1.5$ is 1<br>$\ominus E$ is 3 12<br>$X$ is $\ominus X$   |
| Format<br>(Dyadic)                                | $V \oplus A$ | $4 \ 1 \oplus P$ is 2.0 3.0 5.0 7.0<br>$6 \ \ominus 1 \oplus P$ is 2E000 3E000 5E000 7E000<br>$'0,55' \oplus P$ is 0,02 0,03 0,05 0,07 |

Notes:

1. Restrictions on argument ranks are indicated by: *S* for scalar, *V* for vector, *M* for matrix, and *A* for any array (see Figure 5-6 on page 5-29).

Conformability requirements are given in the text where each function is defined.

2. Arrays used in examples:

```

      P
2 3 5 7
      E
  1 2 3 4
  5 6 7 8
  9 10 11 12
      X
ABCD
EFGH
IJKL

```

3. The function is applied along the last axis; the symbols  $\dagger$ ,  $\backslash$ , and  $\Theta$  are equivalent to  $/$ ,  $\backslash$ , and  $\Phi$ , respectively, except that the function is applied along the first axis. In general, the relevant axis is determined by  $[V]$  or  $[S]$  after the function symbol.
4. Function depends on index origin.
5. Elision of any index selects all along that axis.

Figure 5-5 (Part 4 of 4). Primitive Mixed Functions

Figure 5-6 on page 5-29 shows for what mixed functions and under what conditions scalar and vector arguments may be substituted for each other.

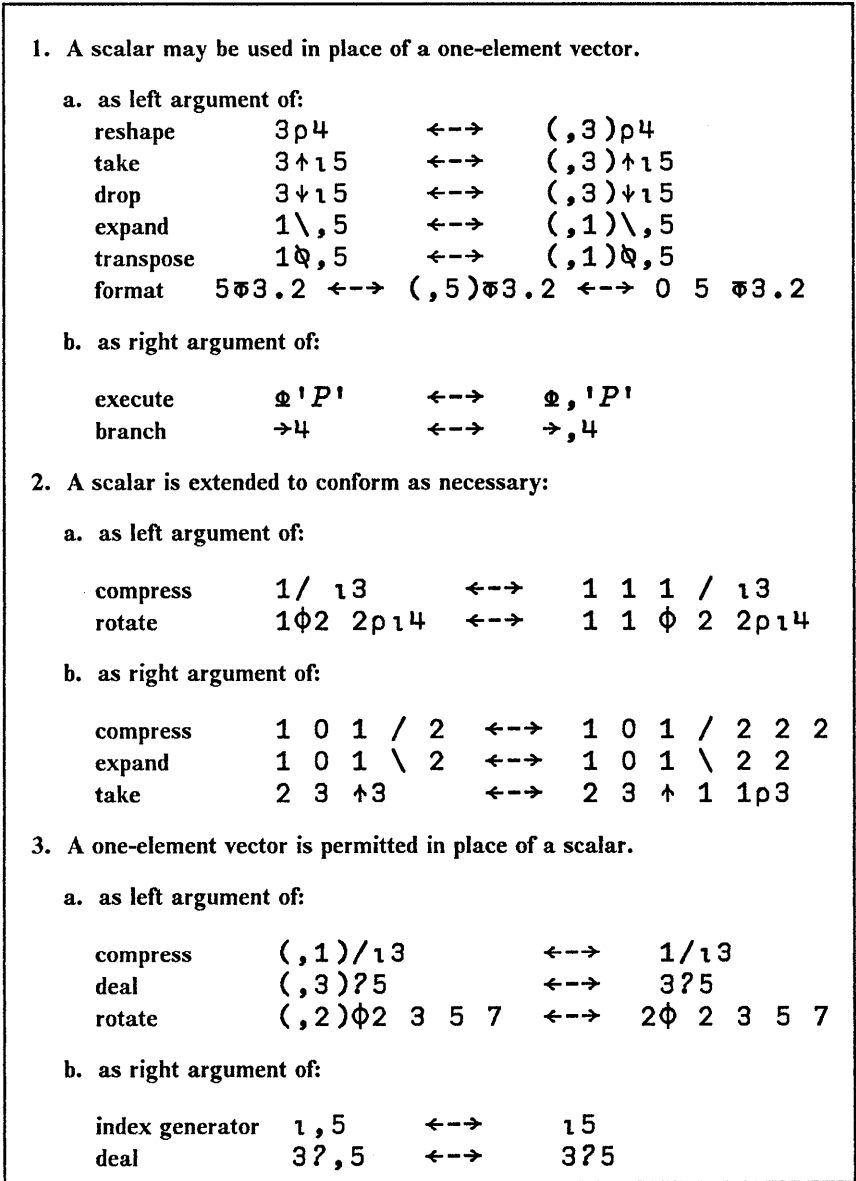


Figure 5-6. Scalar Vector Substitutions for Mixed Functions



## Structural Functions

In the monadic structure functions, the argument may be any type: numeric or character. In the dyadic selection and structure functions, one argument may be any type, and the other (which serves as an index or other selection indicator) must be numeric, and in one case (expansion), is further restricted to be boolean.

### Shape, Reshape, and Ravel

The *shape* function is the monadic function  $\rho$ . When applied to an array  $A$ , it yields the *shape* of  $A$ ; that is, a vector whose components are the dimensions of  $A$ . For example, if  $A$  is the matrix of three rows and four columns:

```

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

```

then  $\rho A$  is the vector 3 4.

Because  $\rho A$  has one component for each axis of  $A$ , the expression  $\rho\rho A$  is the rank of  $A$ . The following table shows the values of  $\rho A$  and  $\rho\rho A$  for arrays of rank 0 (scalars) up to rank 3. In particular, the function  $\rho$  applied to a scalar yields an empty vector.

| <u>Type of Array</u> | <u><math>\rho A</math></u> | <u><math>\rho\rho A</math></u> |
|----------------------|----------------------------|--------------------------------|
| Scalar               |                            | 0                              |
| Vector               | $N$                        | 1                              |
| Matrix               | $MN$                       | 2                              |
| 3-Dimensional        | $LMN$                      | 3                              |

The monadic function *ravel* is denoted by a comma. When applied to any array  $A$ , it produces a vector whose elements are the elements of  $A$  in row order. For example, if  $A$  is the matrix:

```

      A←3 4ρ2 4 6 8 10 12 14 16 18 20 22 24
      A
    2  4  6  8
   10 12 14 16
   18 20 22 24
  
```

and if  $V \leftarrow A$  then  $V$  is a 12-element vector containing the integers 2 4 6 8 10...24. If  $A$  is a vector, then  $\rho A$  is equivalent to  $A$ ; if  $A$  is a scalar, then  $\rho A$  is a vector of length 1.

The *reshape* function is the dyadic function  $\rho$ , which reshapes its right argument to the shape specified by its left argument. If  $M \leftarrow D\rho V$ , then  $M$  is an array of dimension  $D$  whose elements are the elements of  $V$ . For example,  $2\ 3\rho 1\ 2\ 3\ 4\ 5\ 6$  is the matrix:

```

  1 2 3
  4 5 6
  
```

If  $N$ , the total number of elements required in the array  $D\rho V$ , is equal to the dimension of the vector  $V$ , then the ravel of  $D\rho V$  is equal to  $V$ . If  $N$  is less than  $\rho V$ , then only the first  $N$  elements of  $V$  are used; if  $N$  is greater than  $\rho V$ , then the elements of  $V$  are repeated cyclically. For example:

```

      2 3ρ1 2          3 3ρ1 0 0 0
    1 2 1          1 0 0
    2 1 2          0 1 0
                  0 0 1
  
```

More generally, if  $A$  is any array, then  $D\rho A$  is equivalent to  $D\rho \rho A$ . For example:

If:

```

      A←2 3ρ1 2 3 4 5 6
      A
    1 2 3
    4 5 6
  
```

Then:

```

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

The expressions  $0\rho X$  and  $0\ 3\rho X$  and  $0\ 0\rho X$  are all valid; any one or more of the axes of an array may have zero length. Such an array is called an *empty array*. If  $D$  is an empty vector, then  $D\rho A$  is a scalar.

## Reverse and Rotate

The monadic function *reverse* is denoted by  $\phi$ ; if  $X$  is a vector and  $K\leftarrow\phi X$ , then  $K$  is equal to  $X$ , except that the items appear in reverse order. The axis operator applies to reversal and determines the axis along which the vectors are to be reversed. For example:

$$\begin{array}{ccc} & A & \\ \begin{array}{ccc} 1 & 2 & 3 \\ 4 & 5 & 6 \end{array} & & \begin{array}{ccc} 4 & 5 & 6 \\ 1 & 2 & 3 \end{array} \end{array} \quad \phi[1]A \quad \begin{array}{ccc} 3 & 2 & 1 \\ 6 & 5 & 4 \end{array} \quad \phi[2]A$$

The expression  $\phi A$  denotes reversal along the last coordinate of  $A$ , and  $\Theta A$  denotes reversal along the first coordinate. For example, if  $A$  is of rank 3, then  $\phi A$  is equivalent to  $\phi[3]A$ , and  $\Theta A$  is equivalent to  $\phi[1]A$ . The axis operator applies to  $\Theta$ , and  $\Theta[J]A$  is equal to  $\phi[J]A$ .

The dyadic function *rotate* is also denoted by  $\phi$ . If  $K$  is a scalar or one-element vector, and  $X$  is a vector, then  $K\phi X$  results in a cyclic rotation of  $X$ , where  $K$  specifies the number of positions that every element is to be shifted. For  $K>0$ , the elements are rotated to the left; for  $K<0$ , the rotation occurs to the right. If the magnitude of  $K$  is larger than the number of elements in  $X$ , the rotation will be more than one full cycle. Formally,  $K\phi X$  is defined as  $X[1+(\rho X)|^{-1+K+1\rho X}]$ . For example, if  $X\leftarrow 2\ 3\ 5\ 7\ 11$ , then  $2\phi X$  is equal to  $5\ 7\ 11\ 2\ 3$ , and  $^{-2}\phi X$  is equal to  $7\ 11\ 2\ 3\ 5$ . In zero-origin indexing, the definition for  $K\phi X$  becomes  $X[(\rho X)|K+1\rho X]$ .

If the rank of  $X$  exceeds 1, the coordinate  $J$ , along which rotation is to be performed, may be specified by the axis operator in the form  $Z\leftarrow K\phi[J]X$ . Moreover, the shape of  $K$  must equal the remaining dimensions of  $X$ , and each vector along the  $J$ th axis of  $X$  is rotated as specified by the

corresponding element of  $K$ . A scalar or one-element vector  $K$  is extended to conform as required.

For example, if  $\rho X$  is 3 4, and  $J$  is 2, the shape of  $K$  must be 3, and  $Z[I;]$  is equal to  $K[I]\Phi X[I;]$ . If  $J$  is 1,  $\rho K$  must be 4, and  $Z[;I]$  is equal to  $K[I]\Phi X[;I]$ . For example:

```

      M←3 4ρ1 2 3 4 ... 12
      M
1     2   3   4
5     6   7   8
9    10  11  12
      0  1  2  3  ϕ[1]M
1     6  11   4
5    10   3   8
9     2   7  12
                                     1  2  3  ϕ[2]M
                                     2  3  4   1
                                     7  8  5   6
                                     12  9 10  11
    
```

The expression  $K\Theta X$  denotes rotation along the first axis of  $X$ . The axis operator applies to  $\Theta$ , and  $K\Theta[J]X$  is equal to  $K\Phi[J]X$ .

### Catenate and Laminate

*Catenate*, denoted by a comma, chains vectors (or scalars) to form a vector. For example:

```

      X←2 3 5 7 11
      X,X
2 3 5 7 11 2 3 5 7 11
    
```

For vectors, the dimension of  $X,Y$  is equal to the total number of elements in  $X$  and  $Y$ . A non-empty numeric vector cannot be catenated with a non-empty character vector.

The axis operator applies to catenation and determines the axis along which vectors are to be catenated. In the absence of an axis operator, catenation occurs along the last axis. For example:

$M \leftarrow 3 \text{ } \rho \text{ 'ABCDEFGHI'}$   
 $M$

ABC  
 DEF  
 GHI

$M, [1]M$

$M, [2]M$

$M, M$

ABC  
 DEF  
 GHI  
 ABC  
 DEF  
 GHI

ABCABC  
 DEFDEF  
 GHIGHI

ABCABC  
 DEFDEF  
 GHIGHI

Two arrays are conformable for catenate along axis  $I$  if all *other* elements of their shapes agree. Moreover, two arrays may be catenated along axis  $I$  if they differ in rank by 1, and if the shape vector of the array of lower rank is identical to the shape vector of the array of higher rank after dropping its  $I$ th dimension. For example:

$V \leftarrow \text{'PQR'}$   
 $M, [1]V$

$M, [2]V$

$M, V$

ABC  
 DEF  
 GHI  
 PQR

ABCP  
 DEFQ  
 GHIR

ABCP  
 DEFQ  
 GHIR

A scalar argument of catenate will be replicated to form a vector, or higher rank array, as required. For example:

$C \leftarrow \text{'Ⓜ'}$   
 $C, (C, [1]M, [1]C), C$

```
ⓂⓂⓂⓂⓂ
ⓂABCⓂ
ⓂDEFⓂ
ⓂGHIⓂ
ⓂⓂⓂⓂⓂ
```

*Laminate* joins two arrays of the same rank and shape along a new axis. The position of the new axis relative to the existing axes is indicated by a fractional axis number. For example, if the new axis is to be inserted between the existing axes, 1 and 2, the axis number must have a value between 1 and 2. If the new axis is to be inserted ahead of the present first axis of the right argument, the axis number must be between 0 and 1 (or, if zero-origin indexing is used, between -1 and 0). Similarly, if the new axis is to be after the last of the present axes, the axis

**IBM Internal Use Only**

number must exceed the index of the present last axis by a fraction between 0 and 1.

The result of lamination has rank 1 greater than the rank of the arguments, and has the same shape except for the interpolation of the new axis, along which it has length 2. The comma, which normally denotes catenation, followed by an axis operator associated with a non-integral index, produces lamination. For example:

```

M←3 3ρ'ABCDEFGHI'
N←3 3ρ'123456789'
M
ABC
DEF
GHI
N
123
456
789
M,[.5]N
ABC
DEF
GHI
123
456
789
M,[1.5]N
ABC
123
DEF
456
GHI
789
M,[2.5]N
A1
B2
C3
D4
E5
F6
G7
H8
I9

```

The shapes of the preceding laminations are 2 3 3 and 3 2 3 and 3 3 2; the position of the 2 shows the point where the new axis is inserted in each case.

A scalar argument of laminate is extended as required. For example:

```

      B←2 2ρ'1234'
      B,[2.5]'x'
1x
2x

3x
4x
      ,B,[2.5]'x'
1x2x3x4x
    
```

## Transpose

The expression  $2 \ 1 \circ M$  yields the *transpose* of the matrix  $M$ ; that is, if  $R \leftarrow 2 \ 1 \circ M$ , then each element  $R[I;J]$  is equal to  $M[J;I]$ . For example:

```

      M←3 4ρ1 2 3 ... 12
      M
1  2  3  4
5  6  7  8
9 10 11 12
      2 1◊M
1  5  9
2  6 10
3  7 11
4  8 12
    
```

If  $P$  is any permutation of the indexes of the axes of an array  $A$ , then the *dyadic transpose*  $P \circ A$  is an array similar to  $A$ , except that the axes are permuted: the  $I$ th axis becomes the  $P[I]$ th axis of the result. Hence, if  $R \leftarrow P \circ A$ , then  $(\rho R)[P]$  is equal to  $\rho A$ . For example:

```

      A←2 3 5 7ρ1210
      ρA
2 3 5 7
      P←2 3 4 1
      ρP◊A
7 2 3 5
    
```

More generally,  $Q \circ A$  is a valid expression if  $Q$  is any vector equal in length to the rank of  $A$ , which is complete in the sense that if its items include any integer  $N$ , they also include all positive integers less than  $N$ . For example, if  $\rho \rho A$  is 3, then  $1 \ 2$  and  $2 \ 1$  and  $1 \ 1 \ 1$  are suitable values for  $Q$ , but  $1 \ 3 \ 1$  is not. Just as for  $P \circ A$ , where  $P$  is a permutation, the  $I$ th axis becomes the  $Q[I]$ th axis of  $Q \circ A$ . However, in this case, two or more of the axes of  $A$  may map into a single axis

of the result, thus producing a diagonal section of  $A$ , as shown by the following:

|                             |                              |
|-----------------------------|------------------------------|
| $A \leftarrow 3 \ 3\rho 19$ | $B \leftarrow 3 \ 5\rho 115$ |
| $A$                         | $B$                          |
| 1 2 3                       | 1 2 3 4 5                    |
| 4 5 6                       | 6 7 8 9 10                   |
| 7 8 9                       | 11 12 13 14 15               |
| 1 1QA                       | 1 1QB                        |
| 1 5 9                       | 1 7 13                       |

The *monadic transpose*  $QA$  reverses the order of the axes of its argument. Formally,  $QA$  is equivalent to  $(\Phi_{1\rho\rho}A)QA$ . In particular, for a matrix  $A$ , this reduces to  $2 \ 1QA$  and commonly is called the *transpose* of a matrix.

### Selection Functions

The selection functions are all dyadic. One of the arguments may be an array of any type. The other, which will be called the *selector*, because it specifies the selection to be made, must be numeric and, for `expand`, is further restricted to boolean.

### Take and Drop

The *take* function is denoted by the up arrow ( $\uparrow$ ). If  $S$  is a non-negative scalar integer, and  $V$  is a vector, then  $S\uparrow V$  results in a vector of shape  $S$ , which is obtained by taking the first  $S$  elements of  $V$  followed (if  $S > \rho V$ ) by zeros if  $V$  is numeric, and by spaces if it is not. For example:

|                           |                             |
|---------------------------|-----------------------------|
| $3\uparrow 2 \ 3 \ 5 \ 7$ | $7\uparrow 2 \ 3 \ 5 \ 7$   |
| 2 3 5                     | 2 3 5 7 0 0 0               |
| $3\uparrow 'ABCDE'$       | $(7\uparrow 'ABCDE')$ , '⊠' |
| ABC                       | ABCDE ⊠                     |

If  $S$  is a negative integer, then  $S\uparrow V$  takes elements as above, but takes the last elements of  $V$  and fills as needed on the left. The resulting vector is thus right-justified, and the original ordering of the elements is maintained. For example:

|                                 |                                 |
|---------------------------------|---------------------------------|
| $\bar{3}\uparrow 2 \ 3 \ 5 \ 7$ | $\bar{7}\uparrow 2 \ 3 \ 5 \ 7$ |
| 3 5 7                           | 0 0 0 2 3 5 7                   |



If  $A$  is any array, then  $W \uparrow A$  is valid only if the vector  $W$  has one element for each axis of  $A$ , and  $W[I]$  determines how many elements are to be taken along the  $I$ th axis of  $A$ . For example:

|                                  |                |
|----------------------------------|----------------|
| $A \leftarrow 3 \ 4 \ 1 \ 1 \ 2$ |                |
| $A$                              | $2 \ 3 \ 4$    |
| 1 2 3 4                          | 6 7 8          |
| 5 6 7 8                          |                |
| 9 10 11 12                       | $2 \ 3 \ 4$    |
| 2 3 4                            | $2 \ 3 \ 4$    |
| 5 6 7                            | 5 6 7 8 0 0    |
|                                  | 9 10 11 12 0 0 |

The function *drop* ( $\downarrow$ ) is defined similarly, except that the indicated number of elements is dropped rather than taken. For example,  $2 \ 1 \downarrow A$  is the same matrix as the result of  $2 \ 3 \uparrow A$  displayed in the preceding paragraph. If the number of elements to be dropped along any axis equals or exceeds the length of that axis, the resulting shape has a zero length for the axis.

The rank of the result of take and drop functions is the same as the length of the left argument.

### Compress, Replicate and Expand

*Compression* of  $X$  by  $U$  is denoted by the expression  $U/X$ . If  $U$  is a boolean vector, and  $X$  is a vector of the same dimension, then  $U/X$  produces a vector of  $+/U$  elements chosen from those elements of  $X$  that correspond to non-zero elements of  $U$ . For example, if  $X \leftarrow 2 \ 3 \ 5 \ 7 \ 11$  and  $U \leftarrow 1 \ 0 \ 1 \ 1 \ 0$ , then  $U/X$  is  $2 \ 5 \ 7$ , and  $(\sim U)/X$  is  $3 \ 11$ .

```

C ← 'THIS IS AN EXAMPLE'
D ← C ≠ ' '
C1 ← D / C
C1
THISISANEXAMPLE

```

If  $U$  is all zeros, then  $U/X$  is an empty vector.

To be conformable, the dimensions of the arguments must agree, except that a scalar or one-element vector argument on

the left, or a scalar on the right, is extended. So,  $1/X$  and  $(, 1)/X$  are equal to  $X$ .

*Replicate* is an extension of *compress* allowing a non-boolean left argument:

$$Z \leftarrow U/X$$

where  $X$  may be any array.  $U$  must be a scalar or a vector of integers.  $Z$  is an array with the same rank as  $X$ , but with each sub-array along the last axis replicated according to the format indicated by  $U$ .  $\bar{1} \uparrow \rho Z$  is  $\bar{1} \uparrow \rho X$ .

If  $U$  is a scalar or one element vector, it will be extended to  $\bar{1} \uparrow \rho X$  elements before application of the function. If  $X$  is a scalar, then it will be extended to a vector of as many elements as  $+/U \geq 0$ . In any other case,  $\bar{1} \uparrow \rho X$  must be equal to  $+/U \geq 0$ .

Non-negative elements of  $U$  correspond to sub-arrays of  $X$  along its last axis. If  $U[I]$  (an element of  $U$ ) is non-negative, then the corresponding subarray of  $X$  will be replicated  $U[I]$  times. If  $U[I]$  is negative, then  $Z$  is filled with  $|U[I]|$  fill elements (0 if  $X$  is numeric, spaces if  $X$  is literal). If  $U$  is not extended, then  $\bar{1} \uparrow \rho Z$  is  $+/|U$ .

Examples:

```

      2 / 1 2 3
1 1 2 2 3 3
      2 3 / 4
4 4 4 4 4
      1 0 2 3 / 1 2 3 4
1 3 3 4 4 4
      1 1 2 0 0 0 1 / 'MERCURY'
MERRY
      1 0 2  $\bar{1}$  3  $\bar{2}$  / 1 2 3 4
1 3 3 0 4 4 4 0 0
      0 4 0 1 / 3 4  $\rho$  1 1 2
      2 2 2 2 4
      6 6 6 6 8
     10 10 10 10 12
    
```

*Expansion* is the converse of *compression* and is denoted by  $U \setminus X$ . If  $Y \leftarrow U \setminus X$ , then  $U/Y$  is equal to  $X$  and  $(\sim U)/Y$  is an

array of zeros or spaces, depending on whether  $X$  is numeric or character. In other words,  $U \setminus X$  expands  $X$  to the format indicated by the ones in  $U$  and fills in zeros or spaces. To be conformable,  $+ / U$  must equal  $\rho X$ . Continuing our previous example:

$D \setminus C1$   
**THIS IS AN EXAMPLE**

The axis operator applies to compress, replicate and expand and determines the axis along which they apply. If the axis operator is omitted, the last axis is used. The symbols  $\neq$  and  $\setminus$  also denote compression (or replication) and expansion, but when used without an axis operator, apply along the first axis. For example:

|             |                     |                         |                                 |
|-------------|---------------------|-------------------------|---------------------------------|
|             | $Q \leftarrow 3$    | $4 \rho 'ABCDEFGHIJKL'$ |                                 |
|             | $Q$                 |                         |                                 |
| <b>ABCD</b> |                     |                         |                                 |
| <b>EFGH</b> |                     |                         |                                 |
| <b>IJKL</b> |                     |                         |                                 |
|             | $0 \ 1 \ 1 \ 0 / Q$ |                         | $1 \ 1 \ 0 \ 1 \setminus [1] Q$ |
| <b>BC</b>   |                     | <b>ABCD</b>             |                                 |
| <b>FG</b>   |                     | <b>EFGH</b>             |                                 |
| <b>JK</b>   |                     |                         |                                 |
|             | $1 \ 0 \ 1 / [1] Q$ | <b>IJKL</b>             |                                 |
| <b>ABCD</b> |                     |                         | $1 \ 1 \ 0 \ 1 \setminus Q$     |
| <b>IJKL</b> |                     | <b>ABCD</b>             |                                 |
|             | $1 \ 0 \ 1 \neq Q$  | <b>EFGH</b>             |                                 |
| <b>ABCD</b> |                     | <b>IJKL</b>             |                                 |
| <b>IJKL</b> |                     |                         |                                 |
|             | $2 \ 0 \ 1 / [1] Q$ |                         |                                 |
| <b>ABCD</b> |                     |                         |                                 |
| <b>ABCD</b> |                     |                         |                                 |
| <b>IJKL</b> |                     |                         |                                 |
|             | $2 \ 0 \ 1 \neq Q$  |                         |                                 |
| <b>ABCD</b> |                     |                         |                                 |
| <b>ABCD</b> |                     |                         |                                 |
| <b>IJKL</b> |                     |                         |                                 |

If the right argument is a scalar, the result is a vector; otherwise, the rank of the result of compress, replicate or expand equals the rank of the right argument.

## Indexing

*Indexing* may be either 0-origin or 1-origin, as discussed in "Arrays" on page 4-10. The following discussion assumes 1-origin. If  $X$  is a vector and  $I$  is a scalar, then  $X[I]$  denotes the  $I$ th element of  $X$ . For example, if  $X \leftarrow 2 \ 3 \ 5 \ 7 \ 11$  then  $X[2]$  is 3.

If the index  $I$  is a vector, then  $X[I]$  is the vector obtained by selecting from the elements indicated by successive components of  $I$ . For example,  $X[1 \ 3 \ 5]$  is  $2 \ 5 \ 11$  and  $X[5 \ 4 \ 3 \ 2 \ 1]$  is  $11 \ 7 \ 5 \ 3 \ 2$ . If the elements of  $I$  do not belong to the set of indexes of  $X$ , the expression  $X[I]$  causes an *index error* report.

In general  $\rho X[I]$  equals  $\rho I$ . In particular, if  $I$  is a scalar, then  $X[I]$  is a scalar, and if  $I$  is a matrix, then  $X[I]$  is a matrix. For example:

```

A ← 'ABCDEFG'
I ← 4 3 1 4 2 1 4 4 1 2 4 1 4
I
3 1 4          CAD
2 1 4          BAD
4 1 2          DAB
4 1 4          DAD
A[I]

```

If  $M$  is a matrix, it is indexed by a two-part list of the form  $I;J$ , where  $I$  selects the row (or rows), and  $J$  selects the column (or columns). For example:

```

M ← 3 4 1 12
M
1 2 3 4      7      M[2;3]
5 6 7 8
9 10 11 12
           2 3 4
          10 11 12

```

In general,  $\rho M[I;J]$  is equal to  $(\rho I), \rho J$ . Hence, if  $I$  and  $J$  are both vectors, then  $M[I;J]$  is a matrix; if both  $I$  and  $J$  are scalars,  $M[I;J]$  is a scalar; if  $I$  is a vector and  $J$  is a scalar (or vice versa),  $M[I;J]$  is a vector. The indexes are not limited to vectors, but may be of higher rank. For example, if  $I$  is a 3-by-4 matrix, and  $J$  is a vector of dimension 6, then  $M[I;J]$  is of dimension 3 4 6, and  $M[J;I]$  is of

dimension 6 3 4. In particular, if  $T$ ,  $P$ , and  $Q$  are matrices, and if  $R \leftarrow T[P;Q]$ , then  $R$  is an array of rank 4, and  $R[I;J;K;L]$  is equal to  $T[P[I;J];Q[K;L]]$ .

The form  $M[I;]$  indicates that all columns are selected; the form  $M[;J]$  indicates that all rows are selected. For example,  $M[2;]$  is 5 6 7 8, and  $M[;2 1]$  is the matrix with rows 2 1 and 6 5 and 10 9

The following example shows a matrix indexing a matrix to get a three-dimensional array:

|                  |            |   |   |   |   |   |   |   |         |
|------------------|------------|---|---|---|---|---|---|---|---------|
| $M \leftarrow 2$ | $4 \rho 3$ | 1 | 4 | 2 | 1 | 4 | 4 | 1 |         |
| $M$              |            |   |   |   |   |   |   |   | $M[;M]$ |
| 3                | 1          | 4 | 2 |   |   |   |   |   | 4       |
| 1                | 4          | 4 | 1 |   |   |   |   |   | 3       |
|                  |            |   |   |   |   |   |   |   | 4       |
|                  |            |   |   |   |   |   |   |   | 1       |
|                  |            |   |   |   |   |   |   |   | 1       |
|                  |            |   |   |   |   |   |   |   | 1       |
|                  |            |   |   |   |   |   |   |   | 1       |

An indexed variable may appear to the left of a specification arrow if (1) the expression is executable in the environment, and (2) the values of the expression on the left and right are denoted by  $L$  and  $R$ , then  $1 = \times / \rho R$  or  $(1 \neq \rho L) / \rho L$  must equal  $(1 \neq \rho R) / \rho R$ . For example:

|                       |   |   |   |    |
|-----------------------|---|---|---|----|
| $X \leftarrow 2$      | 3 | 5 | 7 | 11 |
| $X[1 3] \leftarrow 6$ | 8 |   |   |    |
| $X$                   |   |   |   |    |
| 6                     | 3 | 8 | 7 | 11 |

## Selector Generators

All functions in this group have integer results which, although they are commonly useful as the selector argument in selection functions, are often used in other ways as well. For example, the grade-up function ( $\Delta$ ) is commonly used to produce indexes needed to order a vector into ascending order (as in  $X[\Delta X]$ ), but may also be used in the treatment of permutations as the inverse function; that is,  $\Delta P$  yields the permutation inverse to  $P$ . Similarly,  $1N$  generates a vector of

$N$  successive indexes, but  $0.1 \times 1N$  generates a grid of values with an interval of 0.1.

## Index Generator and Index Of

The *index generator*  $\mathbf{1}$  applies to a non-negative scalar integer  $N$  to produce a vector of length  $N$  that contains the first  $N$  integers in order, beginning with the value of the index origin  $\square IO$ . For example,  $\mathbf{1}5$  yields  $1\ 2\ 3\ 4\ 5$  (in 1-origin) or  $0\ 1\ 2\ 3\ 4$  (in 0-origin), and  $\mathbf{1}0$  yields an empty vector. A one-element array argument is treated as a scalar.

The *index of* function is dyadic. If  $V$  is a vector and  $S$  is a scalar, then  $V\mathbf{1}S$  yields the index (in the origin in force) of the earliest occurrence of  $S$  in  $V$ ; that is, the index of  $S$  in  $V$ . If  $S$  differs from all items of  $V$ , then  $\mathbf{1}S$  yields the first index outside the range of  $V$ ; that is,  $\square IO + \rho V$ .

If  $S$  is any array, then  $V\mathbf{1}S$  yields an array with the shape of  $S$ , each item being determined as the index in  $V$  of the corresponding item of  $S$ . For example:

```

      A ← 'ABCDEFGHIJKLMNOPQRSTUVWXYZ '
      J ← A  $\mathbf{1}$  'HEAD CHIEF'
      J
      8 5 1 4 27 3 8 9 5 6
      A[J]
      HEAD CHIEF
      A[ $\Phi$ J]
      FEIHC DAEH
      A  $\mathbf{1}$  'VAR3'
      22 1 18 28

      M ← 2 5  $\rho$  'HEAD CHIEF'
      M
      HEAD
      CHIEF
      A  $\mathbf{1}$  M
      8 5 1 4 27
      3 8 9 5 6

```

## Membership

The *membership* function,  $X \in Y$ , yields a boolean array of the same shape as  $X$ . Any particular element of  $X \in Y$  has the value 1 if the corresponding element of  $X$  belongs to  $Y$ ; that is, if it occurs as some element of  $Y$ . For example,  $(17) \in 3\ 5$  is equal to  $0\ 0\ 1\ 0\ 1\ 0\ 0$  and  $'ABCDEFGH' \in 'COFFEE'$  equals  $0\ 0\ 1\ 0\ 1\ 1\ 0\ 0$ . The right argument  $Y$  may be of any rank.

The selector argument of compression is commonly given by applying the membership function, alone or in combination with the scalar boolean and relational functions.

## Grade Functions

The *grade-up* function,  $\Delta V$ , grades the items of vector  $V$  in ascending order; that is, it yields a result of the same dimension as  $V$  whose first item is the index (in the origin in force) of the smallest item of  $V$ , whose second item is the index of the next smallest item, and so on. Consequently,  $V[\Delta V]$  yields the elements of  $V$  in ascending order. For example, if  $V \leftarrow 8\ 3\ 7\ 5$ , then  $\Delta V$  is  $2\ 4\ 3\ 1$ , and  $V[\Delta V]$  is  $3\ 5\ 7\ 8$ .

If the items of  $V$  are not all distinct, the ranking among any set of equal elements is determined by their position. For example,  $\Delta 4\ 3\ 1\ 3\ 4\ 2$  yields  $3\ 6\ 2\ 4\ 1\ 5$ .

The *grade-down* function,  $\Psi V$ , grades the items of  $V$  in descending order. Among equal elements, the ranking is determined by position, just as for grade-up. Consequently,  $\Psi V$  equals the reversal of  $\Delta V$  only if the items of  $V$  are distinct. For example:

|       |                               |       |                                 |
|-------|-------------------------------|-------|---------------------------------|
|       | $A \leftarrow 7\ 2\ 5\ 11\ 3$ |       | $B \leftarrow 4\ 3\ 1\ 3\ 4\ 2$ |
|       | $\Delta A$                    |       | $\Delta B$                      |
| 2 5 3 | 1 4                           | 3 6 2 | 4 1 5                           |
|       | $\Psi A$                      |       | $\Psi B$                        |
| 4 1 3 | 5 2                           | 1 5 2 | 4 6 3                           |

The monadic grade functions apply only to numeric vectors.

**Grade Down (Dyadic):**  $Z \leftarrow L \Psi R$

$R$  may be any non-scalar character array, as may  $L$ .  $Z$  is an integer vector of shape  $1 \uparrow \rho R$ , containing the permutation of  $1 \uparrow \rho R$  that puts the sub-arrays along the first axis of  $R$  in non-ascending order according to the collating sequence  $L$ .

Grading works by searching in  $L$  (in row-major order) for each element of  $R$ , and then attaching a significance dependent on where it was first found. The significance depends on both the location and the rank of  $L$ .

Any elements of  $R$  not found in  $L$  have collating significance as if they were found immediately past the end of  $L$ .  $Z$  leaves the order among elements of equal collating significance undisturbed.

Examples:

```

      'ABCDE' Ψ 'DEAL'
4 2 1 3

      R ← 5 4ρ 'DEALLEADDEADDEEDDALE'
      R
DEAL
LEAD
DEAD
DEED
DALE

      'ABCDE' Ψ R
2 4 1 3 5

```

The last axis of  $L$  is the most significant for grading, and the first axis of  $L$  is the least significant. Thus, in the following example, differences in spelling have higher significance than differences in case:



```

      R ← 5 4ρ 'dealDealdeadDeadDEED'
      R
deal
Deal
dead
Dead
DEED

      L ← 2 5ρ 'abcdeABCDE'
      L
abcde
ABCDE

      Z ← L ∇ R
      Z
5 2 1 4 3
      R[Z;]
DEED
Deal
deal
Dead
dead

```

∇IO is an implicit argument of dyadic grade down.

### Grade Up (Dyadic): $Z \leftarrow L \uparrow R$

$R$  may be any non-scalar character array, as may  $L$ .  $Z$  is an integer vector of shape  $1 \uparrow \rho R$ , containing the permutation of  $1 \uparrow \rho R$  that puts the sub-arrays along the first axis of  $R$  in non-descending order according to the collating sequence  $L$ .

Grading works by searching in  $L$  (in row-major order) for each element of  $R$ , and then attaching a significance dependent on where it was first found. The significance depends on both the location and the rank of  $L$ . Any elements of  $R$  not found in  $L$  have collating significance, as if they were found immediately past the end of  $L$ .  $Z$  leaves the order among elements of equal collating significance undisturbed.

```

      'ABCDE' Δ 'DEAL'
3 1 2 4
R ← 5 4p 'DEALLEADDEADDEEDDALE'
R
DEAL
LEAD
DEAD
DEED
DALE
      'ABCDE' Δ R
5 3 1 4 2

```

The last axis of *L* is the most significant for grading, and the first axis of *L* is the least significant. Thus, in the following example, differences in spelling have higher significance than differences in case:

```

      R ← 5 4p 'dealDealdeadDeadDEED'
R
deal
Deal
dead
Dead
DEED
      L ← 2 5p 'abcdeABCDE'
L
abcde
ABCDE
      Z ← L Δ R
Z
3 4 1 2 5
R[Z;]
dead
Dead
deal
Deal
DEED

```

□*IO* is an implicit argument of dyadic grade up.

## Deal

The *deal* function,  $M?N$ , produces a vector of length  $M$ , which is obtained by making  $M$  (pseudo-) random selections, without replacement, from the population  $1N$ . Both arguments are limited to scalars or one-element vectors. Each selection is made by appropriate application of the scheme described for the function *roll*.

The expression,  $N?N$ , yields a random permutation of the items of  $1N$ . The expression,  $P[M?ρP]$ , selects  $M$  distinct elements from the population defined by the items of a vector  $P$ . For example:

```

)CLEAR
CLEAR WS
      P←'ABCDEFGH'
      P[3?ρP]
BGE          P[(ρP)?ρP]
            ECBGDHAF

```

## Numeric Functions

The numeric mixed functions apply only to numeric arguments and produce numeric results.

## Matrix Inverse and Matrix Divide

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

When applied to a non-singular matrix  $A$ , the expression,  $\boxtimes A$  (*matrix inverse*), yields the inverse of  $A$ , and  $X←B\boxtimes A$  (*matrix divide*) yields a value of  $X$  that satisfies the relation

$\wedge/, B=A+. \times X$  and is therefore the solution of the system of linear equations conventionally represented as  $AX=B$ .

For example:

|                                     |   |   |   |               |    |    |   |                          |   |   |   |
|-------------------------------------|---|---|---|---------------|----|----|---|--------------------------|---|---|---|
| $A \leftarrow (14) \circ . \geq 14$ |   |   |   |               |    |    |   |                          |   |   |   |
| $A$                                 |   |   |   | $\boxtimes A$ |    |    |   | $A+. \times \boxtimes A$ |   |   |   |
| 1                                   | 0 | 0 | 0 | -1            | 0  | 0  | 0 | 1                        | 0 | 0 | 0 |
| 1                                   | 1 | 0 | 0 | -1            | -1 | 0  | 0 | 0                        | 1 | 0 | 0 |
| 1                                   | 1 | 1 | 0 | 0             | -1 | -1 | 0 | 0                        | 0 | 1 | 0 |
| 1                                   | 1 | 1 | 1 | 0             | 0  | -1 | 1 | 0                        | 0 | 0 | 1 |

|   |    |   |    |                  |    |  |  |                             |   |  |  |
|---|----|---|----|------------------|----|--|--|-----------------------------|---|--|--|
| $B \leftarrow 1 \ 3 \ 6 \ 10$                             |    |   |    |                  |    |  |  |                             |   |  |  |
| $X \leftarrow B \boxtimes A$                              |    |   |    | $A+. \times X$   |    |  |  | $(\boxtimes A) +. \times B$ |   |  |  |
| $B$   |    |   |    | $1 \ 3 \ 6 \ 10$ |    |  |  | $1 \ 2 \ 3 \ 4$             |   |  |  |
| 1   | 3  | 6 | 10 |                  |    |  |  |                             |   |  |  |
| $X$   |    |   |    | $1 \ 2 \ 3 \ 4$  |    |  |  | $1 \ 2 \ 3 \ 4$             |   |  |  |
| 1   | 2  | 3 | 4  |                  |    |  |  |                             |   |  |  |
| $C \leftarrow 4 \ 2 \rho 1 \ 2 \ 3 \ 5 \ 6 \ 9 \ 10 \ 14$ |    |   |    |                  |    |  |  |                             |   |  |  |
| $Y \leftarrow C \boxtimes A$                              |    |   |    | $A+. \times Y$   |    |  |  | $(\boxtimes A) +. \times C$ |   |  |  |
| $C$   |    |   |    | $1 \ 2$          |    |  |  | $1 \ 2$                     |   |  |  |
| 1   | 2  |   |    |                  |    |  |  |                             |   |  |  |
| 3   | 5  |   |    |                  |    |  |  |                             |   |  |  |
| 6   | 9  |   |    |                  |    |  |  |                             |   |  |  |
| 10  | 14 |   |    |                  |    |  |  |                             |   |  |  |
| $Y$   |    |   |    | $1 \ 2$          |    |  |  | $1 \ 2$                     |   |  |  |
| 1   | 2  |   |    | 1                | 2  |  |  | 1                           | 2 |  |  |
| 2   | 3  |   |    | 3                | 5  |  |  | 2                           | 3 |  |  |
| 3   | 4  |   |    | 6                | 9  |  |  | 3                           | 4 |  |  |
| 4   | 5  |   |    | 10               | 14 |  |  | 4                           | 5 |  |  |

The last example above shows that if the left argument is a matrix  $C$ , then  $C \boxtimes A$  yields a solution of the system of equations for each column of  $C$ .

If  $A$  is non-singular, and  $I$  is an identity matrix of the same dimension, then the matrix inverse  $\boxtimes A$  is equivalent to the matrix divide  $I \boxtimes A$ . More generally, for any matrix  $P$ , the expression  $\boxtimes P$  is equivalent to the expression  $((1R) \circ . = 1R) \boxtimes P$ , where  $R$  is the number of rows in  $P$ .

The domino functions apply more generally to non-square matrices, and to vectors and scalars; any argument of rank greater than 2 is rejected (*RANK ERROR*). For matrix arguments  $A$  and  $B$ , the expression  $X \leftarrow B \boxtimes A$  is executed only if:

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

2. The columns of  $A$  are linearly independent.

If  $X \leftarrow B \mathbb{Q} A$  is executable, then  $\rho X$  is equal to  $(1 \downarrow \rho A), 1 \downarrow \rho B$ , and  $X$  is determined so as to minimise the value of  $+ / (B - A + . \times X) * 2$ .

The domino functions apply to vector and scalar arguments as follows, except that:

1. The shape of the result is determined as specified above.
2. A vector is treated as a one-column matrix.
3. A scalar is treated as a one-by-one matrix.

The reasoning for this interpretation of a vector as a one-column (rather than one-row) matrix is that the right argument is treated *geometrically* (as will be seen in a later example) as defining a space spanned by its column vectors, and the left argument was seen (in an earlier example) to be treated so as to yield a solution for each of its column vectors. Indeed, a one-row matrix, right argument (unless 1-by-1) would be rejected under condition 2 above.

For scalar arguments  $X$  and  $Y$ , the expression  $\mathbb{Q} Y$  is equivalent to  $\div Y$  and, except that it yields a domain error for  $0 \mathbb{Q} 0$ , the expression,  $X \mathbb{Q} Y$ , is equivalent to  $X \div Y$ .

The use of  $\mathbb{Q}$  for a non-square right argument can be illustrated as follows: if  $X$  is a vector, and  $Y \leftarrow F X$ , then  $Y \mathbb{Q} X \circ . * 0, 1 D$  yields the coefficients of the polynomial of degree  $D$ , which best fits (in the least-square sense) the function  $F$  at the points,  $X$ .

The definition of  $B \mathbb{Q} A$  has certain useful geometric interpretations. If  $B$  is a vector, and  $A$  is a matrix, then saying that  $+ / (B - A + . \times B \mathbb{Q} A) * 2$  is a minimum, is equivalent to saying that the length of vector  $B - A + . \times B \mathbb{Q} A$  is a minimum. But  $A + . \times B \mathbb{Q} A$  is a point in the space spanned by the column vectors of  $A$ , and is therefore the point in this space that is closest to  $B$ . In other words,  $P \leftarrow A + . \times B \mathbb{Q} A$  is the projection of  $B$  on the space spanned by the columns of  $A$ . Moreover, the

vector  $B-P$  must be normal to every vector in the space; in particular,  $(B-P) \cdot A$  is a zero vector.

If  $A$  and  $B$  are single-column matrices, then  $B \cdot A$  is a 1-by-1 matrix, and  $A \cdot B \cdot A$  is equivalent to  $A \cdot S$ , where  $S$  is the scalar  $B \cdot A$ . If  $A$  and  $B$  are vectors, then  $B \cdot A$  is a scalar, and the projection of  $B$  on  $A$  is therefore given by the simpler expression,  $A \cdot B \cdot A$ . For example:

$$\begin{aligned}
 A &\leftarrow 4.5 \quad 1.7 \\
 B &\leftarrow 2 \quad 5 \\
 P &\leftarrow A \cdot B \cdot A \\
 P & \\
 3.403197926 \quad 1.28565255 \\
 N &\leftarrow B - P \\
 N & \\
 -1.403197926 \quad 3.71434745 \\
 N &\cdot A \\
 3.552713679E^{-15}
 \end{aligned}$$

Similar analysis shows that if  $A$  is a vector, then  $A \cdot A$  is a vector in the direction of  $A$ ; that is,  $A \cdot A$  is equal to  $S \cdot A$  for some scalar  $S$ . Moreover,  $A \cdot A \cdot A$  is equal to 1. In other words,  $A \cdot A$  is the image of vector  $A$  obtained by inversion in the unit circle (or sphere).

## Decode and Encode

For vectors  $R$  and  $X$ , the *decode* (or *base-value*) function  $R \cdot X$  yields the value of the vector  $X$  evaluated in a number system with radices  $R[1], R[2], \dots, R[\rho R]$ . For example, if  $R \leftarrow 24 \quad 60 \quad 60$ , and  $X \leftarrow 1 \quad 2 \quad 3$  is a vector of elapsed time in hours, minutes, and seconds, then  $R \cdot X$  has the value 3723, and is the corresponding elapsed time in seconds. Similarly,  $10 \quad 10 \quad 10 \quad 10 \cdot 1 \quad 7 \quad 7 \quad 6$  is equal to 1776, and  $2 \quad 2 \quad 2 \quad 1 \cdot 1 \quad 0 \quad 1$  is equal to 5. Formally,  $R \cdot X$  is equal to  $\sum W \cdot X$ , where  $W$  is the weighting vector determined as follows:  $W[\rho W]$  is equal to 1 and  $W[I-1]$  is equal to  $R[I] \cdot W[I]$ . For example, if  $R$  is 24 60 60, then  $W$  is 3600 60 1.

Scalar (or one-element vector) arguments are extended to conform, as required. For example,  $10 \cdot 1 \quad 7 \quad 7 \quad 6$  yields 1776. The arguments are not restricted to integers; for

example, if  $X$  is a scalar, then  $X \downarrow C$  is the value of the polynomial, with coefficients  $C$  arranged in descending order on the powers of  $X$ .

The decode function is extended to arrays in the manner of the inner product: each of the radix vectors along the last axis of the first argument is applied to each of the vectors along the first axis of the second argument. There is one difference; if either of these distinguished axes is of length 1, it will be extended as necessary (by replication of the element) to match the length of the other argument. Except for this different treatment of unit axes, the shape of the result of  $A \downarrow B$  is determined as the shape of the inner product, namely  $(\uparrow 1 \downarrow p A), 1 \downarrow p B$ .

The *encode* or *representation* function  $R \uparrow X$  is, for certain arguments, inverse to the decode function. For example:

```

      R←10 10 10 10
      R↓1 7 7 6
1776
      R↑1776
1 7 7 6

```

For a radix  $R$  having positive integer elements,  $R \downarrow (R \uparrow X)$  equals  $(\times/R) \downarrow X$  rather than  $X$ . For example:

```

      10 10 10 10↑123
0 1 2 3
      10 10↑123
2 3
      10 10 10↑123
      1 2 3
      10↑123
      3

```

More precisely, the definition of the encode function is based on the definition of the *residue* function; for a vector left argument and scalar right argument, encode is equivalent to the function, **ENCODE**, whose representation is shown at the left below:

```

Z←A ENCODE B;I          2 2 2T13
Z←0×A                   1 0 1
I←ρA                     -2 -2 -2T13
L:→(I=0)/0              -1 -1 -1
Z[I]←A[I]|B             2 0 2T13
→(A[I]=0)/0              0 6 1
B←(B-Z[I])÷A[I]         2 2 2T-13
I←I-1                    0 1 1
→L                        -2 2 -2T13
                          0 1 -1
    
```

The basic definition of  $R\tau X$  concerns a vector  $R$  and a scalar  $X$ , and produces a result of the shape of  $R$ . It is extended to arrays as follows: each radix vector along the first axis of  $R$  is applied to get the representation of each item of  $X$ , the resulting representations being arrayed along the first axis of the result. For example:

```

      10 10 10T215 486 72 219 3
2 4 0 2 0
1 8 7 1 0
5 6 2 9 3
      R←10 10 10,[1.5]8 8 8
      R
10 8
10 8
10 8
      RT123
1 1
2 7
3 3
    
```

The expression for the shape of the result of  $R\tau X$  is the same as for the shape of the outer product, namely  $(\rho R), \rho X$ .

## Data Transformations

Of the two functions in this class, the format is a true type transformation, being designed to produce a character array that represents the data in its numeric argument. Over a certain class of arguments, the execute function is inverse to the format and is therefore considered as a type transformation as well, although its applicability is, in fact, much broader.



## Execute and Format

Any character vector or scalar can be regarded as a representation of an APL statement (which may or may not be well-formed). The monadic function denoted by  $\Phi$  (execute) takes as its argument, a character vector or scalar, and *evaluates* or *executes* the APL statement it represents. When applied to an argument that might be interpreted as a system command or the opening of function definition, an error will necessarily result when evaluation is attempted, because neither of these is a well-formed APL statement.

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

```

       $\Phi$  ' '
      Z $\leftarrow$  $\Phi$  ' '
VALUE ERROR
      Z $\leftarrow$  $\Phi$  ' '
      ^

```

The domain of  $\Phi$  is any character array of rank less than 2, and *RANK* and *DOMAIN* errors are reported in the usual way:

```

      C $\leftarrow$ '3 4'
      +/ $\Phi$ C
7
       $\Phi$ 1 3pC
RANK ERROR
       $\Phi$ 1 3pC
      ^
       $\Phi$ 3 4
DOMAIN ERROR
       $\Phi$ 3 4
      ^

```

An error can also occur in the attempted execution of the APL expression represented by the argument of  $\Phi$ ; such an indirect error is reported by the error type prefaced by the symbol  $\Phi$  and followed by the character string and the caret marking the point of difficulty. For example:

```

    ⚡'4⚡0'
⚡ DOMAIN ERROR
    4⚡0
    ^
    ⚡')WSID'
⚡ VALUE ERROR
    )WSID
    ^

```

The symbol ⚡ denotes two format functions, which convert numeric arrays to character arrays. These functions have several significant uses, besides the obvious one for composing tabular output. For example, the use of *format* is complementary to the use of *execute* in treating bulk input and output, and in the management of combined alphabetic and numeric data.

The monadic format function produces a character array that will display the same as the display normally produced by its argument, but makes this character array explicitly available. For example:

```

    )CLEAR
CLEAR WS
    M←2= ?4 4ρ2
    R←⚡M
    M
    R
    0 1 0 1          0 1 0 1
    0 0 1 1          0 0 1 1
    1 0 1 1          1 0 1 1
    0 0 1 1          0 0 1 1
    ρM              ρR
4 4                4 8
    R[;2×14]        ρ⚡2 5
0101              3
0011              ^/,R=⚡R
1011              1
0011              ⚡'ABCD'
                  ABCD
    X←34
    'THE VALUE OF X IS ',⚡X
THE VALUE OF X IS 34

```

The monadic format function applied to a *character* array yields the array unchanged, as shown by the last two examples. For a *numeric* array, the shape of the result is the same as the shape of the argument, except for the required expansion along the last coordinate, with each number going, in general, to

several characters. The format of a scalar number is always a vector.

The printing normally produced by APL systems may vary slightly from system to system, but the result produced by the monadic format will have no final column of all spaces, and no initial spaces for a vector or scalar argument.

The dyadic format function accepts only numeric arrays as its right argument, and uses variations in the left argument to provide progressively more detailed control over the result. Thus, for  $Z \leftarrow L \overline{\text{P}} R$ , the argument  $L$  may be a numeric or character vector.

If numeric,  $L$  may be a single number, a pair of numbers, or a vector of length  $2 \times^{-1} 1 \uparrow 1, \rho R$ . In general, a pair of numbers controls the result: the first determines the total *width* of a number field, and the second sets the precision. For decimal form, the precision is specified as the number of digits to the right of the decimal point, and for scaled form, it is specified as the number of digits in the multiplier. The form to be used is determined by the sign of the precision indicator, with negative numbers indicating scaled form. Thus:

```

      A←3 2ρ12.34 -34.567 0 12 -0.26 -123.45
      ρ□←A
12.34          -34.567
  0           12
-0.26         -123.45
3 2
      ρ□←12 3ϕA
12.340        -34.567
  .000         12.000
  -.260        -123.450
3 24
      R←9 2ϕA
      S←9 -2ϕA
      ρ□←R
12.34         -34.57          12 ρ□←6 0ϕA
  .00         12.00          0    35
  -.26        -123.45        0    12
3 18          3 12          0   -123
      ρ□←S
1.2E001      3.5E001          1E001      3E001
  0.0E000    1.2E001          0E000      1E001
-2.6E-001   -1.2E002         -3E-001  -1E002
3 18          3 14

```

## IBM Internal Use Only

If the width indicator of the control pair is 0, a field width is chosen so that at least one space will be left between adjoining numbers. If only a single control number is used, it is treated as a number pair with a width indicator of 0:

|  |  |
|--|--|
| <pre>       ρ□←2ϕA 12.34  34.57   .00  12.00  -.26 -123.45 3 15       ρ□←0 2ϕA 12.34  34.57   .00  12.00  -.26 -123.45 3 15 </pre> | <pre>       ρ□←-2ϕA   1.2E001  3.5E001  -0.0E000  1.2E001  -2.6E-001 -1.2E002 3 20       ρ□←0 -2ϕA   1.2E001  3.5E001  -0.0E000  1.2E001  -2.6E-001 -1.2E002 3 20 </pre> |
|--|--|

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

|   |  |
|---|--|
| <pre>       ρ□←0 2 0 2ϕA 12.34  34.57   .00  12.00  -.26 -123.45 3 15       ρ□←6 2 12 -3ϕA 12.34  3.46E001   .00  1.20E001  -.26  -1.23E002 3 18 </pre> | <pre>       ρ□←8 3 0 2ϕA 12.340  34.57   .000  12.00  -.260 -123.45 3 16       ρ□←8 0 0 -2ϕA   12  3.5E001    0  1.2E001    0 -1.2E002 3 18 </pre> |
| <pre>       6 2 8 3 3 0 4 0 5 0 12 4ϕ,A 12.34 -34.567 0 12 0 123.4500 </pre>  |  |

The format function applied to an array of rank greater than 2 applies to each of the planes defined by the last two axes. For example:

```

)CLEAR
)CLEAR WS
  L←2=?2 2 5ρ2
  L
0 1 0 1 0      .0 1.0 .0 1.0 .0
0 1 1 1 0      .0 1.0 1.0 1.0 .0

1 1 0 0 1      1.0 1.0 .0 .0 1.0
1 0 0 0 0      1.0 .0 .0 .0 .0

```

Fabular displays incorporating row and column headings, or other information between columns or rows, are easily set up using the format function and catenation. For example:

```

)CLEAR
CLEAR WS
ROWS←4 3ρ'JANAPRJULOCT'
YEARS←75+14
TBL←.001×-4E5+?4 4ρ8E5
(' ', [1]ROWS), (2φ9 0φYEARS), [1]9 2φTBL
      76      77      78      79
JAN   -294.77   -204.48   -33.08    26.21
APR   -224.83   -362.36   143.09   143.44
JUL   347.75    -93.20    15.53   264.77
OCT   -372.34   -357.23    23.76   136.92

```

The left argument of format has obvious restrictions, because the width of a field must be large enough to hold the requested form. If the specified width is inadequate, the result will be a *DOMAIN* error. However, the width does not have to provide open spaces between adjoining numbers. For example, boolean arrays can be tightly packed:

```

)CLEAR
CLEAR WS
      1 0φ2=?4 4ρ2
0101
0011
1011
0011

```

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

- The least width needed for a column of numbers *C* with precision *P* is:

$$W \leftarrow (\vee/R < 0) + (\sim P \in 0 \text{ } ^{-1}) + (|P|) + (5, \lceil /0, (R \neq 0) + \lfloor 10 \otimes |R+R=0 \rfloor [1+P \geq 0])$$

where *R* is the rounded value of *C* given by

$$R \leftarrow (\lfloor .5 + C \times 10 * |P| \rfloor) \div 10 * |P|$$

- The expressions,  $(M \phi A)$ ,  $N \phi B$  and  $(M, N) \phi A, B$ , are equivalent if *M* and *N* are full control vectors; that is, if  $((\rho M) = 2 \times ^{-1} \uparrow \rho A) \wedge (\rho N) = 2 \times ^{-1} \uparrow \rho B$ . If  $2 = \rho M$ , then  $(M \phi A)$ ,  $M \phi B$  and  $M \phi A, B$  are equivalent.

## Picture Format

If the left argument  $L$  is a character vector, it is a pattern for the result  $Z$ . The length of the last dimension of  $Z$  will be an exact multiple of the length of  $L$ , and numbers in  $R$  will appear in numerical field positions shown in the pattern, along with different kinds of decorations. Formally,  $\text{format}(R, L)$  will equal  $K \times L$ , where  $K$  is an integer. If  $L$  has more than one numerical field, then  $K$  will be 1. The system variable `FC` is an implicit argument of picture format.

A numerical field is defined as a sequence of characters bounded by blanks and containing at least one decimal digit (numeric character). The digits appearing in a field are both place holders and control characters for that field. Non-digits in the pattern are decorators, which fall into three classes: simple, controlled, or conventional.

A simple decoration may be imbedded in a numerical field or stand alone. Such a decoration always appears in the result in the same relative position as in the pattern, regardless of the numerical values being formatted.

A candidate for a controlled decoration is one that is immediately adjacent to the leftmost or rightmost digit in a numerical field. It becomes controlled if one of the digits 1, 2 or 3 appears in the field.

The dot and comma are conventional decorators because they specify decimal points or group separators according to known conventions. If a dot appears in the pattern between two digits, and it is the only such dot in the field, then it will be regarded as a decimal point and be reproduced in the result if there are fractional digits to be displayed. Similarly, a comma in the pattern that is bordered by digits on both sides will be regarded as a conventional decoration. In this case, any number of occurrences in a field are admissible, and the corresponding commas in the result will be included only if ordered by digits there as well.

Control functions of numeric characters are:

- 0 Pad zeros outward from the decimal point
- 1 Float nearest decorators if number is negative
- 2 Float nearest decorators if number is non-negative
- 3 Float nearest decorators
- 4 Do not float nearest decorator
- 5 Normal digit
- 6 Field ends at first non-digit character other than a decimal point or a comma
- 7 Exponential symbol replaced by next non-digit character other than a decimal point or a comma
- 8 Fill with  $\square FC[3]$  (\* for “check-protection”) when otherwise blank
- 9 Pad zeros outwards to this position if non-zero

If more than one of the numeric control characters (1, 2, 3 or 4), appears in a field, the outermost ones control the sides of the field that each is nearest to.

The normal digit to use in the pattern is 5. A field of only 5's will suppress leading and trailing zeros.

If there is only one field, it is used for every column of numbers in  $R$ :

```

      Z←' 555.55'⊖ 1 0 10.1 100
      Z
1     10.1 100
28   ρZ

```

If there is more than one field, there must be one for every column of numbers in  $R$ :

## IBM Internal Use Only

```
Z←' 5 5.5 5.55'⌘ 1.12 2.12 3.12
Z
1 2.1 3.12
ρZ
11
```

A 0 can be used in the field to pad zeros to a particular point:

```
Z←' 005 5.50 5.550'⌘ 1.12 2.12 3.12
Z
001 2.12 3.120
ρZ
15
```

Embedded decorators may be included:

```
Z←'HERE: 5 ;THERE: 5.55'⌘ 1.12 3.12
Z
HERE: 1 ;THERE: 3.12
ρZ
24
```

A single field may have embedded decorators:

```
Z←'05/55/55'⌘ 70481
Z
07/04/81
ρZ
8
```

A 1 can be used in the field to float a decorator in against a number for negative values only:

```
Z←' -551.50'⌘ -1 0 10 -100
Z
-1.00 .00 10.00 -100.00
ρZ
32
```

A floating decorator may be on both sides of a number:

```
Z←' (551.50)'⌘ -1 0 10 -100
Z
(1.00) .00 10.00 (100.00)
ρZ
32
```

A 2 can be used in the field to float a decorator in against a number for non-negative values only:



```

      Z←' +552.50'⌘ -1 0 10 -100
      Z
1.00      +.00  +10.00  100.00
      ρZ

```

32

A 3 can be used in the field to float a decorator in against a number for all values:

```

      Z←' £553.50'⌘ 1 0 10 100
      Z
£1.00      £.00  £10.00  £100.
      ρZ

```

32

A 4 can be used with a 1, 2, or 3 in the field to mix non-floating and floating decorators. It blocks the floating effect of a 1, 2, or 3 on its side of the pattern.

```

      Z←' -551.45*'⌘ -1 0 10.1 -100
      Z
-1 *          *  10.1 * -100 *
      ρZ

```

36

A 6 can be used to end a field that is otherwise continued. It allows any character other than a digit, decimal point, to end a field.

```

      Z←' 06/06/05'⌘ 7 4 81
      Z
07/04/81
      ρZ

```

8

A 7 can be used to specify a double field for scaled formatting. The next decorator to the right of a 7 replaces the *E* in scaled form.

```

      Z←' 1.70*00'⌘ 12345
      Z
1.23*04
      ρZ

```

7

An 8 can be used in the field to have otherwise blank positions in the result filled with `FC[3]`:

## IBM Internal Use Only

```
      Z←' 8555.50'⌘ 1 0 10 100
      Z
***1.00 ****.00 **10.00 *100.00
      ρZ
32
```

A 9 can be used in the field to pad zeros to a particular point only for non-zero numbers:

```
      Z←' 555.59'⌘ 1 0 100
      Z
      1.00          100.00
      ρZ
21
```

If  $\square FC[4]$  is not a 0, then it is used to fill a field that would otherwise be an error, because the number is too large.

```
      Z←' 555.59'⌘ 1 1000 100
DOMAIN ERROR
      Z←' 555.59'⌘1 1000 100
      ^

      □FC[4]←'? '
      Z←' 555.59'⌘ 1 1000 100
      Z
      1.00 ????.?? 100.00
      ρZ
21
```

For more examples, refer to the Format Control system variable ( $\square FC$ ).

**Notes:**

# Chapter 6. System Functions and System Variables

|   |      |
|---|------|
| System Functions                        | 6-3  |
| Canonical Representation - $\square CR$ | 6-5  |
| Delay - $\square DL$                    | 6-6  |
| Execute Alternate - $\square EA$        | 6-6  |
| Expunge - $\square EX$                  | 6-7  |
| Function Establishment - $\square FX$   | 6-7  |
| Name Classification - $\square NC$      | 6-8  |
| Name List - $\square NL$                | 6-9  |
| Peek/Poke - $\square PK$                | 6-9  |
| Transfer Form - $\square TF$            | 6-11 |
| System Variables                        | 6-14 |
| Account Information - $\square AI$      | 6-16 |
| Atomic Vector - $\square AV$            | 6-16 |
| Comparison Tolerance - $\square CT$     | 6-17 |
| Format Control - $\square FC$           | 6-17 |
| Index Origin - $\square IO$             | 6-18 |
| Horizontal Tabs - $\square HT$          | 6-18 |
| Latent Expression - $\square LX$        | 6-18 |
| Line Counter - $\square LC$             | 6-19 |
| Printing Precision - $\square PP$       | 6-19 |
| Printing Width - $\square PW$           | 6-20 |
| Random Link - $\square RL$              | 6-20 |
| Terminal Control - $\square TC$         | 6-20 |
| Terminal Type - $\square TT$            | 6-20 |
| Time Stamp - $\square TS$               | 6-20 |
| User Load - $\square UL$                | 6-20 |
| Workspace Available - $\square WA$      | 6-20 |

**Notes:**

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

In principle, all necessary interaction between APL and its environment could be managed with a complete set of system variables. However, in some situations it is more convenient, or otherwise more desirable, to use functions based on the use of system variables that may not themselves be made explicitly available. Such functions are called *system functions*.

System variables and system functions are denoted by *distinguished names* that begin with a *quad* ( $\square$ ). The use of such names is reserved for the system and cannot be applied to user-defined objects. They cannot be erased; those that denote system variables can appear in function headers, but only to be localised (see Chapter 8, "Function Definition"). Within APL statements, distinguished names are subject to all the normal rules of syntax.

## System Functions

Like the primitive abstract functions of APL, the system functions are available throughout the system, and can be used in defined functions. They are monadic or dyadic, as appropriate, and have explicit results. In most cases they also have implicit results, in that their execution causes a change in the environment. The explicit result always indicates the status of the environment relevant to the possible implicit result. Altogether, seventeen system functions are provided. Six of these are for managing the shared-variable facility and

are described in Chapter 7, "Shared Variables". The other eleven are shown in Figure 6-1 and are described after the figure.

| Function         | Requirements                                 |  | Effect on Environment   | Explicit Result  |
|------------------|--|--|---|--|
|                  | Rank   | Domain   |   |  |
| $\square CR A$   | $1 \geq \rho \rho A$                         | Array of characters  | None  | Canonical Representation of object named by <i>A</i> . The result of anything other than an unlocked defined function is size 0 0.                 |
| $\square DL S$   | $0 = \rho \rho S$                            | Numeric value  | None, but requires <i>S</i> secs to complete.   | Scalar value of actual delay.  |
| $\square EX A$   | $2 \geq \rho \rho A$                         | Array of characters  | Erase objects named by rows of <i>A</i> , except labels or halted functions.  | A boolean vector whose <i>i</i> th element is 1 if the <i>i</i> th name is now free.   |
| $\square FX M$   | $2 = \rho \rho A$                            | Matrix of characters   | Fix definition of function represented by <i>M</i> , unless its name already used for an object other than function that is not halted. | Vector that represents name of function established, or scalar row index of fault that prevented establishment.                                    |
| $\square NC A$   | $2 \geq \rho \rho A$                         | Array of characters  | None  | Vector giving the usage of the name in each row of <i>A</i> :<br>0 - name available<br>1 - label<br>2 - variable<br>3 - function<br>4 - other      |
| $A \square NL N$ | $1 \geq \rho \rho N$                         | $\wedge / N \in 1 \ 2 \ 3$<br>Elements of <i>A</i> must be alphabetic. | None  | Same as monadic form, except only names starting with letters in <i>A</i> will be included.  |
| $\square NL N$   | $1 \geq \rho \rho N$                         | $\wedge / N \in 1 \ 2 \ 3$   | None  | Matrix of rows (in alphabetic order) that represent names of designated kinds in dynamic environment: 1, 2, 3 for labels, variables and functions. |
| $A \square EA B$ | $1 \geq \rho \rho B$<br>$1 \geq \rho \rho A$ | Characters   | None  | Executes <i>B</i> . For error, executes <i>A</i> .   |

Figure 6-1 (Part 1 of 2). System Functions

| Function         | Requirements         |  | Effect on Environment                           | Explicit Result   |
|------------------|----------------------|--|---|---|
|                  | Rank                 | Domain                                   |   |   |
| $N \square PK A$ | $0 = \rho \rho N$    | $N$ is a scalar positive integer.        | None  | Peek memory contents. Result is character vector of elements of $\square AV$ .                            |
|                  | $1 = \rho \rho A$    | $A$ is a numeric vector of two elements. | Changes memory                                  | Poke memory contents. Result is character vector with previous contents.                                  |
|                  | $1 \geq \rho \rho N$ | Character scalar/vector.                 |   |   |
|                  | $1 = \rho \rho A$    | Numeric vector of two elements.          |   |   |
| $\square PK A$   | $1 = \rho \rho A$    | See dyadic $\square PK$                  | Depends on user programs.                       | Executes machine language program. Returns register contents and flags.                                   |
| $\square TF A$   | $1 \geq \rho \rho A$ | Character scalar/vector.                 | Generate transfer form or fix new object in WS. | If $A$ is a name, result is the transfer form. If $A$ is a transfer form, result is name of object fixed. |

Figure 6-1 (Part 2 of 2). System Functions

## Canonical Representation - $\square CR$

The *canonical representation* of a defined function, as defined in Chapter 8, "Function Definition", is obtained by applying the system function  $\square CR$  to the character array representing the name of the function. When applied to any argument that does not represent the name of an unlocked defined function, it yields a matrix of dimension  $0$  by  $0$ . Possible error reports for  $\square CR$  are RANK error, if the argument is not a vector or scalar, or DOMAIN error if the argument is not a character array. The use of  $\square CR$  is further described in Chapter 8, "Function Definition".



## Delay - $\square DL$

The *delay* function, denoted by  $\square DL$ , causes a pause in the execution of the statement in which it appears. The argument of the function determines the duration of the pause, in seconds, but the accuracy is limited by other possible demands on the system at the moment of release. Moreover, the delay can be ended by a strong interrupt. The explicit result of the delay function is a scalar value equal to the actual delay. If the argument of  $\square DL$  is not a scalar with numeric value, a RANK or DOMAIN error will be reported.

The delay function may be used freely in situations where repeated tests may be required at intervals to determine if an expected event has taken place. This is useful in certain kinds of interaction between users and programs.

## Execute Alternate - $\square EA$

If you execute the statement

$$Z \leftarrow L \ \square EA \ R$$

and there is an error in the expression  $R$ , or if  $R$  is interrupted, then execution of  $R$  is ended without an error message, and  $L$  is executed instead.

$R$  and  $L$  must be character vectors or scalars. Both must contain only valid APL characters.

$R$  is taken to represent an APL expression, and is executed in the context of the statement in which it is found.  $Z$  is the value of the APL expression in  $R$  if  $R$  executes successfully. If the expression has no value, then  $L \ \square EA \ R$  has no value. If  $R$  fails to execute successfully,  $L$  is executed instead and  $Z$  is the value of the expression in  $L$ . If this expression has no value, then  $L \ \square EA \ R$  has no value. Execution of  $L$  is subject to normal error handling.

## IBM Internal Use Only

```
1 2 3 4 '12' □EA '14'  
1 2 '12' □EA '14.5'  
1 2 '→' □EA '14.5'  
    '12.3' □EA '14.5'  
◆ DOMAIN ERROR  
  12.3  
  ^
```

If *R* calls a defined function *F*, then the statements executed by *F* are also under the control of the error trap. In particular, *R* could call a long running function, and *L* could be an error recovery function.

An indication of the type of error that caused the right argument to fail to execute may be obtained using the *ΔET* function from the UTIL workspace. See “The UTIL Workspace” on page 11-92.

## Expunge - □EX

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

## Function Establishment - □FX

The definition of a function can be established or fixed by applying the system function □FX to its character representation. The function □FX produces as an explicit result, a character vector that represents the name of the function being fixed while replacing any existing definition of a function with the same name.

An expression of the form  $\square FX M$  will establish a function if both the following conditions are met:

1.  $M$  is a valid representation of a function. Any matrix that differs from a canonical matrix only in the addition of non-significant spaces is a valid representation. A row of  $M$  consisting of only spaces will appear as an empty statement in the resulting function.
2. The name of the function to be established does not conflict with any existing use of the name for a halted function (defined in Chapter 9, "Function Execution") or for a label or variable.

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

## Name Classification - $\square NC$

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

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

## **Name List - $\square NL$**

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

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

The uses of  $\square NL$  include the following:

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

## **Peek/Poke - $\square PK$**

This function has three different uses that may be requested as follows:

1. Peek the memory contents.

$R \leftarrow N \square PK AR, ADDR$

where:

$N$  is the number of bytes desired.

$ADDR$  is the starting address (in decimal code).

$AR$  may be 0 or 1. If 0,  $ADDR$  is absolute. If 1,  $ADDR$  is relative to the workspace origin.

$R$  is a character vector with the contents of the selected memory positions as elements of  $\square AV$  (that is, if the bit configuration of a byte is the base-2 representation of 120, the corresponding result will be the 120th element of  $\square AV$  in zero origin).

2. Poke the memory contents.

$R \leftarrow V \square PK \ AR, ADDR$

where:

$V$  is a character vector with the values to be inserted in memory as elements of  $\square AV$ .

$AR$  and  $ADDR$  are interpreted as in 1 above.

$R$  is the previous contents of the changed memory.

3. Execute memory.

$R \leftarrow \square PK \ AR, ADDR$

executes the machine language program contained in the indicated address and returns in  $R$  the final contents of the registers and flags in the following order: AL, AH, BL, BH, CL, CH, DL, DH, low SI, high SI, low DI, high DI, low BP, high BP, low flags, high flags.

Executable programs must end with a long RET assembly instruction (it is considered as a far procedure). The program must return the stack as it was found on entry.

## IBM Internal Use Only

An example of using  $\square PK$  to execute a small machine code program may be found in the *WHISTLE* function in the UTIL workspace. See “The UTIL Workspace” on page 11-92.

**Warning:** If you give control to an address that does not contain executable code, the system is likely to hang.

Some uses of  $\square PK$ :

- Both the weak attention and the strong attention keys can be disabled by turning on bit 128 in byte X'7AC' relative to the origin of the workspace. The current value of this byte may be obtained by  $1 \square PK 1 1964$ . See the *ESC\_OFF* and *ESC\_ON* functions described in “The UTIL Workspace” on page 11-92, for functions to set and reset this bit. BEWARE: If you turn this bit on, you will be unable to interrupt the execution of APL functions until you turn it off again!
- Terminal output can also be inhibited by turning on bit 64 in the same byte. However, this bit will be automatically turned off at the first terminal input (including stacked input). See the *HT* and *RT* functions described in “The UTIL Workspace” on page 11-92, for functions to set and reset this bit.
- | • The use of the printer as a system log is controlled by bit  
| 16 in the same byte. Turning the bit on will cause any  
| output displayed on the screen to be printed as well. This  
| bit is normally toggled on and off by Ctrl-PrtSc or  
| Alt-PrtSc. See the *PRT\_ON* and *PRT\_OFF* functions  
| described in “The UTIL Workspace” on page 11-92, for  
| functions to set and reset this bit.

## Transfer Form - $\square TF$

In the expression:

$$Z \leftarrow \square TF R$$

if  $R$  is the name of a variable or a defined function, then  $Z$  is a character vector, that is the transfer form for that object. If the transfer form cannot be formed, then  $Z$  is an empty character vector (' ').

$R$  must be a character scalar or vector.  $Z$  is a character vector.

If  $R$  is the transfer form of a variable or a defined function, then that object is established in the workspace, and  $Z$  is a character vector containing its name. If the transfer form is invalid, then  $Z$  is an empty character vector (' '). This is called the *inverse transfer form*.

Inverse transfer form ignores name class conflicts. That is, if there is a variable named  $X$  in the active workspace, an inverse transfer form may be performed to establish a function with the same name  $X$ . Similarly, if there is a function named  $X$  in the active workspace, an inverse transfer form may be performed to establish a variable with the same name  $X$ . Additionally, if there is a shared variable named  $X$  in the active workspace, and an inverse transfer form is performed to establish a variable with the same name  $X$ , then the old variable is expunged before the new variable is formed, so that any share on that variable is retracted.

The *transfer form* is a character vector. It represents the name and value of a variable, or a displayable defined function. It is produced by the monadic system function  $\square TF R$ , where  $R$  is the name of the object.

The transfer form is a character vector with four parts:

1. A data type code header character:
  - “ $F$ ” for a function
  - “ $N$ ” for a numeric array
  - “ $C$ ” for a character array
2. The name of the object, followed by a blank.

**IBM Internal Use Only**

3. A character representation of the rank and shape of the array, followed by a blank.
4. The body, consisting of a character representation of the array elements in row major order (the ravel of the array). Numeric conversions are carried to 15 digits.

The body of the transfer form of a defined function is the ravel of its canonical form character matrix.

Examples:

```

      PW←45
      THIS ← 2 3 ρ 16
      Z←□TF 'THIS'
      Z
NTHIS 2 2 3 1 2 3 4 5 6
      ρZ
23
      THAT←3 4 ρ 'ABCDEFGHIJKL'
      Z←□TF 'THAT'
      Z
CTHAT 2 3 4 ABCDEFGHIJKL
      ρZ
24

      ∇ Z←L PLUS R
[1]   Z←L+R
[2]   ∇
      Z←□TF 'PLUS'
      Z
FPLUS 2 2 10 Z←L PLUS RZ←L+R
      ρZ
33

      ∇ V←PRIMES N;□IO;M
[1]   □IO←1
[2]   M←1N
[3]   V←(1=0+.=(1+M)◦.|M)/M
[4]   ∇
      Z←□TF 'PRIMES'
      Z
FPRIMES 2 4 21 V←PRIMES N;□IO;M      □IO←1
      M←1N                               V←(1=0
      +.=(1+M)◦.|M)/M
      ρZ
99

```



## System Variables

System variables are instances of *shared variables* (see Chapter 7, “Shared Variables”). The characteristics of shared variables that are most significant here are:

- If a variable is shared between two processors, the value of the variable when used by one of them may well be different from what that processor last specified, and
- Each processor is free to use or not use a value specified by the other, according to its own internal workings.

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

Figure 6-2 on page 6-15 lists the system variables and gives their significance and use. Two classes can be distinguished:

1. Comparison tolerance, format control, horizontal tabs, index origin, latent expression, printing precision, printing width and random link. In these cases, the value you specify (or that available in a clear workspace) is used by the APL processor during the execution of operations to which they relate. Except for the latent expression (see below), if this value is inappropriate, or if no value has been specified after localisation, an IMPLICIT error will result at the time of execution.
2. Account information, atomic vector, line counter, time stamp, terminal control, terminal type, user load, and workspace available. In these cases, localisation or your setting is immaterial. The APL processor will always reset the variable before it can be used again.

| Name            | Value in Clear WS | Meaningful Range | Purpose   |
|-----------------|-------------------|------------------|---|
| <b>Class 1:</b> |                   |                  |   |
| $\square CT$    | $1E^{-13}$        | $0-1E^{-4}$      | Comparison tolerance used in monadic $\square L$ dyadic $\langle \leq = \geq \rangle \neq \in \uparrow \downarrow$                        |
| $\square FC$    | $\cdot, * 0 \_$   |                  | Format control used in dyadic $\Phi$ (Picture format)   |
| $\square HT$    | 10                |                  | This variable is ignored by the system.   |
| $\square IO$    | 1                 | 0 or 1           | Index origin: used in indexing and in ? $\uparrow \Delta \Psi \Phi \square FX$  |
| $\square LX$    | ' '               | Characters       | Latent expression executed on activation of a workspace.  |
| $\square PP$    | 10                | 1 15             | Printing precision: affects numeric output and monadic $\Phi$   |
| $\square PW$    | 79                | 30-132           | Printing width: affects all but bare output and error reports.  |
| $\square RL$    | 7*5               | $1^{-1}+2*31$    | Random link: used in ?  |
| <b>Class 2:</b> |                   |                  |   |
| $\square AI$    |                   |                  | Account information: identification, computer time, connect time, keying time (all times in milliseconds and cumulative during sessions.) |
| $\square AV$    |                   |                  | Atomic vector.  |
| $\square LC$    | 10                |                  | Line counter: statement numbers of functions in execution or halted, most recently activated first.                                       |
| $\square TS$    |                   |                  | Time stamp: year, month, day (of month), hour (on 24-hour clock minute, second, millisecond.  |

Figure 6-2 (Part 1 of 2). System Variables

| Name            | Value in Clear WS | Meaningful Range | Purpose  |
|-----------------|-------------------|------------------|--|
| Class 2 (cont): |                   |                  |  |
| $\square TC$    |                   |                  | Terminal control: a three-element vector containing the backspace, new line, and line-feed characters in that order. |
| $\square TT$    | 0                 |                  | Terminal type: always 0  |
| $\square UL$    | 1                 |                  | User load: always 1  |
| $\square WA$    |                   |                  | Workspace available (in bytes): main workspace plus elastic workspace size.  |

Figure 6-2 (Part 2 of 2). System Variables

## Account Information - $\square AI$

Account information returns a length four numeric vector containing: user identification (always 1 in APL/PC 2.1), computer time, connect time and keying time (all times in milliseconds and cumulative during session).

*Note:* This is provided to be compatible with other APL systems. The times returned may not be sufficiently accurate to use in benchmarks against other APL implementations.

## Atomic Vector - $\square AV$

The *atomic vector*  $\square AV$  is a 256-element character vector, containing all possible characters. Certain elements of  $\square AV$  may be screen-control characters, such as new line or line feed. The indexes of any known characters can be determined by an expression such as  $\square AV_1 'ABCabc'$ .

## Comparison Tolerance - $\square CT$

Comparison tolerance: used in monadic  $\Gamma$  and  $L$  and dyadic  $<$ ,  $\leq$ ,  $=$ ,  $\geq$ ,  $>$ ,  $\neq$ ,  $\epsilon$ ,  $\mathbf{1}$  and  $|$ . The value in a clear workspace is  $1E^{-13}$ .

## Format Control - $\square FC$

This is a five-element character vector containing control characters implicitly used by the picture format. The value in a clear workspace is `'.,*0_'`.

The element definitions are:

$\square FC[1]$  Use for conventional decimal point

$\square FC[2]$  Use for conventional comma

$\square FC[3]$  Fill when otherwise blank for digit

$\square FC[4]$  Fill when otherwise *DOMAIN ERROR* for overflow

$\square FC[5]$  Display as blank (may not be `.,0123456789`)

Elements of  $\square FC$  beyond 5 are not defined.

$\square FC[1]$  is used wherever a decimal point is needed in picture format:

```

       $\square FC[1] \leftarrow '.,'$ 
      '5.5555'  $\boxtimes$  3.1415
3,1415
    
```

$\square FC[2]$  is used wherever a comma is needed in picture format:

```

       $\square FC[2] \leftarrow '.,'$ 
      '555,555,555'  $\boxtimes$  123456789
123.456.789
    
```

$\square FC[3]$  is used where a field containing an 8 would otherwise be blank in picture format:

```

FC[3]←'□'
'855555'⊕ 1234

```

```

□□1234

```

$\square FC[4]$  is used to fill where a field is too small for a number or a non-scalar item in picture format:

```

FC[4]←'? '
'5555'⊕ 123456

```

```

????

```

If  $\square FC[4]$  is '0' (which is the default), then a field that is too small will result in a DOMAIN error.

$\square FC[5]$  is replaced by a blank without ending a field wherever it is used in picture format:

```

FC[5]←'⊖'
'£⊖355'⊕ 12

```

```

£ 12

```

## Index Origin - $\square IO$

Index origin: used in indexing and in  $\square$ ,  $\square$ ,  $\square$ ,  $\square$ ,  $\square$  and  $\square FX$ . The value in a clear workspace is 1.

## Horizontal Tabs - $\square HT$

The value of  $\square HT$  is ignored by the IBM Personal Computer APL System.

## Latent Expression - $\square LX$

The APL expression represented by the latent expression is automatically executed whenever the workspace is activated. The value in a clear workspace is an empty vector.

Formally,  $\square LX$  is used as an argument to the execute function  $\square \square LX$ , and any error message, will be appropriate to the use of that function.

Common uses of the latent expression include the form  $\square LX \leftarrow 'G'$ , used to invoke an arbitrary function  $G$ ; the form,

```
 $\square LX \leftarrow ' 'FOR CHANGES, ENTER: NEW' ' '$ 
```

is used to print a message upon activation of the workspace, and the form  $\square LX \leftarrow ' \rightarrow \square LC'$  is used to automatically restart a suspended function. The variable  $\square LX$  may also be localised within a function and respecified therein to furnish a different latent expression when the function is suspended. For example:

```

 $\square LX \leftarrow 'F'$ 
 $\nabla F; \square LX$ 
[1]  $\square LX \leftarrow ' \rightarrow \square LC, \rho \square \leftarrow ' 'RESUME LESSON' ' '$ 
[2]  $'WE NOW BEGIN LESSON 2'$ 
[3]  $DRILLFUNCTION$ 
[4]  $\nabla$ 
 $)SAVE ABC$ 
```

On the first activation of workspace ABC, the function  $F$  would be automatically invoked; if it were later saved with  $F$  halted, subsequent activation of the workspace would automatically continue execution from the point of interruption.

### Line Counter - $\square LC$

The line counter contains a vector of line numbers of defined functions either, currently being executed, or halted (suspended or pendent), with the most recently activated line number first. The value in a clear workspace is an empty vector.

### Printing Precision - $\square PP$

Printing precision affects numeric output and monadic  $\pi$ . The value in a clear workspace is 10.

## Printing Width - $\square PW$

Printing width affects all but bare output and error reports. The value in a clear workspace is 79.

## Random Link - $\square RL$

Random link is used in ?. The value in a clear workspace is 16807.

## Terminal Control - $\square TC$

Terminal control is a length 3 character vector containing the backspace, new line, and line feed characters (in that order).

## Terminal Type - $\square TT$

Terminal type is always set to a value of 0 in APL/PC 2.1.

## Time Stamp - $\square TS$

Date and time stamp is a length 7 numeric vector giving: year, month, day (of month), hour (on 24 hour clock), minute, second and millisecond.

## User Load - $\square UL$

User load is always set to a value of 1 in APL/PC 2.1.

## Workspace Available - $\square WA$

Workspace available (in bytes).

## Chapter 7. Shared Variables

|                      |      |
|----------------------|------|
| Offers .....         | 7-6  |
| Access Control ..... | 7-7  |
| Retraction .....     | 7-11 |
| Inquiries .....      | 7-12 |



**Notes:**

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

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

At any instant, a shared variable has only one value - the value last assigned to it by one of its owners. Characteristically however, a processor using a shared variable will find its value different from what it might have set earlier.

A given processor can simultaneously share variables with several other processors. However, each sharing is *bilateral*; that is, each shared variable has only two owners. This restriction does not represent a loss of generality in the systems that can be constructed, and commonly useful arrangements are easily designed. For example, a shared file can be made directly accessible to a single control processor that communicates bilaterally with (or is integral with) the file processor itself. In turn, the central processor shares variables bilaterally with each of the using processors, controlling their individual access to the data, as required.

It was noted in Chapter 6, "System Functions and System Variables" that system variables are instances of shared variables in which the sharing is automatic. It was not pointed out, however, that access sequence disciplines are also imposed on certain of these variables, although one effect of this was noted; namely, variables, like the time stamp, accept any value specified, but continue to provide the proper information when used. The discipline that accomplishes this effect is an

inhibition against two successive accesses to the variable, unless the sharing processor (the system) has set it in the interim.

When ordinary, "undistinguished" variables are to be shared, explicit actions are necessary to accomplish the sharing and establish a desired access discipline.

Sharing a variable is an act of communication between two processors. There are several aspects to such communication, involving:

- Establishing
- Controlling
- Querying
- Retracting

Thus it should not be surprising that there are system functions to deal with each of these aspects of the communication and these are described below. Six system functions are provided for these purposes - three for the actual management, and three to provide related information. These are summarised in Figure 7-1.

| Function          | Requirements [ 1 ]   |  |   | Effect on Environment   | Explicit Result   |
|-------------------|----------------------|--|---|---|---|
|                   | Rank                 | Length                                       | Domain  |   |   |
| $P \square SVO N$ | $2 \geq \rho \rho N$ | $(x/\rho P) \in 1, \neg 1 \downarrow \rho N$ | $P \in 1 \downarrow 1 \neg 1 + 2 * 15$<br>[ 2 ] | Tenders offer to processor $P$ if first (or only) name of pair is not previously offered and not already in use as the name of an object other than a variable. | Degree of coupling now in effect for the name pair.<br>Dimension:<br>$, x/\neg 1 \downarrow \rho N$ |
| $\square SVO N$   | $2 \geq \rho \rho N$ | None   | [ 2 ]   | None  | Degree of coupling now in effect for the name pair.<br>Dimension:<br>$, x/\neg 1 \downarrow \rho N$ |

Figure 7-1 (Part 1 of 2). System Functions

| Function            | Rank   | Requirements [ 1 ]   |                                     | Effect on Environment         | Explicit Result   |
|---------------------|--|--|-------------------------------------|-------------------------------|---|
|                     |  | Length   | Domain                              |                               |   |
| $C \square SVC \ N$ | $2 \geq \rho \rho N$<br>$2 \geq \rho \rho C$ | $(1 \geq \rho \rho C) \wedge$<br>$1 = x / \rho C$<br>or<br>$(\rho C) =$<br>$(\neg 1 \vee \rho N), 4$ | $\wedge / C \in 0 \ 1$<br><br>[ 2 ] | Sets access control.          | New setting of access control.<br><br>Dimension:<br>$(\neg 1 \vee \rho N), 4$   |
| $\square SVC \ N$   | $2 \geq \rho \rho N$                         | None   | [ 2 ]                               | None                          | Existing access control.  |
| $\square SVR \ N$   | $2 \geq \rho \rho N$                         | None   | [ 2 ]                               | Retracts offer (ends sharing) | Degree of coupling before retraction.<br>Dimension:<br>$, x / \neg 1 \vee \rho N$   |
| $\square SVQ \ P$   | $1 \geq \rho \rho P$                         | $1 \geq \rho, P$   | $P \in$<br>$1 \vee \neg 1 + 2 * 15$ | None                          | If $0 = \rho P$ : Vector of IDs of processors making offers to this user.<br><br>If $1 = x / \rho P$ : Matrix of names offered by processor $P$ but not yet shared. |

Notes:

1. If a requirement is not met, the function is not executed, and a corresponding error report is printed.
2. Each row of  $N$  (or  $N$  itself if  $2 \geq \rho \rho N$ ) must represent a name or pair of names. If a pair of names is used for an offer (dyadic  $\square SVO$ ), either the pair, or the first name only, can be used for the other functions.

Figure 7-1 (Part 2 of 2). System Functions

## Offers

A single offer to share is of the form offer  $P \square SVO N$ , where  $P$  is the identification of another processor, and  $N$  is a character vector representing a pair of names. The first of this pair is the name of the variable to be shared, and the second is a surrogate name. The name of the variable may be its own surrogate, in which case only the one name need be used, rather than two.

*Note:* The surrogate names have no effect in this implementation of APL and are provided to allow code compatibility with other APL systems.

The explicit result of the expression  $P \square SVO N$  is the degree of coupling of the name in  $N$ : 0 if no offer has been made, 1 if an offer has been made but not matched, 2 if sharing is completed.

An offer to any processor (other than the offering processor itself) requests an increase in the coupling of the name offered, if the name has zero coupling and is not the name of a label or a function. An offer never decreases the coupling.

The monadic function  $\square SVO$  does not affect the coupling of the name represented by its argument, but does report the degree of coupling as its explicit result. If the degree of coupling is 1 or 2, a repeated use of dyadic  $\square SVO$  has no further implicit result, and either monadic or dyadic  $\square SVO$  may be used for inquiry.

The following is an example of a defined function for offering a name (to be entered on request) to a processor  $P$ :

```
[0] Z←OFFER P;Q
[1]  □←'NAME:'
[2]  →(' '∧.=Q←5↓□)/Z←0
[3]  Z←P □SVO Q
[4]  →(2=Z←□SVO Q)/L
[5]  'NO DEAL'
[6]  →0
[7]  L:'ACCEPTED'
```

If the arguments of  $\square SVO$  fail to meet any of the basic requirements listed in Figure 7-1 on page 7-4, the appropriate error report results, and the function is not executed. An offer to a processor will be acknowledged, whether or not the processor happens to be available.

The value of a shared variable, when sharing is first completed, is determined thus: if both owners had assigned values previously, the value is that assigned by the first to have offered; if only one owner had, it has that value; if neither had, the variable has no value. Names used in sharing are subject to the usual rules of localisation.

A set of offers can be made by using a vector left argument (or scalar or one-element vector that is automatically extended) and a matrix right argument, each of whose rows represent a name or a name pair. A vector left argument should have as many items as the right argument has rows. The offers are then treated in sequence and the explicit result is the vector of the resulting degrees of coupling.

Auxiliary processors are identified by positive integers between 2 and 32767.

## **Access Control**

In most practical applications, it is important to know that a new value has been assigned between successive uses of a shared variable, or that use has been made of an assigned value before a new one is set. Because, as a practical matter, this cannot be left to chance, an access control mechanism is embodied in the shared variable facility.

The access control operates by inhibiting the setting or use of a shared variable by one owner or the other, depending on the access state of the variable and the value of an access control matrix, which is set jointly by the two owners, using the dyadic form of the system function  $\square SVC$ . If one user had followed his offer to share  $V$  by the expression  $1 \ 1 \ 1 \ 1 \ \square SVC \ 'V'$ ,

the following sequence would have been enforced: the use of  $V$  by the second processor would be automatically delayed until  $V$  is set by the first one, and the use by the latter would be delayed until  $V$  is set by the former.

Figure 7-2 on page 7-9 shows the three access states possible for a shared variable, the possible transitions between states, and the potential inhibitions imposed by the access control matrix (ACM). The first row of the ACM is associated with setting of the variable by each owner, and the second with its use. The permissible operations for any state are indicated by the zeros in  $ACM \wedge ASM$ , where  $ASM$  is the representation of the access state shown in the figure. This can be confirmed by using the figure to validate each of the following statements:

- If  $ACM[1;1]=1$ , then two successive sets by A require an intervening access (set or use) by B.
- If  $ACM[1;2]=1$ , then two successive sets by B require an intervening access by A.
- If  $ACM[2;1]=1$ , then two successive uses by A require an intervening set by B.
- If  $ACM[2;2]=1$ , then two successive uses by B require an intervening set by A.

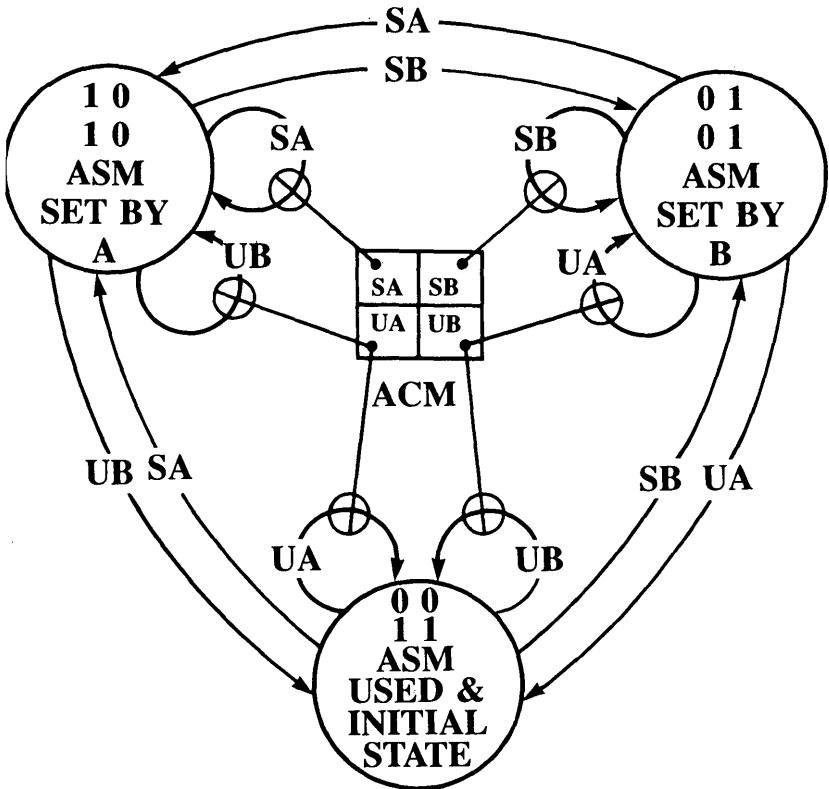


Figure 7-2. Access Control of a Shared Variable

The value of the access state representation is not directly available to you, but the value of the access control matrix is given by the monadic function  $\square SVC$ . For a shared variable  $V$ , the result of the expression  $\square SVC 'V'$  executed by user A is the *access control vector*,  $\phi ACM$  (the four-element ravel of  $ACM$ ). However, if user B executed the same expression, he would obtain the result  $\phi \phi ACM$ . The reason for the reversal is that sharing is *symmetric*: neither owner has precedence over the other, and each sees a control vector in which the first one of each pair of control settings applies to his own accesses. This symmetry is evident in the figure; if it were redrawn to interchange the roles of A and B, the control matrix would be the row-reversal of the matrix shown.

The setting of the access control matrix for a shared variable is determined in a way that maintains the functional symmetry.



An expression of the form  $L \square SVC 'V'$  executed by user A assigns the value of the logical left argument  $L$  to a four-element vector which, for the purposes of the present discussion, will be called  $QA$ . Similar action by user B sets  $QB$ . The value of the access control matrix is determined as follows:

$$ACM \leftarrow (2 \ 2 \rho QA) \vee \phi \ 2 \ 2 \rho QB$$

Because ones in ACM inhibit the corresponding actions, it is clear from this expression that one user can only increase the degree of control imposed by the other (although he can, by using  $\square SVC$  with a left argument of zeros, restore the control to that minimum level at any time).

Access control can be imposed only after a variable is offered, either before or after the degree of coupling reaches 2. The initial values of  $QA$  and  $QB$  when sharing is first offered are zero.

The access state when a variable is first offered (degree of coupling is 1) is always the initial state shown in the figure. If the variable is set or used before the offer is accepted, the state changes accordingly. Completion of sharing does not change the access state.

Figure 7-3 on page 7-11 lists a number of settings of the access control vector that are of common practical interest. Any one of them can be represented by a simplification of Figure 7-2 on page 7-9 obtained by omitting the control matrix and deleting the lines representing those accesses that are inhibited in the particular case. For example, with maximum constraints, all the inner paths would be removed from the figure.

| Access Control Vector<br>as seen by: |         | Comments   |
|--------------------------------------|---------|--|
| A                                    | B       |  |
| 0 0 0 0                              | 0 0 0 0 | No constraints   |
| 0 0 1 1                              | 0 0 1 1 | Half-duplex. Ensures each use is preceded by a set by partner.     |
| 1 1 0 0                              | 1 1 0 0 | Half-duplex. Ensures each set is preceded by an access by partner. |
| 1 1 1 1                              | 1 1 1 1 | Reversing half-duplex. Maximum constraint.                         |
| 0 1 1 0                              | 1 0 0 1 | Simplex. Controlled communications from B to A.                    |

Figure 7-3. Some Useful Settings for the Access Control Vector

A group of  $N$  access control matrices can be set at once by applying the function  $\square SVC$  to an  $N$ -by-4 matrix left argument and an  $N$ -rowed matrix right argument of names. The explicit result is an  $N$ -by-4 matrix giving the current values of the (ravel of) control matrices. When control is being set for a single variable, the left argument may be a single 1 or 0 if all inhibits or none are intended. A scalar, a one-element vector, or a four-element vector left argument  $L$  is treated as the  $N$ -by-4 matrix  $(N, 4) \rho L$ .

## Retraction

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

Retraction of sharing is automatic if you sign off or load a new workspace. Sharing of a variable is also retracted by its

erasure or, if it is a local variable, upon completion of the function in which it appeared.

The nature of the shared-variable implementation is often such that the current value of a variable set by a partner will not be represented within a user's workspace until actually required to be there. If this is the case, the value will be copied to the user's workspace when the variable is to be used, when sharing is ended, or when a *)SAVE* command is issued (since the current value of the variable must be stored). Under any of these conditions, it is possible for a *WS FULL* error to be reported. In all cases, the prior access state remains in effect, and the operation can be retried after corrective action.

## Inquiries

There are three monadic inquiry functions that produce information concerning the shared variable environment but do not alter it: the functions  $\square SVO$  and  $\square SVC$ , already discussed, and the function  $\square SVQ$ . A user who applies the  $\square SVQ$  function to an empty vector obtains a vector result containing the identification of each user making a specific and unmatched sharing offer to him. A user who applies this function to a non-empty argument obtains a matrix of the names offered to him by the processor identified in the argument. This matrix includes only those names that have not been accepted by counter-offers.

The expression  $(0 \neq \square SVO M) / [1] M \leftarrow \square NL \ 2$  can be used to produce a character matrix whose rows represent the names of all shared variables in the dynamic environment.

## Chapter 8. Function Definition

|   |      |
|---|------|
| Canonical Representation and Function Establishment | 8-3  |
| The Function Header                                 | 8-5  |
| Ambi-Valent Functions                               | 8-5  |
| Local and Global Names                              | 8-6  |
| Branching and Statement Numbers                     | 8-7  |
| Labels  | 8-9  |
| Comments  | 8-9  |
| Function Editing - The $\nabla$ Form                | 8-10 |
| Adding a Statement                                  | 8-10 |
| Inserting or Replacing a Statement                  | 8-11 |
| Replacing the Header                                | 8-11 |
| Deleting a Statement                                | 8-11 |
| Modifying a Statement or Header                     | 8-12 |
| Function Display                                    | 8-12 |
| Leaving the $\nabla$ Form                           | 8-13 |
| Quitting the $\nabla$ Form                          | 8-14 |

**Notes:**

A defined function can be established in an APL workspace in three ways:

1. It can be copied from a stored workspace using the system command `⌋IN`, as described in Chapter 10, “System Commands”.
2. It can be established in execution mode, using either of the system functions `⌈FX` or `⌈TF`, either in direct keyboard entry or in the course of execution of another defined function.
3. It can be established in function definition mode.

Regardless of which method has been used for establishing a function, its definition can be displayed or modified either in the function definition mode, in which certain editing capabilities are built-in, or by the combined use of the system functions `⌈CR` and `⌈FX`.

## Canonical Representation and Function Establishment

The *character representation* of a function is a character matrix satisfying certain constraints: the first row of the matrix represents the *function header* and must be one of the forms specified below under “The Function Header” on page 8-5. The remaining rows of the matrix, if any, constitute the *function body*, and may consist of any sequence of characters. If the character representation satisfies additional constraints, such as left justification of the non-blank characters in each row, then it is said to be a canonical representation.

Applying `⌈CR` to the character array representing the name of an already established function will produce its canonical representation. For example, if *OVERTIME* is an available function:

```

DEF←□CR 'OVERTIME'
DEF
PAY←R OVERTIME H;TIME
TIME←0↑H-40
PAY←R×1.5×TIME
ρDEF
3 21

```

The function  $\square CR$  applied to any argument that does not represent the name of an unlocked defined function yields a matrix of shape 0 0.

The use of  $\square CR$  does not change the status of the function *OVERTIME*, which remains established and can be used for calculations. Thus:

```

7 5 8 OVERTIME 35 40 45
0 0 60

```

If *OVERTIME* should be expunged:

```

□EX 'OVERTIME'
1

```

it is no longer available for use:

```

7 5 8 OVERTIME 35 40 45
SYNTAX ERROR
7 5 8 OVERTIME 35 40 45
^

```

The function can be re-established by  $\square FX$ :

```

□FX DEF
OVERTIME

```

The function  $\square FX$  produces as its explicit result, the vector of characters that represents the name of the function being fixed, while replacing any existing definition of a function with the same name. The function *OVERTIME* can now be used again:

```

7 5 8 OVERTIME 35 40 45
0 0 60

```

An expression of the form  $\square FX M$  will establish a function if the conditions described under "Canonical Representation and Function Establishment" on page 8-3 are met.

## The Function Header

The *valence* of a function is defined as the number of explicit arguments that it takes. A defined function may have a valence of 0, 1, or 2, and may or may not yield an explicit result. These cases are represented by six forms of header as follows:

| <u>Type</u> | <u>Valence</u> | <u>Result</u>            | <u>No Result</u> |
|-------------|----------------|--------------------------|------------------|
| Dyadic      | 2              | $R \leftarrow A \ F \ B$ | $A \ F \ B$      |
| Monadic     | 1              | $R \leftarrow F \ B$     | $F \ B$          |
| Niladic     | 0              | $R \leftarrow F$         | $F$              |

The names used for the arguments of a function become local to the function, and additional local names may be designated by listing them after the function name and argument, separated from them and from each other by semicolons; the name of the function is *global*. The significance of these distinctions is explained below.

Except that the function name itself may be repeated in the list of local names, a name may not be usefully repeated in the header. Nor is it obligatory for the arguments of a defined function to be used within the body, or for the result variable to be specified in the course of function execution.

## Ambi-Valent Functions

Defined functions with a valence of 2 may be called either monadically (without a left argument) or dyadically. All dyadic defined functions are thus *ambi-valent*; that is, a left argument is not required when the function is called in context. In such a case, the left argument will be undefined (will have no value) inside the function, and its name class will be zero.

For example, the function *ROOT* calculates the *N*th root of its right argument. If no left argument is provided when the function is called, a default value is supplied:



```

[0]  Z←N ROOT A
[1]  →(0≠□NC 'N')/RN
[2]  N←2
[3]  RN:Z←A*÷N

      2 ROOT 64 729 4096
8 27 64

      ROOT 64 729 4096
8 27 64

      3 ROOT 64 729 4096
4 9 16

```

## Local and Global Names

In the execution of a defined function, it is often necessary to work with intermediate results or temporary functions that have no significance either before or after the function is used. The use of local names for these purposes, so designated by their appearance in the function header, avoids cluttering the workspace with many objects introduced for such transient purposes, and allows greater freedom in the choice of names. Names used in the function body, and not so designated, are said to be *global* to that function.

A *local* name may be the same as that for a global object, and any number of names local to different functions may be the same. During the execution of a defined function, a local name will temporarily exclude from use a global object of the same name. If the execution of a function is interrupted (leaving it either suspended, or pendent, as described in Chapter 9, "Function Execution"), the local objects keep their dominant position during the execution of later APL operations, until such time as the *halted* function is completed. Even the system commands and the *del* form of function definition (see below) reference local objects under these circumstances.

The localisation of names is dynamic in the sense that it has no effect except when the defined function is being executed. Furthermore, when a defined function uses another defined function during its execution, a name localised in the first (or outer) function continues to exclude global objects of the same

name from the range of the second (or inner) function. This means that a name localised in an outer function has the significance assigned to it in that function when used without further localisation in an inner function. The same name localised in a sequence of nested functions has the significance assigned to it at the innermost level of execution. The *shadowing* of a name by localisation is complete, in the sense that once a name has been localised, its global and outer values are nullified, even if no significance is assigned to it during execution of the function in which it is localised.

## **Branching and Statement Numbers**

Statements in a function are normally executed successively, from top to bottom, and execution stops at the end of the last statement in the sequence. This normal order can be modified by *branches*. Branches are used in the construction of iterative procedures, in choosing one out of a number of possible continuations, or in other situations where decisions are made during the course of function execution.

To facilitate branching, the successive statements in a function definition have reference numbers associated with them, starting with the number 1 for the first statement in the function body and continuing with successive integers, as required. Thus, the expression  $\rightarrow 4$  signifies a branch to the fourth statement in the function body, and when executed, causes statement 4 to be executed next, regardless of where the branch statement itself occurs. (In particular  $\rightarrow 4$  may be statement 4, in which case the system will simply execute this “tight loop” indefinitely, until interrupted by an action from the keyboard. This is a trap to be avoided.)

A branch statement always starts with the *branch arrow* (or *right arrow*) on the left, and this can be followed by any expression. For the statement to be effective, however, the expression must be an integer, or a vector whose first element is an integer, or an empty vector; any other value results in a *DOMAIN* or *RANK* error. If the result of the expression is a valid result, the following rules apply:

1. If the result is an empty vector, the branch is empty and execution continues with the next statement in the function if there is one, or else the function ends.
2. If the result is the number of a statement in the function, then that statement is the next to be executed.
3. If the result is a number out of the range of statement numbers in the function, then the function ends. The number 0 and all negative integers are outside the range of statement numbers for any function.

Because zero is often a convenient result to compute, and is not the number of a statement in the body of any function, it is often used as a standard value for a branch intended to end the execution of a function. It should be noted that in the function definition mode described below, zero is used to refer to the header. This has no bearing on its use as a target for a branch.

An example of the use of a branch statement is shown in the following function, which computes the greatest common divisor of two scalars:

```
[0]  Z←M GCD N
[1]  L:Z←M
[2]  M←M|N
[3]  N←Z
[4]  →(0≠M)/L
```

The *compression* function in the form  $U/V$  gives  $V$  if  $U$  is equal to 1, and an empty vector if  $U$  is equal to 0. Thus, the fourth statement in *GCD* is a branch statement that causes a branch to the first statement (labelled “L”, see below), when the condition  $0≠M$  is true, and a branch with an empty vector argument, that is, normal sequence, when the condition is false. In this case, there is no next statement and so execution of the function ends.

## Labels

If a statement occurring in the body of a function definition is prefaced by a name and a colon, the name is assigned a value equal to the statement number. A name used in this way is called a *label*. Labels are used to advantage when it is expected that a function definition may be changed for one reason or another, since a label automatically assumes the new value of the statement number of its associated statement as other statements are inserted or deleted.

The name of a label is local to the function in which it appears, and must be distinct from other label names and from the local names in the header.

A label name may not appear immediately to the left of a specification arrow. In effect, it acts as a (local) constant.

## Comments

The *lamp* symbol  $\mathfrak{A}$  (the *cap-null*) signifies that what follows is a comment for illumination only and is not to be executed; it may occur anywhere in a statement. Everything to the left of the first lamp character in a line is treated as executable code and everything to its right is assumed to be a comment.

If there is executable code to the left of a lamp character, the blanks between the last character of the executable code and the lamp will be preserved. This may be used to align the lamp characters vertically to make the comments easier to read.

## Function Editing - The $\nabla$ Form

The functions  $\square CR$  and  $\square FX$  together form a basis for establishing and revising functions. Convenient definition and/or editing with them, however, requires the use of prepared editing functions, which must be defined, stored in a library, and explicitly activated when needed. The *del* form described here provides another means for function entry and revision, which is always present for use.

When you enter the *del* character ( $\nabla$ ) followed by the name of a defined function, the system responds by displaying  $[N+1]$ , where  $N$  is the number of statements in the function. It is now possible to:

- add, insert, or replace statements
- replace the header
- modify the header or a statement
- delete statements
- display all or part of the definition

A new function is started by entering the desired header on the same line as the opening  $\nabla$ . Once the function definition mode has thus been entered, the treatment of a new function is identical to that for a function already defined.

### Adding a Statement

If the response to the display of statement number  $[N+1]$  is a statement, it is accepted as a line added at the end of the definition. The system response is  $[N+2]$ . Additional statements may continue to be added to the definition in this way. If an empty statement is entered, the system will re-display the line number in brackets.

## Inserting or Replacing a Statement

If the response to the statement number displayed by the system is [*N*], where *N* is any positive number with or without a fractional part, the system will display [*N*]. A statement entered will replace an existing statement *N*.

The system continues by displaying the next appropriate number. For example, if the statement number entered was [3], the next number displayed will be [4]; if [3.02], then [3.03]; if [3.29], then [3.3], and so forth.

A statement may be submitted with line number [*N*]; it will be inserted or will replace an existing statement in the way described. The response of the system in this case is to display the next statement number.

## Replacing the Header

If you enter [0], the system responds with [0]. You may now enter any legal header, which will replace the existing header. Following this, the system displays [1]. The entire operation may be done by entering [0] and, on the same line, the header.

## Deleting a Statement

A statement may be deleted by entering a *delta* in brackets followed by the statement number, for example, [ $\Delta$ 2]. The response of the system is to display the next statement number. In the example, the response will be [3]. Several statements may be deleted at a time, as in [ $\Delta$ 2 3 5]. A range of statements may be deleted by [ $\Delta$ 25], which deletes statements 2 to 5 (inclusive).

## Modifying a Statement or Header

One or more characters can be added to the end of statement *N*, or statement *N* can be corrected, by entering [*N*□□]. In response, the system displays statement *N*; the cursor moves to the end of the statement, and the keyboard unlocks. The statement may be extended, or modified, by using the normal revision procedures for entry. In response, the system displays the next statement number and awaits entry.

The header may be modified in this way by entering [0□□].

Any line displayed anywhere on the screen during function definition can be modified in place, and incorporated in the function definition by using the Enter key after modification. Since the modified line is brought to the bottom of the screen when entered, this facility may be used for transferring selected lines from one function, whose definition is still on the display, to another that is currently being defined or edited.

You can modify several lines of a function wherever they are displayed on the screen, and run through them in order with the alternate Enter key (the “+” key associated with the numeric keypad), which will enter them in place. This avoids the scrolling of lines off the screen, and maintains the same line on the display for further modification or study.

## Function Display

The canonical representation of a function includes the header and body displayed as a character matrix. The ▽ form permits display of a canonical representation modified as follows:

1. The header, labelled lines and comment statements are offset one space to the left.
2. Statement numbers in brackets are appended to the left of the statements.
3. A statement number of 0, in brackets, is prefixed to the header.

The following is the function display of function *OVERTIME*, the canonical representation of which was displayed at the beginning of this chapter:

```

      ∇OVERTIME[ ]∇
[0]  PAY←R OVERTIME H;TIME
[1]  TIME←0[H-40
[2]  PAY←R×1.5×TIME

```

While in function definition mode, display of the entire definition can be requested by responding with [ ]]. The statements will be listed in numeric order, taking into account deletions and insertions. Following the last statement, the next appropriate line number will be displayed. The definition from statement *N* onward can be similarly displayed by entering [ ]N].

Statement *N* alone can be displayed by entering [N ]]; in this case the statement number *N* is repeated by the system after the display of the statement itself. Statements *N* to *M* can be displayed by entering [N ]M].

## Leaving the ∇ Form

The *del* form may be left, and the function in the active workspace updated, by typing a ∇ on a line by itself, or as the last character on any entry that does not contain a comment. In particular, it can follow a request for display or a function statement, and either can be included in the same entry that both opens and closes the definition mode. For example, ∇DET[ ]∇ displays the function *DET*, and ∇DET[10] ↵L ∇ modifies the contents of line 10 in the function *DET*. On leaving the *del* form, the statements are reordered according to their statement numbers, and the statement numbers are replaced by the integers 1, 2, 3, and so on.

A function definition can be locked by either opening or closing the definition mode with a *del-tilde*, ∇̃. The use of this is explained in Chapter 9, "Function Execution".



## Quitting the $\nabla$ Form

Typing [→] will quit the *del* form. Any changes made to the function during the current editing session will be cancelled, and immediate execution mode will be resumed.

## Chapter 9. Function Execution

|   |      |
|---|------|
| Halted Execution .....                    | 9-4  |
| State Indicator .....                     | 9-4  |
| State Indicator Damage .....              | 9-6  |
| Trace Control .....                       | 9-6  |
| Stop Control .....                        | 9-7  |
| Locked Functions .....                    | 9-8  |
| Recursive Functions .....                 | 9-9  |
| Input and Output .....                    | 9-10 |
| Evaluated Input .....                     | 9-11 |
| Character Input .....                     | 9-12 |
| Interrupting Execution during Input ..... | 9-12 |
| Normal Output .....                       | 9-12 |
| Bare Output .....                         | 9-13 |

**Notes:**

A defined function may be used like a primitive function, except that it cannot be the argument of a primitive operator. In particular, a defined function may be used within its own definition or that of another defined function. When a function is *called*, or put into use, its execution begins with the first statement, and continues with successive statements, except as this sequence is altered by branch instructions.

Consider the function *OVERTIME*.

```
[0]  PAY←R OVERTIME H;TIME
[1]  TIME←0⌈H-40
[2]  PAY←R×1.5×TIME
```

If this function is invoked by a statement such as *X OVERTIME Y*, the effect is to assign to the local name *R* the value of *X*, and to *H* the value of *Y*, and then execute the body of the function *OVERTIME*. Except for having a value assigned initially, the argument variable is treated as any other local variable and, in particular, may be respecified within the function.

A function like *OVERTIME*, which produces an explicit result, may properly be used in compound expressions. In the *OVERTIME* function, the last value received by *PAY* during execution is the explicit result of the function. For example:

```
YTDAT←100 200 150
YTDAT←YTDAT+OT←5 7 6 OVERTIME 35 40 45
OT
0 0 45
YTDAT
100 200 195
```

*PAY*, itself, is a local variable and therefore has no significance after the function is executed:

```
PAY
VALUE ERROR
PAY
^
```

Defined dyadic functions may be called monadically (without a left argument). In such a case, the left argument will not have a value during execution, and its name class will be 0.

## Halted Execution

The execution of a function  $F$  may be stopped before completion in a variety of ways: by an error report, by an attention signal, or by the *stop control*, which is explained below. When this happens, the function is said to be *suspended*, and its progress can be resumed by entering a branch statement from the keyboard. Whatever the reason for suspension, the name of the function is displayed, with a statement number beside it. In the case of an error stop or an interrupt, the statement itself is also displayed, with an appropriate message and an indication of the point of interruption. Unless a specification appears in the statement to the right of this point, the state of the computation has been restored to the condition obtaining before the statement started to execute.

In general, therefore, the displayed number is that of the statement that should be executed next if the function is to continue normally. Execution can be resumed at that point by entering a branch to that number specifically, a branch to an empty vector, or a branch to  $\square LC$ . Entering  $\rightarrow 0$ , or a branch to another number outside the range of statement numbers, causes an immediate exit from the function and it is no longer suspended.

In the suspended state, all normal activities are possible, but names used refer to their local significance, if any. The system can execute statements or system commands, resume execution of the function at an arbitrary point, or enter definition mode to work on the suspended function, or some other. Pending functions can be edited with the del ( $\nabla$ ) editor.

## State Indicator

Entering the system command  $)SI$  causes a display of the *state indicator*; a typical display has the following form:

```

    )SI
*  H[7]
   G[2]
   F[3]

```

This display indicates that execution was halted before completing (perhaps before starting) execution of statement 7 of function *H*, that the current use of function *H* was invoked in statement 2 of function *G*, and that the use of function *G* was in turn invoked in statement 3 of *F*. The \* appearing to the left of *H*[7] indicates that the function *H* is suspended. The functions *G* and *F* are said to be *pendent*, because their execution cannot be restarted directly, but only as a consequence of function *H* resuming its course of execution. The term *halted* is used to describe a function that is either pendent or suspended.

Further functions can be invoked in the suspended state. Thus, if *G* were now invoked and a further suspension occurred in statement 5 of *Q* (*Q* was invoked in statement 8 of *G*), a subsequent display of the state indicator would appear as follows:

```

    )SI
*  Q[5]
   G[8]
*  H[7]
   G[2]
   F[3]

```

Because the line counter,  $\square LC$ , holds the current statement numbers of functions that are executing, its value at this point would be the vector 5 8 7 2 3.

The sequence from the last to the preceding suspension can be cleared by entering a right arrow ( $\rightarrow$ ). This behaviour is illustrated by continuing the foregoing example as follows:

```

    →
    )SI
*  H[7]
   G[2]
   F[3]
  □LC
7 2 3

```

Repeated use of  $\rightarrow$  will clear the state indicator completely and restore  $\square LC$  to an empty vector. The same effect can be obtained with a single use of the  $\rangle RESET$  command. The cleared state indicator displays as if a blank line had been entered.

## State Indicator Damage

If the name of a function occurs in the state indicator list, then erasure of the function or replacement of the function by copying a function with the same name (even another instance of the same function) will make it impossible for the original course of execution to be resumed. In such an event, an *SI DAMAGE* report is given. In addition, the APL system will give an *SI DAMAGE* report if a halted function is edited to change the order of its labels or to modify its header.

If an *SI DAMAGE* report is given for a suspended function, it will not be possible to resume its execution by entering a branch statement, but the function can be invoked again, with or without prior clearance of the state indicator.

In case of *SI DAMAGE*, display of the state indicator will show the damage by giving  $\bar{1}$  as the current statement number of the affected function.

## Trace Control

A *trace* is an automatic display of information generated by the execution of a function as it progresses. In a complete trace of a function, the number of each statement executed is displayed in brackets, preceded by the function name and followed by the final value produced by the statement. The trace of a branch statement shows a branch arrow followed by the number of the next statement to be executed. The trace is useful in analysing the behaviour of a defined function, particularly during its design.

The tracing of a function *PROFIT* is controlled by the trace control for *PROFIT*, denoted by *TΔPROFIT*. If one sets *TΔPROFIT←2 3 5*, then statements 2, 3, and 5 will be traced in any later execution of *PROFIT*. *TΔPROFIT←1 0* discontinues tracing of *PROFIT*. A complete trace of *PROFIT* is obtained by *TΔPROFIT←1 N*, where *N* is the number of statements in *PROFIT*. In general, the trace control for any function is designated by prefixing *TΔ* to the function name.

## Stop Control

A function can be caused to execute up to a certain statement and then stop in the suspended state. This is frequently useful in analysing a function, for example by experimenting with local variables or intermediate results. The stops are set by the *stop control* in the same manner as the trace. For example, stops that will stop execution of the function *PROFIT* before lines 4 and 12 are executed can be set by entering *SΔPROFIT←4 12*.

At each stop, the function name and line number are displayed, as described above for suspended functions. To go to the next stopping point after the first, execution must be explicitly restarted by entering an appropriate branch statement.

Trace control and stop control can be used in conjunction. Moreover, either of the controls may be set within functions. In particular, they may be set by expressions that initiate tracing or stops as a result of certain conditions that may develop during function execution, such as a particular variable taking on a particular value. They may only be used as the left argument of specification. They may not be used by themselves or as the argument to a function.



## Locked Functions

If the symbol  $\nabla$  (called *del-tilde*) is used instead of  $\nabla$  to open or close a function definition, the function becomes *locked*. A locked function cannot be revised or displayed in any way. Any associated stop control or trace control is nullified after the function is locked.

A locked function is treated essentially as a primitive, and its inner workings are concealed as much as possible. Execution of a locked function is ended by any error occurring within it, or by a strong interrupt. If execution stops, the function is never suspended but is immediately abandoned. The message displayed for a stop is *DOMAIN ERROR*, if an error of any kind occurred; *WORKSPACE FULL* and the like, if the stop resulted from a system limitation, or *INTERRUPT*.

Moreover, a locked function is never pendent, and if an error occurs in any function invoked either directly or indirectly by a locked function, the execution of the entire sequence of nested functions is abandoned. If the outermost locked function was invoked by an unlocked function, that function will be suspended; if it was invoked by a keyboard entry, the error message will be displayed with a copy of that statement.

Similarly, when a weak interrupt is encountered in a locked function, or in any function that was ultimately invoked by a locked function, execution continues normally up to the first interruptable point - either the next statement in an unlocked function that invoked the outermost locked function, or the completion of the keyboard entry that used this locked function. In the latter case, the weak interrupt has no net effect.

Locked functions may be used to keep a function definition proprietary, or as part of a security scheme for protecting other proprietary information. They are also used to force the kind of behaviour just described, which sometimes simplifies the use of applications.

## Recursive Functions

A defined function whose name has not been made local and is used in the body of the function definition is said to be defined *recursively*. For example, one definition of the greatest-common-divisor function states that the greatest common divisor of zero and any number  $N$  is  $N$ ; for any other pair of numbers it is the greatest common divisor of the residue of the second number by the first, and the first number. The words "greatest common divisor" are used in the definition. This suggests that a greatest-common-divisor function  $GCDR$  can be written whose canonical representation is:

```

      □CR' GCDR'
R←A GCDR B
R←B
→(0=A)/A
R←(A|B)GCDR A

      18 GCDR 45
9

```

This can be compared to the equivalent function  $GCD$  defined iteratively in Chapter 8, "Function Definition".

Executing an erroneously-defined recursive function will often result in a **STACK FULL** report. The non-trivial execution of a properly-defined recursive function may also have this effect because of the very deep nesting of function calls that is often required.

## Input and Output

In many applications, such as text processing, it is necessary to supply information as the execution of the application program progresses. It is also often convenient, even in the use of an isolated function, to supply information in response to a request, rather than as arguments to the function as part of the original entry. This is illustrated by considering the use of the function *CI*, which determines the growth of a unit amount invested at periodic interest rate *R* for a number of periods *T*:

$$A \leftarrow R \text{ CI } T$$

$$A \leftarrow (1+R)^*T$$

For example, the value of 1000 pounds at 5 percent for 7 years, compounded quarterly, might be found by:

$$1000 \times (0.05 \div 4) \text{ CI } 7 \times 4$$

1415.992304

The casual user of such a function might, however, find it difficult to remember which argument of *CI* is which, how to adjust the rate and period stated in years for the frequency of compounding, and whether the interest rate is to be entered as the actual rate (for example, 0.05) or as a percentage (for example, 5). An exchange of the following form might be more suitable:

```

      INVEST
ENTER CAPITAL AMOUNT IN POUNDS
□:      1000
ENTER NUMBER OF TIMES COMPOUNDED EACH YEAR
□:      4
ENTER ANNUAL INTEREST RATE IN PERCENT
□:      5
ENTER PERIOD IN YEARS
□:      7
VALUE IS 1415.992304

```

Each of the entries (1000, 4, 5, and 7) occurring in such an exchange must be accepted, not as an ordinary entry (which would only evoke the response 1000, etc.), but as data to be used within the function *INVEST*. Facilities for this are provided in two ways - *evaluated input* and *character input*. A definition of the function *INVEST*, which uses evaluated input, is as follows:

```

      □CR 'INVEST'
INVEST;C;R;T;F
'ENTER CAPITAL AMOUNT IN POUNDS'
C←□
'ENTER NUMBER OF TIMES COMPOUNDED EACH YEAR'
F←□
'ENTER ANNUAL INTEREST RATE IN PERCENT'
R←□÷F×100
'ENTER PERIOD IN YEARS'
T←F×□
'VALUE IS ',⊖C×R CI T
    
```

## Evaluated Input

The *quad* symbol (□) appearing anywhere other than immediately to the left of a specification arrow signifies a request for keyboard input as follows: the two symbols □: are displayed, and the keyboard is unlocked on the next line, indented from the left margin. Any valid expression entered at this point is evaluated, and the result substituted for the *quad*. Suppose *F* is a function whose definition includes a *quad* symbol:

```

      □CR 'F'
Z←F
Z←4×□
      F
□:
      3+2
20
    
```

An invalid entry in response to a request for *quad* input causes an appropriate error report, after which input is again awaited. For example, entering an expression that has no result produces a *value* error. Function definition mode (the editing or display of functions, or creation of new functions) is not permitted during □ entry. In general, a system command

entered during  $\square$  input is executed, but the system's response to the command is not treated as a response to  $\square$ . After execution of a command, valid input is again awaited (unless the command was one that replaced the contents of the active workspace). An empty input (one containing nothing other than zero or more spaces) is rejected and the system again awaits input.

## Character Input

The *quote-quad* symbol  $\square$  (that is, a *quad* superimposed on a quote) appearing anywhere other than immediately to the left of a specification arrow is a request for character input; entry is permitted at the left margin and data entered is accepted as characters. For example:

```

          X← $\square$ 
CAN'T      (Quote-quad input, not indented)
          X
CAN'T

```

## Interrupting Execution during Input

The response  $\rightarrow$  entered in response to  $\square$  abandons execution of the function and any pendent functions leading up to it.

A request for  $\square$  input can be interrupted by entering the Ctrl-Break key combination.

## Normal Output

The *quad* symbol appearing immediately to the left of a specification arrow indicates that the value of the expression to the right of the arrow is to be displayed in the standard format (subject to the printing precision  $\square PP$  and the printing width  $\square PW$ ). Hence,  $\square \leftarrow X$  is equivalent to the statement  $X$ . The longer form  $\square \leftarrow X$  is useful when employing multiple specification. For example,  $\square \leftarrow Q \leftarrow X * 2$  assigns to  $Q$  the value  $X * 2$ , then prints the value of  $X * 2$ .

The maximum length of a line of normal display (measured in characters) is called the *printing width* and is given by the value of the system variable `□PW`. A display whose lines exceed the printing width is ended at or before the maximum length, and continued on subsequent lines.

## Bare Output

Normal output includes a concluding new-line signal so that the succeeding display (either input or output) will begin at a standard position on the following line. *Bare output*, denoted by expressions of the form `□←X`, does not include this signal if it is followed either by another bare output or by character input (of the form `X←□`).

Character input following a bare output is treated as though you had spaced over to the position occupied at the conclusion of the bare output, so that the characters received in response will be prefixed by the characters displayed in the bare output. This allows for the possibility that, after the keyboard is unlocked, you backspace into the area occupied by the preceding output. The following function prompts you with whatever message is supplied as its argument, and evaluates the response:

```
□CR 'PROMPT'  
Z←PROMPT MSG  
□←MSG  
Z←□
```

Using such a function, the expression

```
PROMPT 'ENTER CAPITAL: '
```

would have the following effect:

```
Displayed by system:  
ENTER CAPITAL: 1000  
Entered by user
```

The value of `Z` is the string of characters contained in `MSG`, followed by the characters you entered, not including explicitly-entered trailing blanks.

The new-line signals that would be supplied by the system to break lines that exceed the printing width are not supplied with bare output. However, because an expression of the form  $\square \leftarrow X$  entered directly from the keyboard (rather than being executed as part of a defined function) must necessarily be followed by another keyboard entry, the output it causes is concluded with a new-line signal.

# Chapter 10. System Commands

- Active Workspace - Action Commands ..... 10-9
- Active Workspace - Inquiry Commands ..... 10-13
- Workspace Storage and Retrieval - Action  
Commands ..... 10-14
  - Libraries of Saved Workspaces ..... 10-14
  - Workspace Names ..... 10-14
- Workspace Storage and Retrieval - Inquiry  
Commands ..... 10-20
- Sign-Off ..... 10-21



**Notes:**

An APL system recognises two broad classes of instructions - *statements* and *system commands*. System commands control the start and end of a work session, saving and reactivating copies of a workspace, and transferring data from one workspace to another.

System commands are prefixed by a right (closing) parenthesis. System commands can be invoked only by individual entries from the keyboard and cannot be executed dynamically as part of a defined function.

(However, some system commands may be emulated by APL functions, such as the *IN* and *OUT* functions in the FILE workspace. Also the AP101 Auxiliary Processor may be used to stack system commands for execution as though they had been entered from the keyboard).

The system commands are summarised in Figure 10-1 on page 10-4, and will be discussed under three main headings:

1. The active workspace.
  - a. Action.
  - b. Inquiry.
2. Workspace storage and retrieval.
  - a. Action.
  - b. Inquiry.
3. Access to the system.

| Form   | Purpose  | Normal Response                                  | Trouble Reports       |
|--|--|--|-----------------------|
| <b>Active Workspace - Action Commands</b>                |  |  |                       |
| <code>)CLEAR</code>                                      | Activate a clear WS                                    | <code>CLEAR WS</code>                            | 3                     |
| <code>)SYMBOLS pi</code>                                 | Set size of symbol table                               | <code>WAS number</code>                          | 3                     |
| <code>)STACK pi</code>                                   | Set size of execution stack                            | <code>CLEAR WS</code><br><code>WAS number</code> | 3                     |
| <code>)ERASE nms</code>                                  | Erase objects from active WS                           | <code>CLEAR WS</code>                            | 3, 5, 7               |
| <code>)IN tf</code>                                      | Copy all objects from transfer file to active WS       | <code>SAVED time date</code>                     | 1, 3, 8, 9, 11, 14    |
| <code>)IN tf nms</code>                                  | Copy named objects from transfer file to active WS     | <code>SAVED time date</code>                     | 1, 3, 4, 8, 9, 11, 14 |
| <code>)RESET</code>                                      | Clear the state indicator                              |  | 3                     |
| <b>Active Workspace - Inquiry Commands</b>               |  |  |                       |
| <code>)SYMBOLS</code>                                    | Give size of symbol table and available space in bytes | <code>number number</code>                       | 3                     |
| <code>)STACK</code>                                      | Give size of execution stack                           | <code>number</code>                              | 3                     |
| <code>)FNS</code>  | List defined functions                                 | <code>(names)</code>                             | 3                     |
| <code>)VARS</code>                                       | List variables   | <code>(names)</code>                             | 3                     |
| <code>)SI</code>   | List halted functions                                  | <code>state indicator</code>                     | 3                     |
| <code>)SINL</code>                                       | List halted functions and names                        | <code>state indicator and names</code>           | 3                     |
| <b>Workspace Storage and Retrieval - Action Commands</b> |  |  |                       |
| <code>)WSID wsid</code>                                  | Change ID of active workspace                          | <code>WAS ws</code>                              | 3                     |
| <code>)SAVE wsid</code>                                  | Replace named WS with copy of active WS                | <code>time date wsid</code>                      | 2, 3, 6, 11, 13, 14   |
| <code>)SAVE</code>                                       | Place copy of active workspace in library              | <code>time date wsid</code>                      | 2, 3, 6, 11, 13, 14   |
| <code>)LOAD wsid</code>                                  | Activate copy of named workspace                       | <code>time date</code>                           | 1, 3, 10, 11, 12, 14  |

Figure 10-1 (Part 1 of 2). System Commands

| Form  | Purpose   | Normal Response | Trouble Reports     |
|---|---|-----------------|---------------------|
| <code>)OUT tf</code>                                      | Generate a transfer file with all objects in the active workspace | time date       | 2, 3, 11, 13, 14    |
| <code>)OUT tf nms</code>                                  | Generate a transfer file with the objects in nms                  | time date       | 2, 3, 4, 11, 13, 14 |
| <code>)DROP wsid</code>                                   | Drop workspace or file from library                               |                 | 1, 3, 11, 13        |
| <b>Workspace Storage and Retrieval - Inquiry Commands</b> |   |                 |                     |
| <code>)WSID</code>  | Give identification of active workspace                           | (number) name   | 3                   |
| <code>)LIB</code>   | List workspaces or files in desired library                       | (names)         | 3, 11               |
| <b>Access to the System</b>                               |   |                 |                     |
| <code>)OFF</code>   | End use of APL  | Return to DOS   | 3                   |

Notes:

1. Items in parentheses are optional.
2. Abbreviations and Meanings:
  - *WS*: workspace
  - *wsid*: a workspace name possibly preceded by a library number
  - *tf*: transfer file name possibly preceded by a library number
  - *pi*: positive integer
  - *nms*: list of names
3. The commands, `)ERASE`, `)FNS`, and `)VARS` have variants that are system functions.

Figure 10-1 (Part 2 of 2). System Commands

A system command that is not recognisable, or is improperly formed, is rejected with the report *COMMAND ERROR*. Certain commands may also result in more specific trouble reports; these are discussed in the appropriate context and are summarised in Figure 10-2 on page 10-7.

| No | Message           | Meaning   | Remedy  |
|----|-------------------|---|---|
| 1  | NOT FOUND         | No stored workspace with given ID   | Change the disk or drop unneeded WS<br><br>Give a name to the workspace<br><br>OR<br>Rename active workspace then store<br><br>Clear SI ( )RESET)<br><br>Erase objects not needed, then:<br>)OUT ws, )CLEAR,<br>)SYMBOLS n,<br>)IN ws, )WSID ws<br>1. Erase unneeded objects<br>2. Clear SI ( )RESET)<br><br>Ensure that diskette is inserted correctly and that the drive door is closed.<br><br>Check file. Convert to APL/PC 2.1 format. (See description of )LOAD).<br>Check diskette. Remove write-protect tab.<br>Increase number of file handles in CONFIG.SYS |
| 2  | LIB FULL          | No room on the disk   |   |
| 3  | COMMAND ERROR     | Workspace does not contain objects with purported names                                   |   |
| 4  | nms NOT FOUND     |   |   |
| 5  | nms NOT ERASED    | Purported names could not be erased   |   |
| 6  | NOT SAVED         | A clear workspace with no name cannot be saved  |   |
|    |                   | OR  |   |
|    |                   | Attempted replacement of a stored workspace whose ID does not match that of the active WS |   |
| 7  | SI DAMAGE         | State indicator damaged by )ERASE   |   |
| 8  | SYMBOL TABLE FULL | Too many names used   |   |
| 9  | WS FULL           | Workspace full  |   |
| 10 | WS TOO LONG       | Workspace does not fit in main storage  |   |
| 11 | NOT READY         | The door of the drive you want to access is open or diskette is not inserted correctly.   |   |
| 12 | INVALID WS        | Workspace was not saved by APL/PC 2.1.  |   |
| 13 | PROTECTED         | Diskette is write protected.  |   |
| 14 | TOO MANY FILES    | Insufficient DOS file handles.  |   |

WARNING: Changing diskettes during an input/output operation, or when you have open files, may damage your diskette.

Figure 10-2. Trouble Reports

Once the execution of a system command has started, it cannot be interrupted, although display of the system's response to the command can be suppressed by an interrupt signal.

In the text that follows, each system command is shown in a sample form. The meaning of the symbols used in the sample command forms is shown in Figure 10-3. In use, the appropriate names or numbers should, of course, be substituted.

|                 |  |
|-----------------|--|
| <b>A</b>        | A letter of the alphabet   |
| <b>LIBNO</b>    | A library number (that is, the number of a disk drive).<br>If this field is not given, the default drive is assumed.   |
| <b>WSNAME</b>   | A workspace name   |
| <b>FILENAME</b> | A DOS file name  |
| <b>EXT</b>      | A DOS file extension   |
| <b>NAME</b>     | A string formed by numbers and uppercase letters, starting with a letter   |
| <b>OBJ</b>      | The name of an object within a workspace (that is, a function or a variable)   |
| <b>( )</b>      | Items enclosed in parentheses may, in some cases, be omitted.  |
| <b>[ ]</b>      | Items enclosed in brackets may, in some cases, be omitted; when items are omitted, the system supplies default values. |

Figure 10-3. Symbols Used in Command Definitions

## Active Workspace - Action Commands

The following system commands affect or modify the active workspace, the environment in which computation takes place and, in which, names have meaning. In particular, the active workspace contains the settings of the state indicator (discussed in Chapter 9, "Function Execution") and other elements of the computing environment, mediated by several of the system variables (discussed in Chapter 6, "System Functions and System Variables").

### **)CLEAR**

This command is used to make a fresh start, discarding the contents of the active workspace, and resetting the environment to standard initial values (see Figure 10-4). At sign-on, you receive a clear workspace characterised by these same initial values, unless a system command has been included in the starting APL invocation line, and successfully executed.

The environment in a clear workspace is as follows:

|                                    |                           |
|------------------------------------|---------------------------|
| Workspace name                     | None ( <i>CLEAR WS</i> )  |
| Symbol table size                  | 2048 bytes                |
| Stack size                         | 128 elements              |
| State indicator                    | Cleared                   |
| Comparison tolerance, $\square CT$ | $1E^{-13}$                |
| Format control, $\square FC$       | . , * 0 _                 |
| Horizontal tabs, $\square HT$      | Empty                     |
| Index origin, $\square IO$         | 1                         |
| Latent expression, $\square LX$    | Empty                     |
| Line counter, $\square LC$         | Empty                     |
| Printing precision, $\square PP$   | 10                        |
| Printing width, $\square PW$       | 79                        |
| Random link, $\square RL$          | 16807                     |
| Work area available, $\square WA$  | Depends on PC memory size |

Figure 10-4. Environment Within a Clear Workspace



**)STACK N**

Sets the size of the execution stack, in number of elements. The following table indicates the APL operations that use the stack, and the stack size they need:

| <u>APL operation</u> | <u>Number of<br/>stack elements</u> |
|----------------------|-------------------------------------|
| Pending parenthesis  | 1                                   |
| Pending bracket      | 1                                   |
| Execute              | 1                                   |
| Quad input           | 2                                   |
| Execute alternative  | 1                                   |
| Niladic defined fn   | 2                                   |
| Monadic defined fn   | 1                                   |
| Dyadic defined fn    | 1                                   |

A minimum of 64, and a maximum of 4096 stack elements may be selected by the **)STACK** command.

This command may be executed in unclear workspaces, the only restriction being that the stack itself must be empty. An attempt to change the maximum with a non-empty **SI**, or to set it outside the range permitted by the system, is rejected with the message **COMMAND ERROR**. Valid use of the command results in a report showing the former limit.

**)SYMBOLS N**

Sets the size of the symbol table in bytes. New values of the maximum size may be set only in a clear workspace. A minimum of 512 and a maximum of 32766 bytes may be selected by the **)SYMBOLS** command.

A typical series of commands to change the size of the symbol table of a **)SAVE**-ed workspace is:

```

)LOAD WS
)OUT WS
)CLEAR
)SYMBOLS N
)IN WS
)WSID WS
)SAVE

```

An attempt to change the maximum once the workspace is no longer clear, or to set it outside the range permitted by the system, is rejected with the message *COMMAND ERROR*. Valid use of the command results in a report showing the former limit.

**)ERASE OBJ1 (OBJ2 (OBJ3 ...))**

The objects named are erased from the workspace; shared variable offers with respect to any of them are retracted.

If a halted function is erased, the report *SI DAMAGE* is displayed. It is not possible to resume the execution of an erased function, and you should enter one or more right arrows to clear the state indicator of indications of damage.

If an object named in the command cannot be found, the report *NOT ERASED* is displayed, followed by a list of the objects not found.

**)IN [LIBNO] FILENAME [OBJ1 (OBJ2 ...)]**

The indicated objects or system variables are copied from the indicated transfer file (FILENAME) into the active workspace. The system reports the date and time at which the transfer file was last written to disk.

If the list of objects to be copied is omitted, all objects and system variables are copied from the transfer file.

If the indicated transfer file is unavailable for some reason, copying cannot take place. In this case the message *NOT FOUND* will be reported. If any objects are specifically requested but not found in the transfer file, then a list of such names is reported, followed by *NOT FOUND*.

When an object to be copied has the same name as an object in the active workspace, the copied object replaces it. If there was a shared variable offer with respect to the variable thus replaced, the offer is retracted.

The latent expression ( $\square LX$ ) is not executed by  $\rangle IN$ , even if a complete workspace transfer form file is copied into the active workspace.

The  $\rangle IN$  command can be used even when the execution stack is not empty. Note, however, that this can redefine local variables as well as global objects, so that care is needed when using this command in a workspace with an unclear execution stack.

The following trouble reports may arise during copying:

- **WORKSPACE FULL**

There is not enough space to accommodate all the material to be copied. However, those objects copied before space was exhausted remain in the active workspace.

- **SYMBOL TABLE FULL**

New names occurring in the copied material exhaust the capacity of the symbol table. Those objects copied before the symbol table was exhausted remain in the active workspace.

- **NOT READY**

The door of the drive you want to access is open or the inserted diskette is not properly formatted.

- **I/O ERROR**

Data error on the read operation.

**$\rangle RESET$**

The state indicator is cleared.

## Active Workspace - Inquiry Commands

The following commands report aspects of the workspace environment, but produce no change in it.

### *)SYMBOLS*

Gives two numbers: the first one shows the current size of the symbol table, in bytes; the second one, shows the current size of the available space in the symbol table, in bytes.

### *)STACK*

Gives the current size of the execution stack, in number of elements.

### *)FNS*

Reports a list of the functions in the active workspace, in alphabetic order. It also accepts the specification of one or more “starting letters” for the names listed, e.g. *)FNS FG* lists all function names beginning with either *F* or *G*.

### *)VARS*

Reports a list of the variables in the active workspace, in alphabetic order. It also accepts the specification of one or more “starting letters” for the names listed, e.g. *)VARS XY* lists all variable names beginning with either *X* or *Y*.

### *)SI*

Displays the state indicator, showing the status of halted functions, with the most-recently-halted first. The list shows the name of the function and the number of the statement at which work is halted. The actions that you can take with respect to a halted function are described in Chapter 9, “Function Execution”.

Suspended functions are marked in the state indicator by an asterisk, while pendent functions appear in the state indicator

*without* an asterisk. Damage to the state indicator is shown by a statement number of **-1** beside the name of the affected function.

**)SINL**

Displays the state indicator in the same way as **)SI**, but in addition, with each function listed, lists names that are local to its execution.

## **Workspace Storage and Retrieval - Action Commands**

You may request that a duplicate of the currently active workspace be saved for later use. When a duplicate of a saved workspace is reactivated later, the entire environment of computation is restored, except that variables that were shared in the active workspace are not automatically shared again when the workspace is reactivated.

### **Libraries of Saved Workspaces**

Each disk drive in the IBM Personal Computer is called a *library* (see Chapter 1, "Introduction"). Library identifications are usually consecutive numbers.

### **Workspace Names**

A *saved workspace* must be named. The name of a workspace may duplicate a name used for an APL object within the workspace.

Workspace names are subject to DOS file naming restrictions and may be composed of up to eight upper case alphabetic and numeric characters, but not spaces or special symbols; workspace names must begin with an alphabetic character.

**)WSID** [*LIBNO*] *WSNAME*

Assigns the name indicated and, optionally, the library number indicated, to the active workspace

Setting of the active workspace's identification is acknowledged by the report *WAS* . . . followed by the former name.

**)SAVE** [[*LIBNO*] *WSNAME*]

A duplicate of the active workspace is saved (optionally, in the indicated library) under the indicated name. If the workspace name is omitted, it is supplied from the workspace identification. After saving, the active workspace has the same identification (including library number and name) as the saved workspace.

Current values of the shared variables are saved in the stored copy. This does not affect the state of sharing in the active workspace.

Saving is acknowledged by a report showing the date and time at which the workspace was saved, and the name of the saved workspace.

The file written to disk will have a filename of the specified *WSNAME*, padded to eight characters with underbar characters and will have an extension of ".APL". Any existing file of the same name will be overwritten.

The padding of names with underbars provides some protection from DOS device names. This allows workspaces to have identical names to DOS devices. For example, **)SAVE NUL** saves a workspace with a file name of NUL\_\_\_\_\_.APL, rather than writing the file to the NUL device, which would result in nothing being saved.

The command to save the active workspace may be rejected, with trouble reports as follows:

- NOT SAVED

Saving is not permitted when the name given in the command matches the identification of an existing saved workspace but does not match the identification of the active workspace. This restriction prevents you from accidentally overwriting one workspace with another.

This message may also appear if the workspace has no name (is a *CLEAR WS*) and the *)SAVE* command does not assign a new name to it.

- LIBRARY FULL

There is not enough space on the disk to accommodate the workspace.

- NOT READY

The door of the drive you want to access is open or the inserted diskette is not properly formatted.

- PROTECTED

The diskette in the indicated drive is write-protected.

- I/O ERROR

Data error on the write operation.

*)OUT [LIBNO] FILENAME [OBJ1 (OBJ2 ...)]*

This command writes the transfer form of objects in the active workspace to a transfer file. The optional list specifies what objects to transfer. The default is to transfer all the objects and system variables in the workspace.

The *FILENAME* can be different to the current *WSID* and will not affect the *WSID* of the active workspace. The file written to disk will have a filename of the specified *FILENAME*, padded to eight characters with underbar characters and will have an extension of ".AIO". Any existing file of the same name will be overwritten. No provision is made to append extra objects to an existing "AIO" file.

The *)OUT* command can be used even when the execution stack is not empty. Note, however, that the local definition of a localised object will be written to the transfer file. To provide further protection, *)OUT* of a full workspace with a non-empty stack is not allowed, and the error report “*INVALID COMMAND*” will be given.

This command may be rejected with trouble reports as follows:

- NOT FOUND

If any objects are specifically requested but not found in the active workspace, a list of such names is reported followed by *NOT FOUND*.

- LIBRARY FULL

The LIBRARY FULL error message may be reported if the selected disk does not have enough space for the transfer file.

- NOT READY

The door of the drive you want to access is open or the inserted diskette is not properly formatted.

- PROTECTED

The diskette in the indicated drive is write-protected.

- I/O ERROR

Data error on the write operation.

*)LOAD [LIBNO] WSNAME*

A duplicate of the indicated workspace (including its entire computing environment) becomes your active workspace.

Shared variable offers in the former active workspace are retracted. Following a successful *)LOAD*, the system reports the date and time at which the loaded workspace was last



saved. The system then immediately executes the latent expression ( $\square LX$ ).

Invalid requests to load a workspace may result in the reports:

- NOT FOUND

If the indicated workspace cannot be found on the selected drive.

*Note:* A file that has not been created by a  $\rangle SAVE$  command cannot be  $\rangle LOADED$ , even though its extension is “.APL”.

A valid workspace that has been renamed must have its name padded to eight characters with underbars in order to be found by the  $\rangle LOAD$  command.

- INVALID WS

If the indicated workspace has not been created by the APL/Personal Computer Version 2.1. If the workspace was created under Personal Computer APL Version 1.0, you can convert it by executing the following procedure:

1. Invoke APL/PC Version 1.0 from DOS.
2.  $\rangle LOAD$  the workspace.
3.  $\rangle OUT$  the whole workspace in transfer form.
4.  $\rangle OFF$  from APL Version 1.0.
5. Invoke APL/PC Version 2.1 from DOS, including AP210 as a required auxiliary processor.
6.  $\rangle LOAD$  the MIGRATE workspace and enter the name of the transfer form file when requested. For more details on using the MIGRATE workspace see Appendix B, “APL/PC 1.0 Workspace Migration”.
7.  $\rangle CLEAR$  the active workspace.

8. *)IN* the workspace from the transfer form file.
9. *)WSID* to set the appropriate name.
10. *)SAVE* it to the disk.

- NOT READY

The door of the drive you want to access is open or the inserted diskette is not properly formatted.

- I/O ERROR

Data error on the read operation.

*)DROP [LIBNO] FILENAME[.EXT]*

The named file is removed from the indicated library. If no extension is given, the default is ".APL". Dropping a workspace has no effect on the active workspace.

You may get the following trouble reports:

- NOT FOUND

If the workspace you want to drop does not exist.

- NOT READY

The door of the drive you want to access is open or the inserted diskette is not properly formatted.

- PROTECTED

The diskette in the indicated drive is write-protected.

- I/O ERROR

Data error on the write operation.

# Workspace Storage and Retrieval - Inquiry Commands

## *)WSID*

Reports the identification of the active workspace, showing the library number if explicitly stated, and the workspace name.

## *)LIB [LIBNO] [NAME][.EXT]*

Displays those files, the names of which start with *NAME*, and that have the given extension *.EXT* in the indicated drive *LIBNO*. If no extension is given, all files starting with *NAME* are listed. If no name is given, all files with the given extension are listed.

### *Examples:*

|                    |   |
|--------------------|---|
| <i>)LIB 1</i>      | Lists all files in drive 1.   |
| <i>)LIB 1 APL</i>  | Lists all files with name starting with "APL".  |
| <i>)LIB 1 .APL</i> | Lists all saved workspaces.   |
| <i>)LIB 1 .AIO</i> | Lists all workspaces in transfer form.  |
| <i>)LIB 1 .EXE</i> | Lists all "EXE" files in the drive.   |
| <i>)LIB 1 .</i>    | Lists all files with a blank extension.   |
| <i>)LIB AP.EXE</i> | Lists all files on the default drive with names starting with "AP" and having an extension of ".EXE". |

You can get the following error reports:

- NOT READY

The door of the drive you want to access is open or the inserted diskette is not properly formatted.

- I/O ERROR

Data error on the read operation.

## Sign-Off

*)OFF*

Gets out of APL and gives control back to the Disk Operating System. The active workspace is lost.

**Notes:**

## Part 3. Application Guide

|  |             |
|--|-------------|
| <b>Chapter 11. Application Workspaces</b> .....                | <b>11-1</b> |
| The AP2 Workspace .....  | 11-5        |
| Example Session .....  | 11-8        |
| The AP124 Workspace .....                                      | 11-8        |
| Fundamentals .....   | 11-9        |
| Building a Menu .....  | 11-10       |
| The AP190 Workspace .....                                      | 11-19       |
| The AP205 Workspace .....                                      | 11-21       |
| The AP206 Workspace .....                                      | 11-21       |
| The AP232X Workspace .....                                     | 11-26       |
| The AP488 Workspace .....                                      | 11-28       |
| Requirements .....   | 11-28       |
| Reference Documentation .....                                  | 11-28       |
| Hints to Avoid Trouble .....                                   | 11-28       |
| Description of AP488 Functions .....                           | 11-30       |
| The APLFILE Workspace .....                                    | 11-42       |
| The DEMO124 Workspace .....                                    | 11-47       |
| The DEMO206 Workspace .....                                    | 11-48       |
| The DOSFNS Workspace .....                                     | 11-48       |
| The EDIT Workspace .....                                       | 11-51       |
| The EXCHG Workspace .....                                      | 11-55       |
| The FILE Workspace .....                                       | 11-56       |
| Functions .....  | 11-57       |
| Terminology .....  | 11-58       |
| Examples of Use .....  | 11-71       |
| The FOIL Workspace .....                                       | 11-73       |
| The FORTRAN Workspace .....                                    | 11-74       |
| Restrictions on FORTRAN Programs .....                         | 11-74       |
| Generation Process .....                                       | 11-75       |
| Usage Protocol .....   | 11-78       |
| PFORTPAR Parameter Management Program .....                    | 11-79       |
| Sample FORTRAN Subroutines (IBM PC Professional FORTRAN) ..... | 11-81       |
| The GEDIT Workspace .....                                      | 11-82       |
| The GRAPHPAK Workspaces .....                                  | 11-84       |
| The MUSIC Workspace .....                                      | 11-86       |

|   |        |
|---|--------|
| The PLOT Workspace .....                    | 11-87  |
| The PRINT Workspace .....                   | 11-89  |
| The PROFILE Workspace .....                 | 11-91  |
| The UTIL Workspace .....                    | 11-92  |
| The VM232 Workspace .....                   | 11-95  |
| Selecting a Terminal .....                  | 11-96  |
| Saving Your Line Parameter Definition ..... | 11-102 |
| Connection with the Host .....              | 11-103 |
| Functions .....                             | 11-105 |
| Example of Connection with the Host .....   | 11-108 |
| Auxiliary Files on the Host .....           | 11-112 |

## **Chapter 12. Auxiliary Processors .....** 12-1

### The Non-APL Program Interface Auxiliary

|   |      |
|---|------|
| Processor: AP2 .....  | 12-4 |
| Basic Functions .....   | 12-5 |
| Auxiliary Functions .....                                     | 12-8 |
| Sample AP2 session .....                                      | 12-9 |
| Return Codes (Returned through the control<br>variable) ..... | 12-9 |

|   |       |
|---|-------|
| The Printer Auxiliary Processor: AP80 ..... | 12-10 |
| Patching AP80 for Other Printers .....      | 12-12 |

|   |       |
|---|-------|
| The Stack and Profile Auxiliary Processor: AP101 .. | 12-14 |
| Error Return Codes .....                            | 12-17 |

### The BIOS/DOS Interrupt Auxiliary Processor:

|  |       |
|--|-------|
| AP103 .....                            | 12-18 |
| BIOS/DOS Interrupt Function Call ..... | 12-19 |
| I/O Port IN/OUT Request .....          | 12-22 |
| Joystick Algorithm .....               | 12-23 |

### The Full Screen Management Auxiliary Processor:

|                          |       |
|--------------------------|-------|
| AP124 .....              | 12-24 |
| AP124 Operation .....    | 12-24 |
| Error Return Codes ..... | 12-33 |

### The Host Communications Auxiliary Processors:

|  |       |
|--|-------|
| AP190 and AP190I .....                   | 12-34 |
| Possible uses for AP190 .....            | 12-35 |
| Getting Started .....                    | 12-35 |
| Sending Keystrokes .....                 | 12-35 |
| Setting Keyboard Translation Table ..... | 12-36 |
| Getting Host Status .....                | 12-36 |
| Getting the Physical Screen .....        | 12-36 |
| Get the Operator Information Area .....  | 12-37 |
| Simulate a Power On Reset .....          | 12-37 |

- Get Cursor Position and Beep Indication . . . . . 12-37
- Get the Keyboard Translation Table . . . . . 12-37
- Get the Screen Format Array . . . . . 12-38
- The Full-Screen Auxiliary Processor: AP205 . . . . . 12-38
- The Graphic Auxiliary Processor: AP206 . . . . . 12-39
  - Storage Management . . . . . 12-39
  - Parameters . . . . . 12-40
  - Use of AP206 . . . . . 12-47
  - Functions . . . . . 12-48
  - Return codes . . . . . 12-53
- The File Auxiliary Processor: AP210 . . . . . 12-53
  - Control Commands . . . . . 12-54
  - Control Subcommands . . . . . 12-57
  - AP210 Return Codes . . . . . 12-59
  - Examples of use . . . . . 12-60
- The Asynchronous Communications Auxiliary Processor: AP232 . . . . . 12-62
  - Control Commands . . . . . 12-63
- The Extended Asynchronous Communications Auxiliary Processor: AP232X . . . . . 12-69
  - Hardware Notes . . . . . 12-70
  - AP232X Operation . . . . . 12-70
  - AP232X Return Codes . . . . . 12-75
- The Music Auxiliary Processor: AP440 . . . . . 12-76
  - AP440 Command Syntax . . . . . 12-77
- The IBM GPIB Support Auxiliary Processor: AP488 . . . . . 12-79
  - Description of AP488 Functions . . . . . 12-80
- Chapter 13. How to Build an Auxiliary Processor . . . . . 13-1**
  - Access Control . . . . . 13-4
  - Format of Shared Data . . . . . 13-5
  - Shared Variable Processor Services and Return Codes . . . . . 13-7
    - Processor Sign-on: 00H . . . . . 13-8
    - Return to APL via Shared Variable Processor:
      - 01H . . . . . 13-9
      - Share or Query the State of a Variable: 02H . . . . . 13-10
      - Get the Present Value of a Shared Variable: 03H . . . . . 13-11
      - Get a Block of Memory From the Workspace:
        - 04H . . . . . 13-12
        - Release Storage to the Workspace: 05H . . . . . 13-13
        - Pass a Variable to APL and Release the Space:
          - 06H . . . . . 13-14
          - Pass a Scalar Integer Return Code to APL: 07H . . . . . 13-15



|  |       |
|--|-------|
| Convert an APL Object from Type Boolean to Integer: 08H        | 13-10 |
| Convert from APL Z-code to ASCII: 09H                          | 13-11 |
| Convert from ASCII to APL Z-code: 0AH                          | 13-11 |
| Share or Query the State of a Variable: 0BH                    | 13-19 |
| Pre-read a Variable: 0CH                                       | 13-20 |
| Read a Previously Pre-read Variable: 0DH                       | 13-20 |
| Release a Previously Pre-read Variable: 0EH                    | 13-20 |
| Pass a Value to a Variable: 0FH                                | 13-20 |
| Processor Sign-off: 10H  | 13-24 |
| SVP Reserved Function: 11H                                     | 13-24 |
| Locate an Associated Variable: 12H                             | 13-25 |
| Change the Keyboard / Screen Mode: 13H                         | 13-26 |
| Get Loop Count for Delay: 14H                                  | 13-27 |
| Change the Keyboard / Screen Mode Without Clearing Screen: 15H | 13-28 |
| Notes  | 13-29 |
| Return Codes (Returned in CX Register)                         | 13-30 |
| Sample Auxiliary Processors                                    | 13-30 |
| APL Interrupt Usage  | 13-31 |
| How to Debug Auxiliary Processors                              | 13-32 |
| Exchange Assembly Programs                                     | 13-33 |

## Chapter 11. Application Workspaces

|  |       |
|--|-------|
| The AP2 Workspace  | 11-5  |
| Example Session  | 11-8  |
| The AP124 Workspace                                      | 11-8  |
| Fundamentals   | 11-9  |
| Building a Menu  | 11-10 |
| The AP190 Workspace                                      | 11-19 |
| The AP205 Workspace                                      | 11-21 |
| The AP206 Workspace                                      | 11-21 |
| The AP232X Workspace                                     | 11-26 |
| The AP488 Workspace                                      | 11-28 |
| Requirements   | 11-28 |
| Reference Documentation                                  | 11-28 |
| Hints to Avoid Trouble                                   | 11-28 |
| Description of AP488 Functions                           | 11-30 |
| The APLFILE Workspace                                    | 11-42 |
| The DEMO124 Workspace                                    | 11-47 |
| The DEMO206 Workspace                                    | 11-48 |
| The DOSFNS Workspace                                     | 11-48 |
| The EDIT Workspace                                       | 11-51 |
| The EXCHG Workspace                                      | 11-55 |
| The FILE Workspace                                       | 11-56 |
| Functions  | 11-57 |
| Terminology  | 11-58 |
| Examples of Use  | 11-71 |
| The FOIL Workspace                                       | 11-73 |
| The FORTRAN Workspace                                    | 11-74 |
| Restrictions on FORTRAN Programs                         | 11-74 |
| Generation Process                                       | 11-75 |
| Usage Protocol   | 11-78 |
| PFORTPAR Parameter Management Program                    | 11-79 |
| Sample FORTRAN Subroutines (IBM PC Professional FORTRAN) | 11-81 |
| The GEDIT Workspace                                      | 11-82 |
| The GRAPHPAK Workspaces                                  | 11-84 |
| The MUSIC Workspace                                      | 11-86 |
| The PLOT Workspace                                       | 11-87 |

|   |        |
|---|--------|
| The PRINT Workspace .....                   | 11-89  |
| The PROFILE Workspace .....                 | 11-91  |
| The UTIL Workspace .....                    | 11-92  |
| The VM232 Workspace .....                   | 11-95  |
| Selecting a Terminal .....                  | 11-96  |
| Saving Your Line Parameter Definition ..... | 11-102 |
| Connection with the Host .....              | 11-103 |
| Functions .....                             | 11-105 |
| Example of Connection with the Host .....   | 11-108 |
| Auxiliary Files on the Host .....           | 11-112 |

**Notes:**

The APL diskettes have a number of *workspaces* in transfer form (extension “.AIO”) or APL form (extension “.APL”). These workspaces contain functions that you can call from your programs to perform the following applications:

- Dynamic loading of auxiliary processors and non-APL programs (AP2).
- Basic functions to help in building full-screen applications (AP124 and AP205).
- Basic functions to help in building graphic applications (AP206).
- Management of APL files (APLFILE).
- Emulation of the DOS operating system functions (DOSFNS).
- Using the APL full-screen function editor (EDIT).
- Using Personal or Professional Editor to edit APL functions (EDIT).
- Loading and use of exchange assembly programs (EXCHG).
- Using DOS file management routines (FILE).
- Communicating with FORTRAN subroutines (FORTRAN).
- Samples for the music auxiliary processor (MUSIC).
- Using the printer from APL programs (PRINT).
- Initialising the APL session (PROFILE).
- Various graphic applications (PLOT, FOIL, GEDIT, GRAPHPAK).
- Various general purpose utility programs (UTIL).

- Communication with other machines and equipment (VM232, AP232X, AP190, AP488).

These functions also can be used as examples of how to program with the corresponding auxiliary processors in the APL/Personal Computer 2.1 system:

- AP2, EXCHG and FORTRAN use the non-APL program interface auxiliary processor (AP2).
- PRINT uses the printer auxiliary processor (AP80).
- PROFILE uses the stack and profile auxiliary processor (AP101).
- DOSFNS and UTIL use the DOS-BIOS interface auxiliary processor (AP103).
- The EDIT function in the EDIT workspace and the AP124 workspace use the VSAPL compatible full-screen auxiliary processor (AP124).
- The EDAPL function in the EDIT workspace uses AP2 and AP210.
- AP190 uses the IBM PC 3278/79 emulation card communications auxiliary processor (AP190).
- AP205 uses the APL/PC 1.0 compatible full-screen auxiliary processor (AP205).
- AP206, PLOT, FOIL, GEDIT and GRAPHPAK use the graphics auxiliary processor (AP206).
- FILE and APLFILE use the file auxiliary processor (AP210).
- VM232 uses the RS232 interface auxiliary processor (AP232) as well as the file manager (AP210 and FILE).
- AP232X uses the extended RS232 interface auxiliary processor (AP232X).

MUSIC uses the music generator auxiliary processor (AP440).

AP488 uses the GPIB/IEEE488 interface auxiliary processor (AP488).

## **The AP2 Workspace**

The AP2 auxiliary processor allows you to define special areas of memory (*partitions*) where non-APL programs may be loaded and executed, and non-executable files may be loaded. Memory for the partitions is taken out of the APL active workspace, and returned to it when the partitions are discarded. A certain number of partitions (usually 8) may be defined at the same time. Memory allocation is dynamic - there is no restriction in the memory size or the address where a partition may reside at a given time.

Files of four different types may be loaded in a partition:

- 1: Non-executable file.
- 2: APL auxiliary processor.
- 3: Exchange-assembly program, i.e. a program capable of accepting data from APL and returning a result. For an explanation of how these programs may be built, see the description of the AP2 auxiliary processor in Chapter 12, "Auxiliary Processors" ("The Non-APL Program Interface Auxiliary Processor: AP2" on page 12-4), as well as Chapter 13, "How to Build an Auxiliary Processor").
- 4: Standard DOS program. No direct data exchange takes place in this case, though data may be passed through files (see the EDAPL function in the EDIT workspace, as an example).

If a program of type 3 or 4 is to be executed, an additional parameter (restore type) indicates the information to be restored after execution ends:

- 0: Neither the interrupt vector nor the program code are restored after execution of these programs. The programs are assumed not to destroy the interrupt vector.
- 1: The interrupt vector is automatically restored after execution of these programs. The program code is not restored.
- 2: The program code is automatically restored after execution of these programs. They therefore become reusable. (This is applicable, for instance, to the Personal Editor). However, the interrupt vector is not restored, and is assumed to be maintained by execution of the program.
- 3: Both the interrupt vector and the program code are automatically restored after execution of these programs.

The default value of this parameter is 0.

The AP2 workspace provides a set of simple functions that perform non-APL program operations. The functions included are:

- *HELP* - Provides a listing of all the functions in the workspace together with their syntax.
- *CLEAR n* - Unloads all programs currently active, and redefines the maximum number of partitions to *n*.
- *MAP* - Provides a listing of all the partitions and their contents.
- *n GET p* - Assigns space to partition *p*. Space is measured in paragraphs of 16 bytes, i.e. a size of *n* paragraphs is equivalent to  $16 \times n$  bytes.
- '*filespec*' *LOAD p[,t[,k]]* - Loads the indicated file into partition *p*. *filespec* can be given both in

DOS or in APL library format (see description of *filespec* in the FILE workspace, "Terminology" on page 11-58). If *t*, the type of the file, is not given, it is assumed to be 4. If *k*, the restore information, is not given, it is assumed to be 0. Space for the file is assumed to have been requested previously, by means the *GET* function.

For file types 2-4, if the *filespec* does not contain the extension, ".COM" is assumed. If the file is not found, the operation is retried with an extension of ".EXE".

- '*filespec*' *GLOAD* *p* [, *t* [, *k*]] - Loads the indicated file into partition *p*. If the type is not given, 4 is assumed. If the restore information is not given, 0 is assumed. The appropriate space to load the file is automatically requested.
- *UNLOAD* *p* - Unloads any file previously loaded in partition *p*.
- *FREE* *p* - Frees the space currently occupied by partition *p*.
- *UFREE* *p* - Unloads the file and frees the space occupied by partition *p*.
- [*p*] *AUXP* '*filespec*' - Loads the indicated auxiliary processor in partition *p*, or in the first available partition, if *p* is not given.
- ['*parameters*'] *RUN* *p* - Executes the program contained in partition *p*, passing to it the indicated parameters.
- [*p*] *PE* '*filespec*' - Loads the Personal Editor in partition *p* (1, if *p* is not given) and edits the indicated file. The partition is assigned a size of 64 K bytes.



*Notes:*

1. *Programs to be executed using AP2 should not use the DOS 4A or 4B functions.*
2. *Programs that grow into storage below themselves will need a partition of sufficient size to allow for this.*
3. *Programs should not rely on Ctrl-Break being available, as this is disabled by AP2.*
4. *Programs that expect the entire memory below COMMAND.COM to be available to them should not be executed from AP2.*

**Example Session**

```

4000 GET 1 A Get a 64k partition
4000x16 Bytes reserved for partition 1
Return code 0
      A Load the DOS FORMAT Program
      'C:\DOS21\FORMAT' LOAD 1
Loading module C:\DOS21\FORMAT in partition 1
Return code 0
      A Execute the FORMAT program
      'A: /V' RUN 1
Executing program loaded in partition 1
Parameters A: /V
Insert new diskette for drive A:
and strike any key when ready
(Output from Format, reply N to return to APL).
End of execution. Return code 0

```

**The AP124 Workspace**

This workspace contains a set of cover functions to assist in the use of the full screen auxiliary processor, AP124, in an application. A screen definition facility makes it possible to define a screen as a set of fields, each one with its own name or number, where information can be sent or retrieved by means of appropriate functions.

Using a fullscreen interface adds a professional appearance and, more importantly, can yield substantial productivity gains to the APL applications you build. To that end, APL/Personal Computer 2.1 is provided with a powerful auxiliary processor and an easy APL interface to assist you in this area.

## Fundamentals

The display you are using on your IBM Personal Computer should be considered as a character array for the purposes of the following discussion.

The size of this array depends on the screen mode: it can be either a 25 by 80 or a 25 by 40 character array. This area may be sub-divided into smaller rectangular sections, more convenient to your data processing needs. These rectangular areas are called fields. The fullscreen auxiliary processor, AP124, works only in terms of fields. Functions in the AP124 workspace perform actions to a field or a group of fields.

Fields have type, determining how a field may be acted upon: There are two types of fields: “input/output”, allowing input from the keyboard to be typed and displayed; and “output only”, capable of receiving data from an APL function but not from the keyboard.

Fields also have attributes which define how the information contained in the fields is to be displayed. Attributes are expressed as an integer value, depending on the actual display to be used. Some of these attributes define the colour of the field or whether it should be displayed as blinking, reverse video, etc.

## Building a Menu

As an example of the use of the AP124 workspace, we will define a screen, containing some information and requesting data from the user. The AP124 workspace includes functions allowing you to define menus quickly, and making it easy to maintain them.

Let's assume the menu you wish to define is for an overtime payment system. You should collect and process the following data:

- Employee clock number.
- Employee name.
- Number of overtime hours worked each day (Monday to Sunday).
- Per-day overtime rate. This will be fixed at the moment, but may be adjusted at a later date.

Now we have to step through some simple decisions to design the screen: First, how big should the screen be, 25 by 40 or 25 by 80 characters? We will choose 25 by 40, since this size is more appropriate to the manual page width. Second, we must sketch the layout for the fields. The following is a representation of the desired menu:

```

000000000111111111222222222233333333334
1234567890123456789012345678901234567890
01          APL Overtime System
02-----
03
04
05Input the following data, press Enter:
06
07   Employee Number   XXXXXXXX
08   Employee Name     XXXXXXXXXXXXXXXXX
09
10       Mon  Tue  Wed  Thu  Fri  Sat  Sun
11Hours  X.XX X.XX X.XX X.XX X.XX X.XX X.XX
12Rate  1.25 1.25 1.25 1.25 1.25 1.50 2.00
13
14
15
16
17
18
19
20-----
21The following options may also be used:
22
23  F1 - Help           F3 - Exit
24
25

```

The row and column numbers at the top and left of the diagram are included just for clarity.

We also need a name for the screen: in this example, we will use the name OVERTIME.

Now everything is planned and ready, function *FSDEF* from the API24 workspace will be used to build the menu. The function is invoked thus:

```

FSDEF 'Menu_Name'

```

the first time it is used, and:

```

Field_Def FSDEF 'Field_Data'

```

once per field to be defined.

*Menu\_Name* defines the name by which this screen is to be known. This name should begin with an upper or lower case

letter, the delta ( $\Delta$ ) character or the delta underbar ( $\underline{\Delta}$ ) character; and may continue with any of these, plus the digits 0-9.

*Field\_Def* is a six element vector with the following information:

1. Start row of the field
2. Start column of the field
3. Field height
4. Field length
5. Field type: either 0 (Input/Output) or 2 (Output only).
6. Field Attribute: an integer between 0 and 255. The following are some attribute examples applicable to the monochrome and the colour monitor in the alphanumeric modes (see the IBM Technical Reference Manual for full details):
  - 0 - No display
  - 1 - Underlined
  - 7 - Normal
  - 9 - Highlighted underlined
  - 15 - Highlighted
  - 112 - Reverse video
  - 120 - Highlighted reverse video
  - 129 - Blinking reverse video
  - 135 - Blinking normal

If the field length is given as 0, the length will be automatically derived from the value of *Field\_Data*. This is the data to be written on the field at initial menu usage. It must be a character vector.

Alternatively, the following field definition format may be used to assign a name to the field being defined:

```
Field_Def FSDEF 'Field_Name' FSDEF 'Field_Data'
```

*Field\_Def* and *Field\_Data* are the same as above.  
*Field\_Name* is the name you wish to assign to this field.  
This name should consist of upper or lower case letters, digits, delta ( $\Delta$ ) or delta underbar ( $\underline{\Delta}$ ).

*FSDEF* also allows you to define a group of fields so that they may be referred to collectively. This is especially useful for changing the attribute of a complete set of fields, or to switch the type of a set of fields from Input/Output to Output only, or vice versa. This is done in the following way:

*'Group\_Name' FSDEF Number\_of\_Fields*

*Number\_of\_Fields* is the number of fields defined immediately before the execution of this line that are to be included in the group. Each of these previously defined fields must have been named.

The following function can be used to display a menu after it has been completely defined:

*FSSHOW 'Menu\_Name'*

Now let's write a function to define our sample menu and display it on the screen:

```

[0] DEFINE;A
[1] FSDEF 'OVERTIME'
[2] 1 12 1 20 2 7 FSDEF ' APL Overtime System'
[3] 2 1 1 40 2 7 FSDEF 40p'-'
[4] A←'Input the following data, press enter:'
[5] 5 1 1 0 2 7 FSDEF 'PROMPT' FSDEF A
[6] 7 4 1 16 2 7 FSDEF 'Employee Number'
[7] 8 4 1 16 2 7 FSDEF 'Employee Name'
[8] 7 22 1 7 0 7 FSDEF 'Emp_No' FSDEF ''
[9] 8 22 1 15 0 7 FSDEF 'Emp_Name' FSDEF ''
[10] A←'Mon Tue Wed Thu Fri Sat Sun'
[11] 10 7 1 0 2 7 FSDEF A
[12] 11 1 1 0 2 7 FSDEF 'Hours'
[13] 12 1 1 0 2 7 FSDEF 'Rate'
[14] 11 7 1 4 0 7 FSDEF 'Hours_Mon' FSDEF ''
[15] 11 12 1 4 0 7 FSDEF 'Hours_Tue' FSDEF ''
[16] 11 17 1 4 0 7 FSDEF 'Hours_Wed' FSDEF ''
[17] 11 22 1 4 0 7 FSDEF 'Hours_Thu' FSDEF ''
[18] 11 27 1 4 0 7 FSDEF 'Hours_Fri' FSDEF ''
[19] 11 32 1 4 0 7 FSDEF 'Hours_Sat' FSDEF ''
[20] 11 37 1 4 0 7 FSDEF 'Hours_Sun' FSDEF ''
[21] 'Hours' FSDEF 7
[22] 12 7 1 4 0 7 FSDEF 'Rate_Mon' FSDEF '1.25'
[23] 12 12 1 4 0 7 FSDEF 'Rate_Tue' FSDEF '1.25'
[24] 12 17 1 4 0 7 FSDEF 'Rate_Wed' FSDEF '1.25'
[25] 12 22 1 4 0 7 FSDEF 'Rate_Thu' FSDEF '1.25'
[26] 12 27 1 4 0 7 FSDEF 'Rate_Fri' FSDEF '1.25'
[27] 12 32 1 4 0 7 FSDEF 'Rate_Sat' FSDEF '1.50'
[28] 12 37 1 4 0 7 FSDEF 'Rate_Sun' FSDEF '2.00'
[29] 'Rates' FSDEF 7
[30] 18 1 1 40 2 7 FSDEF 'Msg_Area' FSDEF ''
[31] 20 1 1 40 2 7 FSDEF 40p'-'
[32] A←'The following options may also be used:'
[33] 21 1 1 0 2 7 FSDEF A
[34] A←'F1 - Help           F3 - Exit'
[35] 23 3 1 0 2 7 FSDEF A
[36] FSSHOW 'OVERTIME'

```

You will notice we named some of the fields. These are the ones we wish to operate with. You will also observe we have defined an extra field called *Msg\_Area*, where the program can output any errors found in input validation, or any other system message.

| We would normally write a function like this for each panel in  
| our system. Each of these panel definition functions then only  
| needs to be executed once in order to create the global  
| variables that are used by other functions in this workspace.  
| (Of course, they would have to be re-executed if any changes

to the panels are made by editing the appropriate panel definition function).

Let's look at the next stage of combining the screen with the function that will drive the input-output operation (a "driver"). The following cover functions can be used in the driver:

*FSMODE* 'Mode\_Name'

This function sets the screen mode, where *Mode\_Name* is either an integer or a name:

- 0 CO40 Colour 25 × 40
- 1 BW40 Black and White 25 × 40
- 2 BW80 Black and White 25 × 80
- 3 CO80 Colour 25 × 80
- 4 COGR Colour 320 × 200
- 5 BWGL Black and White 320 × 200
- 6 BWGH Black and White 640 × 320
- 7 MONO Monochrome display

*FSUSE* 'Menu\_Name'

This function initialises the menu named *Menu\_Name*. This is the basic call used to allow you to start using a menu. It will share variables with AP124, load the indicated pre-defined menu, and leave you ready to use it.

*Cursor\_Offset FSSETCURSOR* 'Field\_Name'

This call sets the cursor in a specific field or at any position on the screen. The left argument may be omitted, and the cursor offset will be defaulted to the first position in that field. *Field\_Name* is the name you selected for the field during definition of the menu.

'Data' *FSWRITE* 'Field\_Name'

This function writes *Data* to a given field, the one named *Field\_Name* during menu definition.

*FSWAIT*



This function will display the active menu, and wait for user input. When the user presses a certain key (see below), control returns to APL, and the result of function *FSWAIT* is the following (call it *R*):

- *R*[1] - Return code: 0 if ok, otherwise 1
- *R*[2 3] - Key pressed to complete the call:
  - 0 0 - Enter
  - 0 1 - Alternate Enter (Large Plus Key)
  - 1 *n* - An F Key, where *n* is the key number (1 to 30)
  - 4 1 - Esc Key
  - 4 2 - Ctrl Break
  - 6 1 - Home
  - 6 2 - End
  - 6 3 - PgUp
  - 6 4 - PgDn
- *R*[4] - Field number where cursor was located at return to APL, or zero if it was outside all the fields.
- *R*[5 6] - Cursor offset (row/column) into that field. If field was zero, then offset is from the top-left corner of the screen.
- *R*[7 . . .] - List of fields updated during this *FSWAIT* request.

The list of updated fields will particularly improve your processing time, if used properly. In general, it is only necessary to read and validate those fields, rather than reading back and checking ALL the fields, which, for a very large screen can be a very long process.

Two other functions perform a similar task to *FSWAIT*. Their calling syntax and return are the same, with the following exceptions:

1. *FSSCAN* will refresh the screen, check if any key has been pressed, and return to APL. The BIOS scan code of the

key will be returned in position  $R[2\ 3]$ . If no key had been pressed,  $R[2\ 3]$  will be  $\bar{1}\ \bar{1}$ .

2. *FSINKEY* will refresh the screen, wait until any key is pressed, and return to APL. The BIOS scan code of the key will be returned in position  $R[2\ 3]$ .

*FSREAD 'Field\_Name'*

This function reads data from one field or a group of fields. If the result is numeric, it is a return code indicating that the operation failed. Otherwise, the result will be the contents of the requested field or Group field.

Additional functions assisting in the use of the screen:

- *FSAPLOFF* or *FSAPLON*

Turn the keyboard from APL to National mode (*FSAPLOFF*), and vice versa (*FSAPLON*).

- *FSBEEP*

Sets the beep flag. A beep will sound at the next read and wait call (*FSWAIT*, *FSSCAN*, *FSINKEY*).

- *FSCLEAR*

Clears the display.

- *FSCLOSE*

Purges all global variables and retracts all shared variables. It should be used at the end of your session.

- *FSCOPY*

Sends the current active screen to the printer. AP80 must have been loaded explicitly at APL initialisation, or through AP2.

- *FSFIELD A*

This function translates a field name to the corresponding field number, or vice versa.

- *FSFORMAT*

Returns the active format array.

- *'Data' FSIWRITE 'Field\_Name'*

Immediate write of *Data* to the display and buffer areas.

- *FSOPEN*

Shares variables with AP124. This function is used internally by *FSUSE* or *FSMODE*.

- *FSSCREEN*

Returns a copy of the current screen as a character array.

- *A FSSETFI 'Field\_Name'*

Changes the attribute of a field or a group of fields.

- *A FSSETFT 'Field\_Name'*

Changes the type of a field or a group of fields.

- *FSSTATUS*

Returns the status of the session,

*R[1]* - Return code of call  
*R[2]* - 1: Keyboard in APL mode.  
*R[3]* - 1: Monochrome adapter installed.  
*R[4]* - 1: Colour adapter installed.  
*R[5]* - 1: Beep request pending.

The following is an example of a driver function that uses the above defined menu:

```

[0] DRIVER;R;A
[1] →(FSMODE 'MONO')/E124
[2] →(FSUSE 'OVERTIME')/E124
[3] →(FSSETCURSOR 'Hours_Mon')/E124
[4] ASK:→(1↑R←FSWAIT)/E124
[5] →(1=R[2])/FK
[6] a Examine fields changed
[7] a These are listed by 6↓R
[8] A←FSREAD 'Hours'
[9]
[.] FK:.:.
[.] E124:'Fullscreen Error in Driver'

```

## The AP190 Workspace

This workspace provides some functions to make the use of either: the AP190 (for the IBM PC 3278/79 emulator card), or the AP190I (for the IRMA PC 3278/79 emulator card) versions of the AP190 communications auxiliary processor easier. These functions use a shared variable called *C190*, but do not test for presence of the auxiliary processor. You should do this before using the functions: Execute the instruction `190 □SVO 'C190'`, twice. If the result of the second instance of this instruction is not 2, then the AP has not been loaded.

- *SETUP\_IBM* - Defines a set of variables containing the most commonly used communication control characters under appropriate names (such as *Enter*, *PF1*, *PA1*, etc) for the IBM PC 3278/79 emulation card. These variables can be used to send data to the host in the following way:

*C190←character\_string\_vector*

Example:

*C190←'LOGOFF',Enter*

- *SETUP\_IRMA* - Defines a set of variables containing the most commonly used communication control characters under appropriate names (such as *Enter*, *PF1*, *PA1*, etc) for the IRMA PC 3278/79 emulation card.

- *WAIT\_HOST* - Waits until the host is in a “ready to receive input” state.
- *KEY 'text'* - Sends characters in *text* as keystrokes to host. This function includes a delay to allow time for the keystrokes to be sent, and this may need to be tuned for different systems. See text in function *INFO* for more information.
- *READSCR* - Returns the screen as an  $n \times 80$  character array.
- *READBASE* - Returns the operator information area as an 80 character vector.
- *CURSOR* - Returns a three element numeric vector. The first two element give the (0-origin) row and column of the cursor on the screen, and the third is a beep flag (0 for no beep, 1 for beep requested by host).
- *GET\_TABLE* - Returns a  $256 \times 2$  array containing the current keyboard translation table.
- *LOAD\_TRANS table* - Sets the keyboard translation table to  $256 \times 2$  array, *table*.
- *TRANS* - An editor for keyboard translation tables. The following keyboard translation tables are supplied in the AP190 workspace: *UK\_ENGLISH*, *UK\_APL*, *US\_ENGLISH*, *US\_APL*, *FRENCH*, *FRENCH\_APL*, *GERMAN* and *GERMAN\_APL*.

Note that the APL translation tables may only be used with AP190I.

- *APL\_ON\_OFF* - Switches between non-APL and APL (*UK\_ENGLISH* and *UK\_APL*) keyboard modes (AP190I only).
- *PWR\_RESET* - Issues a simulated power-on reset.

- *SET\_QUADTS* - A sample program that demonstrates how to set the PC date and time clock using a host VM session.

## The AP205 Workspace

The AP205 workspace has been included for compatibility with those workspaces in the Personal Computer APL system, Version 1.0, that used the old full screen auxiliary processor, AP205. Its use for new applications is not recommended.

## The AP206 Workspace

The AP206 workspace contains a set of basic functions to help using the AP206 graphics auxiliary processor. This processor maintains a set of graphic parameters that may be independently modified by means of a set of functions:

- *SHARE* - Shares variable *G* with the graphic auxiliary processor. This function must be executed before any of the others.

The following functions modify what we call “the parameters” of the graphic processor.

- *MODE n* - Sets the screen mode to *n*. Legal modes are 4, 5 (320-200 graphic screen), 6 (640-200 graphic screen), 7 (monochrome, no graphics), 8 (virtual mode, optionally used while changing other parameters from an APL mode different from the present graphic screen mode. When this mode is used, other parameters may be changed without the screen being erased).
- *BG n* - Changes the background colour to *n* (0-31, see the appropriate PC Technical Reference Manual).

- **PALETTE** *n* - Changes the colour palette to *n* (0-1). This parameter is only useful in mode 4.
- **COLOR** *n* - Sets the colour parameter to *n* (0-255) for all graphics and characters drawn subsequently, until this parameter is changed again.
- **STYLE** *n* - Sets the style parameter to *n* (0-255) for all graphics and characters drawn subsequently, until this parameter is changed again. This sets the top 16 bits of the colour/style parameter and defines the pattern of dots that are used to form the line. A style value of 0 gives a solid line. For an 8 element boolean vector *V* (1 = dot, 0 = space), the style parameter needed is given by **STYLE**  $(8\rho 2)_{1\sim V}$ .
- **THICKNESS** *n1 n2* - Sets the X and Y thickness parameters to *n1* (1-4 in modes 4, 5; 1-8 in mode 6) and *n2* (1-200), respectively. This parameter affects all subsequent graphics and characters until redefined.
- **WINDOW** *n1 n2 m1 m2* - Sets the current window on the graphic screen to a rectangle defined by *n1 n2* (left and right horizontal boundaries) and *m1 m2* (bottom and top vertical boundaries).
- **VIEWPORT** *n1 n2 m1 m2* - Sets the current viewport, mapping a coordinate system into the current window. *n1 n2* are the X coordinates to be assigned to the window left and right horizontal boundaries. *m1 m2* are the Y coordinates to be assigned to the window bottom and top vertical boundaries. The values of *n1, n2, m1* and *m2* must lie in the interval  $[-16383, 16383]$ .
- **POSITION** *p1 p2* - Sets the "incremental position" parameters. All graphics and characters subsequently drawn are automatically displaced *p1* positions in the horizontal direction, *p2* in the vertical direction.
- **SCALE** *s1 s2* - Sets the "scale parameters". All subsequent graphics and characters are applied a percentage scaling (100 means no change) in the horizontal

axis (*s1*) and the vertical axis (*s2*). If *s2* is omitted, *s1* is used for both directions.

Scales that are multiples of 33.33 (rounded to the nearest integer) are recommended for text drawing. Otherwise (or if both scales are unequal) some characters may appear a little distorted.

- ***INCLINATION*** *n* - Sets the "inclination displacement" (a percentage of the value of Y that is added to the value of X for all subsequent graphics and characters). Values of 0, 100, or -100 are recommended for texts.
- ***DELTA*** *n1 n2* - Sets the "automatic displacement" to be added to the normal character separation in character string drawings. *n1* is the displacement in the horizontal direction, *n2* in the vertical direction. This function should be used only for character strings. However, the same parameters have a different interpretation for filling of graphic drawings, and in this case the following function should be used:
- ***n1 FPATTERN n2*** - *n1* and *n2* are the definitions of the fill pattern to be used in the odd and even lines of subsequent filled graphic drawings. Both arguments may contain up to four colour elements (0-3).

Example: **2 0 2 0 *FPATTERN* 0 2 0 2** will fill drawings with a set of red points (in the appropriate palette).

- ***AUTOΔ*** *n* - Where *n* may be one or zero, sets the autodisplacement switch for literals. If *n* is 1, subsequent characters in a string are automatically displaced an amount dependent on the character being drawn. The DELTA displacements are added to this automatic displacement. If *n* is 0, no autodisplacement is included and only the DELTA displacement is applied. Thus, if the DELTA parameters are also zero, all characters in a string will overstrike.



- **AUTOCAT** *n* - Where *n* may be one or zero. If *n* is 1, successive strings are automatically catenated (the POSITION parameter is adjusted to the end of the precedent string). If *n* is 0, the POSITION parameter remains unchanged.
- **HORX** *n* - Where *n* may be one or zero. If *n* is 1, the viewport X axis appears horizontal in the screen. Otherwise, the viewport is rotated 90 degrees counterclockwise with respect to the window, and the X axis becomes vertical.
- **VSB** *n* - Where *n* may be one or zero. If *n* is 1, all graphic operations are performed on a "Virtual Screen Buffer" (VSB) rather than the physical screen buffer (SB). Otherwise, operations are performed directly on the physical screen buffer.

The following two functions affect the parameters as a whole.

1. **PDEFAULT** - Sets all the parameters to their default values.
2. **PARMS** - Displays the present values of all the parameters, in a way that makes it very easy to adjust them using one of the preceding functions.

The following functions draw texts or graphics on the screen:

- **CLEAR** - Erases the graphic screen.
- **FILLW** *n* - Where *n* is a number in the interval 0-255. The current window is filled with the bit configuration defined by *n* (e.g. if *n* is zero, the window is erased).
- **PRTSC** - Copies the window to the graphics printer.
- **TYPE** *x* - Where *x* is a character string, draws the string in the current screen (VSB or SB) according to the present values of the parameters.

- ***DRAW*  $x$**  - Where  $x$  is a three column integer matrix (a graphic matrix), draws a graphic in the current screen (VSB or SB). The second and third columns in  $x$  are the X and Y coordinates of each point in the graphic. The first column is the “visibility” of the movement from the previous point to the next (0, move; 1, draw). For more information on this function, see the description of the AP206 auxiliary processor, “The Graphic Auxiliary Processor: AP206” on page 12-39.
- ***FILL*  $x$**  - Where  $x$  is a graphic matrix, modifies the graphic matrix so that function *DRAW* fills it according to the patterns defined through function *FPATTERN* (see above. Remember the DELTA parameters are the same as those changed by *FPATTERN*). (E.g. *DRAW FILL* 4 3p0 0 0 1 100 0 1 50 100 1 0 0 draws and fills a triangle).
- ***GINPUT*  $x$**  - Where  $x$  is a graphic matrix, draws the graphic in the screen (similarly to *DRAW*) and accepts graphic input modifications from the keyboard, passing back the resulting matrix to APL. For more information on this, see the description of the AP206 auxiliary processor, “The Graphic Auxiliary Processor: AP206” on page 12-39.
- ***COMPACT*  $x$**  - Where  $x$  is a graphic matrix, returns another graphic matrix where unneeded rows have been eliminated. It can be used on the result of *GINPUT* to optimise space.
- ***REVERSE*** - Interchanges the colours drawn on the present window.

The following functions use the screen buffers (VSB and SB) as well as special memory buffers (window buffers, or WB), where copies of the whole or part of the screen may be kept. Screen images can be copied between the APL active workspace and the screen buffers; or between the screen buffers and the screen. The screen buffers may be repeatedly copied to the screen in order to produce an animated display.

- *SCR2BUF n* - Where *n* is an integer in the interval 1-128, copies the current window in the current screen (VSB or SB) to the window buffer (WB) recognised by number *n*. If this WB did not exist, it is created. Otherwise, it is replaced.
- *DELBUF n1 [n2]* - Where *n1 n2* are WB numbers, deletes all WBs in the interval *n1-n2*. If *n2* is not given, only *n1* will be deleted.
- *BUF2SCR n1 [n2 [n3 [n4]]]* - Performs an animated copy from the WBs in the interval *n1-n2* to the current active (VSB or SB). The full copy operation is repeated *n3* times (default is 1). A delay of *n4* hundredths of a second (default is 4) will be included between any two subsequent copies.
- *BUF2APL n* - Returns to APL the present contents of WB number *n*, as a literal matrix.
- *APL2BUF x* - Where *x* is a literal matrix as the one returned to APL by function *BUF2APL*, copies the graphic information in *x* to the first available WB, and returns its number.
- *VSB2SCR* - Copies all the information contained in the virtual screen buffer (VSB) to the physical screen buffer.

## The AP232X Workspace

To assist in using the AP232X asynchronous communication auxiliary processor, a workspace called AP232X is available. This provides an interface to all of the AP232X calls, and includes error checking to provide useful diagnostics on call failures. Two functions called *SHARE\_232* and *RETRACT\_232* are provided to share and retract the *C232* and *D232* shared variables that are used by the other

functions in the workspace. The *SHARE\_232* function should be executed before any of the other functions are used.

Other functions are:

- *SETUP* - Helps you to define the initialisation parameters. You are prompted for them in their proper order, and a global variable called *PARMS* is generated.
- *port INIT232 data* - Initialises the indicated *port* (1 if not given) to the indicated *data* (such as the *PARMS* variable generated by *SETUP*).
- *PARMS232 port* - Recovers the initialisation data for the indicated *port*.
- *port TRANSMIT data* - Transmits the indicated (character) *data* to the indicated *port* (1 if not given).
- *RECEIVE port* - Gets the characters currently in the receive buffer, up to the first turnaround character, or to the buffer end.
- *flag BUFSTAT port* - Returns the transmit and receive buffer sizes for the indicated *port* and the number of characters in each, as a four element vector. If "*flag*" is present, the result is displayed with proper explanations.
- *flag LCR port* - Returns the present value of the LCR register for the indicated *port*. If "*flag*" is present, then the register is presented visually in a proper way.
- *flag MCR port* - Returns the present value of the MCR register for the indicated *port*. If "*flag*" is present, then the register is presented visually in a proper way.
- *flag LSR port* - Returns the present value of the LSR register for the indicated *port*. If "*flag*" is present, then the register is presented visually in a proper way.

- *flag MSR port* - Returns the present value of the MSR register for the indicated *port*. If "*flag*" is present, then the register is presented visually in a proper way.
- *port SETLCR value* - Sets the LCR register for the indicated *port* (1 if not given) to the indicated *value*.
- *port SETMCR value* - Sets the MCR register for the indicated *port* (1 if not given) to the indicated *value*.
- *RESET232 port* - Resets the transmit and receive buffers for the indicated *port*, purging them.
- *SETBRK port* - Sends a break to the indicated *port*. It contains a delay, which should be tuned to your device.

## The AP488 Workspace

### Requirements

- IBM GPIB/IEEE-488 Adapter Card(s)
- IBM Software Support for the GPIB Adapter

### Reference Documentation

- IBM Guide to the General Purpose Interface Bus Support

### Hints to Avoid Trouble

AP488 is an interface between APL/PC 2.1 and the device driver software that is supplied with the programming support for the IBM GPIB/IEEE-488 hardware. Most, but not all, of the functions that are available in the IBM software package have been implemented. This auxiliary processor may only be used with the IBM General Purpose Interface Bus

Programming Support (Part number 6024201, Feature code 4201).

The following functions are NOT supported:

- **IBRDA** - Read Data Asynchronously
- **IBWRTA** - Write Data Asynchronously
- **IBCMDA** - Write Commands Asynchronously

The following variables are reserved:

- **C488** - The AP488 control variable
- **D488** - The AP488 character vector variable
- **IBSTA** - The GPIB status variable
- **IBERR** - The GPIB error number variable
- **IBCNT** - The GPIB auxiliary count variable

The three functions that process character vectors (**IBRD**, **IBWRT**, **IBCMD**) are modified by the high order bit of the device number. If this bit is zero, translation will take place between the APL internal character set and ASCII. If the bit is one, then no translation will occur. The high order bit may be reset with the function "**ASCII**" and set with "**BINARY**".

In all functions, the word 'device' refers to the integer that is returned by the **IBFIND** function. This always refers to an instrument. The word "adapter" is also an integer returned by **IBFIND**, but in this case it refers to the IBM adapter board itself, not an instrument. The word "either" means either an instrument or an adapter board.

If you have not already done so, please read the introductory chapters in the IBM GPIB Support Package documentation before continuing with AP488. It is easy to become frustrated unless you are familiar with the basic concepts of the IEEE-488 standard.

Some additional hints to avoid trouble:

- Few instruments can support tri-state timing (the default with the IBM support package). Unless you are certain that all instruments on your interface adapter can support this option, you should select the open collector interface instead.
- Local Lockout is good in a production environment, but not when you are setting up an experiment.
- Automatic Serial Polling is not always a good idea. Some instruments can not respond to a serial poll.
- Do not try to read from, or write to a file that has the same name as an instrument. For example, if you have an instrument named "DVM", do not try to access a file named "DVM.DAT". DOS will intercept this access and send the data to your instrument with sometimes amusing but always unpredictable results.
- Read your instrument manual *carefully*. Some instruments are *very sensitive* to the format of data that is sent to them. For example, one instrument may *require* line feed terminator on every message, while another may totally lock up if it receives one. *Data format is instrument specific and not part of the IEEE-488 standard.*
- Changes to "GPIB.COM" do not take effect until the next reboot of your system. Be sure to reinitialise your system via Ctrl-Alt-Del after making any changes with the *IBCONF* configuration program.
- The entry "DEVICE = GPIB.COM" must be in your "CONFIG.SYS" file. In addition, "CONFIG.SYS" must be in the root directory of your boot disk. "GPIB.COM" must also reside on your boot disk. After booting your system, neither "CONFIG.SYS" nor "GPIB.COM" are required. You may change diskettes without fear of problems until the next time that you reboot.

## Description of AP488 Functions

All functions in this workspace return one or more values. For those functions that do not return a data value, the *IBSTA* status status word is returned.

### Translation On, ASCII

*ASCII address*

This function takes an integer argument (the device number returned by *IBFIND*) and turns off the no translation bit. This is the default, and is not normally necessary unless you have turned on this bit with the *BINARY* function. The returned value is the new device number that you should use for further accesses to the device or adapter.

### No Translation, BINARY

*BINARY address*

This function takes an integer argument (the device number returned by *IBFIND*) and turns on the no translation bit. This bit prevents the automatic translation of data between ASCII and APL's internal representation. You would normally use this function only when the data is a binary data stream such as that returned by a digital oscilloscope. The returned value is the new device number that you should use for further accesses to the device.

### Change Adapter, IBBNA

*device IBBNA 'adaptername'*

This function changes the adapter used to access the specified device. This change is temporary and disappears after you leave APL. '*adaptername*' is a string containing 'GPIBx' where 'x' is a number from zero through three.



## Active Controller, IBCAC

*flag IBCAC adapter*

The flag is zero to take control immediately (possibly asynchronously), and non-zero to force synchronous assumption of control with respect to data transfer. Adapter refers to a GPIBx adapter handle obtained from *IBFIND*.

## Selected Device Clear, IBCLR

*IBCLR device*

This function sends the listen address(es) of the specified device followed by selected device clear (SDC), unlisten and untalk. It usually clears a device to some specified initial state. Not all devices respond to SDC.

## Send GPIB Commands, IBCMD

*'gpib\_commands' IBCMD adapter*

This function sends data out through the specified adapter with ATN true. It is up to you to ensure that the string contains valid GPIB commands. See Figure 11-1 on page 11-33 for a table of IEEE-488 addresses with their character equivalents (in ASCII). This is one of the commands that is modified by the state of the high order bit of the file handle. Also see *ASCII* and *BINARY*.

| Listen |    | Talk |   | Device # |
|--------|----|------|---|----------|
| 20h    | SP | 40h  | @ | 00       |
| 21h    | !  | 41h  | A | 01       |
| 22h    | "  | 42h  | B | 02       |
| 23h    | #  | 43h  | C | 03       |
| 24h    | \$ | 44h  | D | 04       |
| 25h    | %  | 45h  | E | 05       |
| 26h    | &  | 46h  | F | 06       |
| 27h    | '  | 47h  | G | 07       |
| 28h    | (  | 48h  | H | 08       |
| 29h    | )  | 49h  | I | 09       |
| 2Ah    | ⌘  | 4Ah  | J | 10       |
| 2Bh    | +  | 4Bh  | K | 11       |
| 2Ch    | ,  | 4Ch  | L | 12       |
| 2Dh    | .  | 4Dh  | M | 13       |
| 2Eh    | -  | 4Eh  | N | 14       |
| 2Fh    | /  | 4Fh  | O | 15       |
| 30h    | 0  | 50h  | P | 16       |
| 31h    | 1  | 51h  | Q | 17       |
| 32h    | 2  | 52h  | R | 18       |
| 33h    | 3  | 53h  | S | 19       |
| 34h    | 4  | 54h  | T | 20       |
| 35h    | 5  | 55h  | U | 21       |
| 36h    | 6  | 56h  | V | 22       |
| 37h    | 7  | 57h  | W | 23       |
| 38h    | 8  | 58h  | X | 24       |
| 39h    | 9  | 59h  | Y | 25       |
| 3Ah    | :  | 5Ah  | Z | 26       |
| 3Bh    | ;  | 5Bh  | [ | 27       |
| 3Ch    | <  | 5Ch  | \ | 28       |
| 3Dh    | =  | 5Dh  | ] | 29       |
| 3Eh    | >  | 5Eh  | - | 30       |

These addresses may be used with Parallel Polling.

Primary Listen Address = Device number + 32

Primary Talk Address = Device number + 64

Secondary Addresses extent from X'60' through X'7E' and are always device dependent.

Figure 11-1. IEEE-488 Addresses

### Enable/Disable DMA, IBDMA

*flag IBDMA adapter*

This function enables or disables DMA on the specified adapter provided that DMA was not disabled when you configured your device driver. If the flag is zero, programmed I/O is used (temporarily), and when non-zero, DMA is reactivated.

**Change/Disable EOS Termination Method, IBEOS***flag IBEOS either*

This routine changes the way the EOS termination byte is handled by *IBRD* and *IBWRT*. See the IBM documentation for a full description.

**Change/Disable END Termination Method, IBEOT***flag IBEOT either*

If the supplied flag is zero, the END message is not sent concurrently with the last byte of an *IBWRT*. If the flag is non-zero, then it is. This can be very useful when you're making adapter level writes.

**Return Unit Descriptor, IBFIND***IBFIND 'defined488thing'*

This is always the first thing that has to be done before an instrument or controller may be accessed. It returns an integer value that is used in all subsequent device or board level calls. If the integer returned is negative, then an error has occurred. See *IBSTA*, *IBERR* and *IBCNT* for a complete description of the error.

**Active Controller to Standby, IBGTS***flag IBGTS adapter*

If zero, the integer simply disables the controller function. If non-zero, then the controller is disabled, but monitors the bus waiting for an END message. When the END message is detected, the adapter enters the NRFD holdoff state. This is normally used in board level I/O calls.

## Set/Clear Individual Status Bit, IBIST

*flag IBIST adapter*

Although an adapter is specified, this function is used when the PC is NOT the active controller but rather a device being controlled elsewhere. If zero, flag sets the parallel poll status bit false and if non-zero, it sets this bit true. The actual state of the bit (0 or 1) is specified by the external controller when it sends the parallel poll configure message.

## Go to Local, IBLOC

*IBLOC either*

IBLOC sends unlisten, listen address(es) of the specified device, Go to Local (GTL), unlisten and untalk. This temporarily overrides the Local Lockout state. (Local Lockout is useful when you have knob twiddlers coming into your lab but horrible when you're trying to set up an experiment).

## Online/Offline, IBONL

*flag IBONL either*

If the flag is zero, the device or adapter is placed in an offline state (essentially a close function). The device descriptor is no longer valid and can not be used to place the device back online! *IBFIND* is the inverse of this function.

## Change Primary Address, IBPAD

*address IBPAD either*

The address specifies a new primary address for subsequent GPIB activity. It may range from zero through thirty (be sure not to conflict with anything already on the bus). This is primarily useful when you are adding a device to a system temporarily and don't want to configure it in permanently. Also see *IBSAD*, *IBEOS* and *IBEOT*.

## Pass Control, IBPCT

*IBPCT device*

This function passes control of the GPIB bus to another controller. The adapter enters controller idle state at the end of this function (CIDS). Be sure that the device you specify can act as a controller.

## Parallel Poll Configure, IBPPC

*flag IBPPC either*

See the IBM manual for a description of the flag word.

## Read Data, IBRD

*IBRD either*

This routine can access either an adapter or a device. It reads data until

- EOS is detected (if active)
- END is detected (always)
- Buffer is full (always)

Be careful! This is one of the routines affected by the state of the high order bit of the file handle. Also see *ASCII* and *BINARY*.

## Read from a Device into a DOS file, IBRDF

*filename IBRDF either*

This routine performs a read from the specified device or adapter and sends the output to a DOS file rather than back to APL. The DOS file is opened for output, not append, so only one record may be placed in each unique file. The

filename is a character vector that may include a full drive, path, filename and extension specification.

## **Conduct a Parallel Poll, IBRPP**

*IBRPP either*

This routine returns a parallel poll byte. If you specify a device, it is mapped into the correct adapter instead.

## **Request/Release System Control, IBRSC**

*adapter IBRSC integer*

If the integer is zero, all system control functions are disallowed until a *IBONL* followed by an *IBFIND* occurs.

## **Request Serial Poll, IBRSP**

*IBRSP device*

This function returns the serial poll byte from the specified instrument as an integer. If the integer is negative, an error has occurred. Analyse *IBSTA*, *IBERR* and *IBCNT* to find the exact problem.

## **Set Serial Poll Status, IBRSV**

*flag IBRSV adapter*

This function puts the specified flag into the serial poll response register of the specified controller. If bit 6 (X'40') is true, then service is requested as well. Normally used when the adapter is not the system controller.

## Change Secondary Address, IBSAD

*address IBSAD either*

The address specifies a new secondary address for subsequent GPIB activity. It may range from X'60' through X'7E' normally. If it is zero or X'7F' then the secondary address is disabled. This is a temporary function. Also see *IBPAD*, *IBEOS* and *IBEOT*.

## Send Interface Clear, IBSIC

*IBSIC adapter*

This routine sends the interface clear message for 100 microseconds.

## Set Receive Buffer Size, IBSIZE

*IBSIZE integer*

This routine specifies the maximum data size for *IBRD*. Must be less than 32000.

## Set/Clear Remote Enable Line, IBSRE

*flag IBSRE adapter*

If the flag is zero, the remote enable line of the specified adapter is turned off; if one then it is turned on.

## Change or Disable Timeout Limit, IBTMO

*time IBTMO either*

Time may range from zero through seventeen. Changes the timeout limit on the specified device or adapter. Range is from 10us through 1000 seconds. See Figure 11-2 on page 11-39 for a list of the available control codes.

| Mnemonic | Code | Time Out |
|----------|------|----------|
| TNONE    | 0    | Infinite |
| T10US    | 1    | 10 us    |
| T30US    | 2    | 30 us    |
| T100US   | 3    | 100 us   |
| T300US   | 4    | 300 us   |
| T1MS     | 5    | 1 ms     |
| T3MS     | 6    | 3 ms     |
| T10MS    | 7    | 10 ms    |
| T30MS    | 8    | 30 ms    |
| T100MS   | 9    | 100 ms   |
| T300MS   | 10   | 300 ms   |
| T1S      | 11   | 1 s      |
| T3S      | 12   | 3 s      |
| T10S     | 13   | 10 s     |
| T30S     | 14   | 30 s     |
| T100S    | 15   | 100 s    |
| T300S    | 16   | 300 s    |
| T1000S   | 17   | 1000 s   |

Figure 11-2. Timeout Control Codes

## Trigger Device, IBTRG

*IBTRG device*

This routine sends a group execute trigger message (GET) to the specified device. Not all instruments respond to GET.

## Wait for Selected Event, IBWAIT

*integer IBWAIT either*

The integer is a mask (See Figure 11-3 on page 11-40). This routine permits waiting for a specified event or events to occur.



| Mnemonic | Bit | Description             |
|----------|-----|-------------------------|
| ERR      | 15  | Error Detected          |
| TIMO     | 14  | Timeout                 |
| END      | 13  | EOI or EOS              |
| SRQI     | 12  | SRQ detected by CIC     |
| RQS      | 11  | Device requires service |
| CMPL     | 8   | DMA Completed           |
| LOK      | 7   | Local Lockout State     |
| REM      | 6   | Remote State            |
| CIC      | 5   | Controller-In-Charge    |
| ATN      | 4   | Attention Asserted      |
| TACS     | 3   | Talker Active           |
| LACS     | 2   | Listener Active         |
| DTAS     | 1   | Device Trigger State    |
| DCAS     | 0   | Device Clear State      |

Figure 11-3. Mask Layout

## Write Data, IBWRT

*data IBWRT either*

This routine can access either an adapter or a device. It sends data until:

- EOS is detected (if active)
- Buffer is empty (always)

Be careful! This is one of the routines affected by the state of the high order bit of the file handle. Also see *ASCII* and *BINARY*.

## Write Data from DOS File to Device, IBWRTF

*filename IBWRTF either*

This routine writes all of the data from the specified DOS file to the device or adapter. No translation is done; ALL characters are sent. This includes any CR/LF's in the file and the Ctrl-Z that is placed in the file by many editors. EOI is sent concurrent with the last byte of data. A full drive, path, filename and extension specification may be given for filename. Debug is sometimes useful for modifying the data file.

**Check Return Code, CHK\_488**

This function may be used to validate the return codes generated by the functions listed above. It takes a right argument of the value returned by these functions, and a left argument listing the valid return codes that should be accepted.

Example:

```

A First get device handle
HANDLE ← IBFIND 'GPIB0'
A Set Remote Enable Line
A This puts device into remote mode
A Suppress return code of 256
256 CHK_488 1 IBSRE HANDLE
A Set Interface Clear
304 CHK_488 IBSIC HANDLE
A Send command to GPIB:
A Unlisten (ASCII "?"), PC Talk address,
A DVM Listen address.
A Suppress both 376 and 312 return codes
376 312 CHK_488 '?@(' IBCMD HANDLE
A Now send command string to the DVM to
A initiallise it such that it will send
A data. This is device dependent!
372 296 CHK_488 'Command string' IBWRT HANDLE
A DVM now ready to send out data.
A Send commands to GPIB:
A Unlisten, DVM Talk address,
A PC Listen address
372 CHK_488 '?H...' IBCMD HANDLE
A Now listening to DVM
A Read in the data
VOLTS ← IBRD HANDLE
VOLTS
+1.234567E+02(cr,lf)

```

## The APLFILE Workspace

APLFILE is a set of functions which may be used to create and use a PC DOS file of APL arrays. The functions are most useful when a file must contain APL arrays of arbitrary rank and dimension, when variable length records must be accessed randomly, or when records are longer than the maximum length otherwise permitted. APLFILE uses auxiliary processor AP210 supplied with APL/PC 2.1.

It is designed to be as compatible as possible with the VAPLFILE workspace distributed with VSAPL and APL2. The main functions are:

- ***L CREATE F***

*F* is a character vector containing the PC DOS name to be assigned to the file. *L* is a physical description of the file: *L*[0] = the maximum number of arrays which may be written. Default = 100. *L*[1] = The blocksize of the file used to store the data. Default = 512. Each array requires an integral number of blocks. The maximum allowable block size with AP210 is 32512. *L*[2] = The number of blocks for data. Default =  $1.1 \times L[0]$ . The left argument "*L*" may be elided, in which case the default file size parameters will be used unless the right argument is composed with the "*AS*" function. The *AS* function may be used to specify both the file name and the file parameters as *CREATE F AS L*.

- ***USE F***

*F* is a character vector containing the name of an existing APLFILE file. This function shares appropriately named variables with the file processor, opens the file, and defines global variables associated with the file in use. A left argument consisting of the single character '*R*' will cause *USE* to open the file for read/only access.

- ***RELEASE F***

*F* is a character vector containing the name of a file. This function retracts and expunges the variables shared with AP210 and expunges the global variables associated with the file in use. The file is closed. The explicit result of the function 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.

- ***( F AT I ) SET A***

Sets *A* as the *I*-th. array in the file whose name is in the character vector *F*. (*F*[*I*]←*A*). *I SET A* may 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 index 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 will never lose an existing array.

- ***GET F AT I***

Returns the array set in the *I*-th. position of the file whose name is in the character vector *F*. (*F*[*I*]). *GET I* may 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 index origin.

- ***DELETE F***

Erases the file whose name is in the character vector *F*.

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

- ***RENAME F AS G***

Renames file *F* as *G*. The old and new file names may also be specified as left and right arguments (e.g. *F RENAME G*), or as a single right argument containing the two names separated by a space.

- ***REPLICATE F AS G***

Replicates file *F* as *G*. The old and new file names may also be specified as left and right arguments (e.g. *F REPLICATE G*), or as a single right argument containing the two names separated by a space.

- *REFORMAT F AS L*

Reformats file *F* as a new file whose size is described by containing all the components of the existing file *F*. *L* is as defined for the *CREATE* function. *F* and *L* may also be given as left and right arguments (e.g. *F REFORMAT L*).

- *FILEIN A AS F*

Reads contents of AIO file *A* into APLFILE file *F*. The format of the AIO file is assumed to be as generated by the *FILEOUT* function described below. The AIO and APLFILE file names may also be specified as left and right arguments (e.g. *A FILEIN F*), or as a single right argument containing the two names separated by a space.

- *FILEOUT F AS A*

Writes contents of APLFILE file *F* into AIO file *F*. Each record is written to the AIO file as an APL variable. The *N*th record is written as a variable called "*RECORDN*". The APLFILE and AIO file names may also be specified as left and right arguments (e.g. *F FILEOUT A*), or as a single right argument containing the two names separated by a space.

- *GET1 F AT I*

Returns the account number of the person who last set the *I*-th. element of *F* (always 1 in APL/PC 2.1), and the time stamp of the set.

- *GETTs F AT I*

Returns the time stamp when the *I*-th. array was set. If *I* is negative, the time stamp of the file creation is returned.

- *SIZE A*

Returns the size of array *A* in bytes.

- *RHO F*

Returns the number of arrays which may be written in the file *F*, where file *F* is a file which is in use.

- *ERASE F AT I*

Undefines the *I*-th. element of *F* and releases the space used by it in the file. *I* may be a vector.

- *FREEBLOCKS F*

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 minimise the fragmentation. The result of *FREEBLOCKS* could change after executing *COMBINE*.

*Note:* Small scalar numbers take zero blocks.

- *EXIST F AT I*

Returns 1 if *F[I]* has been set, 0 if *F[I]* does not contain a value, or -1 if *I* is out of range. *I* may be a vector.

- *SHVARS*

Returns a matrix of the names of currently shared variables.

The following namelists exist in the workspace:

*FILEREAD* - Functions needed for get access only.

**FILEWRITE** - Additional functions needed for set access.

**APLFILE** - All the functions in the above list and the optional ones.

When an error is encountered during the use of the APLFILE functions, an appropriate message is printed, after which execution of the current function and any associated pendant functions is abandoned. However, if the variable "o" contains a negative number, then after any error message, execution will be suspended with a normal APL error message. This may be useful when debugging new applications.

It is recommended that the functions "**CHK**" and "**TRY**" normally be locked.

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. This is formed from the specified filename with any drive letter, library number, path or file extension removed. For this reason, any files to be accessed simultaneously must have different names.

'**CA**', **FILEID** and '**DA**', **FILEID** are variables shared with AP210.

'**FA**', **FILEID** contains the file description as follows:

- 0 - Account number of file creator (always 1 in APL/PC 2.1).
- 1 - Number of arrays permitted.
- 2 - Blocksize.
- 3 - Number of data blocks.
- 4 - Row dimension of an index array.
- 5 - Number of index arrays.
- 6 - Number of salvage index arrays.
- 7 - Total number of blocks.
- 8-14 - **ZS** at creation.

## The DEMO124 Workspace

This workspace is designed to give the user a sample of the capabilities of the AP124 auxiliary processor.

The demonstration provides an on-line reference to the various calls to AP124, and shows some interesting applications for it, both in industry and education.

AP124 is a simple to use and yet powerful interface to both the keyboard and video services provided by the IBM Personal Computer hardware.

This demonstration requires AP124 to be in the APL session, and, if you wish to print any of the screens, also AP80. This may be accomplished by specifying the APs at session start-up time as in:

```
APL AP80 AP124
```

Alternatively, they may be loaded by using AP2.

This workspace is supplied in “.APL” form on the system diskette and may be loaded by a `)LOAD DEMO124` command.

*Note:* A confusing situation can arise if this demonstration workspace is run on a PC with a colour adapter installed but no screen attached, as all output will be directed to this adapter giving the appearance of a hung PC. Press ESC twice and then Alt-F1 to return control to the monochrome adapter.



## The DEMO206 Workspace

*DEMO206* is a sample workspace demonstrating most of the AP206 capabilities. APL should be loaded with AP206. The starting function in this workspace (which is executed through  $\square LX$ ) is called *DEMO*. The demonstration is bilingual (English/Spanish). The minimum storage requirement is 320k.

This workspace is supplied in “.APL” form on the system diskette and may be loaded by a *LOAD DEMO206* command.

*Note:* A confusing situation can arise if this demonstration workspace is run on a PC with a colour adapter installed but no screen attached, as all output will be directed to this adapter giving the appearance of a hung PC. Press ESC twice and then Alt-F1 to return control to the monochrome adapter.

## The DOSFNS Workspace

The DOSFNS workspace uses AP103 to provide emulation of some of the more useful DOS commands. Except where noted, these are monadic functions. The commands available are:

- *CHDIR* - Queries or changes the current directory. The argument is of the form '*d:path*' where '*d:*' is the drive to be changed and '*path*' is the path leading to the directory required. If only a drive is specified, the current directory is displayed. If the argument is an empty vector, the current directory of the default drive is displayed.
- *CHDRIVE* - Queries or changes the current default drive. The argument is of the form '*d:*' where '*d:*' is the drive to become the current default drive. (The “:” is optional). If the argument is an empty vector, the drive letter of the current default drive is displayed.

- *CLS* - Clears the screen. It is a niladic function.
- *DATE* - Allows the current date to be set. *DATE* is a niladic function. It displays the current value and allows a new value to be entered.
- *DIR* - Displays a directory list. The argument is of the form '*d:path*' where '*d:*' is the drive of which the directory is to be displayed and '*path*' is the path leading to the directory whose contents are to be displayed. More finely defined searches are also possible by specifying a filespec after the path (e.g. '*\*.APL*' will list all files with an extension of "APL"). Hidden and system files are included in the display and are marked with the letters "H" and "S" respectively. Read-only files are similarly marked with an "R".
- *ERASE* - Erases a file. The argument is of the form '*d:filename*' where '*d:*' is the drive on which the file is stored and '*filename*' is the name of the file to be erased, and may include a path definition.
- *MKDIR* - Creates a new sub-directory. The argument is of the form '*d:path*' where '*d:*' is the drive on which the new sub-directory is required and '*path*' is the path leading to the directory required. A new directory is created at the end of the specified path.
- *RENAME* - Renames a file. The argument is of the form '*d:oldname newname*' where '*d:*' is the drive on which the file is stored, '*oldname*' is the name of the file to be renamed (and may include a path definition), and '*newname*' is the new name by which the file is to be known. *RENAME* may also be used dyadically, with the old name as the left argument and the new name as the right argument.
- *RMDIR* - Removes a sub-directory. The argument is of the form '*d:path*' where '*d:*' is the drive on which the sub-directory is to be deleted and '*path*' is the path leading to the directory.

- **SETDATE** - Allows a file's time-stamp to be changed. The argument is of the form '*d:filename*' where '*d:*' is the drive on which the file is stored and '*filename*' is the name of the file, and may include a path definition. It displays the current time-stamp and allows a new value to be entered. It will then update the time-stamp and re-display the new value. Press "Enter" when satisfied with the time-stamp displayed.
- **SETMODE** - Allows a file's attributes (achive, system, hidden or read-only) to be changed. The argument is of the form '*d:filename*' where '*d:*' is the drive on which the file is stored and '*filename*' is the name of the file, and may include a path definition. It displays the current attributes and allows new attributes to be entered. It will then update the attributes and re-displays the new settings. Press "Enter" when satisfied with the attributes displayed.
- **SPACE** - Queries the available space on a drive. The argument is of the form '*d:*' where '*d:*' specifies the drive for which space usage is to be displayed. (The ":" is optional). If the argument is an empty vector, the space on the current default drive is displayed.
- **TIME** - Allows the time-of-day clock to be set. **TIME** is a niladic function. It displays the current value and allows a new value to be entered.
- **VERIFY** - Queries or changes the current verify switch setting. An argument of **1** or **0** will set or reset (respectively) the verify switch. An empty vector argument will display the current setting.
- **VOL** - Queries the volume id of a drive. The argument is of the form '*d:*' where '*d:*' specifies the drive for which the volume label is to be displayed. (The ":" is optional).

# The EDIT Workspace

This workspace contains two functions:

## EDIT

This is an APL full-screen, defined-function editor. It is used with AP124.

To use the *EDIT* function, you must include the full-screen auxiliary processor, AP124, as a parameter to the APL command at load time before you begin an APL work session, or load it dynamically through AP2. For example,

```
APL AP124
```

To edit an APL function with the full-screen editor, you must copy the EDIT workspace into your active workspace with the command:

```
)IN EDIT EDIT
```

If the name of the function you want to create or edit is *FN1*, enter the following line:

```
EDIT 'FN1'
```

The screen is cleared and the first page of the function definition appears. You may now move the cursor, using the four arrow keys on the numeric keypad, change any character in the lines displayed, insert characters (with the Ins key), delete characters (with the Del key), delete to the end of a line (with the Ctrl-End key combination), delete to the beginning of a line (with the Ctrl-Home key combination), and move the cursor to the beginning of the next line (by pressing the Tab key). The function keys can also be used as indicated in the lowest line of the screen. The function keys are described next.

**F1** TOP - Displays the first or top page of the function.

- F2 BOT - Displays the last or bottom page of the function.
- F3 END - Ends function definition. All modifications to the function are kept and the new definition of the function will be established in the active workspace. If this process fails, the bottom line of the display will be updated with an error message to indicate the line number found to be in error.
- F4 LIN - Clears the screen and displays only the line pointed to by the current cursor position. You can use this to edit lines longer than the screen width. The maximum line length this method allows is 160 characters.
- F5 INS - Inserts a new line after the current cursor position.
- F6 COP - Copies a line: You must first move the cursor to the line you want to be copied, then press F6. An asterisk (\*) will be displayed by the word COP on the bottom line of the screen. The system is now in "copy" state. Then move the cursor to the line after which the indicated line is to be copied (possibly on another page). Finally, press F6 again to cause the copy to take place. Alternatively, press Esc to cancel the copy operation. The asterisk is erased and the system is no longer in "copy" state.
- F7 XEC - Executes the line pointed to by the cursor. The line is executed under the control of  $\square EA$ , so that any error that occurs will not suspend the execution of EDIT.
- F8 EOL - Moves the cursor to the end of the line pointed to by the cursor.
- F9 DEL - Deletes the line pointed to by the cursor.
- F20 (Shift-F10) CAN - Cancels function definition. No changes are kept. The function remains as it was at the beginning of the edit session.

## IBM Internal Use Only

All other function keys are ignored.

### Other Special Keys:

Tab Moves the cursor to beginning of the next line.

Shift-Tab Moves the cursor to beginning of the preceding line.

PgDn Displays the next page.

PgUp Displays the preceding page.

End Displays the last page.

Home Displays the first page.

Enter Moves the cursor down one line at a time. If the cursor is in the last displayed line when this key is pressed, the whole function will be scrolled up one line.

In addition to using the F9 key, a line may be deleted by moving the cursor to the beginning of that line and pressing the Ctrl-End key combination. The line will remain on the screen as a blank line, but will be automatically deleted when F3 is pressed to end the edit session. Only the part of the line contained in the currently displayed page will be erased. If the line to be erased extends beyond the right edge of the screen, you must press F4 with the cursor on this line, and then erase it using the Ctrl-End key.

Locked functions cannot be edited using this function.

You should also be warned that no check for function suspension is made. If the function being edited is suspended, the END command (F3) will fail, though you may still rename the function so as not to lose the changes made.

This full-screen function editor can be used to create new defined functions and modify existing ones.

You may also find this editor useful to copy a function to a new name, leaving the old version intact. Just invoke *EDIT*

for the original function, change the name in the header line, and press END (F3).

## EDAPL

For larger functions you might prefer to use a system editor, and APL provides a method for doing just that.

The EDIT workspace contains a function called *EDAPL*. This may be used to call the IBM Personal Editor, the IBM Professional Editor, or any other well-behaved editor that runs under PC DOS.

*EDAPL* performs the following operations:

1. Validates the function name and generates its canonical representation (if it already exists).
2. Creates a temporary file with the name of the function and an extension of ".PEA" and writes the canonical representation to this file.
3. Loads the required editor into a memory partition and executes it to edit the created file.
4. When the editor releases control to APL, the file is read back and the new function is fixed.
5. The temporary file is deleted.

To invoke the editor use:

```
'Editor_Name' EDAPL 'Function_Name'
```

If *Editor\_Name* is omitted, it will default to using the IBM Personal Editor. To use the IBM Professional Editor, specify EDIT as the *Editor\_Name*.

*Note:* Function names must not exceed 8 characters, as DOS restricts the file name to 8 characters. Similarly, the function name must not contain lower case letters or the overbar

character. If necessary, use a temporary name for editing and later replace it using the Del-editor.

## **The EXCHG Workspace**

Several exchange assembly programs have been included in the package. These functions are invoked through the AP2 auxiliary processor, and are capable of receiving information from APL and passing back their results. For an explanation of how they operate, see description of "The Non-APL Program Interface Auxiliary Processor: AP2" on page 12-4.

APL cover functions to execute these programs are provided in the EXCHG workspace:

1. *HEXOBJ X* - Displays the internal Hex representation (including header) of APL object *X*. Its result contains the header block plus the variable in hexadecimal code.

The source for the invoked exchange program is included in the package to provide an example of how to code this type of programs.

2. *HEXCONV X* - Converts integers to hexadecimal and hexadecimal to integers.

This function is designed to give assembler language assistance to the conversion of APL objects to hexadecimal and vice versa. It loads and executes the *HEXCONV.COM* exchange program. If *X* is an integer, it is translated into hexadecimal. If *X* is a literal hex string, it is translated into integer.

Failure to translate is signalled by the return of a value of  $\bar{1}$ . The program will only translate integers in the range 0-255 and hex codes 00-FF. Only scalars, or vector arrays will be accepted by the program. Matrices and higher rank objects will be rejected.



Example:

```

          HEXCONV 1 2 3 4 10
010203040A
          HEXCONV '01020A0B'
1 2 10 11

```

3. *A FINDST B* - Searches a long character vector (*B*) for all instances of a particular string of characters (*A*).

This function loads and executes the *FINDST.COM* exchange program. It returns either: an empty vector, if no matches are found; or the 0-origin positions of the matches in the string. If insufficient space is available to generate the result, a value of  $\bar{1}$  will be returned.

Example:

```

          'JILL' FINDST 'JACK AND JILL'
9
          'A' FINDST 'JACK AND JILL'
1 5

```

4. *FERRET X* - Finds all instances of a character string (*X*) in all the functions of the active workspace. It uses the *FINDST.COM* exchange program.

## The FILE Workspace

The FILE workspace has been designed to help you work with DOS files, and allows either sequential or random access. It uses the file auxiliary processor, AP210. This workspace enables you to create a file, *WRITE* into it, and *READ* from it. To do so, you *WOPEN* an old or new file, and *WRITE* data into it. You then *CLOSE* the file to save it on disk. If you only want to read data from an old file, without writing any more data into it, on the next access simply *OPEN* the file and *READ* in records, either randomly or sequentially.

## IBM Internal Use Only

To use the FILE workspace from APL programs, you must include the file auxiliary processor, AP210, as a parameter to the APL command at load time before you begin an APL work session, or you may load it dynamically with the non-APL program interface auxiliary processor, AP2. For example, the APL invocation line could be:

```
APL AP210
```

When APL is ready, you must copy the FILE workspace into your active workspace by entering:

```
)IN FILE
```

If this command executes successfully, the following set of functions will be loaded into your active workspace.

## Functions

The transfer file, FILE.AIO, contains a number of functions for manipulating DOS files, including:

- *WOPEN*
- *OPEN*
- *SIZE*
- *READ*
- *READD*
- *READV*
- *WRITE*
- *WRITED*
- *WRITEV*
- *CLOSE*
- *DELETE*
- *RENAME*

Other functions in this file that are used for related purposes are:

- *APLPATCH*
- *PATCH*
- *GEN\_TV*
- *IN*

- *PIN*
- *OUT*
- *COMPARE*
- *TYPE*
- *TYPEV*

## Terminology

The following terms are used in the descriptions of the syntax for the functions:

Brackets are used to indicate that a parameter is optional.

“*code*” can be any of the following characters:

- A (APL) The records in the file are APL objects and their headers in APL internal form. Matrices, vectors, and arrays of any rank may be stored and recovered. Different records of a file may contain objects of different types (for example, characters, integers, or real numbers). An APL object in a record may occupy up to the actual record length (not necessarily the same number of bytes), but the header fills a part of that area. (See Chapter 13, “How to Build an Auxiliary Processor” for the structure and memory requirements of an APL header).
- B (Bool) The records in the file contain strings of bits without any header (packed eight bits per byte). The equivalent APL object will be a boolean vector. In this case, all records must be equal to the selected record length.
- C (Chars) The contents of the record is a string of characters in APL internal code, without any header. All records must be equal to the selected record length, with each character occupying one byte.
- D (ASCII) The contents of the record is a string of characters in ASCII code, without any header. Each character occupies one byte.

“*file\_no*” is a positive integer that you define for future reference to a file when you open it.

“*filespec*” must be in the following DOS syntax (see DOS manual):

```
[d:][\subdirectory\]filename[.ext]
```

or in the APL library syntax:

```
[library_number] filename[.ext]
```

**Warning:** Changing diskettes during an input/output operation, or when you have open files, may damage your diskette.

Errors encountered during the execution of these functions may cause a message containing an AP210 return code to be displayed. The meanings of these return codes are listed in “AP210 Return Codes” on page 12-59.

## **WOPEN**

This function opens a DOS data file for reading or writing, with sequential or random access. A maximum of ten files may be open (through *WOPEN* or *OPEN*) at any one time.

The syntax of the function is:

```
[file_no] WOPEN 'filespec[,code]'
```

If no file by that name exists in the indicated drive or directory, a new file is created. If “*file\_no*” is omitted, 1 is assumed. If “*code*” is omitted, A is assumed.

## **OPEN**

This function opens a DOS data file for read-only sequential or random access. A maximum of ten files may be open (through *WOPEN* or *OPEN*) at any one time.

```
[file_no] OPEN 'filespec[,code]'
```

If no file by that name exists in the indicated drive or directory, an error will result; see “AP210 Return Codes” on page 12-59 for a list of all possible return codes. If “*file\_no*” is omitted, 1 is assumed. If “*code*” is omitted, A is assumed.

## SIZE

This function returns the size of a file when it was last opened. The syntax is:

*SIZE file\_no*

*SIZE* can only be used after the file has been (*W*)*OPEN*ed successfully.

## READ

This function reads a DOS data file, sequentially or randomly, that was opened using (*W*)*OPEN*. The syntax is:

*READ file\_no [record\_no [record\_size]]*

where

$0 \leq \textit{record\_no} \leq 32767$

$0 < \textit{record\_size} \leq 32512$

“*file\_no*” matches the number that you specified in (*W*)*OPEN*ing the file.

If no “*record\_no*” is specified, the default is *sequential access* to the file. Under sequential access, the first record (record 0) will be accessed by either a *READ* or *WRITE* command immediately after the (*W*)*OPEN*; the second record (record 1) will be accessed on the next command, and so on. The *READ*, *READD*, *READV*, *WRITE*, *WRITED* and *WRITEV* functions work from the same access point, meaning that the access point is advanced sequentially to the next record each time any of these commands are issued.

## IBM Internal Use Only

Random access is designated by specifying a particular record. The *record\_size* can only be specified when using random access. If the *record\_size* is not specified, the default is the *record\_size* specified in the previous operation. If *record\_size* is not specified on the first *READ* or *WRITE*, the default is 128 bytes.

## READD

This function reads a DOS data file, sequentially or randomly, that was opened using *(W)OPEN*. The syntax is:

```
READD file_no [byte_no [record_size]]
```

where:

$0 \leq \text{byte\_no}$

$0 < \text{record\_size} \leq 32512$

“*file\_no*” matches the number that you specified in *(W)OPEN*ing the file.

“*file\_no*”, “*byte\_no*” and “*record\_size*” must all be integer.

If *byte\_no* is not specified, the default is *sequential access* to the file. Random access is designated by specifying a particular “*byte\_no*” position in the file. “*record\_size*” can only be specified when using random-access.

## READV

This function sequentially reads a variable-length record APL or DOS character file that was previously opened using *(W)OPEN*. The syntax is:

```
READV file_no
```

The *file\_no* matches the number that you defined in *(W)OPEN*ing the file.

This function may only be used if file was opened with codes A or D.

## WRITE

This function writes to a DOS data file, either sequentially or randomly, that was previously opened using *WOPEN*. (Trying to *WRITE* to a file opened by *OPEN* will result in an error; see "AP210 Return Codes" on page 12-59 for a list of all possible return codes.) When the *WRITE* function is issued, it will write over any existing data in the currently accessed record.

The syntax for this function is:

```
[file_no [rec_no [rec_size]]] WRITE DATA
```

where:

$$0 \leq \text{rec\_no} \leq 32767$$

$$0 < \text{rec\_size} \leq 32512$$

"*file\_no*" matches the number arbitrarily defined when *WOPEN*ing the file. If not given, 1 is assumed.

If the *rec\_no* is not specified on the first *READ* or *WRITE* the default is *sequential access* to the file. Under sequential access, the first record (record 0) will be accessed by either a *READ* or *WRITE* command immediately after the (*W*)*OPEN*; the second record (record 1) will be accessed on the next command, and so on. The *READ*, *READD*, *READV*, *WRITE*, *WRITED* and *WRITEV* functions work from the same access point, meaning that the access point is advanced sequentially to the next record each time any of these commands are issued.

Random access is designated by specifying a particular record. If the record size, *rec\_size*, is not specified, the default is the *rec\_size* specified on the previous *READ* or *WRITE*

If the *rec\_size* has not been specified, the default is 128 bytes.

## WRITED

This function writes to a DOS data file, either sequentially or randomly, that was previously opened using *WOPEN*. (Trying to *WRITED* to a file opened by *OPEN* will result in an error; see “AP210 Return Codes” on page 12-59 for a list of all possible errors.) When the *WRITED* function is issued, it will write over any existing data in the currently accessed record.

The syntax for this function is:

```
[file_no [byte_no [rec_size]]] WRITED DATA
```

where:

$0 \leq \text{byte\_no}$

$0 < \text{rec\_size} \leq 32512$

“*file\_no*” matches the number that you arbitrarily defined in *WOPEN*ing the file. If not given, 1 is assumed.

“*file\_no*”, “*byte\_no*” and “*rec\_size*” must all be integer.

If “*byte\_no*” is not specified, the default is *sequential access* to the file. Random access is designated by specifying a particular “*byte\_no*”. If the record size, *rec\_size*, has not been specified, the default is 128 bytes.

## WRITEV

This function sequentially writes a variable-length record APL or DOS character file that was previously opened using *WOPEN*. The syntax is:

```
[file_no] WRITEV DATA
```

The *file\_no* matches the number that you defined in *WOPEN*ing the file. If not given, 1 is assumed.



This function may only be used if file was opened with codes A or D.

## CLOSE

This function closes a file that was previously opened using *(W)OPEN*. The previously assigned *file\_no* is now available for reuse. (*(W)OPEN*ing a *file\_no* without having closed the corresponding file will cause the current file to be closed, and then re-opened according to the new request).

The syntax for *CLOSE* is:

```
CLOSE file_no
```

## DELETE

This function deletes DOS data files. (Files may also be erased in DOS using ERASE, or in APL using *)DROP*.) The syntax for *DELETE* is:

```
DELETE 'filespec'
```

## RENAME

This function changes the name of the file specified in the right argument to the name and extension specified in the left argument. The left argument drive/directory (or APL library number) must be specified. If a different subdirectory in the same drive is specified, a move is performed instead of a rename. Renaming to a different drive is not allowed. The syntax is:

```
'new_filespec' RENAME 'old_filespec'
```

*Note:* If renaming an APL workspace (".APL") or a transfer file (".AIO"), the file name must be padded to eight characters with underbars.

## **APLPATCH**

This function allows you to make hexadecimal patches in DOS files (including .EXE files). It reads a DOS file listing the files to be changed, and the changes to be made. It is used as:

***APLPATCH 'filespec'***

Where *filespec* is the name of a patch file to be processed. Records in this file are of the form:

- Comment - any record beginning with an asterisk.
- File name - a record containing the name of the file to be patched as: "FILE filespec". This must be immediately followed by one or more groups of three records of the form:
  - Address - hexadecimal address of start of data to be patched.
  - Old data - hexadecimal representation of data expected to be in the file to be patched. The patch will not be made unless the contents of the file exactly matches the data specified.
  - New data - hexadecimal representation of the data to be patched into the file.

The address and data specifications must consist of pairs of characters (0 to 9, A to F) with no intervening spaces.

## **PATCH**

This function allows you to make hexadecimal patches in DOS files (including .EXE files). It works interactively. The patches are made one byte at a time. First the hexadecimal address of the byte (relative to the beginning of the file) is requested, then the present contents are displayed, and finally, a prompt is made for the new value. (It must be given as two hexadecimal digits). After the patch has been made, a new one

can be entered. Entering an empty line (pressing the Enter key with no data) exits the function.

The syntax for PATCH is:

```
PATCH 'filespec'
```

Example:

```
      PATCH 'FILE.EXE'
GIVE ADDRESS: 129A
IS 00
GIVE NEW VALUE OR EMPTY LINE TO CANCEL PATCH
: 07
GIVE ADDRESS: (press Enter key to leave PATCH)
```

## | GEN\_TV

| This function allows a copy of the APL program to be  
 | modified for correct operation under TopView. The function  
 | must be run on the same machine as will be used for  
 | APL/TopView use, because machine-dependent timing  
 | information will be stored in the generated program. However,  
 | to run the *GEN\_TV* function, APL must be started without  
 | TopView being active.

| To create a TopView compatible version of APL:

- | 1. COPY APL.EXE APLTV.EXE
- | 2. APL AP210
- | 3. )IN FILE
- | 4. *GEN\_TV 'APLTV.EXE'*
- | 5. )OFF

| Then the APLTV.EXE program can be used in place of the  
 | supplied APL.EXE. For example, use "APLTV AP210" to  
 | start the TopView compatible version of APL with AP210.  
 | The level identification of the modified version displayed when  
 | APL is started will be "2.10TV".

## IN

This function imitates the *IN* command (see Chapter 10, “System Commands”) under control of AP210. It can be called from another APL function, thus effectively providing a powerful IN facility. You can call this function in two different ways:

- If you want to copy a whole file into your active workspace, you must call the *IN* function in the following way:

```
IN '[d:]filename'
```

where *filename* is the name of the file you want to copy. You must not give an extension, but the name should be completed to eight characters with underbars if a file created by the *OUT* command is to be read. APL assumes an extension of *.AIO* and appends it to the file name. The result is a 1 if the file exists; otherwise the result is 0.

Example:

```
IN 'MYFILE__'
```

This line will copy the whole file, MYFILE\_\_.AIO, into your active workspace.

- If you want to copy only part of a file (some functions and/or variables) into your active workspace, you must call the *IN* function in the following way:

```
namelist_matrix IN '[d:]filename'
```

In *namelist\_matrix*, you have to give the names of the functions and variables (APL objects) you want to copy. If there is more than one object, each name must be given as a row of a character matrix. For *filename*, see above. Only the indicated objects are copied into the active workspace. The function returns a logical vector result - a 1 per object copied and a 0 per object not copied.

Example:

```
(2 3p'FUNVAR') IN 'MYFILE__'
```

The left argument of the *IN* function in the preceding example is a 2-by-3 character matrix, the first row of which is *FUN* and the second is *VAR*. This line copies into your active workspace the objects (functions and/or variables), *FUN* and *VAR*, from MYFILE\_\_.AIO.

## PIN

This function is a *protected IN*. It works like *IN*, except that an object is copied only if the outstanding object in the active workspace has no current value. You can call this function in two different ways:

- If you want to copy a whole file into your active workspace, you must call the *PIN* function in the following way:

```
PIN '[d:]filename'
```

where *filename* is the name of the file you want to copy. You must not give an extension, but the name should be completed to eight characters with underbars if a file created by the *OUT* command is to be read. APL assumes an extension of .AIO and appends it to the file name. The result is a 1 if the file exists; otherwise the result is 0.

Example:

```
PIN 'MYFILE__'
```

This line will copy the whole file, MYFILE\_\_.AIO, into your active workspace.

- If you want to copy only part of a file (some functions and/or variables) into your active workspace, you must call the *PIN* function in the following way:

```
namelist_matrix PIN '[d:]filename'
```

In *namelist\_matrix*, you have to give the names of the functions and variables (APL objects) you want to copy. If there is more than one object, each name must be given as a row of a character matrix. For *filename*, see above. Only the indicated objects are copied into the active workspace. The function returns a logical vector result - a 1 per object copied and a 0 per object not copied.

Example:

```

)ERASE FUN
VAR←7
(2 3p'FUNVAR') PIN 'MYFILE__'

```

The left argument of the *IN* function in the preceding example is a 2-by-3 character matrix, the first row of which is *FUN* and the second is *VAR*. This line copies into your active workspace only the object *FUN* because *VAR* had a value before *PIN* was executed (in *VAR←7* we set *VAR* to the value of 7), and therefore the result of *PIN* will be  
1 0.

## OUT

This function emulates the *OUT* command (see Chapter 10, “System Commands”) under control of AP210, and can be called from another APL function, thus effectively providing a powerful *OUT* facility. You can call this function in two different ways:

- If you want to copy your entire active workspace (all functions and all variables) into an .AIO file (that is, a transfer file), you must call the *OUT* function in the following way:

```
OUT '[d:]filename'
```

where *filename* is the name of the transfer file. You must not give an extension. APL assumes an extension of .AIO and appends it to the file name. If you want to generate a file that is compatible with the *IN* command, you should complete the name to 8 characters with the

appropriate numbers of underbars. The result is a 1 if the operation is successful; otherwise, the result is 0.

Example:

```
OUT 'MYFILE__'
```

This line will copy all functions and variables of your active workspace into the file, MYFILE\_\_.AIO.

- If you want to copy only part of your workspace (some functions and/or variables) into a file, you must call the *OUT* function in the following way:

```
namelist_matrix OUT '[d:]filename'
```

In *namelist\_matrix*, you have to give the names of the functions and variables (APL objects) you want to copy. If there is more than one object, each name must be given as a row of a character matrix. For *filename*, see above. Only the indicated objects will be included in the file. The function returns a logical vector result - a 1 per object copied and a 0 per object not copied.

Example:

```
(2 3p 'FUNVAR') OUT 'MYFILE__'
```

The left argument of the *OUT* function in the preceding example is a 2-by-3 character matrix, the first row of which is *FUN* and the second is *VAR*. This line creates a transfer file called MYFILE\_\_.AIO and writes into it, the objects *FUN* and *VAR* in the transfer form.

## COMPARE

This function compares two files. The syntax is:

```
record_size COMPARE filespec_matrix
```

The right argument is a two-row character matrix, each row containing the filespec of one of the files to be compared, followed by a comma, followed by the code in which the file is

to be read. The left argument indicates the length of the record with which the files are to be read.

The *COMPARE* function gives no result if both files are identical. Otherwise, it lists the pairs of corresponding records that are different. The function also indicates which of the files is shorter, if applicable.

Example:

```
80 COMPARE 2 11p 'FILE1.EXT,DFILE2.EXT,D'
```

This example compares files, FILE1.EXT and FILE2.EXT, both of which are read with a record length of 80 in ASCII code.

## **TYPE**

This function emulates the DOS TYPE command. The syntax is:

```
[record_size [n]] TYPE 'filespec[,code]
```

The file with the indicated *filespec* is displayed at the terminal.

“*record\_size*”, if given, specifies the record length of a fixed record length file, the *n* first characters of which are to be typed. If *n* is not given, the full *record\_size* is typed. If *record\_size* is not given, the file is assumed to contain variable length records.

## **Examples of Use**

Following are examples of using the various DOS file-handling functions.

```
WOPEN 'FILE.EXT'
```

Creates a new file. Records will contain APL objects with header (default code). File number will be defaulted to 1.



*WRITE* 1 10

First record will be a vector of elements from 1 to 10 (origin 1). Default *record\_no* is 0; default *record\_size* is 128 bytes.

1 *WRITE* 2 3p16

A matrix of two rows and three columns, of elements from 1 to 6, is written sequentially to the file.

*CLOSE* 1

The file is closed.

*OPEN* 'FILE.EXT'

Open the same file for read-only operation, with the same file number.

*READ* 1 1

Read the second record first: the following matrix is displayed:

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

*READ* 1 0

Now ask for the first record: the result is the vector of integers:

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

Close the file.

*DELETE* 'FILE.EXT'

Delete the file.

## The FOIL Workspace

This workspace contains a set of functions that make it easy to create text foils or pie charts. The created objects are stored in the workspace and may be redrawn later at will.

- **DEFFOIL** - Interactively defines a text foil, made up of the following parts:
  1. A name for the APL variable where the foil will be kept.
  2. Text for each line in the foil. The following parameters may be defined: colour, size, and margin width (if the latter is given a value of  $\bar{1}$ , this line will be centred).

An empty line ends the foil definition. The **FOIL** function is then automatically invoked to display the resulting text foil.

- **FOIL x** - Where **x** is an APL variable containing a text foil definition, displays it on the screen. Lines are automatically spaced to optimise the appearance of the display.
- **DEFPIC** - Interactively defines a pie chart. The result is kept in an APL variable, the name of which is requested.
- **PIC x** - Where **x** is an APL variable containing a pie chart definition, displays it on the screen.

The FOIL workspace requires the AP206 graphic auxiliary processor to be active.

## **The FORTRAN Workspace**

| This workspace contains a set of functions designed to assist in  
| the generation and calling of FORTRAN subroutines from  
| APL as if they were APL functions. The FORTRAN  
| language implementations supported are:

- | • IBM PC Professional FORTRAN
- | • IBM PC FORTRAN 2.0

| In the following text, the word FORTRAN is used to imply  
| both versions unless a particular implementation is explicitly  
| specified.

| The workspace uses the services of the PFORTPAR program  
| which is used to pass FORTRAN parameters.

| Calls to FORTRAN subroutines compiled under FORTRAN  
| are supported. Programs in other languages (like IBM Macro  
| Assembler, Pascal or C) written to be called as FORTRAN  
| subroutines are also supported.

## **Restrictions on FORTRAN Programs**

| Communication with the FORTRAN subroutine must be  
| through explicit arguments (no COMMON, limited I/O).

| IBM PC Professional FORTRAN may only be compiled or  
| run on a machine with a Math Co-processor option installed.  
| However, it should be noted that although programs must be  
| compiled on a machine with the Math Co-processor option  
| installed, they may be run on any machine under APL, as the  
| services of APL's 8087 emulator module will automatically be  
| invoked, if required. Programs making use of the 8087  
| emulator should be very carefully tested as not all of the 8087  
| instructions are included in the emulator code.

| AP2, like other auxiliary processors, may only pass up to a  
| 32512 byte object (regardless of the number of elements). In

order to pass more than 32512 bytes to a FORTRAN subroutine in one variable, such objects should be passed as a series of 32512 byte objects and concatenated in the FORTRAN subroutine.

## **Generation Process**

Before FORTRAN subroutines may be called from APL, a generation process must be performed. The following procedure should be used to generate a module containing one or more FORTRAN compatible subroutines:

1. AP2 and AP210 must be included in your APL session.
2. The FORTRAN subroutines are assumed to have been compiled outside APL. The “.OBJ” modules should be available.
3. Generate the assembler interface (IBM PC Professional FORTRAN):

```
'name' GEN_ASM fmv 'SUB1;SUB2;...'
```

where ‘name’ is the name to be given to the module, and SUB1, SUB2, . . . are the names of the FORTRAN compatible subroutines.

The source program “name.ASM” will be generated.

or,

4. Generate the assembler interface (IBM PC FORTRAN 2.0):

```
'name' GEN_ASM_MS fmv 'SUB1;SUB2;...'
```

where ‘name’ is the name to be given to the module, and SUB1, SUB2, . . . are the names of the FORTRAN compatible subroutines.

The source program “name.ASM” will be generated.

## | 5. Assemble the interface:

```
|   ASM 'name;'
```

| The assembler will be invoked. The object program  
| 'name.OBJ' will be generated.

| Only the small assembler may be used via AP2, this is  
| designated as ASM.EXE on the diskette.

## | 6. Link the interface with the subroutines and generate the executable module.

| Depending on the nature of the FORTRAN code, it may  
| be necessary to include a library in the LINK command.  
| Example:

```
|   LINK 'name+SUB1+SUB2+...,,CON;'
```

## | 7. Generate the APL driver functions. For each subroutine, a function should be written (using any of the APL editors) with as many definition lines as the arguments to be passed to the subroutine, and in the same order. We recommend that the name of this function is the same as the name of the corresponding subroutine. Each definition line will start by two cap null characters (␣), not separated by blanks, and will define the argument type and initial value in the following way:

```
|   ␣␣[RESULT_NAME<] TYPE ARG_NAME<INIT_VALUE
```

| where *RESULT\_NAME*, if given, is the name of the APL  
| variable where the value of the argument after execution  
| should be passed back to APL.

| *TYPE* is one of the following:

```
|   INTEGER*2  
|   INTEGER*4  
|   REAL*4 or REAL  
|   REAL*8  
|   LOGICAL*1  
|   LOGICAL*4 or LOGICAL
```

Additionally, for IBM PC Professional FORTRAN, the following *TYPE*s may be used:

CHARACTER  
APL

Similarly, for IBM PC FORTRAN 2.0, to signal that the special Microsoft conventions for character data are to be used:

CHARASCII  
CHARDATA

CHARACTER and CHARASCII types specify that the data should be translated between the internal APL character representation and ASCII. APL and CHARDATA types specify that the data is not to be translated.

INTEGER is not permitted, since the Professional FORTRAN compiler allows it to be user defined to be either INTEGER\*2 or INTEGER\*4.

An additional *TYPE* is also allowed:

TEMPREAL

Which allows numeric data to be translated to 10 byte temporary real for use with IBM PC Macro Assembler or other special applications.

*ARG\_NAME* is a dummy name (usually the same as the name given to the corresponding argument in the FORTRAN subroutine). The name may be followed by a DIMENSION term (e.g. A(1)).

*INIT\_VALUE* is any APL expression generating the initial value(s) to be passed to the FORTRAN argument.

Example:

```

|  RRREAL*8 A(1)←A
|  RAR←REAL*8 SUM←0
|  RAINTEGER*2 N←ρA

```

The example defines the interface to a subroutine with three arguments: the first one is a double-precision vector, the length of which is given by the integer third argument. The result is computed on the double-precision second argument and passed back to APL through variable *R*.

The block of definition lines may be preceded by, or followed by, as many APL-lines as are necessary to build a complete function.

8. For each FORTRAN subroutine, the following function should be executed:

```

|      GEN_FORT 'driver_function_name'

```

After this function has been executed, the driver function will be ready to be used.

The compiler adds a lot of AP2 shared variable references to the program, which, once compiled should not be edited. To change anything in the driver function, you should uncompile it first, by:

```

|      UNGEN_FORT 'driver_function_name'

```

## Usage Protocol

Before the APL driver function can be used, the following two loading functions must have been executed once:

1. *PFORT 'drive:'*

to load the parameter management program in partition 2.

2. *1 40 LOAD 'name'*

to load the executable module in partition 1. The "40" represents the size assigned to partition 1. Its actual value may depend on the size of the module.

You should ensure that the usage of these partitions does not cause any conflicts with other parts of your application.

From this point, the APL driver functions can be used to call the corresponding subroutines, as if they were normal APL functions. The functions assume the programs to be loaded in partitions 1 and 2, as specified.

## **PFORTPAR Parameter Management Program**

*Note:* This section may be ignored by users who are not interested in building their own parameter passing interfaces for use with the FORTRAN workspace. There are several functions included in the workspace, which assist in the direct usage of PFORTPAR, which are documented in the workspace.

PFORTPAR is an exchange assembly program callable from AP2, that provides different type conversions and storage allocation for APL objects.

This program operates on the 2 exchange assembly variables *D* and *E*. It can be used to build the parameter list for a FORTRAN subroutine.

*E* is a data variable for the operation. Original APL data will be passed through it, one parameter at a time. On return, parameters will be passed back to APL through this variable.

*D* is a 2 element control variable for the PFORTPAR exchange assembly program. There are several operations depending on the first element of *D*.

- 0,n - Reset the environment for n parameters.
- 0,0 - Total reset. All storage is freed. This call must be done before the last use of PFORTPAR, otherwise some main storage may be lost.



- A number between 1 and 20: Set parameter  $n$  to the value of  $E$ .
- A number between  $\bar{1}$  and  $\bar{20}$ : Get parameter  $n$  through  $E$ .

The second element in the last two cases determines the data type:

- 1 - Integer\*2
- 2 - Real\*8
- 3 - APL Character (for Professional FORTRAN)
- 4 - Integer\*4
- 5 - Real\*4
- 6 - Logical\*1
- 7 - Logical\*4
- 8 - ASCII Character (for Professional FORTRAN)
- 9 - Temporary Real (only valid for Assembler)
- 10 - APL Character (for FORTRAN 2.0)
- 11 - ASCII Character (for FORTRAN 2.0)

Return codes will be passed through variable  $D$ :

| Code       | Meaning   |
|------------|---|
| 99         | All storage freed, routine reset.   |
| 0          | Success: Element 2 is the segment offset of the parm-table. Element 3 is the maximum number of parameters. Element 4 is the number of parameters already specified. |
| $\bar{1}$  | Value Error. The requested parameter is not there.  |
| $\bar{2}$  | Type Error, $D$ must be boolean or integer.   |
| $\bar{3}$  | Rank Error, $D$ must be vector.   |
| $\bar{4}$  | Length Error, $D$ must have 2 elements.   |
| $\bar{5}$  | Already in use. RESET should be called first.   |
| $\bar{6}$  | Invalid second element. More parameters than allowed.   |
| $\bar{7}$  | Invalid sequence. RESET must be called first.   |
| $\bar{8}$  | Parameter is out of previously defined range.   |
| $\bar{9}$  | Parameter already defined, cannot be redefined.   |
| $\bar{10}$ | Requested parameter has not yet been defined.   |
| $\bar{11}$ | Some space could not be released to APL.  |
| $\bar{12}$ | Specified type is not valid.  |

| -27 - No space available. Parameters too large to fit in  
| memory.

## | **Sample FORTRAN Subroutines (IBM PC | Professional FORTRAN)**

- | • **FADD.FOR**

| This FORTRAN subroutine, adds a vector of numbers,  
| like APL +/.

- | • **F1.FOR**

| Generates two matrices, one real, the other integer\*4. Also  
| assigns a value to a character variable depending on a  
| logical argument.

- | • **F2.FOR**

| Performs the product of two complex vectors. In APL,  
| each element in the complex vector is represented by two  
| consecutive elements.

### | Sample Case:

| Assume ASM and LINK are located in drive A:, and the  
| subroutine's ".OBJ" modules in drive B:. The following will  
| generate the APL-FORTRAN interfaces.

```
|      'B:SAMPLE' GEN_ASM fmv 'FADD;F1;F2'  
|      ASM 'B:SAMPLE,B:;' '  
|      LINK 'B:SAMPLE+B:FADD+B:F1+B:F2,B:,CON;'
```

| The starting APL driver functions (*FADD*, *F1*, *F2*) are already  
| included in the FORTRAN workspace.

| The APL driver functions will be generated thus:

```
|      GEN_FORT 'FADD'  
|      GEN_FORT 'F1'  
|      GEN_FORT 'F2'
```

| Load the executable modules:

```
PFORT 'B:'
1 40 LOAD 'B:SAMPLE'
```

Execute the APL driver functions:

```
FADD 11000
500500
F1 1 1
3 5 4 6
I
3 5 4 6
C
0123456789
F1 0
C
ABCDEFGHIJ
0 1 F2 0 1
-1 0
```

*Note:* For IBM PC FORTRAN 2.0, only *FADD* is correctly defined, because the other two are dependent on special IBM PC Professional FORTRAN data types. *GEN\_ASM\_MS* must be used in place of *GEN\_ASM*.

## The GEDIT Workspace

This workspace implements a graphic input application that makes it possible to generate complex “images” graphically, store them in the workspace, and reproduce them later on the screen. An “image” is stored in the workspace as a set of variables, all of which start by the same prefix (the “image” name) followed by the APL character  $\Delta$  and a three digit number. Each variable contains graphic information in the form of:

1. Parameter settings for AP206.
2. Text strings to be drawn.
3. Graphic matrices to be drawn.

The following functions are included in the workspace:

- `[mode] GEDIT 'image_name'` - The graphic editor. The optional left argument is the screen mode to be used (the default is 4). The right argument is the name of the "image" to be created or modified. If the image exists, it is drawn and the terminal opens for graphic input. If the image is new, the terminal opens on an empty screen.

When the terminal is open for graphic input, you may use the numeric keypad to move the cursor, fix points, add straight lines, delete lines, and so forth. The Tab key and the Backspace key are also useful to move the cursor along the image. The F-keys can be used to change the cursor movement mode (i.e. the length of cursor step or the speed of cursor advance). For additional details on the use of these keys, see the description of "interactive input mode" in "The Graphic Auxiliary Processor: AP206" on page 12-39.

Besides these keys, directly supported by AP206, *GEDIT* uses other keys to perform special operations, such as changing background colour, drawing or filling circles or polygons, drawing text, etc. A full description of these additional operations may be displayed by executing function *HELPGEDIT* in the workspace, or pressing the H key when in graphic input mode (when the *GEDIT* function is active).

- *DEMO* - Shows on the screen an image generated through *GEDIT*. Graphic input mode is entered, allowing changes or additions. To leave this state (automatically entered when loading the workspace through the *)LOAD* command) press the E key.
- *SHARE* - Shares variable *G* with the graphic auxiliary processor (AP206). It is not needed if *GEDIT* is executed. However, it must be invoked before *IMAGE* or *PRINT* is used, if *GEDIT* has not been called since the workspace was loaded.

In the next three functions, *nlist* is a numeric vector of image component numbers.

- [*nlist*] *IMAGE* '*image\_name*' - Shows on the screen, the "image" defined by the prefix *image\_name*. If *nlist* is given, only the indicated components of the image are shown.
- [*nlist*] *DISPLAY* '*image\_name*' - Lists the different variables making up the image defined by the prefix *image\_name*. If *nlist* is given, only the indicated components are listed.
- [*nlist*] *ERASE* '*image\_name*' - Erases the full image defined by the prefix *image\_name*. If *nlist* is given, only the indicated components are erased. The remaining components are renumbered.
- *PRINT* - Equivalent to *IMAGE*, but the image is also printed.

## The GRAPHPAK Workspaces

There are seven workspaces on the *APL Workspaces - 2* diskette together containing a set of functions compatible with the GRAPHPAK workspace, as implemented under IBM APL2 Program Product (5668-899). These are included for use by APL2 and VS APL programmers wishing to develop applications which will run on both mainframe and personal computers, or to use the PC for prototyping and program development purposes. The Math Co-processor option is strongly recommended to users of these workspaces.

GRAPHPAK has been divided to correspond to the seven groups of the same name in the mainframe product. These are:

- GPBASE contains the fundamental drawing and writing functions;
- GPCHT contains functions for drawing charts;

## IBM Internal Use Only

GPCONT contains functions for drawing contour maps;

GPDEMO contains functions illustrating aspects of GRAPHPAK. This workspace is supplied in ".APL" form and should be *)LOADED* to run the demonstration.

GPFIT contains functions for curve fitting;

GPGEOM contains descriptive geometry functions;

GPLOT contains plotting functions.

A list of the functions in these workspaces is provided in Appendix F, "The GRAPHPAK Workspaces - Functions".

*Note:* Developers of programs intended to run only on IBM PC family machines should use the functions provided in the cover workspace AP206. These will generally prove faster and more convenient for such applications.

To run the demonstration contained in the GPDEMO, the complete GRAPHPAK package must be assembled:

```
)LOAD GPDEMO
)IN GPBASE
)IN GPGEOM
)IN GPLOT
)IN GPCHT (*)
)IN GPCONT (*)
DEMO
```

The last two parts, each marked with a \*, are optional, and need only be included if the full demonstration is to be run.

## The MUSIC Workspace

The MUSIC workspace provides a sample of the use of the AP440 auxiliary processor, which makes it possible to create music in your IBM Personal Computer using the attached speaker.

To use the speaker from APL programs, you must include the music auxiliary processor, AP440, as a parameter to the APL command at load time before you begin an APL work session, or load it dynamically through AP2. For example,

```
APL AP440
```

To copy the MUSIC workspace into your active workspace, you must enter:

```
)IN MUSIC
```

The following melodies are included in the MUSIC workspace. Each melody is a part of a well-known musical piece.

|               |              |              |             |
|---------------|--------------|--------------|-------------|
| <i>SAKURA</i> | <i>POP</i>   | <i>STARS</i> | <i>BLUE</i> |
| <i>BUG</i>    | <i>HUMOR</i> | <i>FORTY</i> | <i>HAT</i>  |
| <i>SCALES</i> | <i>DANDY</i> | <i>MARCH</i> |             |

To perform them, you have to execute the following:

```
PLAY name
```

where name is the title of the melody.

## The PLOT Workspace

This workspace contains a number of functions that make it easy to build histograms and graphic plots of mathematical functions. A set of control variables allow the user to define the plot characteristics, such as where the axis must be drawn, what axis marks should be included, whether linear or logarithmic scales are selected, and so on and so forth. The meaning of these control variables and the values they may be assigned can be obtained from the workspace by typing *HELPPLOT*.

If any of the control variables does not exist, default values will be used instead.

The PLOT workspace requires the AP206 graphic auxiliary processor to be active.

The following user functions are described here:

- [*title*] *PLOT plot\_data* - Plots a graph. "*title*", if given, is the title to be typed at the top of the plot. The plot data is a three-dimensional object, that may be generated by means of the auxiliary functions *VS* and *AND*.

For example, if we want to obtain the plots of two dependent variables (Y1, Y2), as functions of the independent variable X (where X, Y1 and Y2 are assumed to be APL vectors containing the corresponding data, all of them with the same number of elements), the following line will do it:

```
PLOT (Y1 AND Y2) VS X
```

Other *AND/VS* combinations may be obtained.

Function *DEMO PLOT* in the workspace gives an example of the use of the *PLOT* function.



Function *DEFPLOT* allows you to define a plot conversationally.

- *RPLOT* - Same as *PLOT*, but the X and Y axes are reversed.
- [*title*] *HPLOT hist\_data* - Plots a histogram. "*title*", if given, is the title to be typed at the top of the histogram. The histogram data is a bi-dimensional object, that may be generated by means of the auxiliary function *AND*. The independent variable is not given, and assumed to be a set of subsequent integers.

For example, if we want to obtain the plots of two dependent variables (Y1, Y2), (where Y1 and Y2 are assumed to be APL vectors containing the corresponding data, both with the same number of elements), the following line will do it:

```
HPLOT Y1 AND Y2
```

Function *DEMOHPLOT* in the workspace gives an example of the use of the *HPLOT* function.

Function *DEFHPLOT* allows you to define a histogram conversationally.

- *RHPLOT* - Same as *HPLOT*, but the X and Y axes are reversed.
- *CLEARPLOT* - Erases all the plot control variables, so that the default values will be used in successive plots, unless the control variables are redefined.
- *PARMSPLOT* - Allows you to define conversationally the plot control variables.
- *BEGIMAGE 'image\_name'* - Indicates that all subsequent plots will be kept in the workspace as an "image" with the name *image\_name*. See description of "images" in "The GEDIT Workspace" on page 11-82.

- *ENDIMAGE* - Indicates that all subsequent plots should not be kept in the previously defined “image” (given by the last execution of *BEGINIMAGE*).
- Functions *DISPLAY*, *ERASE* and *IMAGE* are described in “The GEDIT Workspace” on page 11-82.

## **The PRINT Workspace**

To use the printer from APL programs, you must include the printer auxiliary processor, AP80, as a parameter to the APL command at load time before you begin an APL work session, or load it later with the non-APL program interface auxiliary processor (AP2). For example,

```
APL AP80
```

This workspace contains the following functions:

- *PRINT* can be used to selectively print any APL object or result, of any rank or type (that is, literal or numeric), from your APL program. It may be called from any other APL user-defined function, thus giving the program control of the printer.

To copy the *PRINT* function to the active workspace, you should execute the following:

```
)IN PRINT PRINT
```

Function *PRINT* may be used in the following way:

```
[Page_length] PRINT object
```

If *Page\_length* is omitted, 60 lines per page are assumed. The current page position is maintained by a global variable named “*lc*” (line count).

The following examples show what is printed for various entries:

| Entry                     | Printed              |
|---------------------------|----------------------|
| <i>PRINT</i> 2+2          | 4                    |
| <i>PRINT</i> 'ABCabc'     | ABCabc               |
| <i>PRINT</i> 110          | 1 2 3 4 5 6 7 8 9 10 |
| <i>PRINT</i> 2 3p'ABCDEF' | ABC<br>DEF           |

(A variable can also be printed)

```
X←'IS A VARIABLE'
PRINT 'X ',X      X IS A VARIABLE
```

If the *PRINT* function is used to print a character string beginning with  $\square AV[\square IO+255]$ , the remaining characters in the string are sent to the printer in alphameric mode. In this way, printer control codes can be included and executed. These control codes are used to obtain emphasised printing, large character sizes, and other special printing functions.

If the first character in the string is not  $\square AV[\square IO+255]$ , the whole string is printed as given. Therefore, a single character can have a dual function, depending on the selected printing mode.

AP80 will automatically use a compressed style of character if  $\square PW$  has a value greater than 80. These smaller characters allow lines of up to 132 characters to be printed.

*Note:* Turning the printer off and then on will reset all options. In some circumstances this can be more convenient than sending all the required codes to reset everything.

- The *PRINT\_DOC* function, uses *PRINT* and the file auxiliary processor (AP210) to obtain a listing at the printer of a DOS ASCII file.

- *PRINT\_GEN* function generates a set of variables that may be used to provide the printer control codes to produce complex documents.
- The *DEMO* function shows how to combine the printer control codes and the *PRINT* function.

## The PROFILE Workspace

This workspace uses the AP101 profile and stack auxiliary processor to initialise the APL session. It includes a function (*PROFILE*), that establishes an F-key profile, defines some APL libraries to be equivalent to certain sub-directories, executes some floating-point APL functions to prime the Math Co-processor emulator, and uses the input stack to *)CLEAR* the active workspace, in this way taking itself out of the memory. This function provides an example of how automatic loading and even disconnection of APL applications may take place.

Function *PROFILE* is invoked by the workspace latent expression, which is automatically executed whenever the workspace is loaded:

```
□LX  
PROFILE
```

Since latent expressions are only executed when a *)LOAD* command is performed, which requires the workspace to be in .APL format, this workspace is provided both in the .APL and the .AIO formats. To use it, invoke APL in the following way:

```
APL AP101 ... )LOAD PROFILE
```

where '...' represents the names of other auxiliary processors that you may wish to load at sign-on time. The *PROFILE* workspace will be loaded automatically, the *PROFILE* function will be executed, the function keys and the libraries will be defined, and a clear workspace will be generated.

If larger default sizes are required for any of the definitions stored by AP101, it may be convenient to add the request for the larger sizes to the *PROFILE* function. For example, adding the line *X←512 50 40*, to the *PROFILE* function between the existing lines 3 and 4, would set the maximum stack size to 512 bytes, the maximum F-key definition to 50 characters, and the maximum library definition to 40 characters.

A second function in the *PROFILE* workspace (*COPY*) emulates the *)COPY* command (which is not supported by APL/PC 2.1). This function has two arguments: the left argument is the name of the copy source workspace. The right argument is the list of objects to be copied (a character string with the names separated by spaces).

*Note:* This function will not work, and the active workspace may be lost, if the default drive does not have enough space to save both the active workspace and the objects to be copied!

## The UTIL Workspace

The UTIL workspace provides a selection of functions that may be of general use. It contains some functions to assist in application writing and some demonstration functions that show how AP103 may be used to: read the joystick port; peek and poke I/O ports and obtain or free memory for DOS function calls.

- *R←APLIST* - Returns (in *R*) a list of the numbers of the currently active Auxiliary Processors.
- *APL\_DEUTSCH* - Sets German keyboard layout on APL keyboard - Y and Z keys are exchanged.
- *APL\_FRANCAIS* - Sets French keyboard layout on APL keyboard - A-Q and W-Z keys are exchanged.
- *APL\_OFF* - Switches the keyboard to national mode.

- *APL\_ON* - Switches the keyboard to APL mode.
- *APL\_RESTORE* - Sets the international keyboard layout on the APL keyboard.
- *R←ASCII* - Returns (in *R*) the APL to ASCII translation table.
- *R←BIGFONT A* - Takes a character string argument (*A*) and returns (in *R*) the string in a large font.
- *R←DATA\_TYPE A* - Returns (in *R*) the type of its argument (0 = boolean, 1 = integer, 2 = real, 3 = literal).
- *R←DIR* - Uses API03 to return the directory of the default drive.
- *R←EQUIP\_AVAIL* - Returns (in *R*) the system equipment flags as a boolean vector. See the Technical Reference manual for further details.
- *ESC\_OFF* - Disables the Esc and Ctrl-Break keys from interrupting APL execution until *ESC\_ON* is executed.

**Warning:** Use with care!

- *ESC\_ON* - Enables the Esc and Ctrl-Break keys to interrupt APL execution.
- *R←FORMAT A* - Formats its right argument (*A*) like APL2's format function.
- *R←FREEMEM A* - Takes an address (*A*) returned by *GETMEM* and returns the storage to the active workspace, using API03.
- *R←GETMEM B* - Where *B* is the number of bytes required, extracts that amount of space from the active workspace and returns (in *R*) the memory address where the space has been found. It uses API03.

- *R←HEX A* - Converts *A* to hex if it is numeric, or from hex to decimal if it is a literal string.
- *HT* - Disables output to the display until *RT* is run or keyboard is used.
- *R←INKEY* - Mimics BASIC inkey function using AP103. Returns (in *R*) the keyboard scan code.
- *R←W INPORT P* - Uses AP103 to read the current value from a specific I/O port (*P*). If *W* is given, a word (two bytes) is retrieved. Otherwise, a byte is retrieved.
- *R←JOYSTICKS* - A cover function for the AP103 joystick algorithm.
- *R←KEYB* - Returns the address of the APL keyboard translate table.
- *KEYBOARD* - Allows interactive modification of the layout of the APL keyboard.
- *R←MEM\_SIZE* - Returns (in *R*) the true size of the machine.
- *R←P OUTPORT V* - Uses AP103 to send data (*V*, one or two integers in the range 0-255) to a system I/O port (*P*).
- *R←N PEEK A* - Gets *N* bytes from the address defined by *A* (the same address definition as in  $\square PK$ ), but result is in hex.
- *N POKE A* - Like  $\square PK$ , but *N* is given in hex format.
- | • *PRT\_OFF* - Turns off the use of the printer as system log.
- | • *PRT\_ON* - Turns on the use of the printer as system log.  
| Any output displayed on the screen will be printed as well.
- | • *QUEUE A* - Places string *A* in keyboard buffer area. This  
| may be used to stack commands that continue to be  
| interpreted even after APL has terminated. For example:

```
CR←TC[IO+1]
QUEUE ' )OFF',CR,'PC_CMD',CR,'APL',CR
```

exits APL, runs "PC\_CMD" as a DOS command and then restarts APL.

- *RT* - Enables output to the display. It is the inverse of *HT* or the *)Q*(quiet) command in the APL invocation line.
- *SPEAKER* - A sample program that uses *INPORT* and *OUTPORT* to produce a noise in the PC speaker.
- *STORY\_BOARD* - Enables PC Storyboard Picture Taker to be executed when Shift-PrtSc is pressed.
- *WHERE 'text'* - Gives a listing of all functions in the active workspace that contain the indicated text.
- *WHISTLE* - A practical demonstration of using *GETMEM*, *FREEMEM*, *PEEK* and *PK* to load a machine code program into memory and execute it.
- *R←WSNAME* - Returns (in *R*) the name of the active workspace. If it has no name,  $8\rho AV[IO]$  is returned.
- *R←DET* - Returns (in *R*) an indication of the last error encountered.

## The VM232 Workspace

The VM232 workspace supports communications with IBM Virtual Machine Facility/370 (VM/370) on an IBM System/370 with an ASCII port or an equivalent machine.

To operate this application, you need:

- The IBM Personal Computer Asynchronous Communications Adapter.



- Either a duplex modem (either acoustic or direct coupled), or a direct cable connection to the host computer. (The communications program does not support communications using a half-duplex modem).

To use this application from APL programs, you must include both the asynchronous communications auxiliary processor, AP232, and the file management auxiliary processor, AP210, as parameters to the APL command at load time, before you begin an APL work session (or load them through AP2). For example:

```
APL AP232 AP210
```

Then you must copy the files, VM232 and FILE, into your workspace using the following commands:

```
)IN VM232  
)IN FILE
```

You are now ready to start communications with the host.

## Selecting a Terminal

When you start up the communications program, you are in the *terminal-selection* phase. A series of menus lets you select which type of terminal the IBM Personal Computer will simulate, and the detailed features of that terminal.

The terminal-selection phase has three levels of menus. The first-level menu lists the different line parameter definitions that can be selected. When you select one of these definitions, a second-level menu lists the terminal options that can be specified for the selected definition. When you select one of the options, a third-level menu lists the possible choices for that option.

To start the terminal-selection phase, you have to call the function *SETUP*. The following will then appear:

*SETUP*  
*LINE PARAMETER DEFINITION. Select:*  
1: *VM*  
2: *Unused*  
3: *Unused*  
4: *Other*  
5: *Current Definition*  
□:

- Menu item 1 (“VM”) gives you a terminal that operates with most IBM VM/370 System Control Programs running on an IBM computer (see “VM/370 Terminal”).
- Menu items 2 and 3 are listed here for future use.
- Menu item 4 (“Other”) lets you specify pertinent parameters to define your own terminal (see “User-Specified Terminal” on page 11-100).
- Menu item 5 (“Current Definition”) lets you use a terminal specification that you have created in a previous call to the function *SETUP*, and that you have saved using the procedure described under “Saving Your Line Parameter Definition” on page 11-102. The application “remembers” whether you created your current definition using menu item 1 or 4; when you type 5 and press Enter, it brings up the corresponding second-level menu.

## VM/370 Terminal

To access VM/370 and have your IBM Personal Computer operate as a VM/370 terminal, you have to type 1 and press the Enter key while in the *LINE PARAMETER DEFINITION* menu. The following menu then appears:

*PARAMETER CHANGE. Select:*  
0: *No change*  
1: *Baud rate*  
2: *Parity*  
3: *Turnaround local*  
□:

This is the *PARAMETER CHANGE* menu. Using this menu, you can change the baud rate, the type of parity checking, and

the line turnaround character sent to the host. You can also return to APL if you type the number 0 and then press the Enter key.

- **Baud rate:** Describes the speed at which characters are sent across the communications line. The higher the rate, the faster the transmission will be. Generally, this rate is determined by the baud rate that the transmission equipment can handle and/or the baud rates available at the input port for the host computer. If you want to change the baud rate for the host computer, type 1 on the *PARAMETER CHANGE* menu and press the Enter key. The following menu appears:

```

BAUD RATE. Select:
  0: No change
  1:   75
  2:  110
  3:  150
  4:  300  *
  5:  600
  6: 1200
  7: 1800
  8: 2400
  9: 4800
 10: 9600
□:

```

The asterisk (\*) in item 4 indicates that the VM/370 terminal will start up with a communication-line speed of 300 baud (or bits per second), unless you change it. This is the *currently-defined value*. Type the item number that corresponds to the baud rate you are using. For example, if you are connecting to a 1200-baud computer port, type 6 and press the Enter key. This sets the line's bit rate to 1200 baud. The *PARAMETER CHANGE* menu appears on the screen again.

- **Parity:** Characters transmitted over an asynchronous communications line are sent serially as sequences of 1's and 0's that represent each character. The parity bit is the eighth bit of the ASCII character code and is added to the 7-bit code, depending on your selection, so that the character may be checked for accuracy at the receiving end. You have to set the parity to match the type expected by

the host computer. To set the parity bit, enter 2 on the *PARAMETER CHANGE* menu and press the Enter key. The following appears:

```
PARITY. Select:
 0: No change
 1: NONE
 2: ODD
 3: EVEN
 4: MARK *
 5: SPACE
□:
```

The types of parity checked are:

- NONE: No parity bit is added to the character transmitted. Eight bits of data are transmitted for each character.
- ODD: The sum of all bits, including parity, of the character transmitted, is odd.
- EVEN: The sum of all bits, including parity, of the character transmitted, is even.
- MARK: The parity is always set to 1. This is the default.
- SPACE: The parity is always set to 0.

To select the type of parity checking your host system uses, type the corresponding item number and press the Enter key. The *PARAMETER CHANGE* menu appears on the screen again.

- Turnaround Local Character: To tell the host computer that you have completed typing a line of text, you press the Enter key. The character produced when you press Enter is called the *turnaround local* or *line turnaround* character sent to the host. The turnaround character indicates the end of a line of input sent to the host computer. The host computer takes action on that line and sends back a response.

The currently-defined value for this character is a Carriage Return. If you wish to change the value of this parameter, type 3 on the *PARAMETER CHANGE* menu, and press the Enter key. The following appears on the screen:

*TURNAROUND LOCAL CHARACTER. Select:*

- 0: *No change*
- 1: *CR (0DH) \**
- 2: *XON (11H)*
- 3: *XOFF(13H)*
- 4: *EOT (04H)*
- 5: *LF (0AH)*

□:

If you want the turnaround character to be, for example, the line feed (LF), type 5 and press the Enter key. The *PARAMETER CHANGE* menu appears on the screen again.

## User-Specified Terminal

When you select item 4 (“Other”) in the *LINE PARAMETER DEFINITION* menu, you can specify all of the terminal features to make your IBM Personal Computer operate as a terminal for your particular host system. The following menu appears:

*PARAMETER CHANGE. Select:*

- 0: *No change*
- 1: *Baud rate*
- 2: *Parity*
- 3: *No. of stop bits*
- 4: *Half/Full dpx.*
- 5: *Turnaround local*
- 6: *Delete chars.*
- 7: *End of line char.*

□:

To return to APL, type 0 and press the Enter key.

- Baud rate: See “VM/370 Terminal” on page 11-97.
- Parity: See “VM/370 Terminal” on page 11-97.
- No. of stop bits: Stop bits are sent by your IBM Personal Computer after each character to keep the line in

synchronisation. These bits let the receiver detect the beginning of the next transmitted character. Usually one stop bit is required (default). The number of stop bits you select must match the number required by your host system. To change the number of stop bits, type 3 on the *PARAMETER CHANGE* menu and press the Enter key. The following menu appears:

```
NO. OF STOP BITS. Select:
  0: No change
  1: 1 *
  2: 2
□:
```

To select two stop bits, type 2 and press the Enter key. Pressing Enter returns you to the *PARAMETER CHANGE* menu.

- Half/Full dpx: Although a full duplex modem is required, this application does not support duplex transmission protocol. Therefore, when you type 4 in the *PARAMETER CHANGE* menu, the following message appears:

***FULL DUPLEX NOT SUPPORTED***

and the *PARAMETER CHANGE* menu is displayed again.

- Turnaround local: “end-of-line” designator. To change this character you have to type 5 in the *PARAMETER CHANGE* menu and press the Enter key. For more information, see “VM/370 Terminal” on page 11-97.
- Delete chars: When you are in communications with the host computer, the host may transmit characters you do not want displayed on your screen. Generally these are special ASCII characters known as *control characters*.

If you want to change the Delete characters, type 6 in the *PARAMETER CHANGE* menu and press the Enter key. The following will appear on your screen:

*DELETE CHARS. Select up to 4:*

0: *No change*  
 1: *Unused*  
 2: *CR (0DH)*  
 3: *LF (0AH)*  
 4: *BELL(07H)*  
 5: *XON (11H)*  
 6: *XOFF(13H)*  
 7: *ESC (1BH)*  
 8: *TAB (09H)*  
 9: *BS (08H)*  
 □:

Type the numbers of the characters you want to delete. You can type a maximum of four numbers. Then press the Enter key to return to the *PARAMETER CHANGE* menu.

- End of line char: The character selected from this menu specifies the end-of-line character sent from the host computer. This character indicates that a new line should be started on the screen.

The default value provided is a Carriage Return. If you wish to change the value of the end-of-line character sent by the host, type 7 on the *PARAMETER CHANGE* menu and press the Enter key. The following is displayed:

*END OF LINE CHAR. Select:*

0: *No change*  
 1: *CR (0DH) \**  
 2: *XON (11H)*  
 3: *XOFF(13H)*  
 4: *EOT (04H)*  
 5: *LF (0AH)*  
 □:

Type the number of the character you wish to use and press the Enter key to return to the *PARAMETER CHANGE* menu.

## Saving Your Line Parameter Definition

After you have defined the line parameters for your system, you can save your new specifications by executing:

*)OUT name*

where “*name*” is the name of the transfer file in which your application will be stored (see Chapter 10, “System Commands” for a description of the *)OUT* command).

The parameter definition you have saved is now your current definition. The next time you use your application, you have to load it using the command:

```
)CLEAR  
)IN name
```

where “*name*” is the name you used when you saved the application with the *)OUT* command (see Chapter 10, “System Commands” for a description of the *)IN* command).

If you do not want to change the new parameters again, you need not call the function *SETUP*.

## Connection with the Host

When you have selected the communications parameters, you must establish a connection with the host computer by executing the following:

```
TERMINAL
```

A beep sounds and the following messages are displayed:

```
Computer connection NOT established  
You are starting up as a terminal  
Check computer or modem connection  
Starting in RECEIVE state  
Press Esc key to go into SEND state
```

Depending on the type of connection between your IBM Personal Computer and the host system, you must do the following:

- **Modem Connection:** Read the instructions for the modem carefully to understand how to use the telephone set for voice and data transmission.



In general, what you must do is dial the number of the host computer, either by using the telephone or by typing the dial-up commands required by the modem. When you use the dial-up commands, you must go into SEND state by pressing the Esc key. When you hear the modem's carrier (a high-pitched tone), the connection has been made and you must go to the following step.

- Direct Cable Connection or Modem Connection Complete (you hear a carrier): At this stage, two things may have happened:
  - Your IBM Personal Computer was not opened as a terminal (cursor not visible on the screen). You will have to press the Esc key to go into SEND state. You may now have to send a BREAK to the host computer (the application will prompt you for it). You will answer YES or NO, depending on the needs of your host system. The use of BREAK is system-dependent; check with the person who has installed your host system. If your host system requires a BREAK to be sent, sending it will cause your IBM Personal Computer to open as a terminal.
  - Your IBM Personal Computer has opened as a terminal to the host computer. You will receive the following:

```
VM/370 ONLINE  
!
```

(cursor placed here)

Connection is established. You can proceed to log on to your host system.

Each line entered is passed to the host for execution. There is no transmission transparency in this mode: Only standard ASCII characters may be transmitted; APL special characters will be lost and not sent to the host. You may, however, go into the host APL system and execute system commands, load workspaces, and call APL functions.

APL statements that are prefixed with the I-beam character ( $\text{␣}$ ) are executed by the IBM APL/Personal Computer 2.1 system, and are not passed to the host. APL system commands cannot be executed in this way.

The entering of a line consisting of a single I-beam character ( $\text{␣}$ ) is considered as a request to exit function *TERMINAL* and go back into local APL mode. However, transmission is not interrupted (that is, the connection is not lost) until you expressly log off from the remote system. You may also reenter terminal mode by executing the *TERMINAL* function again. If you had not disconnected the remote system, you should not log on again at this point.

*Note:* If transmission fails at any point and your terminal does not return control to you, you can press the Esc key and execute the APL line:

→

You can then try to repeat the operation by invoking the *TERMINAL* function again.

## Functions

Four special APL functions are included in the workspace and may be used for transferring files between the host and the IBM Personal Computer.

These functions may be invoked in terminal mode by preceding their names with an *I-beam* character ( $\text{␣}$ ). The functions are:

- *UPLOAD*
- *DOWNLOAD*
- *APLOUT*
- *APLIN*

These functions assume the following prerequisites:

- Transmission has been established.

- The host VM/370 system contains the file, EDIT EXEC, as described below.
- The host VM/370 system contains the file, APL EXEC, to load VSAPL.
- The host VM/370 system contains the APL workspace, OUT, as described below.

## UPLOAD

Sends a file from disk(ette) to a minidisk in the host. The file must be composed of DOS variable-length records separated by a new-line character and a line-feed character (in that order). The last record must also end with these two characters. Transmission is transparent; that is, all remaining 254 characters (except new line and line feed) may be sent.

When this function is invoked, it asks for the filespec of the source file to be sent (*ENTER SOURCE FILE NAME*). The filespec must be given in DOS format (*[drive:]name.ext*). If the file does not exist, *NOT FOUND* is written and the request is repeated. To exit, press Enter.

Next the target file name is requested (*ENTER TARGET FILE NAME*). It must be given in Conversational Monitor System (CMS) format: filename filetype filemode. If the target file name already exists, a warning is given (*FILE EXISTS. DO YOU WANT TO REPLACE?*). If the answer is YES, the old file will be deleted. Otherwise, uploading stops. If everything is correct, the file is transferred and converted to its final form to assure transparency. Some operations (including invoking APL in the host and executing an APL function) are automatically performed by the function.

## DOWNLOAD

Performs the opposite operation to *UPLOAD*. It sends a file from a minidisk in the host to a disk(ette) in your IBM Personal Computer.

## APLOUT

Takes an APL workspace in transfer form (extension .AIO) on the IBM Personal Computer and sends it to the host. The final result of the execution of this function is a CMS file with filetype AIO, which may be loaded into a VSAPL workspace by means of the following instructions:

```
)CLEAR  
)SYMBOLS size  
)COPY OUT IN  
' ' IN 'filename'  
)ERASE IN  
)SAVE wsname
```

## APLIN

Performs the opposite operation to *APLOUT*. The source workspace must be in normal VSAPLWS format (that is, not in AIO form). The function automatically invokes APL, loads the workspace, converts it into AIO form with the help of the OUT workspace (see below) and sends it to the Personal Computer with transmission transparency. The final result is a file in transfer form, which is created on the Personal Computer's disk(ette), and which may be loaded directly into the active workspace by means of the *)IN* command.

The transformations of alphabetic characters in going between the IBM Personal Computer and the VM/370 system are as follows:

| Function | Source Alphabet: |                  |                    |
|----------|------------------|------------------|--------------------|
|          | Capitals         | Lower Case       | Caps Underscored   |
| UPLOAD   | Capitals         | Lower Case       | N/A                |
| DOWNLOAD | Capitals         | Lower Case       | Special Characters |
| APLOUT   | Capitals         | Caps Underscored | N/A                |
| APLIN    | Capitals         | Lower Case       | Lower Case         |

## Example of Connection with the Host

Load the VM232 and FILE workspaces.

```
)IN VM232
)IN FILE
```

Then create a file to be uploaded to the host.

```
1 WOPEN 'B:TEST,D'
A←'FIRST LINE',□TC[□IO+1 2]
A←A,'SECOND LINE',□TC[□IO+1 2]
A←A,'LAST LINE',□TC[□IO+1 2],'→'
ρA
37
1 0 37 WRITE A
CLOSE 1
TYPEV'B:TEST'
FIRST LINE
SECOND LINE
LAST LINE
```

The file just created has three records with the indicated information.

You will now have to call the function *SETUP* (if you have not get done so) to establish the characteristics of the type of terminal your IBM Personal Computer will simulate, and the detailed features of that terminal.

## IBM Internal Use Only

The IBM Personal Computer is connected to the host computer through a duplex modem with a half-duplex protocol.

To connect your IBM Personal Computer to the host computer, you must execute the function *TERMINAL*. The following will appear on your screen:

```
TERMINAL  
Computer connection NOT established  
You are starting up as a terminal  
Check computer or modem connection  
Starting in RECEIVE state  
Press Esc key to go into SEND state
```

You have to dial up here. When the connection is established, you will receive the message

```
VM/370 ONLINE  
!
```

and you can proceed to log on.

```
L user_name  
ENTER PASSWORD:  
*****  
HHHHHHH  
SSSSSSS  
password
```

Connection messages are received here. You may now IPL CMS.

*CMS*

*↑UPLOAD*

(Request to send)

*ENTER SOURCE FILE NAME*

(Prompt from *UPLOAD*)

*B:TEST*

(Our answer)

*ENTER TARGET FILE NAME*

(Prompt)

*TEST TEST A*

(Our answer)

*FILE EXISTS. DO YOU WANT TO REPLACE?*

(Prompt)

*Y*

(Our answer)

*END OF TRANSMISSION*

(From this point the

*3 RECORDS SENT*

system generates a

*APL*

set of lines assuring

*V S A P L 4.0*

transparency of the

transmission.)

*CLEAR WS*

*)LOAD OUT*

*SAVED 10:13:47 02/01/83*

*CMSIN 'TEST TEST A'*

*R0;*

*)OFF HOLD*

*R;*

*ERASE TEST HIO A*

*R;*

At this point, uploading is complete and the terminal opens.  
You are again connected to CMS.

**IBM Internal Use Only**

|                               |   |
|-------------------------------|---|
| <i>TYPE TEST TEST A</i>       | (user CMS command)                          |
| <i>FIRST LINE</i>             | (System answer)                             |
| <i>SECOND LINE</i>            |   |
| <i>LAST LINE</i>              |   |
| <i>R;</i>                     |   |
| <i>⌵DOWNLOAD</i>              | (DOWNLOAD request)                          |
| <i>ENTER TARGET FILE NAME</i> | (Prompt from <i>DOWNLOAD</i> )              |
| <i>B:TEST1</i>                | (Our answer)                                |
| <i>ENTER SOURCE FILE NAME</i> | (Prompt)                                    |
| <i>TEST TEST A</i>            | (Our answer)                                |
|                               | (Send back the file)                        |
| <i>APL</i>                    | (Next commands are generated automatically) |

*V S A P L 4.0*

*CLEAR WS*

*)LOAD OUT*  
*SAVED 10:37:47 02/01/83*  
*CMSOUT 'TEST TEST A'*  
*R28;*  
*)OFF HOLD*  
*R;*  
*10*  
*ERASE TEST HIO A*  
*R;*



END OF TRANSMISSION  
10 RECORDS RECEIVED

(Records sent in blocks  
of 10, number given is  
rounded up to a multiple  
of 10)  
(We are again under CMS)  
(A single I-beam followed  
by ENTER returns to IBM  
APL/Personal Computer 2.1)

I

TYPEV'B:TEST1'  
FIRST LINE  
SECOND LINE  
LAST LINE  
TERMINAL

(Back to terminal state)  
(Startup messages here)  
(This is a CMS command)

Q PRT  
NO PRT FILES  
R;  
LOG  
(Logoff message)

VM/370 ONLINE

The VM/370 system is now in *receive* state. To return to IBM APL/Personal Computer 2.1, you have to press Esc, then Ctrl-Break, and then →.

The terminal opens now, and you are back in IBM APL/Personal Computer 2.1.

## Auxiliary Files on the Host

To be able to use this application, your host system must have the following files in your minidisk A.

- EDIT EXEC
- The APL workspace OUT
- APL EXEC

## EDIT EXEC

```
&CONTROL OFF  
CP TERMINAL ESCAPE OFF CHARDEL OFF  
  LINEND OFF LINEDEL OFF LINESIZE 165  
CP SET MSG OFF WNG OFF ACNT OFF  
SET BLIP OFF  
SET TERMINAL LINESIZE 255  
&STACK CASE M  
&STACK RECFM V  
EDIT &1 &2 &3 &4 &5 &6 &7
```

## The APL Workspace OUT

The functions in this workspace can be divided into three different sets:

- Those that perform EXPORT/IMPORT:
  - CMSOUT* Converts 256-character files into ASCII-compatible files.
  - CMSIN* Converts ASCII-compatible files into 256-character files.
  - APLOUT* Like *CMSOUT*, but underscored letters are replaced by lowercase letters.
  - APLIN* Like *CMSIN*, but lowercase letters are replaced by underscored letters.
- Auxiliary to EXPORT/IMPORT:
  - CIN* Used by *CMSIN*, *APLIN*
  - COU* Used by *CMSOUT*, *APLOUT*
  - GASC* Used by *CMSIN*, *APLIN*, *CMSOUT*, *APLOUT*
  - XUL* Used by *APLOUT*
  - XUL1* Used by *APLIN*
  - CMS* Used by all four and *STACK*
- IN/OUT Functions:
  - IN* Equivalent to the *)IN* command (see "The FILE Workspace" on page 11-56).
  - OUT* Equivalent to the *)OUT* command (see "The FILE Workspace" on page 11-56).

## IBM Internal Use Only

In the function listings on the following pages, some non-APL characters are included. These characters must be generated from a 3270 terminal in "APL-OFF" mode. The only functions containing these non-APL characters are: *CAPL*, *CIN*, *COUT*, *GASC*, *XUL* and *XUL1*.

Functions:

|               |               |             |            |            |              |
|---------------|---------------|-------------|------------|------------|--------------|
| <i>APLIN</i>  | <i>APLOUT</i> | <i>CAPL</i> | <i>CIN</i> | <i>CMS</i> | <i>CMSIN</i> |
| <i>CMSOUT</i> | <i>COUT</i>   | <i>GASC</i> | <i>IN</i>  | <i>OUT</i> | <i>XUL</i>   |
| <i>XUL1</i>   |               |             |            |            |              |

```
∇ APLIN X;A;SH;N;I;∅IO;ASC;N1;N2;AUX
[1]  ∅IO←0
[2]  N←X, ' HIO(192'
[3]  A←110 ∅SVO 'N'
[4]  →(0≠1↑A←N)/E
[5]  →(∧/ 0 1 1 =3↑A)/E
[6]  CMS 'ERASE ',X,' AIO'
[7]  SH←X, ' AIO(192 FIX'
[8]  A←110 ∅SVO 'SH'
[9]  →(∨/ 0 1 1 ≠3↑SH)/E
[10] GASC
[11] L:→(0=ρA←N)/0
[12] SH←80↑XUL1 CIN A
[13] →L
[14] E:'ERROR'
∇
```

```

∇ APLOUT X;A;B;SH;N;I;□IO;ASC;N1;N2;AA
[1] □IO←0
[2] N←X,(4 0[ ' 'εX]↑' AIO'),'(192'
[3] A←110 □SVO 'N'
[4] →(0≠1↑A←N)/E
[5] →(∧/ 0 1 1 =3↑A)/E
[6] CMS 'ERASE ',(X←(X1' ')↑X),' HIO'
[7] SH←X,' HIO(192'
[8] A←110 □SVO 'SH'
[9] →(∨/ 0 1 1 ≠3↑SH)/E
[10] GASC
[11] L:→(0=ρA←N)/0
[12] SH←COUT XUL A
[13] →L
[14] E:'ERROR'

```

∇

```

∇ Z←CAPL X
[1] Z[( '$ '=Z)/1ρZ←X]←'≠'
[2] Z[( '@ '=Z)/1ρZ]←'α'
[3] Z[( '# '=Z)/1ρZ]←'Δ'

```

∇

```

∇ Z←CIN X;□IO;I;J
[1] □IO←0
[2] X←(I←Zε'"_')/1ρZ←X,((~1↑X)ε'"_')/' '
[3] X←(N1,N2)[((ρN1)×Z[X]='_')+ASC1Z[X+1]]
[4] Z←(~I∨J←~1ΦI)/Z
[5] Z←(~I←(~J)/I)\Z
[6] Z[I/1ρI]←X

```

∇

```

∇ CMS X;CP;I
[1] CP←'CMS'
[2] I←100 □SVO 'CP'
[3] CP←X
[4] 'R',(⊖CP),';'

```

∇

```

▽ CMSIN X;A;SH;N;I;□IO;ASC;N1;N2;AUX
[1] □IO←0
[2] N←((X1' ')↑X←CAPL X),' HIO(192'
[3] A←110 □SVO 'N'
[4] →(0≠1↑A←N)/E
[5] →(∧/ 0 1 1 =3↑A)/E
[6] CMS 'ERASE ',X
[7] SH←X,' (192'
[8] A←110 □SVO 'SH'
[9] →(∨/ 0 1 1 ≠3↑SH)/E
[10] GASC
[11] L:→(0=ρA←N)/0
[12] SH←CIN A
[13] →L
[14] E:'ERROR'

```

▽

```

▽ CMSOUT X;A;B;SH;N;I;□IO;ASC;N1;N2;AA
[1] □IO←0
[2] N←(X←(X1' ')↑X),((X1' ')↑X←CAPL X),' (192'
[3] A←110 □SVO 'N'
[4] →(0≠1↑A←N)/E
[5] →(∧/ 0 1 1 =3↑A)/E
[6] CMS 'ERASE ',X,' HIO'
[7] SH←X,' HIO(192'
[8] A←110 □SVO 'SH'
[9] →(∨/ 0 1 1 ≠3↑SH)/E
[10] GASC
[11] L:→(0=ρA←N)/0
[12] SH←COUT A
[13] →L
[14] E:'ERROR'

```

▽

```

▽ Z←COUT X;I;J;□IO
[1] □IO←0
[2] Z←,X
[3] X←~Zε(∼ASCε□AV[23 30])/ASC
[4] I←,Φ(2,ρI)ρ'"'_[I≥ρN1],(ASC,ASC)[I←(N1,
N2)ιX/Z]
[5] Z←(∼X)/Z
[6] J←(∼X)/0,~1↓+∖X+1
[7] X←((+/X)+ρX)ρ0
[8] X[J]←1
[9] Z←X∖Z
[10] Z[(∼X)/ιρX]←I
▽

```

```

▽ GASC;□IO
[1] □IO←0
[2] ASC←' ',□AV[23], '#$%&' '()*+,-./01234567
89:;<=>?@ABCDEFGHIJKLMNPOQRSTUVWXYZ'
[3] ASC←ASC,' ]abcdefghijklmnopqrstuvwxyz',
□AV[30]
[4] N1←□AV[0 248], 'E', □AV[224 244 229 249
12 225 11 245 230 246 226]
[5] N1←N1, □AV[231 234 251 227 247 232 2224
25 228 233 237 238 31]
[6] N1←N1, □AV[15 14], 'ΔΔ', □AV[(1+ι8), (16+ι4
21, (26+ι4), (32+ι3), 131]
[7] N1←N1, 'W', □AV[30], 'XYZΔ~IQ_..▽αωηυςϷιx÷
Γ|∧∨∧∧≤≥'
[8] N2←'z!ριειτφθϑ∕\⊗Δ∇↑↓↔←□□⊕⊗[a', □TC[0
1]
[9] N2←N2, 'UMN', □AV[240 239], 'QD', □AV[241],
'FBGHJ', □AV[20], 'CK', □AV[219 220]
[10] N2←N2, 'L', □AV[222 223 242], 'PRST', □AV[
243], 'V', □AV[(203+ι4), 235 236]
[11] N2←N2, □AV[(207+ι12), 221 250 23 252 253
254 255 132 13]
▽

```

```

∇ ZR←LS IN SH1; ZX2; ZXM; ZXA; ZX1; PP; IO
[1] PP←15+IO←1+ZR←0
[2] SH1←SH1, ((~' '∈SH1)/' AIO'), '(192 FIX'
[3] WA←110 SVO 'SH1'
[4] →(0≠1↑SH1)/0
[5] ZR←1
[6] →(0=ρLS)/ZXL2
[7] ZR←(1↑ρLS)ρ0
[8] ZXL2: ZXA←' '
[9] ZXL3: →(0=ρZX1←SH1)/0
[10] ZXA←ZXA, 1↓-8 0 [1+(1↑ZX1)∈' CE' ]↓80↑ZX1
[11] →((1↑ZX1)∈' C' )/ZXL3
[12] ZXM←(ZXA↑' ' )↓ZXA
[13] ZXA←(ZXA↑' ' )↑ZXA
[14] →(0=ρLS)/ZXL1
[15] →((ρZXA)>2+1↓ρLS)/ZXL2
[16] →(~1∈LS∧.(1↓ρLS)↑1↓ZXA)/ZXL2
[17] ZXL1: ZX1←ϕ(ZXM↑' ' )↑ZXM
[18] ZXM←(ZXM↑' ' )↓ZXM
[19] ZX2←10
[20] ZXL4: →(ZX1=0)/ZXL5
[21] ZX2←ZX2, ϕ(ZXM↑' ' )↑ZXM
[22] ZXM←(ZXM↑' ' )↓ZXM
[23] →ZXL4, ZX1←ZX1-1
[24] ZXL5: →(' FC'=1↑ZXA)/ZXL6, ZXL7
[25] ϕ(1↓ZXA), '←', (ϕZX2), ((0≠ρZX2)/' ρ'), ZXM,
(0≠ρZX2)/' 0'
[26] →ZXL8
[27] ZXL6: ZX1←' '=0\0ρ FX ZX2ρZXM
[28] →ZXL9
[29] ZXL7: →(∧/' FC'=3↑1↓ZXA)/ZXL8
[30] ϕ(1↓ZXA), '←', (ϕZX2), ((0≠ρZX2)/' ρ'), 'ZXM'
[31] ZXL8: ZX1←1
[32] ZXL9: →(0∈ρLS)/ZXL2
[33] ZR[(LS∧.(1↓ρLS)↑1↓ZXA)↑1]←ZX1
[34] →(0=ρLS)/ZXL2
[35] →(0∈ZR)/ZXL2

```

∇



```

∇ ZR←ZXM OUT SH1; ZXA; ZX1; PP; CMS
[1] PP←16
[2] →(∼0∈ρZXM)/ZXL1
[3] ZXM←(∧/ZXM∨.≠((¬1↑ρZXM),3)↑q 3 3 ρ'OUT
SH1ZXM')÷ZXM←NL 2 3
[4] ZXL1:ZR←0≠NC ZXM
[5] ZXL0:→(0=ρZXM)/0
[6] ZXA←1↑((ZX1←3=NC ZXM[IO;] )/'F'), 'CN' [
IO←¬1↑ϕ'0', (2=NC ZXM[IO;] )/' ,ϕ'0=0\
0ρ' ,ZXM[IO;] ,'   ']
[7] ZX1←ϕ(ZX1/'CR' ), ZXM[IO;] ,ZX1/'   '
[8] ZXA←ZXA, (( '   ' ≠ZXM[IO;] )/ZXM[IO;] ),
'   ' , (ϕ(ρZX1), ρZX1), '   ' , , ϕZX1
[9] ZXA←(((¬1+1↑ρZXA)ρ' '   '), 'X' ), ZXA←(ZX1,
79)↑((ZX1←Γ(ρZXA)÷71), 71)ρZXA, 71ρ' '   '
[10] →(2=NCSVO 'SH1')/ZXL2
[11] CMS←'CMS'
[12] WA←100 NCSVO 'CMS'
[13] CMS←'ERASE 'SH1, 'AIO'
[14] 'R', (ϕCMS), ';'
[15] SH1←SH1, 'AIO(192 FIX'
[16] WA←110 NCSVO 'SH1'
[17] ϕ(0≠1↑SH1)/' →ZR←0'
[18] ZXL2:SH1←ZXA[IO;]
[19] ZXA← 1 0 ↓ZXA
[20] →(0≠1↑ρZXA)/ZXL2
[21] ZXM← 1 0 ↓ZXM
[22] →ZXL0

```

∇

```

∇ R←XUL X; I; J
[1] J←26≠I←'ABCDEFGHIJKLMNOPQRSTUVWXYZ' ∖R←
X
[2] R[J/ρR]←'abcdefghijklmnopqrstuvxyz' [J/I]

```

∇

```
▽ R←XUL1 X;I;J
[1] J←26÷I←'abcdefghijklmnopqrstuvwxyz' ιR←
    ,X
[2] R[J/ιρR]←'ABCDEFGHIJKLMNOPQRSTUVWXYZ' [
    J/I]
▽
```

## Accessing Host APL

You should consult your local installation personnel for guidance in accessing host APL (e.g. VS APL Release 4, Program Product 5748-API). In general, certain CMS minidisks may need to be linked, which may require special authorisation or passwords.

**Notes:**

## Chapter 12. Auxiliary Processors

|  |       |
|--|-------|
| The Non-APL Program Interface Auxiliary Processor: AP2 | 12-4  |
| Basic Functions  | 12-5  |
| Auxiliary Functions                                    | 12-8  |
| Sample AP2 session                                     | 12-9  |
| Return Codes (Returned through the control variable)   | 12-9  |
| The Printer Auxiliary Processor: AP80                  | 12-10 |
| Patching AP80 for Other Printers                       | 12-12 |
| The Stack and Profile Auxiliary Processor: AP101       | 12-14 |
| Error Return Codes                                     | 12-17 |
| The BIOS/DOS Interrupt Auxiliary Processor:            |       |
| API03  | 12-18 |
| BIOS/DOS Interrupt Function Call                       | 12-19 |
| I/O Port IN/OUT Request                                | 12-22 |
| Joystick Algorithm                                     | 12-23 |
| The Full Screen Management Auxiliary Processor:        |       |
| API24  | 12-24 |
| API24 Operation  | 12-24 |
| Error Return Codes                                     | 12-33 |
| The Host Communications Auxiliary Processors:          |       |
| API90 and API90I                                       | 12-34 |
| Possible uses for API90                                | 12-35 |
| Getting Started  | 12-35 |
| Sending Keystrokes                                     | 12-35 |
| Setting Keyboard Translation Table                     | 12-36 |
| Getting Host Status                                    | 12-36 |
| Getting the Physical Screen                            | 12-36 |
| Get the Operator Information Area                      | 12-37 |
| Simulate a Power On Reset                              | 12-37 |
| Get Cursor Position and Beep Indication                | 12-37 |
| Get the Keyboard Translation Table                     | 12-37 |
| Get the Screen Format Array                            | 12-38 |
| The Full-Screen Auxiliary Processor: AP205             | 12-38 |
| The Graphic Auxiliary Processor: AP206                 | 12-39 |
| Storage Management                                     | 12-39 |

|   |       |
|---|-------|
| Parameters .....  | 12-40 |
| Use of AP206 .....  | 12-47 |
| Functions .....   | 12-48 |
| Return codes .....  | 12-53 |
| The File Auxiliary Processor: AP210 .....                                     | 12-53 |
| Control Commands .....  | 12-54 |
| Control Subcommands .....   | 12-57 |
| AP210 Return Codes .....  | 12-59 |
| Examples of use .....   | 12-60 |
| The Asynchronous Communications Auxiliary<br>Processor: AP232 .....           | 12-62 |
| Control Commands .....  | 12-63 |
| The Extended Asynchronous Communications<br>Auxiliary Processor: AP232X ..... | 12-69 |
| Hardware Notes .....  | 12-70 |
| AP232X Operation .....  | 12-70 |
| AP232X Return Codes .....   | 12-75 |
| The Music Auxiliary Processor: AP440 .....                                    | 12-76 |
| AP440 Command Syntax .....  | 12-77 |
| The IBM GPIB Support Auxiliary Processor: AP488 .....                         | 12-79 |
| Description of AP488 Functions .....  | 12-80 |

**Notes:**

The auxiliary processors discussed in this chapter are:

- AP2 - Non-APL program interface
- AP80 - IBM Graphics Printer control
- AP101 - Stack and profile
- AP103 - BIOS/DOS interrupt handling
- AP124 - Full-screen display management
- AP190 - IBM 3278/79 emulation card communications
- AP190I - IRMA 3278/79 emulation card communications
- AP205 - Full-screen display management
- AP206 - Graphics
- AP210 - DOS file management
- AP232 - Asynchronous communications
- AP232X - Extended asynchronous communications
- AP440 - Music generator
- AP488 - GPIB/IEEE488 interface

Each auxiliary processor requires storage space in addition to that required for APL and the shared variable processor (\$SVP). When you start APL with at least one auxiliary processor, the shared variable processor is loaded with it. If you load more than one auxiliary processor, only one copy of the shared variable processor is loaded. (Shared variables are described in Chapter 7, “Shared Variables”).

*Note:* The shared variable processor supplied with APL/PC 2.1 has a limit on the maximum size of objects it can handle of 32512 bytes. Passing larger objects to or from auxiliary processors will give unpredictable results.

The following table gives approximate sizes for the APL, the auxiliary processors and other modules. EXAPL (dyadic format) and \$SCR are automatically loaded by APL, so their combined size is given. The shared variable processor, \$SVP, and the 8087 emulator, \$8087, are automatically loaded by APL if they are required.

| <u>Module</u>   | <u>Approximate<br/>Size (K-bytes)</u> |
|-----------------|---------------------------------------|
| APL+EXAPL+\$SCR | 78.1                                  |
| \$SVP           | 3.4                                   |
| \$8087          | 7.6                                   |
| AP2             | 3.2                                   |
| AP80            | 3.1                                   |
| AP101           | 1.2                                   |
| AP103           | 0.7                                   |
| AP124           | 12.4                                  |
| AP190           | 2.1                                   |
| AP190I          | 2.2                                   |
| AP205           | 8.3                                   |
| AP206           | 10.2                                  |
| AP210           | 2.9                                   |
| AP232           | 5.0                                   |
| AP232X          | 2.7                                   |
| AP440           | 0.8                                   |
| AP488           | 2.7                                   |

## The Non-APL Program Interface Auxiliary Processor: AP2

This processor is designed to load and execute non-APL programs written either as programs that can be executed from the DOS command processor, or as APL functions (*Exchange Assembly* programs).

The AP2 processor acquires special areas of memory from the APL workspace, called partitions, where programs can be loaded and executed, and non-executable files may be loaded.

The processor exchanges information with APL through three shared variables:

1. Control variable (the name of which must begin with the letter *C*).
2. Two data variables (whose names must begin with *D* and *E*, respectively).

In the control variable the desired function to be performed is specified, while other parameters are passed through the data variables.

Upon return, the control variable contains the return code (see the Return Codes table) and the data variables will contain the objects generated through the appropriate function.

The functions which can be used are divided into 2 groups:

- Basic Functions
- Auxiliary Functions

## **Basic Functions**

The basic functions are used for acquiring space, loading, executing and unloading programs or files, and returning space to APL. They may be performed either separately or in combination, by using the following format (where brackets enclose an optional argument):

$$C \leftarrow p1, p2, p3 [, p4 [, p5]]$$

where:

- *p1, p2* are the numbers from 1 to 5 corresponding to the initial and final basic operations the user wants to execute. (If just one operation is required, both numbers will be the same). The basic operations and the order of their execution are:
  1. Acquisition of memory and assignment to a partition
  2. Loading programs/files into the acquired memory
  3. Execution of the loaded program
  4. Unloading the loaded program/file
  5. Freeing memory and returning it to APL



- *p3* - the number of the partition to be used
- *p4* - the type of the file (this parameter is required if operation 2 is being requested. It will be ignored in all remaining functions):
  - 1: Non-executable file
  - 2: Auxiliary processor
  - 3: Exchange-assembly program
  - 4: Standard DOS program
- *p5* - the restore information for execution. This parameter is applicable to type 3 and 4 files, and may be given if operation 2 is being requested. It will be ignored with all remaining functions and file types:
  - 0: Neither the interrupt vector nor the program code are restored after execution of these programs. The programs are assumed not to destroy the interrupt vector.
  - 1: The interrupt vector is automatically restored after execution of these programs. The program code is not restored.
  - 2: The program code is automatically restored after execution of these programs. They therefore become reusable. (This is applicable, for instance, to the Personal Editor). However, the interrupt vector is not restored, and is assumed to be maintained by execution of the program.
  - 3: Both the interrupt vector and the program code are automatically restored after execution of these programs.

The default value of this parameter is 0.

Notes:

## IBM Internal Use Only

1. When executing basic operation 1 (acquisition of memory) without function 2 (load), the number of paragraphs of 16 bytes to be acquired must be specified in the variable *D*.
2. When executing basic operation 2, it is always necessary that variable *D* contains the file specification (i.e.  $D \leftarrow '[d:]filename[.ext]'$  or  $D \leftarrow '[libn] filename[.ext]'$ , where *libn* is an APL library number). For file types 2-4, if the extension is omitted, COM is assumed. If the file is not found, the operation is retried with an extension of EXE.
3. With the auxiliary processors (file type 2), only two types of basic operations are permitted: loading ( $C \leftarrow 1, 3, p, 2$ ), or unloading ( $C \leftarrow 4, 5, p$ ), where *p* is the partition number.
4. When exchange-assembly programs (type 3) receive control, positions 80H and 82H of the segment prefix contain segment pointers to the addresses of variables *D* and *E*. That is to say, the value of *D* is at [DS:80]:0, and the value of *E* is at [DS:82]:0. (The segment prefix is at DS:0, according to DOS rules). When control is returned to AP2, the processor tests the values of the objects pointed to by those same positions (80H and 82H at the segment prefix). If they are valid APL objects, they are passed to APL through shared variables *D* and *E*, respectively, and the buffers are freed. If either of these is not a valid APL object, the value will not be passed to APL. In this case, it is the responsibility of the exchange-assembly program to free the respective buffers by means of interrupt X'B6').
5. When executing standard DOS programs (type 4), the value in variable *E* (if any) is passed to the DOS program as a parameter string. The value is assumed to be of type literal.

Example: To call the Personal Editor with an initial file name (i.e. equivalent to PE B:FILE.EXT under DOS) the literal '*B:FILE.EXT*' would be passed through variable *E*.

6. Insufficient storage allocated to a partition for a program to execute in will lead to the loss of your APL session. Some programs grow into storage below themselves, and sufficient storage should be allocated to allow for this. DOS provides no memory protection facilities. Similarly, programs expecting the entire memory to be available to them such as MASM 2.0 will cause problems.
7. Programs to be executed using AP2 should not use the DOS 4A or 4B functions.
8. Ctrl-Break will be disabled by AP2, so programs to be executed using AP2 should not rely on this being available.

## Auxiliary Functions

$C \leftarrow 0[,p]$

This function returns the state of the partition specified in the parameter or of all partitions, if omitted. Variable *E* returns the name and extension of the file contained in the partition, if any. Variable *D* returns 4 parameters per partition: compound type of file in the partition, start address in paragraphs, length of the partition in paragraphs, and actual length used by the file, in paragraphs. For auxiliary processors, this last information is replaced by the processor identification number if sign-on to the SVP was performed successfully.

The compound type of a file is a two digit decimal integer. The first digit is the type of the file; the second one is the restore information for execution.

Example: a compound type of 31 means the file is of type 3 (exchange program) and the interrupt vector is automatically restored after execution.

$C \leftarrow -1,parameter.$

This function unloads and frees the memory of all the partitions and establishes a new maximum number of

partitions with the value specified in the parameter. The valid range of values for *parameter* is from 1 to 100.

### Sample AP2 session

```

      A Share with AP2
      2 □SVO 3 1p'CDE'
1 1 1
      A Verify the share
      □SVO 3 1p'CDE'
2 2 2
      A Request 20K bytes of memory
      D←[20480÷16
      C←1 1 1
      A 1 1=Get Memory, in partition 1
      A Check return code
      C
0
      A Allocation succeeded
      A Define name of the program to load
      D←'PROGRAM'
      C←2 2 1 4 3
      A 2 2=Load, Partition 1, 4= Std DOS,
      A 3=Restore interrupt and program code
      C
0
      A Pass argument to program
      E←'ARGUMENT',□TC[2]
      C←3 5 1
      A 3=Execute, 5=Free memory, 1=Partition
      C
0

```

### Return Codes (Returned through the control variable)

| Code | Meaning                                |
|------|--|
| 0    | - Success                              |
| 2    | - File not found                       |
| 3    | - Path not found                       |
| 4    | - No file handles                      |
| 5    | - Access denied                        |
| 8    | - Insufficient memory for file handles |
| 15   | - Invalid drive                        |
| 21   | - Drive not ready                      |

- 23 - Data error
- 25 - Seek error
- 27 - Sector not found
- 30 - Read fault
- 31 - General failure
- 1 - Data variable *D* required and not shared
- 2 - Invalid object in the control variable
- 3 - Invalid object in data variable *D*
- 4 - Non-executable file requested as executable
- 5 - Loading of this program is expressly prohibited
- 6 - Program to be loaded is larger than partition
- 26 - Insufficient memory to process request

The following return codes will only occur as a result of attempting to pass a previously freed exchange-assembly object:

- 1001 - Unable to free space allocated to *C*
- 1002 - Unable to free space allocated to *D*
- 1003 - Unable to free space allocated to *E*

## The Printer Auxiliary Processor: AP80

The AP80 auxiliary processor can be accessed from APL on the IBM Personal Computer and provides a way to control the IBM Graphics Printer from APL functions. It allows you to specify the printing parameters and to print character strings. The entire APL character set is supported.

To use this auxiliary processor, you must include AP80 as a parameter to the APL command at load time before you begin an APL work session. For example,

```
APL AP80
```

Alternatively, it may be loaded through AP2.

AP80 must be loaded to allow any use of the printer from within APL.

## IBM Internal Use Only

The following APL line must be executed before the auxiliary processor can be used:

```
80 □SVO 'name'
```

where *name* is the name of the APL variable being shared with the auxiliary processor.

The result of the preceding line will be a 1 if the variable *name* has been accepted by the auxiliary processor. This processor accepts only one variable.

The following line must be executed next:

```
□SVO 'name'
```

The execution of this line must give a result of 2. If not, the auxiliary processor is not active, or a different variable has been shared with it and has not been retracted.

Any character string (vector or scalar) assigned to the variable defined by *name*, will be interpreted as a command to the auxiliary processor.

If the first character in the string is □AV[□IO+255], the remaining characters in the string are sent to the printer in alphanumeric mode. In this way, printer control codes can be included and executed. The printer control codes that may be used are listed in the *PRINT\_GEN* function in the PRINT workspace.

When a new-line character is found, the print head returns to the beginning of the same line. A line-feed character sends the print head to the beginning of the next line.

If the first character in the string is not □AV[□IO+255], the whole string is printed as given. A new-line character sends the print head to the beginning of the next line, as does a line-feed character.

Therefore, a character can have a dual function, depending on the selected printing mode.

*Example:*

```

      □IO←1
      80 □SVO 'X'
1
      □SVO 'X'
2

```

Variable *X* has been offered to, and accepted by, AP80.

```
X←'ABCD',□TC[2]
```

The printer prints the string ABCD followed by a return to the beginning of the next line (caused by the New line character, □TC[2], in one-origin).

```
X←□AV[256 159], 'E'
```

Set emphasised mode (printer control code).

```
X←'ABCD',□TC[2]
```

The printer prints the string ABCD in emphasised mode, followed by a new line.

```

      □SVR 'X'
2

```

Finally, the variable is retracted.

If the value of □PW is less than or equal to 80, normal characters are printed. Otherwise, compressed mode is used and the printable line length is increased from 80 to 132. This affects all three modes of using AP80: screen copy, screen log, and direct printing through a shared variable.

## Patching AP80 for Other Printers

| AP80 can be patched for use with Grastrax, the IBM 5182  
| Colour Printer, the IBM 3852-2 Colour Ink Jet Printer, or for  
| wide printers such as the FX100 and MX100 printers.

| To apply a patch to the copy of AP80 on the default drive, the  
| following procedure may be used:

## IBM Internal Use Only

```
APL AP210
)IN FILE
APLPATCH 'AP80GRAF.PAT'
```

| where AP80GRAF.PAT is the name of the DOS file  
| containing the required patch.

| Patch for Graftrax (AP80GRAF.PAT):

```
| Addr Old New
| 347 2B 56 Different setup for compressed print
| 35A 59 4C Change graphics mode from "Y" to "L"
```

The accented national characters and some of the other special characters cannot be printed, but the normal APL characters should print correctly.

| Patch for IBM 5182 Colour Printer (AP805182.PAT):

```
| Addr Old New
| 367 99 0F Point to different
| 368 04 07 character table (optional)
| 36E 03 05 Modify character spacing (Note that
| compressed print does not work
| correctly with the IBM Colour Printer)
```

| Patch for IBM 3852-2 Colour Ink-Jet Printer  
| (AP803852.PAT):

```
| Addr Old New
| 347 2B 56 Different setup for compressed print
| 35A 59 4B Change graphics mode from "Y" to "K"
| 36E 03 01 Modify character spacing
| 3B0 4C 4B Change graphics mode from "L" to "K"
| 3C4 02 01 Modify character spacing
```

| Patch for FX100/MX100 type printers with wider print lines  
| (AP80FXMX.PAT):



| Addr | Old | New |   |
|------|-----|-----|---|
| 344  | 50  | 84  | Switch to compressed mode at $\square PW > 132$   |
| 353  | 50  | 84  | Max length of normal line = 132                   |
| 37E  | 84  | D2  | Max length of compressed line = 210               |
| 3A9  | 84  | D2  | Max length of compressed line = 210<br>(Graftrax) |

*Note:* The addresses given above are for the APL *PATCH* function in the FILE workspace. To use the DOS DEBUG program, add X'100' to each address.

## The Stack and Profile Auxiliary Processor: AP101

AP101 is an auxiliary processor to support the following functions:

- Input line stack. Lines added to the stack are provided as input for subsequent requests for input, in the same order they were given. The default size of the stack is 1024 bytes.
- Function-key definition.
- Library definition. Up to 30 APL libraries with numbers from 1 to 30 may be defined to various DOS directories or sub-directories. APL file commands (*)LIB*, *)LOAD*, *)SAVE*, *)DROP*, *)IN* and *)OUT*) will work automatically with these libraries. The two file auxiliary processors, AP210 and AP2 will also allow the use of these library numbers.
- Screen-keyboard mode change. The screen may be defined as monochrome or one of the colour monitor modes. The keyboard may be defined to be APL or National.

The auxiliary processor must be invoked at APL load time in the following way:

APL AP101

or it may be loaded dynamically by means of AP2.

A variable is shared with AP101 in the usual way. Once the variable has been accepted by AP101, the following commands are recognised (the name of the shared variable is assumed to be X):

- $X \leftarrow 'ST \text{ any char string}'$

Inserts the character string in the input line stack. The string may contain new-line characters, which will provide each of the corresponding lines as input when input is subsequently requested.

- $X \leftarrow 'ST'$

Clears the stack of all lines previously added to it.

- $X \leftarrow 'Fn \text{ any char string}'$

Where  $n$  is a number from 1 to 30, defines the corresponding function key to produce the given character string. New lines may also be included in the string to produce immediate execution. There is a maximum length of the function key string (new line characters included). Its default value is 15.

F1-10 are the normal F-keys. F11-20 are the same keys, together with the shift key. F21-30 are again the same keys, together with the Ctrl key.

- $X \leftarrow 'Fn'$

Undefines function key number  $n$ .

- $X \leftarrow 'F0'$

Undefines all function keys.

- $X \leftarrow 'F?'$

Returns in **X** the current definition of the F-keys, as a matrix with 30 rows and as many columns as the current maximum.

- $X \leftarrow 'Ln \text{ directory string}'$

Where  $n$  is a number from 1 to 30, defines the corresponding library number to refer to the given directory string. Both drive definitions and sub-directories may be included in the string, which should end with a “\”. (For example: “C:\SUB1\SUB2\”). There is a maximum length of the directory string. Its default value is 15.

- $X \leftarrow 'Ln'$

Restores library number  $n$  to its default state. If  $n$  is a library number that corresponds to a valid PC DOS drive id, then it will recover its default mapping. (e.g. library 1 is “A:”, 2 is “B:”, 3 is “C:”, etc.).

- $X \leftarrow 'LO'$

Restores all library numbers to their default state. All library numbers that correspond to valid PC DOS drive ids will recover their default mapping. (e.g. library 1 is “A:”, 2 is “B:”, 3 is “C:”, etc.).

- $X \leftarrow 'L?'$

Returns in **X** the current definition of the libraries, as a matrix with 30 rows and as many columns as the current maximum.

- $X \leftarrow m, n[, p]$

Where:

$m$  - is a number between 256 and 32767, sets  $m$  as the new size of the input stack buffer. The current contents of the stack buffer are lost. The default size is 256.

$n$  - is a number between 10 and 100, sets  $n$  as the new maximum length for F-key definitions. All current F-key definitions are lost. The default size is 15.

$p$  - is a number between 10 and 100, sets  $p$  as the new maximum length for each library definition. The current contents of the library definition buffer are lost. The default size is 15.

A value of 0 for any of these parameters leaves the corresponding setting unchanged.

- $X \leftarrow n$

If  $0 \leq n \leq 7$ , BIOS screen mode  $n$  is selected, if possible. If the monochrome adapter is not present, mode 7 will be rejected with a return code of  $\bar{3}0$ . The same will happen with modes 0-6 if the colour graphics monitor adapter is not present.

With the help of this function, the user may build a PROFILE (see the preceding chapter) to initialise APL to any one of the supported modes.

If  $n = \bar{1}$ , the APL-on keyboard is selected.

If  $n = \bar{2}$ , the national (APL-off) keyboard is selected.

## **Error Return Codes**

### **Code Meaning**

- 0 - Success
- 1 - Invalid object
- 2 - Object too large
- $\bar{2}6$  - No space available
- $\bar{3}0$  - Invalid BIOS screen mode

## The BIOS/DOS Interrupt Auxiliary Processor: AP103

The AP103 auxiliary processor provides an interface to generate BIOS and DOS interrupts or function calls. This processor replaces AP100 (included in the Personal Computer APL System, Version 1.0), and extends its function to include integer specification of the shared variable and I/O port word and byte access. A joystick algorithm has also been included as a special case.

The processor utilises a single shared variable that may have any valid name.

To make the best use of the facilities of this AP, it is advisable to refer to the PC Technical Reference manual for details of the processor registers, port addresses and BIOS interrupts. Refer also to the DOS Technical Reference Manual for details of DOS calls and interrupts.

To use this auxiliary processor, you must include AP103 as a parameter to the APL command at load time before you begin an APL work session. You may also load it dynamically through AP2.

For example,

```
APL AP103
```

The following APL line must be executed before the auxiliary processor can be used.

```
103 □SV0 'name'
```

where *name* is the name of the APL variable being shared with AP103. The result of the previous line will be a 1 if the variable name is accepted.

The following line must be executed next.

```
□SV0 'name'
```

It must give a result of 2. If not, either the auxiliary processor is not active, or a different variable has been shared with it and has not been retracted.

## **BIOS/DOS Interrupt Function Call**

Any integer or character vector with at least 17 elements assigned to the variable name, will be interpreted as a command to the auxiliary processor to generate a BIOS/DOS interrupt. If integer, all values must be positive and in the range 0-255. If character, it will be considered equivalent to the one-byte integers that are equal to the position of each element in the APL atomic vector ( $\square AV$ ), in zero index origin.

The elements of the vector must contain the following information:

```
C[1] - Interrupt number
C[2] - AL register
C[3] - AH register
C[4] - BL register
C[5] - BH register
C[6] - CL register
C[7] - CH register
C[8] - DL register
C[9] - DH register
C[10] - SI register low part
C[11] - SI register high part
C[12] - DI register low part
C[13] - DI register high part
C[14] - BP register low part
C[15] - BP register high part
C[16] - Input: reserved
      - Return: low order flags
C[17] - Translation options(*)
      - Return: high order flags
```

plus none, two or four of the following additional elements:

```
C[18] - DS segment low part
C[19] - DS segment high part
C[20] - ES segment low part
C[21] - ES segment high part
```

## \* AL register translation options:

| Value | Input     | Output    |
|-------|-----------|-----------|
| 0     | as-is     | as-is     |
| 64    | as-is     | ASCII/APL |
| 128   | APL/ASCII | as-is     |
| 192   | APL/ASCII | ASCII/APL |

If element 16 is 128 or 192, AL values are considered as internal APL characters, and are translated to their ASCII equivalents before the interrupt takes effect. If element 16 of the input command was 0 or 128, the return values are passed as they are. However, if element 16 of the input command was 64 or 192, the AL values are considered as ASCII characters, and are translated to their internal APL equivalents. (See the preceding diagram). This feature is to provide compatibility with the AP100 auxiliary processor supplied with APL/PC 1.0 where only characters strings could be passed to or from the AP. When using the numeric form of input to AP103, the translation option should be ignored, and its value set to 0.

The value returned will be a vector with the same type and number of elements as the input. The register values will be updated to their new values after execution of the interrupt. The flags will be returned in elements 16 and 17.

Failure to parse the request will be signified by a return code of -1. Specifying an integer register value outside of the range 0 through 255 will cause unpredictable results.

## Example of Use:

To read an ASCII character struck from the keyboard, using BIOS interrupt 16H and AP103, you can execute the following function (see the *Technical Reference* manual for information about BIOS interrupts):

```
[0] Z←INKEY;X;□IO
[1] □IO←0
[2] Z←103 □SVO 'X'
[3] Z←□SVO 'X'
[4] X←22,16ρ0
[5] Z←X[1 2]
```

- Line 1 sets the origin to 0.
- Lines 2 and 3 share variable X with AP103.
- Line 4 assigns to the shared variable X, the input needed to address interrupt 16H; that is:
  - Element 0 is the number of the interrupt desired (22 is equivalent to X'16').
  - Element 2, the contents of AH, is set to 0.
  - Elements 15 and 16 are set to 0.
  - The contents of the remaining elements are not important.
- Line 5 returns the result of the interrupt:
  - Element 1 returns the contents of AL, the struck key code.
  - Element 2 returns the contents of AH, the key scan code.

When you call the function, *INKEY*, the system stops processing when line 4 is executed. The system waits for you to press any key. When you do so, the function returns a two-element vector: the first element is the key code, and the second is the scan code (see “Keyboard Encoding and Using” in the appropriate *Personal Computer Technical Reference* manual).

The *INKEY* function in the UTIL workspace provides the same capability as this example. See “The UTIL Workspace” on page 11-92.



*Notes:*

1. *AP100, as supplied with APL/PC Version 1.0 had the ability to repeatedly issue interrupts with various values of the AL register. This mode of operation is not supported by AP103.*
2. *If AP103 is used to issue a call to BIOS to switch between 40 and 80 column modes, it is important that the appropriate BIOS equipment flag is changed to reflect this change. Failure to do so will cause problems in graphics modes when backspacing or inserting characters. The mode may be safely changed using AP101, AP124 or AP206.*

**I/O Port IN/OUT Request**

This function is designed to permit a user defined function to read from or write to specific hardware ports with either byte or word data.

**C** ← **DH,DL,AL**

Perform a byte OUT command.

Return: always 0.

**C** ← **DH,DL,AL,AH**

Perform a word OUT command.

Return: always 0.

**C** ← **DH,DL**

Perform a byte IN command.

Return: value of AL register.

**C** ← **1 2 p DH,DL**

Perform a word IN command.

Return: value of AX register.

If any of the register values are not in the range 0-255, unpredictable results will occur. Failure to parse the command will result in a return of -1.

## **Joystick Algorithm**

In addition to the above functions, a special case has been included in this AP to permit the scanning and calculation of the position of devices attached to the Games Control Adapter (e.g. joysticks or paddles).

This function clears the port and then scans it for a change, calculating the joystick position in a convenient way for APL processing.

It should be noted that it is the responsibility of the user defined function to issue the call at regular enough intervals to permit timely interception of changes in joystick requests.

Syntax of this function is:

$$C \leftarrow 1 \ 2 \ p \ 2 \ 1$$

(0201H is the Games Adapter port).

Return code in *C*:

*C*[1] - Joystick-A X Co-ordinate  
*C*[2] - Joystick-A Y Co-ordinate  
*C*[3] - Joystick-B X Co-ordinate  
*C*[4] - Joystick-B Y Co-ordinate  
*C*[5] - Joystick-A Button 1  
*C*[6] - Joystick-A Button 2  
*C*[7] - Joystick-B Button 1  
*C*[8] - Joystick-B Button 2

If it is not possible to obtain enough space to generate the above return code, the result will be  $\bar{2}$ .

# The Full Screen Management Auxiliary Processor: AP124

This auxiliary processor allows you to control the screen from an APL defined function. It permits your application to:

- Write to the formatted screen.
- Read from the formatted screen.
- Copy screen images to a printer.
- Produce colours, highlighting, reverse video, etc.
- Control the keyboard translation.
- Enter “Inkey” mode to trap single key interrupts.
- Run asynchronously.
- Define up to 100 fields.

AP124 requires two shared variables: a data variable, (whose name must start with a “D”), and a command-control variable (whose name must start with a “C”). These variables may be shared in any order but only one pair is allowed.

## AP124 Operation

In the following, the physical screen is the adapter card screen memory, and the logical screen is a buffer in normal memory containing a copy of the screen contents. Operations are performed normally on the logical screen, the information of which is transferred to the physical screen (i.e. the physical screen is refreshed) by certain calls (Read and Wait, Immediate Write).

## Clear Screen/Change Mode

Syntax:

*Ctl*←0 - Clear screen.  
*Ctl*←0,*Mode* - Clear screen and change display mode.  
Both the logical screen and the physical screen are cleared.

where *Mode* is an integer from 0 to 7.

0 = 25 × 40 Black and White Alpha.  
1 = 25 × 40 Colour Alpha.  
2 = 25 × 80 Black and White Alpha.  
3 = 25 × 80 Colour Alpha.  
4 = 200 × 320 Colour Graphics.  
5 = 200 × 320 Black and White Graphics.  
6 = 200 × 640 Black and White Graphics.  
7 = 25 × 80 Monochrome Display.

## Format the Screen into Fields

Syntax:

*Dat*←*n* by 6 numeric array  
*Ctl*←1 - Format the screen.  
*Ctl*←1,*Field\_number(s)* - Reformat field(s).

This call permits you to divide your logical screen into rectangular areas known as fields. Each field is defined in terms of its offset from the left hand corner of the screen, its depth and its width. You must also indicate whether the field is input or input/output and its display "attribute".

Example:

*Dat*←1 6p10 5 1 1 0 7

defines a field in the tenth row, fifth column, one high, one wide. The field has a type of input/output and attribute is normal.

Hint: To provide a complete background colour, define field 1 of your array as `1 1 1, n, 2, c`, (where (where  $n$  is either 1000 for 40 column modes, 2000 for 80 column modes) and  $c$  is the background colour attribute, e.g. 31 will give blue.

This is possible because this implementation of AP124 permits fields to overlap. Fields defined in the format matrix may overlap fields already defined by previous rows of the matrix.

The special case of a 1000 or 2000 character long field covering the entire screen also sets the border colour to the same attribute setting.

The six elements of each row of the format array are defined as:

1. Start row of the field
2. Start column of the field
3. Field height
4. Field length
5. Field type: either 0 (Input/Output) or 2 (Output only).
6. Field attribute: an integer between 0 and 255. The following are some attribute examples applicable to the monochrome and the colour monitor in the alphanumeric modes (see the IBM Technical Reference Manual for full details):

- 0 - No display
- 1 - Underlined
- 7 - Normal
- 9 - Highlighted underlined
- 15 - Highlighted
- 112 - Reverse video

- 120 - Highlighted reverse video
- 129 - Blinking reverse video
- 135 - Blinking normal

## Immediate Write of Data to Screen

Syntax:

*Dat*←*n* by *max\_length* array of character  
*data*

*Ctl*←2, *Field\_number* (*s*)

where *n* is the number of fields you are writing data to. This call permits you to write data to the logical screen. The data are transferred immediately to the physical screen.

*max\_length* is the maximum length of the requested fields, i.e. the total number of characters in the longest field.

## Read and Wait

In normal operation, the physical screen is refreshed. Then, either the auxiliary processor waits for a certain key to be pressed, or it returns to APL with information on whether a key was pressed.

Syntax:

| *Ctl*←3  
| or,  
| *Ctl*←3 0

Allow interactive input and return to APL when a special key is pressed. (See table below for return values). Assigning the control variable a scalar value of 3, or the two element vector | 3 0, are completely equivalent.

*Ctl*←3 1

Return to APL when any key is pressed.

*Ctl*←3 2

Test for a key pressed.

*Ctl*←3 3

Return to APL when any key except a cursor movement key is pressed. ("Semi-inkey" mode).

If any of the above calls has 3 elements, this indicates the auxiliary processor to inhibit refreshing the physical screen.

Return from this call: *Dat* is a vector of at least 5 elements:

*Dat*[1 2] For call 3 0, a code indicating the special key pressed to return to APL. For calls 3 1, 3 2 and 3 3, the BIOS scan code. (For call 3 2, if no key has been pressed, return is  $\bar{1}$   $\bar{1}$ ).

*Dat*[3] Field number where the cursor was located at return to APL (0= the cursor was not in a field).

*Dat*[4 5] Cursor position within that field. (Row/Column). If Field number is 0, these elements give the offset from top-left of the screen. Offsets are given in one origin, i.e. 1 1 specifies the top-left corner of the field.

*Dat*[6...] List of fields updated during this call.

List of special keys and key return codes for the 3 0 case:

- 0 0 - Enter key (New line key)
- 0 1 - Enter key 2 (Large plus key)
- 1, *N* - F key (Where "*N*" ( $1 \leq N \leq 30$ ) is the number of the key that was depressed. 1-10: normal F-keys; 11-20: F-keys in shift mode; 21-30: F-keys in Ctrl mode).
- 4 1 - Esc
- 4 2 - Ctrl Break
- 6 1 - Home

## IBM Internal Use Only

- 6 2 - End
- 6 3 - Pg Up
- 6 4 - Pg Dn

While AP124 is in the 3 0 “Read and Wait” state, the user may type on the screen. The cursor movement keys may be used in the normal way. Three other keys have special functions:

- Ctrl-Backspace This key will toggle the keyboard between APL and National modes.
- Ctrl-End This key will clear to the end of the field. However, if the field has less than 80 columns, only the current line will be cleared to its end.
- Ctrl-Horne This key will clear from the cursor to the start of the field. However, if the field has less than 80 columns, only the current line will be cleared to its beginning.

## Delayed Write of Data to Screen

Syntax:

*Dat←n by max\_length array of character data*

*Ctl←4,Field\_number(s)*

where *n* is the number of fields you are sending data to. This call permits you to write data to the logical screen, which will be displayed at the next refresh of the physical screen.

*max\_length* is the maximum length of the requested fields, i.e. the total number of characters in the longest field.



## Get Data from the Logical Screen

Syntax:

```
Ctl←5,Field_number(s)
```

Return: *Dat* is a character array with so many rows as the number of fields requested, and so many columns as the maximum field length (field length is the total number of characters in a field).

This call enables you to read data from the logical screen.

## Update Field Types

Syntax:

```
Dat←New field Type(s)  
Ctl←6,Field_number(s)
```

where the new type is a number:

0 - Field is input/output.

2 - Field is output only.

This call updates column five of the format array previously specified for the indicated fields.

## Update Field Attributes

Syntax:

```
Dat←Attribute(s)  
Ctl←7,Field_number(s)
```

The new attribute is an integer from 0-255. See "The AP124 Workspace" on page 11-8, or the Technical Reference Manual, for more information on attributes.

## Control and Information Request

Syntax:

- Ctl*←8 - Set to APL Keyboard.
- Ctl*←8 0 - Set to APL Keyboard. (Same as *Ctl*←8).
- Ctl*←8 1 - Set to National Keyboard.
- Ctl*←8 2 - Return Status.

Return:

- Dat*[1] - Keyboard in APL mode.
- Dat*[2] - Monochrome Adapter installed.
- Dat*[3] - Colour Adapter installed.
- Dat*[4] - Beep request pending.

Syntax:

- Ctl*←8 3 - Give address of Logical screen.

This call is designed for use by advanced programmers. The return is a integer vector with the requested address encoded as:

$$Addr \leftarrow (16 \times 256 \downarrow Dat[1 \ 2]) + 256 \downarrow Dat[3 \ 4]$$

The above formula computes the address so as to be useful for a  $\square PK$  of the logical screen. This is 2000 bytes long and is stored with no attributes. (e.g. 2000  $\square PK$  0, *Addr* is the current screen). It should be considered as a ravelled 25x80 character matrix, into which ASCII data may be directly peeked or poked.

## Get Format Table

Syntax:

*Ctl*←9

Return: *Dat* is a *n* by 6 numeric matrix, where *n* is the current number of fields defined.

This call returns the current format array stored by AP124.

If no format array currently exists then *Dat* is set to:

1 1, *No\_of\_rows*, *No\_of\_columns*, 2 1

## Print or Retrieve the Current Logical Screen

Syntax:

*Ctl*←10 - Print the active screen on the printer.

Note 1: AP80 or equivalent must be loaded; otherwise the print request will fail. If the printer runs out of paper, AP124 will return error code 54 in *Ctl*.

Note 2: AP124 will print the physical screen currently being displayed. It is the user's responsibility to make sure the active screen contains the desired image.

*Ctl*←10 1 - Return Logical screen in *Dat* variable.

## Sound a Beep to Alert User

Syntax:

*Ctl*←11 - Delayed Beep.

*Ctl*←11 0 - Delayed Beep. (Same as *Ctl*←11).

*Ctl*←11 1 - Immediate Beep.

*Ctl*←11 2 - Cancel previous delayed Beep.

If the beep is delayed, it will occur at the next "Read and Wait" operation. To find out whether a beep is pending, specify call 8 2 and examine the fourth element.

## Set the Cursor

Syntax:

```
Dat←Field_no,Row offset,Column_offset  
Ctl←12
```

This call is designed to set the cursor in a specific screen location. *Field\_no* is a defined field. If it is zero, then row and column are considered as co-ordinates from the top left corner of the screen. *Dat* must be a three element numeric vector. Offsets are given in one origin, i.e. 1 1 specifies the top-left corner of the field.

## Error Return Codes

### Code Meaning

- 0 - Success
- 11 - *Ctl* Rank Error
- 12 - *Ctl* Length Error
- 13 - *Ctl* Domain Error
- 14 - Invalid call
- 21 - *Dat* Rank Error
- 22 - *Dat* Length Error
- 23 - *Dat* Domain Error
- 24 - *Dat* not shared
- 25 - *Dat* Value Error
- 26 - *Dat* too large
- 30 - Invalid field number
- 31 - Invalid mode or device not attached
- 37 - Invalid field type
- 38 - Invalid attribute
- 54 - Printer out of paper

## The Host Communications Auxiliary Processors: AP190 and AP190I

These auxiliary processors are designed to permit APL/Personal Computer, Version 2.1 to communicate with a host session in a simple way convenient to APL.

You can carry out the following actions with these APs:

- Send keystrokes, including control keys such as “Enter” to a host.
- Get the status of the host (e.g. input inhibited, ready).
- Retrieve the screen image as an  $n \times 80$  character array.

It is necessary to have an IBM PC 3278/79 emulation card installed before AP190 can be used or an IRMA PC 3278/79 emulation card installed for AP190I.

Please note that the IBM 3278/79 emulation card does *NOT* support APL characters. To understand what characters may be sent to the host, regard the terminal as a 3278 model 2 without the APL feature.

The IRMA 3278/79 emulation card does however give full APL support.

Both APs contain sufficient code to run the adapter cards without the respective emulators installed. They are both completely stand-alone. However, you may desire to load the appropriate terminal emulator before starting APL in order to use the facilities provided by the emulator.

*Note:* When using the IBM PC3278 emulator that drives the 3278/79 emulation card, the Alt-R (r) and Alt-S (s) keys are taken over by the emulator. To obtain these characters either use *KEYBOARD* from the UTIL workspace to reassign these keys or use Alt-Backspace to set the keyboard to national mode.

## Possible uses for AP190

- Run IMS/CICS sessions to collect and process data in a PC.
- Run a VSAPL or APL2 session in a cooperative manner.
- Run laboratory equipment, and pass results to host session.
- Any task which is labour intensive and requires operator attendance for an extended period, to acquire data that will be further processed.

## Getting Started

Invoke APL with AP190 for the IBM card, or with AP190I for the IRMA card, or load the appropriate AP using AP2. Once in APL, share with AP190 in the normal manner. Only one shared variable is required (no special naming restrictions). After offering the variable, test for its acceptance by AP190 (a degree of coupling of 2). If not accepted, then AP190 has not been loaded in this APL session or another variable is already shared with it.

## Sending Keystrokes

Syntax:

*C* ← *character\_string\_vector*

Return: zero if OK.

The character string may also contain special control characters (see the AP190 workspace).

In this and the subsequent function descriptions, *C* will be the name of the shared variable.

## | Setting Keyboard Translation Table

|  $C \leftarrow 256 \ 2 \ p \ keyboard\_translation\_table$

| Return: zero if OK. The default keyboard translation tables for both AP190 and AP190I is UK English. Tables for some other keyboards may be found in the AP190 workspace. AP190I may also be loaded with an APL translation table.

## Getting Host Status

Syntax:

$C \leftarrow 0$

Return: numeric integer in  $C$ .

If zero, then host is in a “ready” state. If greater than zero, then host status is either “system not available”, “input inhibited”, or both. A test of host status should always be made after key strokes have been sent to ensure the host is ready to accept more processing and has not gone down!

## Getting the Physical Screen

Syntax:

$C \leftarrow 1$

Return: either a  $24 \times 80$  character array for AP190, or a  $32 \times 80$  character array for AP190I. This array represents the screen as if it were being viewed.

| The screen contents may be assigned to a variable and processed using the full power of APL, providing a very flexible way of extracting data from a standard host enquiry.

|  $C \leftarrow 1 \ 1$

| As above, but the array returned is not translated to the APL character set.

## Get the Operator Information Area

Syntax:

$C \leftarrow 2$

Return: a vector of 80 characters representing this area.

|  $C \leftarrow 2 \ 1$

| As above, but the vector returned is not translated to the APL  
| character set.

## Simulate a Power On Reset

Syntax:

$C \leftarrow 3$

Return: always a numeric zero.

## | Get Cursor Position and Beep Indication

| Syntax:

|  $C \leftarrow 4$

| Return: a three element numeric vector, containing the row  
| and column of the cursor (in 0-origin), and a flag (0 or 1) to  
| indicate that the host has requested the terminal to beep.

## Get the Keyboard Translation Table

Syntax:

$C \leftarrow 5$

Return: the current keyboard translation table.



## | Get the Screen Format Array

| Syntax:

|  $C \leftarrow 6$

| Return: the screen format in the form of an AP124 format array.

## The Full-Screen Auxiliary Processor: AP205

The full-screen auxiliary processor, AP205, is included as a part of the IBM APL/Personal Computer System, Version 2.1, for compatibility with applications developed for the IBM Personal Computer APL System, Version 1.0 (the 1.0 version of AP205 is not compatible with APL/PC Version 2.1). This auxiliary processor will not be described here, since its use is not recommended for the development of new applications (AP124 should be used in this case).

The following commands have been added/replaced to AP205 (in the following,  $C$  is the control shared variable):

- $C \leftarrow 13$  sounds a beep.
- $C \leftarrow 0 [ , M [ , BC ] ]$  clears the screen. If  $M$  is given, screen mode is changed. If  $BC$  is given, it defines a background colour.

In mode 4, normal colour is now yellow. In modes 6 and 8, it is white. In mode 8, the background colour is really the character colour. Therefore, it may be changed with this command.

## **The Graphic Auxiliary Processor: AP206**

This processor represents graphic information defined as a set of straight-line segments, drawing the boundary and/or filling the area. It is oriented towards the colour-graphics adaptor, although it can also generate the image in memory (without displaying it on the screen) for printing on the graphics printer. It works with a single shared variable (no name restrictions), and supports the following basic functions:

- Representation of graphic information.
- Generation of graphic characters with a variable format. The definition of a set of these characters is included in AP206. The entire APL character set is included, but some special PC characters (such as the box characters) are not.
- Storage and fast regeneration of the whole screen or a part of it.
- Animated presentation of images.
- Printing of the whole screen or a part of it.
- Interactive generation of images.

### **Storage Management**

This processor dynamically acquires and frees certain memory buffers, taken from the APL workspace. The acquired areas are restored to the APL workspace as a stack (last-in first-out), though they can be freed in any order, so that when one of them is freed, the space does not become available until all the subsequently requested areas have been liberated.

Certain areas in this processor can be explicitly requested by the user. In these cases, it is the user's responsibility to free them when they are no longer needed. These user-requested areas include:

1. A virtual screen buffer (VSB), usable for the following purposes:
  - To use AP206 in the absence of the colour-graphics monitor adapter and its screen buffer (SB). The generated images may be copied to the graphics printer.
  - To prepare an image slowly and transfer it immediately to the screen with a VSB to SB copy.
2. Up to 128 different window buffers (WB), used to keep images for any of the following purposes:
  - To be used later.
  - To be repeated along the screen.
  - To generate animation.
  - To be transferred to/from APL variables.

## Parameters

The work of this auxiliary processor is controlled by a set of 25 integer parameters. When the shared variable is shared, these parameters receive a set of default values. At any time, one or several parameters may be changed.

The following is a list of all the 25 control parameters, in the order they must be specified, including the range of values allowed for each one:

## IBM Internal Use Only

| <u>NO</u> | <u>NAME</u>      | <u>Description</u>  | <u>Values</u> |
|-----------|------------------|---|---------------|
| 1         | MODE             | Screen mode   | 4-8           |
| 2         | BG               | Background colour (00 colour)   | 0-31          |
| 3         | PAL              | Palette(01-10-11 line colour)   | 0-1           |
| 4         | COLOUR<br>/STYLE | Bit combination defining colour<br>and line style                         |               |
| 5         | REPX             | Pixel repetitions in X axis   | 1-8           |
| 6         | REPY             | Pixel repetitions in Y axis   | 1-200         |
| 7         | WX1              | Left column of window   | 0-(WX2-1)     |
| 8         | WX2              | Right column of window  | (WX1+1)-ncol  |
| 9         | WY1              | Bottom row of window  | 0-(WY2-1)     |
| 10        | WY2              | Top row of window   | (WY1+1)-200   |
| 11        | VX1              | Viewport X coord. of (WX1,WY1) $\neq$ VX2                                 |               |
| 12        | VX2              | Viewport X coord. of (WX2,WY2) $\neq$ VX1                                 |               |
| 13        | VY1              | Viewport Y coord. of (WX1,WY1) $\neq$ VY2                                 |               |
| 14        | VY2              | Viewport Y coord. of (WX2,WY2) $\neq$ VY1                                 |               |
| 15        | X0               | Initial X coord. for next operation                                       |               |
| 16        | Y0               | Initial Y coord. for next operation                                       |               |
| 17        | FX               | Scale factor for X axis (%)   |               |
| 18        | FY               | Scale factor for Y axis (%)   |               |
| 19        | FI               | Inclination factor (%)  |               |
| 20        | PARM20           | Extra character displacement in X axis<br>or filling code for even lines. |               |
| 21        | PARM21           | Extra character displacement in Y axis<br>or filling code for odd lines.  |               |
| 22        | SW22             | Chars autodisplacement switch   | 0-1           |
| 23        | SW23             | Chars autocatenation switch   | 0-1           |
| 24        | SW24             | Horizontality of X axis   | 0-1           |
| 25        | SW25             | VSb switch  | 0-1           |

### 1. Parameters 1, 2, 3: MODE, BG, PAL

The colour-graphics monitor adapter describes a full screen with 128000 bits (approximately 16K bytes). The way in which these 128000 bits define pixels on the screen depends on the adapter mode. There are two basic modes:

- Each bit defines a pixel. The screen is made up of 200 rows, 640 pixels each. If a bit is off, it appears as black. If it is on, it appears in the background colour.

- Each bit *pair* defines a pixel. The screen is made up of 200 rows, 320 pixels each. The four combinations allowed are:
  - a. 00 - background colour
  - b. 01 - colour 1
  - c. 10 - colour 2
  - d. 11 - colour 3

All combinations of these can be selected through the MODE, BG, PAL parameters.

MODE can take any of the following values:

- 4 - activates mode 200×320 with 2 palettes available.
- 5 - activates mode 200×320 with just one palette available.
- 6 - activates mode 200×640.
- 7 - activates the monochrome display, with no change in the internal colour monitor mode.
- 8 - maintains the current mode. Allows change of other parameters without the screen being erased.

The default mode is the active mode when the variable was shared.

BG can have any of the following values:

**BG Background Colour**

- 0,16 black
- 1,17 blue
- 2,18 green
- 3,19 cyan
- 4,20 red
- 5,21 magenta
- 6,22 brown
- 7,23 light grey

- 8,24 dark grey
- 9,25 light blue
- 10,26 light green
- 11,27 light cyan
- 12,28 light red
- 13,29 light magenta
- 14,30 yellow
- 15,31 white

Values 0-15 give low foreground intensity, 16-31 give double foreground intensity. Default value for BG is 16 in modes 4-5, 15 in modes 6-7.

PAL (only active in mode 4) can be assigned the values 0 and 1, with the following effect:

| <u>NAME</u> | <u>BITS</u> | <u>MODE=4<br/>PAL=0</u> | <u>MODE=4<br/>PAL=1</u> | <u>MODE=5<br/>PAL=0,1</u> |
|-------------|-------------|-------------------------|-------------------------|---------------------------|
| Colour 1    | 01          | Green                   | Cyan                    | Cyan                      |
| Colour 2    | 10          | Red                     | Magenta                 | Red                       |
| Colour 3    | 11          | Brown                   | White                   | White                     |

The default value is 1.

More information can be found in the PC Technical Reference Manual.

**2. Parameters 4, 5, 6: COLOUR/STYLE, REPX, REPY**

REPX must have a value between 1 and 4 in modes 4-5, or between 1 and 8 in mode 6. This parameter defines the number of times a pixel must be repeated along the X axis, to the right. The full representation of a pixel (with its X repetitions) will be called a "unit of horizontal representation" (UHR).

REPY must be a number 1-200, that defines the number of times the UHR will be repeated along the Y axis.

If both parameters have a value of 1, the pixel will not be repeated along any axis (this is their default value).

COLOUR/STYLE specifies a value between  $\bar{32768}$  and 32767. This may be broken into the COLOUR and STYLE parts by:  $(16\rho 2)\tau CS$  to give a 16 element vector. The least significant 8 bits define the combination of colours for the UHR: 1 bit per pixel in mode 6, 2 bits per pixel in modes 4-5.

Example: In modes 4-5, COLOUR=85 generates Colour 1, whatever the value of REPX, since  $85 = 01\ 01\ 01\ 01$ .

In a similar way, COLOUR=170 generates Colour 2, since  $170 = 10\ 10\ 10\ 10$ ; COLOUR=255 generates Colour 3, and COLOUR=0 generates the background colour. However, other combinations are possible.

The default value for COLOUR is 255.

The most significant 8 bits define the line style for the UHR. These bits define the pattern of dots to be placed along the line. A 0 gives a dot, and a 1 gives a space so that STYLE=0, causes a solid line to be drawn. The specified pattern of 8 dots is applied cyclically along the line.

The COLOUR and STYLE values may be combined into the two byte signed integer required by AP206 by:

$CS\leftarrow L(2\downarrow STYLE, (8\rho 2)\tau COLOUR) - 65536 \times 128 < STYLE$

### 3. Parameters 7, 8, 9, 10: WX1, WX2, WY1, WY2

They provide the definition of a window on the graphic screen. Rows are numbered 0-199. Columns are numbered 0-639 in mode 6, 0-319 in modes 4-5.

The default window is the maximum allowed by the default mode, i.e. 0 319 0 199 for modes 4-5, 0 639 0 199 for mode 6.

### 4. Parameters 11, 12, 13, 14: VX1, VX2, VY1, VY2

They define a viewport, or system of cartesian coordinates over the previously defined window. The default value is

equal to the window, but other combinations are possible. All four parameters defining the viewport must lie in the interval  $[-16383, 16383]$ .

Example: a viewport equal to  $-160\ 160\ -100\ 100$  would locate the origin of coordinates in the centre of the screen.

If  $VX1 > VX2$ , a symmetry with respect to the vertical axis is performed. Likewise, if  $VY1 > VY2$ , a symmetry with respect to the horizontal axis is performed. If both apply, then a central symmetry takes place.

5. Parameters 15, 16, 17, 18, 19: X0, Y0, FX, FY, FI

The coordinates X, Y, defined explicitly in the graphic matrix, or implicitly in the representation of graphic characters, are transformed in the following way:

$$\begin{aligned} X1 &= X + Y \times FI \div 100 \\ X2 &= X0 + X1 \times FX \div 100 \\ Y2 &= Y0 + Y \times FY \div 100 \end{aligned}$$

FI is an inclination factor, slanting characters and/or graphics the percentage indicated. A positive value slants to the right, a negative value to the left. FX and FY are scale factors affecting the corresponding axis. X0 and Y0 are additional translations of coordinates. Their default values are 0, 0, 100, 100, 0, providing the “identity” transformation.

In the case of texts, the internal definition of the character set is also affected by these transformation parameters. However, the shape of some resulting characters may be somewhat distorted unless the scale parameters are chosen both equal to a multiple of 33.33 (rounded to the nearest integer) and the inclination factor is either zero, or a multiple of 100 (positive or negative).

6. Parameters 20, 21: PARM20, PARM21

Their values are interpreted in two different ways, depending on the function they are applied to.



- During polygon filling, they define the filling colours to be used.
- During text writing, they give the extra displacement between characters.

When an area is filled, the values of these parameters define the UHR for even and odd lines, respectively, with the same criterion explained for parameter COLOUR. Values between 0 and 255 are allowed. The 8 bits in the values define the UHR, 1 bit per pixel in mode 6, two bits per pixel in modes 4 and 5.

When a text is written, these parameters provide an extra displacement between characters in a string. Each character has its own width, which is added to PARM20 to compute the X position of the next character in the string. PARM21 allows the string itself to be written in any angle (with sloping characters).

Both parameters may have negative values. Their default values are both zero.

#### 7. Parameters 22, 23, 24, 25: SW22, SW23, SW24, SW25

The last four parameters are switches. Their values are only 0 or 1.

**SW22:** When its value is 1 (the default) each character is written to the right of the preceding one, at a distance depending on the actual character written before. When SW22 = 0, this translation is not performed. All the characters in the string would then appear one of top of the others, unless PARM20 or PARM21 are in effect.

**SW23:** When its value is 1 (the default) the values of X0 and Y0 are automatically re-adjusted (at the end of an operation) to the coordinates of the last pixel to be drawn. In this way, for instance, successive strings would concatenate on the screen. If SW23 = 0 no re-adjustment of X0, Y0 is performed.

This parameter does not affect graphics, only character strings.

SW24: When its value is 1 (the default) the X axis in the viewport runs horizontally from left to right and the Y axis vertically from bottom to top. If SW24=0, the X axis runs vertically from bottom to top, and the Y axis horizontally from right to left (i.e. a 90 degrees counterclockwise rotation is performed).

All of the above assumes that the viewport-to-window correspondence does not introduce any symmetrical transformation. If these symmetries are present, the 90 degree rotation affects the corresponding viewport axis.

SW25: When its value is 0 (the default if the colour-graphics monitor adapter is installed) graphics are generated directly on the SB. When SW25=1 (the default when the colour-graphics monitor adapter is not installed) images are generated on the VSB. The VSB buffer is acquired automatically the first time SW25 is assigned a value of 0, and it is not freed until the shared variable is retracted. Therefore it is convenient to acquire the VSB before any WB are requested, to make easier the return of the space of the latter to APL.

## **Use of AP206**

The different functions are defined by the rank, type and dimension of APL objects assigned to the shared variable. When this is shared, the existence or absence of the colour-graphics monitor adapter is detected, and the VSB is acquired if the answer is negative. When the variable is retracted, all buffers are freed automatically.

When a given function has been performed, the value returned through the shared variable may be one of the following:

- A return code: 0 (success) or a negative integer (see attached table).
- A positive integer (function “literal matrix”).
- A vector of 25 integers, the parameters (functions 0 and 1).
- A literal matrix (function 8).
- A graphic matrix.

## Functions

The following objects may be assigned to the shared variable:

- 0 - Request for the default parameters. *MODE* is the one active when the variable was shared. *Window* is the maximum possible in that mode. The shared variable returns a vector of all the 25 default parameters.
- 1 - Request for the current parameters. A vector of the 25 values is returned.
- 2 - The whole SB or VSB is erased (filled with zeros).
- 2, *n* - The current window is filled up with  $UHR = n$ .
- 3[, *density*[, *margin*]] - Copies the present window of the active screen (SB or VSB) to the graphic printer. The *density* parameter can be 0 (low density printing, the default) or 1 (high density printing). “*margin*” is number of spaces to be added as a left margin of the printed copy (the default centres the window on the paper).
- 4 - Copies the VSB to the SB.
- 5, *n1* [, *n2*] - Drops WB numbers *n1* through *n2*, freeing their space. If *n2* is not given, only *n1* is dropped.
- 6, *n1* [, *n2* [, *repeat* [, *wait*]]] - Copies buffer *n1* to the current window on the active screen (SB or VSB). If

$n2$  is given, buffers  $n1$  through  $n2$  are copied sequentially (each one erases the preceding one). If “repeat” is given, the whole operation is repeated the indicated number of times. Finally, “wait” specifies number of hundredths of a second between any two buffer copying operations. “repeat” default value is 1, “wait” default value is 4. This function performs an animated presentation of the buffer contents on the screen.

- 7,  $n$  - Erases WB number  $n$ , if it exists, generates it again and copies the current window from the active screen (SB or VSB) into it.
- 8,  $n$  - Sends to APL the contents of WB number  $n$  in the form of a literal matrix. Each row contains the description of a row of the window, each character is a description of 4 pixels (in modes 4-5) or 8 pixels (in mode 6). The first and last characters in each row may contain extra pixels to complete the window buffer.
- A vector of 25 integers - A new set of parameters is defined to the auxiliary processor. In fact, a single or a few parameters may be changed by means of an indexed assignment to the shared variable, in the following way:

```
G←1  A G is the shared variable.  
A Get the current parameters.  
G[2 4 6]←A,B,C  
A Change parameter 2 to A,  
A 4 to B and 6 to C.
```

- A literal vector - The string is written on the active screen, affected by the current parameters.
- A literal matrix - Acquires a new WB, copies on this the contents of the matrix (as defined in function 8) and returns to APL a number between 1 and 128 giving the identification number of the WB. (The first free WB number is always assigned).
- A 3 column integer matrix - Draws on the active screen the polygon shape resulting from the application of the current parameters to the graphic matrix and/or fills the indicated area. Each row in the graphic matrix defines a point on

the viewport coordinate system by means of three values: a control code, and the X-Y coordinates of the point. After the drawing has been completed, this function may go into a "graphic input" state, allowing the user to change the graphic interactively.

The control code is an integer in the range 0-31. Each binary bit in this integer has a different meaning. In the following, bit 0 is assumed to be the least significant bit in the integer:

- Bit 0 - 0: move to position (X,Y)  
1: draw to position (X,Y)
- Bit 1 - 0: (X,Y) directly defines next position  
1: (X,Y) is displacement from present position
- Bit 2 - 0: Do not fill the area  
1: Fill the area
- Bit 3 - 0: Draw the border of the area  
1: Do not draw the border of the area
- Bit 4 - 0: Do not go into graphic input  
1: Go into graphic input

When bit 4 is 1 at least in one row of the matrix, interactive graphic input is entered after the drawing has been completed. The cursor is positioned in the coordinates of the last row where bit 4 had a value of 1.

Examples of typical control codes are:

Draw a polygon:

- 0 - Move to X,Y (do not draw)
- 1 - Draw to X,Y

Draw a polygon and fill the area:

- 4 - Move to X,Y (do not draw)
- 5 - Draw to X,Y

Go into interactive input mode:

- 16 - Set the cursor at this position

While in interactive input mode, the user may modify the drawing by means of the keyboard. The following keys have a special function:

- Cursor movement: the four arrows and the corner keys on the numeric keypad move the cursor in the appropriate direction.
- F1-F8: set the cursor movement step size (F1 - smallest, F8 - largest). Fn sets the step size to a value double as the one set by Fn-1.
- F9: set cursor movement mode to constant speed. Speed is changed when the cursor movement keys are pressed.
- F10: set cursor movement mode to constant acceleration. Acceleration is changed when the cursor movement keys are pressed.
- Del key: Fix point (pen-up from previous fixed point).
- Ins key: Fix point (pen-down from previous fixed point).
- Tab key: Move forward from fixed point to fixed point.
- Backspace key: Move backwards from fixed point to fixed point.

All points previously fixed (both in the drawing and with Del/Ins) can be directly accessed in this way.

- Plus key on the numeric keypad: Draw a line between the fixed point where the cursor is located and the previous one.
- Minus key on the numeric keypad: Delete the line between the fixed point where the cursor is located and the previous one.

The two preceding keys only act if the present point has been reached by means of the Tab or Backspace keys.

- ENTER key: in speed and acceleration modes (F9, F10) the cursor stops.

In the step modes (F1-F8) cursor shape toggles from a small cross to cross-hairs.

- Any other keys: control is passed back to APL. The shared variable will return the following value: a matrix containing the final definition of the drawing appearing on the screen, plus an extra line with the following information: key pressed (as  $\square AV$  index, in zero origin) and final X,Y position of the cursor.
- A colour replacement matrix (the colours of all pixels within the current window may be changed with this function). This is a 4 by 2 integer matrix if mode is 4 or 5, and a 2 by 2 integer matrix for mode 6. Rows define colours (0, 1, 2, 3 for modes 4 and 5; 0, 1 for mode 6). A value of  $\bar{1}$  in column 1 indicates that the corresponding colour will be considered a "boundary colour" and will be replaced to the value in column 2 at the same row. If a row does not contain a value of  $\bar{1}$  the corresponding colour will be changed to the value in column 1, if the pixel with that colour is "exterior"; to the value in column 2 if it is "interior".

To find out whether a pixel is "exterior" or "interior", do the following: Consider the horizontal screen line passing through that pixel and limited by the current window. Compute the number of "boundary" pixels (pixels with "boundary colours"). If it is odd, make it even by eliminating the central pixel. One pixel will be "interior" if the number of boundary pixels to its left is odd. It will be "exterior" if it is even.

Examples:

- The following matrix (in mode 6) changes the whole window to reverse video:

$$\begin{array}{cc} \bar{1} & 1 \\ \bar{1} & 0 \end{array}$$

- Assume we have a single empty triangle drawn in colour 3 in the current window (all pixels are background colour, 0, except the triangle border). We want to change the triangle border to colour 1, and fill its interior with colour 2. The following matrix will do it:

$$\begin{matrix} 0 & 2 \\ 1 & 1 \\ -2 & 2 \\ -1 & 1 \end{matrix}$$

## **Return codes**

### **Code Meaning**

- 0 - Success.
- N* - ( $0 < N < 26$ ) Invalid value for the *N*th parameter.
- 26 - No space available or too many WB's.
- 27 - Invalid function.
- 28 - Invalid value.
- 29 - No active screen exists (SB or VSB).
- 30 - Printer error.

## **The File Auxiliary Processor: AP210**

The file auxiliary processor, AP210, is used to read from, or write to, fixed- or variable-length DOS disk files under control of the DOS file system. The reading and writing can be either sequential or random. Up to ten files may be accessed simultaneously; However, DOS must be configured with a sufficient number of file handles to permit this. See the description of the "FILES" parameter of the "CONFIG.SYS" configuration file in the DOS manual.

To use this auxiliary processor, you must include AP210 as a parameter to the APL command at load time, before you begin an APL work session. You may also load it dynamically through AP2. For example,



## APL AP210

Two shared variables are required to process a file: a data variable and a control variable. They can be offered in any order. The name of the data variable must always begin with the letter “D”, and the control variable must begin with the letter “C”. The remaining characters in both names (possibly none) must be the same, because the coupling of both variables is recognised by their name. Examples of valid pairs are: *C* and *D*, *C1* and *D1*, and *CXjj* and *DXjj*. The control variable is used to select the operation to perform and to control each input/output operation. Up to ten pairs of shared variables may be shared with AP210 at any one time.

The following APL lines must be executed before the auxiliary processor can be used:

```
210 □SVO 'Cxx'
210 □SVO 'Dxx'
```

where *xx* is the common part of the names of both variables.

The preceding two instructions must give a result of 1. You then test if the variables have been accepted by AP210 by executing the following:

```
□SVO 'Cxx'
□SVO 'Dxx'
```

Both must give a result of 2. Otherwise, AP210 is not active or has already accepted ten pairs of variable names.

## Control Commands

Once the control variable has been shared, the first value you assign to it should be a *character* vector, which is considered to be a command that describes the file name and specifies the function to be performed. The following commands are accepted:

## IBM Internal Use Only

|                             |                     |
|-----------------------------|---------------------|
| <i>IR,filespec[,code]</i>   | Open for read-only  |
| <i>IW,filespec[,code]</i>   | Open for read/write |
| <i>DL,filespec</i>          | Delete file         |
| <i>RN,filespec,filespec</i> | Rename file         |

where *filespec* is the DOS file identification, of the form:

`[d:][path]filename[.ext]`

*d*: is a letter that identifies the drive (typically A, B, C, etc.).  
*path* is a valid DOS path. *filename* is a valid DOS file name (up to eight characters), and *ext* (the extension of the name) has no more than three characters (see your DOS manual).

Alternatively, *filespec* may be an APL file identification, of the form:

`[libn] filename[.ext]`

where *libn* is an APL library number (see AP101 in this chapter).

*code* is a single letter selecting a given interpretation of the file data. Four different interpretations are supported:

### Code            Interpretation of Data

A (APL)        The records in the file contain APL objects, with their headers. In this way, matrices, vectors, and arrays of any rank may be stored and recovered. Different records of a file may contain objects of different types (for example, characters, integers, or real numbers). An APL object in a record may occupy up to the actual record length (not necessarily the same number of bytes), but the header fills a part of that area. (See Chapter 13, "How to Build an Auxiliary Processor" for the structure and memory requirements of an APL header).

- B (Bool)** The records in the file contain strings of bits without any header (packed eight bits per byte). The equivalent APL object will be a boolean vector. In this case, all records must be equal to the selected record length.
- C (Chars)** The contents of the record is a string of characters in APL internal code, without any header. All records must be equal to the selected record length.
- D (ASCII)** The contents of the record is a string of characters in ASCII code, without any header. All records must be equal to the selected record length.

If the code is not stated specifically, code *A* is the default.

**Warning:** Changing diskettes during an input/output operation, or when you have open files, may destroy data on the diskettes.

The IR command opens the file for read-only operations. If the operation is successful, the control variable passes into the *subcommand state*. You must then specify which data transfer operation you want to perform. (See "Control Subcommands" below). The IW command works in a similar way, but the file is opened for both read and write operations. If the file cannot be opened, the control variable remains in the command state.

When the DL command is received, the file with the specified filespec is erased from the designated drive (or the default if no drive was specified). Then the control variable returns to the command state.

When the RN command is received, the name and extension of the file specified in the first parameter is changed to the name and extension given in the second parameter. The drive specified in the second parameter **MUST** be the same specified in the first parameter. If a different path is given in the second parameter, a move is performed instead of a rename. Rename to a different drive is not allowed. After this command has

been executed, the control variable returns to the command state.

Once a command has been received and executed, a return code is passed back to APL through the control variable, indicating whether or not the command was executed successfully and, if not, the reason for the failure.

| If an IR or IW command executes successfully (giving a return code of 0 in the control variable), the data variable will be set | to the size, in bytes, of the file just opened.

## Control Subcommands

Once a file has been opened for input (command IR) or input/output (command IW), the control variable passes into the subcommand state. It now accepts the assignment of *numeric* vectors specifying the operation to perform, with the following structure:

$$C \leftarrow op[, rn[, rs]]$$

The following are valid operations:

- 0 - Read a fixed length record. Record size is defaulted to 128 unless specified by the *rs* operand or by a previous subcommand. Files are considered as divided in fixed size records. *rn* is the record number. If not given, sequential operation is assumed.
- 1 - Write a fixed length record. Record size is defaulted to 128 unless specified by the *rs* operand or by a previous subcommand. Files are considered as divided in fixed size records. *rn* is the record number. If not given, sequential operation is assumed.
- 2 - Direct read to a file. Record size is defaulted to 128 unless specified by the *rs* operand or by a previous subcommand. Files are considered as continuous strings of data. *rn* is the starting byte. If not given, sequential operation is assumed.

- 3 - Direct write to a file. Record size is defaulted to 128 unless specified by the *rs* operand or by a previous subcommand. Files are considered as continuous strings of data. *rn* is the starting byte. If not given, sequential operation is assumed.
- 4 - Read a variable length record. This command may only be used if the file was opened with codes A or D. *rs* is the scan distance (where, in the case of code D, the CR/LF combination would be expected to reside). If not given, 128 is assumed unless this parameter has been specified by a previous subcommand. *rn* is the record number. If not given, sequential operation is assumed.
- 5 - Write a variable length record. This command may only be used if the file was opened with codes A or D. *rn* is the record number. If not given, sequential operation is assumed. Direct write of variable records is allowed, but should be done with great care. Records should be replaced by others with exactly the same length. If this is not done, the whole file, starting at the replaced record, may be damaged.

| The valid range of values for *rn* for fixed or variable  
 | read/write operations is 0 to 32767. For direct read/write  
 | operations it may be any non-negative integer. *rn* is always  
 | defined in zero origin (i.e., the first record in a file is record 0;  
 | the first byte in a file is byte 0). If not specifically stated, the  
 | first value of *rn* after opening a file is 0 (that is, the first  
 | record or byte position in the file).

The valid range of values for *rs* is 1 to 32512 for fixed read/write operations, 12 to 32512 for variable operations.

Write operations are not allowed if the subcommand state was entered through the IR command.

If the control variable is assigned an empty vector while in the subcommand state, the file is closed and the control variable reverts to the command state.

Once an operation has been requested, the data variable is used as a buffer, where the actual transfer of records takes place. If the operation is a read, the value of the record can be found in the data variable *after* the successful completion of the requested operation (confirmed by the return code passed through the control variable). If the desired operation is a write, the value of the record must be assigned to the data variable *before* the corresponding subcommand is assigned to the control variable.

## **AP210 Return Codes**

### **Code    Meaning**

- 0 - Success
- 1 - Invalid command
- 2 - File not found
- 3 - Path not found
- 4 - No file handles
- 5 - Access denied
- 6 - Invalid file handle
- 8 - Insufficient memory for file control blocks
- 12 - Invalid access code
- 15 - Invalid drive
- 17 - Not the same device
- 18 - No more files
- 19 - Attempt to write on write protected diskette
- 21 - Drive not ready
- 23 - Data error
- 25 - Seek error
- 27 - Sector not found
- 29 - Write fault
- 30 - Read fault
- 31 - General failure
- 26 - No space available
- 29 - Invalid APL object
- 31 - Invalid type in control variable
- 32 - Control variable rank error
- 33 - Invalid length of control variable
- 36 - Invalid file translation code
- 37 - Data variable value error
- 38 - Invalid type in data variable

- ⌘40 - Data variable not shared
- ⌘41 - File is not active, issue a primary command
- ⌘43 - Invalid object
- ⌘44 - CR,LF not found in scan length
- ⌘45 - End of file
- ⌘46 - Incomplete record, padded with nulls
- ⌘47 - Invalid subcommand
- ⌘48 - Could only write a partial record (disk full)
- ⌘49 - Data variable value exceeds record size for fixed or variable replace

*Note:* These return codes are incompatible with those returned by the version of AP210 supplied with APL/PC 1.0.

## Examples of use

```
1 1      210 ⌘SVO 2 2p 'C1D1'
```

Variables C1 and D1 are offered to AP210.

```
2 2      ⌘SVO 2 2p 'C1D1'
```

Variables C1 and D1 have been accepted by AP210.

```
      C1←'IW,B:FILE.EXT'
      C1
0
```

File B:FILE.EXT is created. Records will contain APL objects with header (default code). We are now in subcommand mode.

```
D1←110
```

First record will be a vector of elements from 1 to 10.

```
      C1←1
      C1
0
```

Subcommand to write the first record in the file. Default record number is 0, default record size is 128 bytes.

**IBM Internal Use Only**

*D1*←2 3ρ16

Second record will be a matrix of 2 rows, 3 columns, of elements from 1 to 6.

```
      C1←1
      C1
0
```

Subcommand to write sequentially in the file.

*C1*←' '

An empty vector closes the file and puts the control variable in command mode.

```
      C1←'IR,B:FILE.EXT'
      C1
0
```

Open the same file for read-only operation.

```
      C1←0 1
      C1
0
      D1
  1 2 3
  4 5 6
```

Read the second record first.

```
      C1←0 0
      C1
0
      D1
  1 2 3 4 5 6 7 8 9 10
```

Read now the first record.

*C1*←10

Close the file and go into command state.

```
      C1←'RN,B:FILE.EXT,B:NEWFILE.XXX'
      C1
0
```

Rename the file to NEWFILE.XXX.



```
C1←'DL,B:NEWFILE.XXX'
C1
```

0

Delete the file.

```
□SVR 2 2p'C1D1'
```

2 2

Finally, retract the shared variables.

## The Asynchronous Communications Auxiliary Processor: AP232

The AP232 auxiliary processor can be accessed from APL on the IBM Personal Computer and provides an interface for communications between the IBM Personal Computer and a host (IBM System/370). (See "Asynchronous Communications Adapter" in the *Technical Reference* manual.)

This auxiliary processor is supplied for compatibility with APL/PC 1.0. New applications should use the new extended version, AP232X.

To use this auxiliary processor, you must include AP232 as a parameter to the APL command at load time before you begin an APL work session, or you may load it dynamically through AP2. For example,

```
APL AP232
```

The following APL line must be executed before the auxiliary processor can be used:

```
232 □SVO 'name'
```

where name is the name of the APL variable being shared with the auxiliary processor.

The result of the preceding line will be a 1 if the variable name has been accepted by the auxiliary processor. This processor accepts only one variable.

The following line must be entered next:

```
□SVO 'name'
```

It must give a result of 2. If not, the auxiliary processor is not active or a different variable has been shared with it and has not been retracted.

## **Control Commands**

Once the control variable has been shared, the first value you assign to it must be a character string representing a command which indicates the function that the auxiliary processor has to perform. The functions are the following:

- Initialise (0)
- Transmit (1)
- Receive (2)
- Get port status (3)
- Set break (4)
- Get buffer size (5)

All commands are strings of at least two characters. The first one is a number that indicates the function to be performed (see above). The second is the port address and must always be 1.

If you do not issue a valid command, an error code is returned (see below for return codes).

This auxiliary processor has three buffers:

1. A 1000-byte buffer to communicate with the APL interpreter. If the buffer ever gets full, a code of 2 is returned.
2. A 255-byte buffer to transmit data to the host. The auxiliary processor does not allow it to get full.

3. A 2000-byte buffer to store the data received from the host. (See the command "Receive" below.)

For an example of how to use this auxiliary processor, look at the functions included in the VM232 workspace.

## Initialise (0)

This command is used to initialise the port. It consists of a string of characters of the form:

*CNBPSX*

where:

- *C* indicates the type of the command. It must be 0.
- *N* is the port address (always 1).
- *B* indicates the desired transmission baud rate. It can have one of the following values:

| Value | Baud Rate |
|-------|-----------|
| 0     | 75        |
| 1     | 110       |
| 2     | 150       |
| 3     | 300       |
| 4     | 600       |
| 5     | 1200      |
| 6     | 1800      |
| 7     | 2400      |
| 8     | 4800      |
| 9     | 9600      |

- *P* indicates the parity, as shown in the following table:

| Value | Parity |
|-------|--------|
| 0     | None   |
| 1     | Odd    |
| 2     | Even   |
| 3     | Mark   |

#### 4 Space

- *S* indicates the number of stop bits you want. It can be either 1 or 2.
- *X* indicates the word length in bits. Its value ranges from 5 through 8.

The return code produced by this command is a numeric scalar indicating:

- $\bar{1}$ : success
- 3: error

### **Transmit (1)**

This command consists of a string of characters of the following form:

*CNS*

where:

- *C* indicates the type of the command. It must be 1.
- *N* is the port address. It must be 1.
- *S* represents the string of ASCII characters that is to be sent.

The return code is always the numeric scalar,  $\bar{1}$ .

## Receive (2)

The command consists of a string of characters of the form:

*CNTED*

where:

- *C* indicates the type of the command. It must be 2.
- *N* is the port address. It must be 1.
- *T* represents the turnaround character.
- *E* is the end-of-line character sent by the host.
- *D* represents four delete characters. If you want to give fewer than four delete characters, the remaining positions must be filled by blanks. Blank is never a delete character.

The system returns a string of characters, the first character of which is one of the following:

```

□AV[□IO]      (Success)
□AV[□IO+9]    (Buffer empty, no character read)
□AV[□IO+12]   (Buffer overflow)
□AV[□IO+13]   (Character error in buffer)

```

The rest of the characters returned form the string received from the host.

## Get Port Status (3)

This command returns the content of both the modem status register (MSR) and the line status register (LSR).

The command consists of a string of characters of the form:

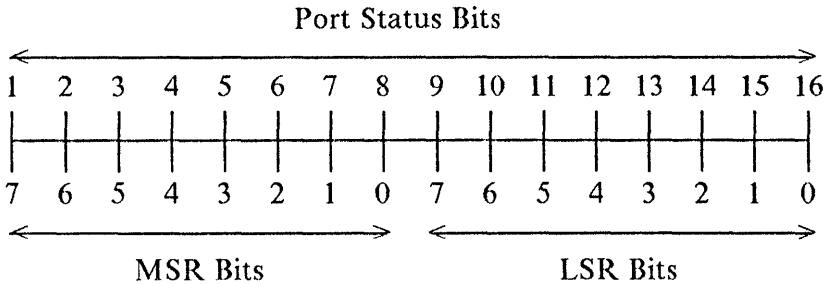
*CN*

where:

## IBM Internal Use Only

- *C* indicates the type of the command. It must be 3.
- *N* is the port address. It must be 1.

The return code is a boolean vector in which bits 1 through 16 (in one origin) represent the content of the MSR, and bits through 16, the content of the LSR, as shown by the following:



### Set Break (4)

The Set Break command sends a break to put the host in the receive state.

The command consists of a string of characters of the form:

*CN*

where:

- *C* indicates the type of the command. It must be 4.
- *N* is the port address. It must be 1.

The return code is always the numeric scalar,  $\bar{1}$ .

## Get Buffer Size (5)

This command is used to ask for the size of contents of the buffer that is currently occupied (either transmit or receive buffer).

The syntax of the command is:

*CNO*

where:

- *C* indicates the operation. It must be 5.
- *N* is the port address (must be 1).
- *O* is the operational type of the buffer (“R” for the receive buffer, “W” for the transmit buffer).

This command returns a two-element numeric vector, in which the first element is one of the following codes:

- $\bar{1}$ : Success
- 10: Buffer more than three-quarters full
- 11: Buffer overflow

The second element is the number of bytes occupied by the contents of the buffer.

# The Extended Asynchronous Communications Auxiliary Processor: AP232X

This Auxiliary Processor allows you to control one or two asynchronous communication ports on your IBM PC from an APL defined function.

In addition it allows your application to:

- Define transmit and receive buffer sizes.
- Set the Line Status Register.
- Set the Modem Status Register.
- Control the retry count on specific lines.
- Set the Request-To-Send line on or off at initialisation.
- Control the translation of input or output.
- Specify end-of-line and turn-around characters.
- Specify up to 4 characters to be removed from data stream.

AP232X requires two shared variables: a data variable (whose name must start with a “*D*”) and a control/command variable (whose name must start with a “*C*”). These variables may be shared in any order, but only one pair can be shared with AP232X at any time. If, on testing the status of the share, a “1” is given as the degree of coupling, the following possibilities should be investigated:

1. AP232X has not been loaded.
2. Another pair of variables are active with it.
3. No asynchronous communication adapters are installed in this PC.



## Hardware Notes

1. The jumper module on the COM2 card should be turned 180 degrees and replugged. (See the PC Technical Reference Manual for details).
2. It should also be noted that ROM BIOS does not verify any card configured as COM2 (2Fn). Therefore the only certain way to verify that the COM2 is O.K., is to:
  - a. Remove COM1 card completely.
  - b. Remove COM2 card and reconfigure it as COM1.
  - c. Replace second card in its normal slot.
  - d. Power on the system.
  - e. If power-on tests run O.K., then the card is good.
3. The wiring of cables and connectors used to attach devices to RS232 ports should be carefully checked. Appropriate connection data is usually provided in the documentation supplied by the manufacturer of the device to be attached.

## AP232X Operation

### Initialise a Port

In this and the following sections, the two shared variables are represented by the names *C* and *D*, respectively for the control and data variables.

This information required below should be obtained from the documentation supplied with the device to be attached.

Syntax:

*C* ← 0 [ ,Port ]

*D* ← *Integer vector parameter list*:

**D[1]:** Baud Rate of Port.

The following rates are supported: 50, 75, 110, 135, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600 or 19200. A rate setting of 135 will give a true rate of 134.5 baud.

**D[2]:** Number of Bits per byte.

This can be one of the following: 5, 6, 7 or 8. A maximum of 7 bits per byte may be used if a stop bit is to be included.

**D[3]:** Stop bit(s) required.

- 0 - No (1 bit per byte regardless of size).
- 1 - Yes (if 5 bits per byte, then generate 1.5, otherwise generate 2).

**D[4]:** Parity type.

- 0 - None
- 1 - Odd
- 2 - Even
- 3 - Mark
- 4 - Space

**D[5]:** Raise Request-to-send line.

- 0 - No
- 1 - Yes

**D[6]:** Add Line Feed Character if New Line detected on output.

- 0 - No
- 1 - Yes

This feature is of particular value if the device is a printer.

**D[7]:** Translate input using ASCII to APL translation table.

- 0 - No translation.
- 1 - Translation takes place.

*D*[8]: Translate output using APL to ASCII translation table.

- 0 - No translation.
- 1 - Translation takes place.

Parameters 7 and 8 give the flexibility of using your own translate table in the workspace, permitting customised translation tables for special devices.

*D*[9]: Clear-to-send line retry count.

*D*[10]: Dataset ready line retry count.

*D*[11]: Carrier detect retry count.

These are the retry counts, to be used when initialising the adapter, indicating whether a potential problem exists with the device attached to them. A value of zero ignores the line. -1 is the maximum number of retries.

*D*[12]: Turn around character.

This is the character used to delimit records in the receive buffer. A value of -1 will cause no turn around character to be used.

*D*[13]: End-of-line character. A value of -1 will cause no end-of-line character to be used.

*D*[14 15 16 17]: Characters to be deleted from datastream.

Values of -1 will cause no characters to be removed.

All of the above characters are specified as indices of  $\square AV$  in zero origin (0-255).

*D*[18]: Receive buffer size.

## IBM Internal Use Only

The receive buffer size has a range of 1K to 32K bytes.

*D[19]*: Transmit buffer size.

The transmit buffer size has a range of 255 to 32K bytes.

Buffer sizes should be selected according to the baud rate and characteristics of the device. For example, a plotter will have a great deal of data sent to it, so it should have a large transmit buffer, but will only need a small receive buffer.

The values in this parameter list should be chosen to agree with those suggested by the manufacturer of the device to be attached.

## Recover Port Initialisation Parameters

Syntax:

*C* ← 1 [ ,*Port*]

The data variable returns a seventeen element vector as specified in the preceding section.

## Transmit Data to Port

Syntax:

*C* ← 2 [ ,*Port*]  
*D* ← *Characters to transmit*

## Receive Data from a Port

Syntax:

*C* ← 3 [ ,*Port*]

The data variable will contain either all characters currently in the buffer or up to the first turnaround character encountered.

## Get Status

Syntax:

$$C \leftarrow 4, Port, 1$$

The data variable returns transmit and receive buffer sizes and number of characters in each as a 4 element integer vector.

$$C \leftarrow 4, Port, 2$$

The data variable returns a four by eight boolean array which is a direct copy of the LCR, MCR, LSR, and MSR registers, in row order.

## Set LCR / MCR Registers

Syntax:

$$C \leftarrow 5, Port, n$$

$D \leftarrow 8$  element boolean vector representing new register value

where n is:

- 1 = Line control register (LCR).
- 2 = Modem control register (MCR).

## Reset the Transmit and Receive Buffers

Syntax:

$$C \leftarrow 6 [, Port]$$

Buffers are frozen and purged.

## AP232X Return Codes

### Code Meaning

- 0 - Success
- 1 - Receive buffer empty
- 2 - Invalid port specified
- 10 - *CtI* has invalid Length
- 11 - *CtI* has invalid Type
- 12 - *CtI* has invalid Rank
- 14 - Invalid function request
- 21 - *Dat* has invalid Rank
- 22 - *Dat* has invalid Length
- 23 - *Dat* has invalid Type
- 24 - *Dat* not shared
- 25 - *Dat* value error / locked
- 26 - *Dat* too large
- 30 - Invalid baud rate
- 31 - Invalid number of bits per byte
- 32 - Invalid Stop bits
- 33 - Invalid Parity code
- 34 - Invalid RTS flag
- 35 - Invalid Linefeed add flag
- 36 - Invalid translation to ASCII flag
- 37 - Invalid translation to APL flag
- 38 - Invalid CTS retry count
- 39 - Invalid RTS retry count
- 40 - Invalid CDD retry count
- 41 - Invalid turnaround character
- 42 - Invalid EOL character
- 43 - Invalid delete char no. 1
- 44 - Invalid delete char no. 2
- 45 - Invalid delete char no. 3
- 46 - Invalid delete char no. 4
- 50 - Time-out on CTS line
- 51 - Time-out on DSR line
- 52 - Time-out on RLSD line
- 53 - Invalid receive buffer size
- 54 - Invalid transmit buffer size

## The Music Auxiliary Processor: AP440

The AP440 auxiliary processor provides an easy way to create music at the attached speaker. To use this auxiliary processor, you should have an elementary knowledge of music and its notation.

To use this auxiliary processor, you must include AP440 as a parameter to the APL command at load time, before you begin an APL work session, or load it dynamically through AP2. For example,

```
APL AP440
```

The APL line:

```
440 □SVO 'name'
```

must be executed before the auxiliary processor can be used. *name* is the name of any APL variable.

The result of the preceding line will be a 1, if the variable name is accepted. This auxiliary processor accepts only one variable.

The line:

```
□SVO 'name'
```

must be executed next and must give a result of 2. If not, the auxiliary processor is not active or a different variable has been shared with it and has not been retracted.

Any character string assigned to name will be interpreted as a set of commands to the auxiliary processor to play music. Commands may be joined within a single character string in any way you desire, or passed through another variable which is then assigned to the shared variable.

## AP440 Command Syntax

{ [tempo] [octave] [mode] [length] [NOTESPEC] [pause] }

where:

[tempo]: T<sub>n</sub> (n = 0 to 6; default 4)

[octave]: On[ {+ -} ] (n = 0 to 6; default 3)

[mode]: Mn (n = 0 to 2; default 0)

[length]: Ln (n = 0 to 6; default 0)

[NOTESPEC]: tone[ {# + -} ] [n] [.] (tone = A to G; n = 0 to 6, default 0)

[pause] P[n] [.] (n = 0 to 6; default 0)

NOTESPEC A to G, optionally followed by #, +, or -, and a digit (0 to 6), optionally followed by a period.

Plays the indicated note in the current octave. # or + specifies a sharp, and - specifies a flat. The digit, if given, specifies the length of the note, according to the following:

- 0 complete note
- 1 half note
- 2 quarter note
- 3 quaver note
- 4 semiquaver note
- 5 quarter quaver note
- 6 half-quarter quaver note

If a period is given, the note is played as a dotted note; that is, its length is multiplied by 3/2. Additional dots are ignored, if present.



- length**       $L_n$ , where  $n$  is a digit from 0 to 6, sets a given length (according to the previous table) applied to all later notes in this or different strings of commands, unless a new  $L_n$  command is found or a note has its own length given, which takes priority. If no  $L_n$  command has ever been given,  $L_0$  is assumed as the default.
- mode**         $M_n$ , where  $n$  is a digit from 0 to 2, selects the music mode, according to the following table:
- 0 Music staccato. Each note will play  $3/4$  of the length. The rest will be a pause.
  - 1 Music normal. Each note will play  $7/8$  of its length.
  - 2 Music legato. Each note will play its full length.
- If no  $M_n$  command has ever been given,  $M_1$  is assumed.
- octave**       $O_n$ , where  $n$  is a digit from 0 to 6, optionally followed by a + or - sets the current octave. Each octave goes from  $C^-$  to  $B^+$ . Octave 3 contains middle A (440 Hertz). If + or - is not present, the number given is the absolute octave. A + sign specifies a relative displacement to higher octaves. A - sign corresponds to a relative displacement to lower octaves. If no  $O_n$  command has ever been given,  $O_3$  is assumed.
- pause**         $P$ , optionally followed by a digit from 0 to 6, optionally followed by a period, defines a pause or rest. The digit, if given, specifies the length of the pause. This length may be enlarged to  $3/2$  its value if a period follows. The length values are interpreted according to the same table indicated in the note-definition command.

tempo         $T_n$ , where  $n$  is a digit from 0 to 6, sets the tempo of the play, according to the following table, where the number of quarter notes per minute equivalent to each tempo is indicated in parentheses:

- 0 Largo (54)
- 1 Largetto (66)
- 2 Adagio (78)
- 3 Andante (96)
- 4 Moderato (120)
- 5 Allegro (156)
- 6 Presto (198 per minute)

If no  $T_n$  command has ever been given,  $T_4$  is assumed.

To play tied notes, connect the expressions of the two notes. You can also assign sub-tunes to any APL variable (not shared with AP440) and call them repetitively with different tempos, octaves, or lengths, by assigning that variable to the shared variable.

For an example of how to use this auxiliary processor, examine the variables included in the MUSIC workspace.

## **The IBM GPIB Support Auxiliary Processor: AP488**

The IBM General Purpose Interface Bus Adapter provides an interface between an IBM Personal Computer and the IEEE-488 General Purpose Interface Bus (GPIB), allowing control of multiple devices or instruments (such as plotters, multimeters, and disk drives).

Auxiliary processor 488 provides an interface between APL and the IBM General Purpose Interface Bus (GPIB) Adapter Programming Support.

|                                      | <u>Part No</u> | <u>Feature</u> |
|--------------------------------------|----------------|----------------|
| <b>General Purpose Interface Bus</b> | <b>6451503</b> | <b>1503</b>    |
| <b>Programming Support</b>           | <b>6024201</b> | <b>4201</b>    |

Each Personal Computer can accommodate up to four GPIB Adapters and provide support for up to 48 devices. The IBM GPIB Adapter can perform as a controller, a talker, or a listener with these compatible devices. The IBM GPIB Adapter also provides capabilities for data transfer between workstations, and the connection of several computers for sharing of instruments or peripheral I/O devices.

## Description of AP488 Functions

Thirty five functions are defined to auxiliary processor 488, numbered 0 through 34. Function calls 21, 29, 31, and 33 are reserved for future use. Each numeric function code has a matching APL function in the AP488 workspace, as follows:

|    |        |   |
|----|--------|---|
| 0  | IBWAIT | Wait for Selected Event                 |
| 1  | IBONL  | Online or Offline                       |
| 2  | IBRSC  | Request or Release System Control       |
| 3  | IBSIC  | Send Interface Clear                    |
| 4  | IBSRE  | Set or Clear Remote Enable Line         |
| 5  | IBLOC  | Go to Local                             |
| 6  | IBRSV  | Request Service                         |
| 7  | IBPPC  | Parallel Poll Configure                 |
| 8  | IBPAD  | Change Primary Address                  |
| 9  | IBSAD  | Change or Disable Secondary Address     |
| 10 | IBIST  | Individual Status Bit                   |
| 11 | IBDMA  | Enable or Disable DMA                   |
| 12 | IBEOS  | Change or Disable EOS Method            |
| 13 | IBTMO  | Change or Disable Timeout Limit         |
| 14 | IBEOT  | Enable or Disable END Message           |
| 15 | IBGTS  | Active Controller Go To Standby         |
| 16 | IBCAC  | Become Active Controller                |
| 17 | IBRDF  | Read Data Into File                     |
| 18 | IBFIND | Open Device Adapter File Handle         |
| 19 | IBRPP  | Conduct Parallel Poll                   |
| 20 | IBSTAT | Return IBSTA, IBERR, IBCNT              |
| 21 |        | Reserved                                |
| 22 | IBCLR  | Clear Device with Selected Device Clear |

## IBM Internal Use Only

|    |        |                      |
|----|--------|----------------------|
| 23 | IBTRG  | Trigger Device       |
| 24 | IBPCT  | Pass Control         |
| 25 | IBRSP  | Conduct Serial Poll  |
| 26 | IBBNA  | Change Adapter Name  |
| 27 | IBSIZE | Set Data Buffer Size |
| 28 | IBRD   | Read Data            |
| 29 |        | Reserved             |
| 30 | IBWRT  | Write Data           |
| 31 |        | Reserved             |
| 32 | IBCMD  | Send Commands        |
| 33 |        | Reserved             |
| 34 | IBWRTF | Write Data From File |

The general format of processor calls is:

```
[D488←optional data]
C488←function code,handle,[optional parm]
VALUE←C488
[DATA←D488]
```

All functions return a value which is unlikely to be zero. In most cases, the integer equivalent of *IBSTA* (the IEEE-488 sixteen bit integer status word) is returned. Two functions (19 - *IBRPP*, and 25 - *IBRSP*) return integer poll responses if no error has occurred.

*C488* accepts only integer values and *D488* accepts only character vectors. This is true for all functions.

## IBSTA, Status Word Layout

The status variable is a sixteen bit integer variable. All sixteen bits mean something:

| <u>Bit</u> | <u>Value</u>     | <u>Meaning</u>                        |
|------------|------------------|---------------------------------------|
| 0          | 1000000000000000 | - ERR GPIB Error                      |
| 1          | 0100000000000000 | - TIMO Time Limit Exceeded            |
| 2          | 0010000000000000 | - END Device Detected END or EOS      |
| 3          | 0001000000000000 | - SRQI SRQ Detected                   |
| 4          | 0000100000000000 | - RQS Device Requires Service         |
| 5          | 0000010000000000 | - Reserved                            |
| 6          | 0000001000000000 | - Reserved                            |
| 7          | 0000000100000000 | - CMPL I/O Completed                  |
| 8          | 0000000010000000 | - LOK Device is in Lockout State      |
| 9          | 0000000001000000 | - REM Device is in Remote State       |
| 10         | 0000000000100000 | - CIC Device is Controller-In-Charge  |
| 11         | 0000000000010000 | - ATN Attention is asserted           |
| 12         | 0000000000001000 | - TACS Device is Talker               |
| 13         | 0000000000000100 | - LACS Device is Listener             |
| 14         | 0000000000000010 | - DTAS Device in Device Trigger state |
| 15         | 0000000000000001 | - DCAS Device in Device Clear state   |

When ERR is true, the sixteen bit integer status word is returned to APL as a negative value. It should be clear that any positive return code is good. A negative value means that ERR is true, an error or abnormal condition has occurred:

- IBSTA bits are set
- IBERR numeric return code is available thru function call 20 (IBSTAT)

### **IBERR, Error number**

- 0 Driver or DOS error
- 1 Function requires GPIB adaptor to be Controller-In-Charge
- 2 Write function detected no listeners
- 3 Interface adapter not addressed correctly
- 4 Invalid argument to function call
- 5 Function requires GPIB adaptor to be System Active Controller
- 6 I/O operation aborted
- 7 Nonexistent interface adapter
- 8 Reserved
- 9 Reserved
- 10 Asynchronous operation not complete
- 11 No capability for operation
- 12 Unable to access file

- 13 Reserved
- 14 Bus command error during device call
- 15 Serial Poll status byte lost
- 16 SRQ remains asserted

## **Notation**

In the following descriptions:

**DEVICE** refers to a device connected to the GPIB

**ADAPTER** refers to the GPIB Adapter board

**EITHER** refers to either an adapter or a device

## **Function 0, Wait for Selected Event (IBWAIT)**

$C488 \leftarrow 0, EITHER, MASK$   
 $IBSTA \leftarrow C488$

This function causes the system to wait for any of the events specified in the mask integer. The mask layout is exactly the same as that of the *IBSTA* variable.

## **Function 1, Online or Offline (IBONL)**

$C488 \leftarrow 1, EITHER, FLAG$   
 $IBSTA \leftarrow C488$

This function is the inverse of *IBFIND*. If *FLAG* is zero, it closes the file handle. If *FLAG* is not zero, it does nothing (except waste time).

## **Function 2, Request or Release System Control (IBRSC)**

$C488 \leftarrow 2, ADAPTER, FLAG$   
 $IBSTA \leftarrow C488$

This function enables or disables System Controller functions (Remote Enable, Interface Clear). The IEEE-488 specification does not specifically permit schemes where System Control may be passed back and forth between instruments, but it does not forbid them either. This routine would be used in such a scheme.

### Function 3, Send Interface Clear (IBSIC)

*C488←3, ADAPTER*  
*IBSTA←C488*

This function causes the adapter board to assert the Interface Clear signal if the adapter is currently supporting System Controller functions.

### Function 4, Set or Clear Remote Enable Line (IBSRE)

*C488←4, ADAPTER, FLAG*  
*IBSTA←C488*

This routine asserts the Remote Enable signal if *FLAG* is non-zero, else it unasserts it.

### Function 5, Go to Local (IBLOC)

*C488←5, EITHER*  
*IBSTA←C488*

This routine temporarily removes "Local Lockout" from the specified instrument. Normally, this will re-enable front panel controls. The next time the instrument is accessed via the interface, "Local Lockout" will resume. There is no way to turn off this condition permanently (except by changing the option in *CONFIG.SYS*).

## Function 6, Request Service (IBRSV)

*C488←6, ADAPTER, STATUS*  
*IBSTA←C488*

This function is used when the adapter is configured as an instrument, not the system controller. It sets the byte that is returned when the system controller performs a serial poll. If *STATUS* has the X'40' bit on, this routine will also assert *SRQ*.

## Function 7, Parallel Poll Configure (IBPPC)

*C488←7, EITHER, CONFIG*  
*IBSTA←C488*

This function configures an instrument or an adapter to respond to a parallel poll. Be certain that the device to be configured has a *GPIB* address in the range of zero through seven. For information on the *CONFIG* integer, see the "Guide to the General Purpose Interface Bus Adapter Programming Support".

## Function 8, Change Primary Address (IBPAD)

*C488←8, EITHER, ADDRESS*  
*IBSTA←C488*

This routine permits you to over-ride the primary address of the device or adapter that was specified by *IBCONF*. This change remains in effect until *IBONL* is called, or your program ends.

## Function 9, Change or Disable Secondary Address (IBSAD)

*C488←9, EITHER, ADDRESS*  
*IBSTA←C488*

This routine permits you to override the secondary address of the device or adapter that was specified by *IBCONF*. This



change remains in effect until *IBONL* is called, or your program ends.

## Function 10, Individual Status Bit (IBIST)

*C488←10, ADAPTER, FLAG*  
*IBSTA←C488*

This routine sets the parallel poll response bit to true if *FLAG* is not zero, else the routine sets the response bit to false. The adapter must have been previously configured to respond to a parallel poll.

## Function 11, Enable / Disable DMA (IBDMA)

*C488←11, ADAPTER, FLAG*  
*IBSTA←C488*

This routine permits you to temporarily disable and re-enable DMA transfers. If *FLAG* is zero, DMA is disabled and the adapter will use programmed I/O exclusively. If *FLAG* is not zero, then the adapter will use DMA.

*Note:* DMA may only be used if you configured your system to use DMA via the *IBCONF* utility supplied by the GPIB Programming Support.

## Function 12, Change or Disable EOS Method (IBEOS)

*C488←12, EITHER, FLAGWORD*  
*IBSTA←C488*

This function changes or disables the End Of String method. *FLAGWORD* specifies both the EOS character, and what to do when it is detected during a read or write. For further information on the *FLAGWORD* integer, see the "Guide to the General Purpose Interface Bus Adapter Programming Support".

### Function 13, Change or Disable Timeout Limit (IBTMO)

*C488←13, EITHER, FLAGWORD*  
*IBSTA←C488*

This function changes the amount of time that the interface will wait before reporting a timeout error. Limits range from 10 microseconds through one thousand seconds, or forever. *FLAGWORD* ranges from zero through seventeen and is described in the "Guide to the General Purpose Interface Bus Adapter Programming Support".

### Function 14, Enable or Disable END Message (IBEOT)

*C488←14, EITHER, FLAG*  
*IBSTA←C488*

This function specifies whether or not *EOI* is set concurrently with the last byte of the data. If *FLAG* is zero, then *EOI* is not sent. If *FLAG* is non-zero, then *EOI* is sent. If the handle refers to an adapter, then the value specified by *FLAG* overrides the specification on all devices attached to the adapter card.

### Function 15, Active Controller Go To Standby (IBGTS)

*C488←15, ADAPTER, FLAG*  
*IBSTA←C488*

This routine is very useful for transferring data between two instruments without bothering to read it into APL first. It is normally used in conjunction with *IBCMD*. The function unasserts the *ATN* line and goes to the standby state. If *FLAG* is non-zero, then the adapter monitors the data transfer and goes into a "Not Ready For Data" state when the *END* message is detected. This permits synchronous resumption of

control via IBCAC. If *FLAG* is zero, then no monitoring is performed.

## Function 16, Become Active Controller (IBCAC)

```
C488←16,ADAPTER,FLAG
IBSTA←C488
```

This function is used to resume control of the GPIB system. If *FLAG* is zero, then control is forced immediately (and possibly asynchronously with respect to data transfer). If *FLAG* is non-zero, then control is resumed synchronously with respect to data transfer.

## Function 17, Read Data Into File (IBRDF)

```
D488←'FULL DOS FILE SPECIFICATION'
(Including Path)
C488←17,EITHER
IBSTA←C488
```

This function reads data from the GPIB and writes it to a DOS file. Any data already in the file is over-written. The transfer will end when either the *END* or the *EOS* message is detected.

## Function 18, Open Device or Adapter File Handle (IBFIND)

```
D488←'UNIT NAME'
C488←18,0
RESULT←C488
```

This function performs an “open” on the specified device. You supply the name of the instrument or adapter board to *D488*, and the function returns a file handle or *IBSTA* if an error occurs. The result will always be a file handle if the integer that is returned is positive, and will always be *IBSTA* if the integer is negative.

## Function 19, Conduct Parallel Poll (IBRPP)

$C488 \leftarrow 19, EITHER, 0$   
 $RESULT \leftarrow C488$

This function causes the adapter to perform a parallel poll. The value returned is either the response byte from the poll (if the value is positive) or *IBSTA* if the value is negative (an error was detected). The *EITHER* may be either an adapter or a device. If the handle refers to a device, the software will actually perform a parallel poll on the adapter board that owns the device.

## Function 20, Return IBSTA, IBERR, IBCNT (IBSTAT)

$C488 \leftarrow 20$   
 $RESULT \leftarrow C488$   
 $IBSTA \leftarrow RESULT[1] + 0$   
 $IBERR \leftarrow RESULT[2]$   
 $IBCNT \leftarrow RESULT[3]$

This function is used to retrieve the current values of *IBSTA*, *IBERR*, and *IBCNT*. No actual GPIB activity results, this function only reads three integers from the auxiliary processor.

## Function 22, Clear Device with Selected Device Clear (IBCLR)

$C488 \leftarrow 22, DEVICE$   
 $IBSTA \leftarrow C488$

This function clears (or is supposed to clear) the internal device dependant functions of the specified instrument. The routine actually sends "Selected Device Clear" to the device. Not all instruments support the "SDC" message.

## Function 23, Trigger Device (IBTRG)

$C488 \leftarrow 23, DEVICE$   
 $IBSTA \leftarrow C488$

This function sends the *GET* (group execute trigger) message to the device specified by *EITHER*. Many devices do not support this function.

### Function 24, Pass Control (IBPCT)

*C488*←24,*DEVICE*  
*IBSTA*←*C488*

This function passes “Controller-In-Charge” authority to the specified device. Be *CERTAIN* that the device specified can support controller functions.

### Function 25, Conduct Serial Poll (IBRSP)

*C488*←25,*DEVICE*,0  
*RESULT*←*C488*

This function performs a serial poll of the specified device and returns the resultant status byte (if no error occurred), or else it returns *IBSTA*. If the returned integer is positive, the response is a status byte. If the returned integer is negative, then the response is *IBSTA*.

### Function 26, Change Adapter Name (IBBNA)

*D488*←'GPIB '  
*C488*←26,*DEVICE*  
*IBSTA*←*C488*

This function changes the adapter (GPIB0-GPIB3) used to access the instrument specified by *DEVICE*. *D488* is assigned a five character name that consists of “GPIB” followed by the character zero through three.

### Function 27, Set Data Buffer Size (IBSIZE)

*C488*←27,*BUFSIZE*  
*IBSTA*←*C488*

## IBM Internal Use Only

This function sets the maximum read buffer size in the auxiliary processor. No I/O is performed.

### Function 28, Read Data (IBRD)

$C488 \leftarrow 28, EITHER$   
 $IBSTA \leftarrow C488$   
 $DATA \leftarrow D488$

This function reads data from the specified device or adapter and returns it to APL through *D488*. If the high order bit (X'8000') of the *EITHER* is true, then no translation of the data from ASCII to Z-CODE (the internal APL character set) is performed. The *BINARY* function is provided to set this bit, and the *ASCII* function to reset it.

### Function 30, Write Data (IBWRT)

$D488 \leftarrow DATA$   
 $C488 \leftarrow 30, EITHER$   
 $IBSTA \leftarrow C488$

This function writes data from a character vector to the instrument that is specified by *EITHER*. If the high order bit (X'8000') of the *EITHER* is true, then no translation of the data from ASCII to Z-CODE (the internal APL character set) is performed. The *BINARY* function is provided to set this bit, and the *ASCII* function to reset it.

### Function 32, Send Commands (IBCMD)

$D488 \leftarrow COMMANDS$   
 $C488 \leftarrow 32, ADAPTER$   
 $IBSTA \leftarrow C488$

This function sends actual GPIB commands out through the adapter. You may send any valid sequence of IEEE-488 commands. For more information on *IBCMD*, see the "Guide to the General Purpose Interface Bus Adapter Programming Support".

## Function 34, Write Data From File (IBWRTF)

*D488* ← 'FULL DOS FILE SPECIFICATION'  
(Including Path)  
*C488* ← 34, EITHER  
*IBSTA* ← *C488*

This function reads data from a DOS disk file, and sends it to the specified device (or adapter) as one long record. No translation is done.

## Chapter 13. How to Build an Auxiliary Processor

|   |       |
|---|-------|
| Access Control .....                                | 13-4  |
| Format of Shared Data .....                         | 13-5  |
| Shared Variable Processor Services and Return Codes | 13-7  |
| Processor Sign-on: 00H .....                        | 13-8  |
| Return to APL via Shared Variable Processor:        |       |
| 01H .....   | 13-9  |
| Share or Query the State of a Variable: 02H ...     | 13-10 |
| Get the Present Value of a Shared Variable: 03H     | 13-11 |
| Get a Block of Memory From the Workspace:           |       |
| 04H .....   | 13-12 |
| Release Storage to the Workspace: 05H .....         | 13-13 |
| Pass a Variable to APL and Release the Space:       |       |
| 06H .....   | 13-14 |
| Pass a Scalar Integer Return Code to APL: 07H       | 13-15 |
| Convert an APL Object from Type Boolean to          |       |
| Integer: 08H .....                                  | 13-16 |
| Convert from APL Z-code to ASCII: 09H ...           | 13-17 |
| Convert from ASCII to APL Z-code: 0AH ...           | 13-18 |
| Share or Query the State of a Variable: 0BH ..      | 13-19 |
| Pre-read a Variable: 0CH .....                      | 13-20 |
| Read a Previously Pre-read Variable: 0DH ...        | 13-21 |
| Release a Previously Pre-read Variable: 0EH ...     | 13-22 |
| Pass a Value to a Variable: 0FH .....               | 13-23 |
| Processor Sign-off: 10H .....                       | 13-24 |
| SVP Reserved Function: 11H .....                    | 13-24 |
| Locate an Associated Variable: 12H .....            | 13-25 |
| Change the Keyboard / Screen Mode: 13H ...          | 13-26 |
| Get Loop Count for Delay: 14H .....                 | 13-27 |
| Change the Keyboard / Screen Mode Without           |       |
| Clearing Screen: 15H .....                          | 13-28 |
| Notes .....   | 13-29 |
| Return Codes (Returned in CX Register) .....        | 13-30 |
| Sample Auxiliary Processors .....                   | 13-30 |
| APL Interrupt Usage .....                           | 13-31 |



|   |       |
|---|-------|
| How to Debug Auxiliary Processors ..... | 13-32 |
| Exchange Assembly Programs .....        | 13-33 |

**Notes:**

The APL/Personal Computer System, Version 2.1, includes a wealth of auxiliary processors implementing interfaces to most of the devices now available for the IBM Personal Computer. However, you may wish to further extend the facilities of your APL system by writing your own, user-tailored auxiliary processors.

To build your own auxiliary processors, you must have a good understanding of APL, APL data types, assembler language, and the information in this chapter. You will need the IBM Personal Computer Macro Assembler if you desire to build your own auxiliary processors.

Essentially, an auxiliary processor (AP) provides a service that involves exchange of data. One obvious service is accessing a file. However, the services that an AP can provide are limited only by the facilities available in the system and the imagination of the designer.

Auxiliary processors exchange information with the APL processor through shared variables. A variable becomes shared when you offer to share it and the auxiliary processor accepts the offer. You and the AP, in effect, then become partners. Each partner can assign a value to the shared variable (specify it) and get its latest value (reference it).

The *shared variable processor* (SVP) is a part of the APL processor and manages all shared-variable offers and information exchange. This processor is loaded in main memory only if at least one auxiliary processor using its services (the name of which begins with "AP") has been selected at APL load time.

## Access Control

It is often necessary for the partners to control the sequence in which they access a shared variable. If the access is not controlled, one partner can specify a variable twice before the other can reference the first value, or one partner can reference a variable twice before the other can specify a second value.

Each shared variable is associated with a 4-bit control vector that provides a means of regulating access to the variable. Each partner presents its own version of the access-control vector to the SVP. The effective, or combined, access-control vector is the logical OR of the two. Thus, each partner can impose more discipline, but neither can reduce the discipline imposed by the other.

The meaning of each of the four bits, as given by an auxiliary processor to the SVP, is:

### Bit Meaning

- 0 If 1, disallow my successive specification until my partner has accessed the variable (either referenced or specified it).
- 1 If 1, disallow my partner's successive specification until I have accessed the variable.
- 2 If 1, disallow my successive reference until my partner has specified the variable.
- 3 If 1, disallow my partner's successive reference until I have specified the variable.

The SVP allows or disallows each access according to the variable's access state. The access state at any point in time depends on the variable's combined access-control vector and the prior accesses by each partner.

## Format of Shared Data

APL data on the IBM Personal Computer has a special internal format. Data passed from APL to the auxiliary processor, and data passed back to APL, must be in that same format. The maximum size of an APL object that may be passed to or from an auxiliary processor is 32512 bytes. If you pass invalid data to APL, unpredictable errors may occur.

Each variable contains information that describes its data type, shape, and size. This information is called its header, and is located at the beginning of the object. It consists of the following fields:

```

$PTR EQU WORD PTR [0] ; A pointer. Should be
                        ; ignored by the A.P.
$NB EQU WORD PTR [2] ; Number of bytes in
                        ; this APL object. It
                        ; MUST be rounded up
                        ; to EVEN.
$NELM EQU WORD PTR [4] ; Number of elements in
                        ; this APL object.
$TYPE EQU BYTE PTR [6] ; APL object type:
                        ; 0=Logical, 1=Integer,
                        ; 2=Real, 3=Character
$RANK EQU BYTE PTR [7] ; Rank of object (0-63).
$DIM1 EQU WORD PTR [8] ; First dimension (if any).
                        ; As many dimensions as
                        ; value of $RANK follow.
    
```

A scalar has a rank of 0 and is a variable with no dimension. It has only one element and contains no size information.

A variable with rank greater than 1 includes size information: as many dimensions as the value of its rank. Each dimension must be a two-byte integer, with a value in the range 0 to 32767. The maximum rank of a variable is 63.

An APL variable may be one of four types:

- real (floating point)
- integer
- logical (boolean)
- literal (character)

When you receive numeric data from APL, you should be prepared to accept the data in any representation and to convert between different representations.

Regardless of a variable's data type, its elements are located immediately after the header, and the whole object is padded up to an even number of bytes. If the variable has more than one dimension, its elements are stored in row order (as if the APL primitive `ravel` had been applied to the variable).

Elements of a logical variable are represented as logical values (0 or 1), with one bit per element. The bytes of a logical variable, and the bits within the byte, are in row order. The word containing the last element can have undefined elements on the right. For example, the elements of a 19-element logical variable are stored in four bytes (two words) in the sequence shown below. Unused elements of the fourth byte are undefined.

0 1 2 3 4 5 6 7    0 1 2 3 4 5 6 7    0 1 2 x x x x x

Elements of an integer variable are represented as binary numbers, with two bytes (one word) per element. Actual values must belong to the interval  $[-32767, 32767]$

Elements of a real variable are represented in long floating-point format, with eight bytes per element.

Elements of a character variable are represented in APL internal code, with one byte per element in row order. The word containing the last element can have one undefined byte on the right. An SVP service function has been provided to translate character data in APL internal form to ASCII and vice versa.

## Shared Variable Processor Services and Return Codes

The IBM Personal Computer APL system Version 1.0 includes a set of SVP services and macros that could be used to build auxiliary processors. Although this interface is still supported by the IBM APL/Personal Computer system Version 2.1, for compatibility reasons, it will not be described here, since a new, easier, and more powerful interface between AP's and SVP has been incorporated to the system. This interface, to be described below, is recommended to all users who want to build new auxiliary processors.

The new SVP interface is accessed through interrupt 0B6H and may be used whenever the SVP has been loaded (i.e. when at least one AP has been invoked at APL load time). The value of the BP register is critical to all SVP interface calls, thus it should not be used by the AP, or restored to its original value whenever an SVP service is requested.

Service calls are passed information and return results in the processor's registers. The value of register AH when the interrupt is executed indicates the function to be performed.

A file (\$AP.MAC) has been included in this package, containing equates for the service calls and the APL object headers. You are advised to include this file in your AP source program.

The following is a short description of all the SVP services supported by the new interface:

## Processor Sign-on: 00H

| On Entry | Register Contents                 |
|----------|-----------------------------------|
| AH       | 00H                               |
| BX       | Processor number range 2 to 32767 |

| On Return | Register Contents         |
|-----------|---------------------------|
| CX        | Return code; if carry set |

### Description

This function will sign on the auxiliary processor to the shared variable processor.

See notes 1 and 2.

# Return to APL via Shared Variable Processor: 01H

| On Entry | Register Contents |
|----------|-------------------|
| AH       | 01H               |

| On Return | Register Contents          |
|-----------|----------------------------|
|           | All register contents lost |

## Description

This call is used to signal the end of an Auxiliary processor call, it is the method of releasing control to the APL interpreter.

When the SVP wishes to re-enter the auxiliary processor it will be at the instruction directly after the Wait request.

Special note, the BP register must be maintained at the value it had on entry to the auxiliary processor following the last wait call.

See notes 1 and 2.



## Share or Query the State of a Variable: 02H

| On Entry | Register Contents   |
|----------|---|
| AH       | 02H   |
| BX       | 0; if variable not shared, else variable identifier   |
| CH       | Access control vector for variable  |
| CL       | Initial character of the variable name, this value is given in APL Z-code or zero if any character should be accepted |

| On Return | Register Contents  |
|-----------|--|
| BX        | Variable identification, or a zero if not accepted or retracted  |
| CX        | Return code  |
| DS        | Aligned to object passed if any (at DS:0)  |
| CF        | If set; the variable if any had no value; see CX for reason.<br>If not set; the variable value has been read |

### Description

If  $BX = 0$  and a matching offer is found, the variable is shared and the access control vector is set.

If  $BX \neq 0$  and the variable has been retracted by APL, the variable is retracted by the Auxiliary Processor.

If this is a control variable (recognised through the access control vector), the present value of the variable, if any, is read.

## IBM Internal Use Only

The variable identification should be kept by the AP to be used whenever a service pertaining to this variable is sent to the SVP.

See notes 1, 3 and 5.

## Get the Present Value of a Shared Variable: 03H

| On Entry | Register Contents       |
|----------|-------------------------|
| AH       | 03H                     |
| BX       | Variable identification |

| On Return | Register Contents  |
|-----------|--|
| CX        | Return code  |
| DS        | Aligned to object passed if any  |
| CF        | If set; value could not be read, reason code in CX<br>If not set; then variable was read |

## Description

Read the contents of a variable set by APL assignment. Variable passed is in APL internal format.

See notes 1, 3 and 5.

## Get a Block of Memory From the Workspace: 04H

|          |                                       |
|----------|---------------------------------------|
| On Entry | Register Contents                     |
| AH       | 04H                                   |
| DX       | Number of 16 byte paragraphs required |

|           |   |
|-----------|---|
| On Return | Register Contents   |
| DS        | Pointer to requested space                                  |
| CX        | Return code   |
| DX        | Number of blocks available                                  |
| CF        | If set; no space available<br>If not; set request succeeded |

### Description

This call provides the auxiliary processor or exchange assembly program with a method of obtaining storage from that unused but allocated to the APL workspace.

See notes 1 and 3.

## Release Storage to the Workspace: 05H

|          |                                   |
|----------|-----------------------------------|
| On Entry | Register Contents                 |
| AH       | 05H                               |
| DS       | Address of storage to de-allocate |

|           |   |
|-----------|---|
| On Return | Register Contents   |
| CX        | Return code   |
| DX        | If carry not set; number of 16 byte paragraphs released                         |
| CF        | If set; operation failed, see CX for reason<br>If not set; operation successful |

### Description

This call allows you to release storage back to the workspace. It is the responsibility of the AP to make sure all areas of storage obtained from the workspace are freed back to APL. All values passed by the SVP to the AP must also be freed by the latter.

Memory allocation is made on a push down stack basis. The following example illustrates the allocation method: Assume A to be the first request, and B the second. If A is freed, no memory will be returned to the workspace, since B is still in use. When B is freed, all the areas previously occupied by A and B are returned to the APL workspace.

See notes 1 and 4.

## Pass a Variable to APL and Release the Space: 06H

| On Entry | Register Contents                    |
|----------|--------------------------------------|
| AH       | 06H                                  |
| BX       | Variable identification              |
| DS       | Pointer to the variable to be passed |

| On Return | Register Contents   |
|-----------|---|
| CX        | Return code   |
| DX        | Number of paragraphs released   |
| CF        | If set; operation failed, see CX for reason<br>If not set; operation successful |

### Description

This call permits you to assign a variable to APL and release it's space.

See notes 1 and 4.

# Pass a Scalar Integer Return Code to APL: 07H

| On Entry | Register Contents       |
|----------|-------------------------|
| AH       | 07H                     |
| BX       | Variable identification |
| CX       | Return code to passed   |

| On Return | Register Contents  |
|-----------|--|
| CX        | Return code  |
| CF        | If set; operation failed, see CX for reason<br>If not set; success |

## Description

This call is designed to ease the passing of a return code to APL to indicate to the caller that a particular operation requested of the auxiliary processor has been carried out.

See notes 1 and 4.

## Convert an APL Object from Type Boolean to Integer: 08H

|          |                                       |
|----------|---------------------------------------|
| On Entry | Register Contents                     |
| AH       | 08H                                   |
| DS       | Points to the variable for conversion |

|           |   |
|-----------|---|
| On Return | Register Contents   |
| CX        | Return code from operation  |
| DS        | Points to converted APL object  |
| CF        | If set; operation failed, see CX for reason<br>If not set; operation successful |

### Description

This call converts APL boolean objects into integers, which are more convenient to work with. If the conversion is not done in place, the SVP will release the storage from the old object. (The value of DS on return may be different from the one on entry).

See notes 1 and 4.

## Convert from APL Z-code to ASCII: 09H

| On Entry | Register Contents   |
|----------|---|
| AH       | 09H   |
| DS<br>SI | This register pair point to an input character string to be converted |
| DS<br>DI | This register pair point at the output location                       |
| CX       | Number of bytes to be converted                                       |

| On Return | Register Contents  |
|-----------|--|
| CX        | Return code  |
| CF        | If set; operation failed, see CX for reason<br>If not set; success |

### Description

This call gives a service to translate a character vector to to ASCII from APL internal code known as Z-codes.

You should always remember to point at the start of data for conversion not at the start of object.

| Note that the input and the output must both be in the same  
| segment.

| Translations may be done “in place” that is DS:SI pair may  
| equal DS:DI pair.

See notes 1 and 4.



## Convert from ASCII to APL Z-code: 0AH

| On Entry | Register Contents   |
|----------|---|
| AH       | 0AH   |
| DS<br>SI | This register pair point to an input character string to be converted |
| DS<br>DI | This register pair point to output location                           |
| CX       | Number of bytes to be converted                                       |

| On Return | Register Contents  |
|-----------|--|
| CX        | Return code  |
| CF        | If set; operation failed, see CX for reason<br>If not set; success |

### Description

This call gives a service to translate a character vector to to APL internal code known as Z-codes from ASCII.

You should always remember to point at the start of data for conversion not at the start of object.

| Note that the input and the output must both be in the same  
| segment.

| Translations may be done "in place" that is DS:SI pair may  
| equal DS:DI pair.

See notes 1 and 4.

## Share or Query the State of a Variable: 0BH

| On Entry | Register Contents   |
|----------|---|
| AH       | 0BH   |
| BX       | 0; if variable not shared, else variable identifier   |
| CH       | Access control vector for variable  |
| CL       | Initial character of the variable name, this value is given in APL Z-code or a zero if any character should be accepted |

| On Return | Register Contents  |
|-----------|--|
| BX        | Variable identification, or a zero if accepted or retracted  |
| CX        | Return code  |
| DX        | If carry not set, and if the variable is a control variable, the number of 16-byte paragraphs needed for its value |
| CF        | If set; the variable, if any, had no value, see CX for reason<br>If not set; the variable value has been read      |

### Description

If  $BX = 0$  and a matching offer is found, the variable is shared and the access control vector is set.

If  $BX \neq 0$  and the variable has been retracted by APL, the variable is retracted by the Auxiliary Processor.

If this is a control variable (recognised through the access control vector), the size needed to read its value, is returned.

The variable identification should be kept by the AP to be used whenever a service pertaining to this variable is sent to the SVP.

Equivalent to AH=02H, but the variable value, if any, is just pre-read.

See notes 1 and 4.

## **Pre-read a Variable: 0CH**

|                 |                                |
|-----------------|--------------------------------|
| <b>On Entry</b> | <b>Register Contents</b>       |
| <b>AH</b>       | <b>0CH</b>                     |
| <b>BX</b>       | <b>Variable identification</b> |

|                  |  |
|------------------|--|
| <b>On Return</b> | <b>Register Contents</b>   |
| <b>CX</b>        | <b>Return code</b>   |
| <b>DX</b>        | <b>Number of paragraphs required</b>   |
| <b>CF</b>        | <b>If set; operation failed, see CX for reason<br/>If not set; operation succeeded</b> |

## **Description**

This call enables you to pre-read a variable to see if it can be successfully copied to your internal buffer area.

See notes 1 and 4.

## Read a Previously Pre-read Variable: 0DH

| On Entry | Register Contents                   |
|----------|-------------------------------------|
| AIH      | 0DH                                 |
| BX       | Variable identification             |
| DS       | Pointer to area to receive variable |

| On Return | Register Contents   |
|-----------|---|
| CX        | Return code   |
| CF        | If set; operation failed, see CX for reason<br>If not set; successful |

### Description

This call gives a method of reading a variable to your internal storage area.

See notes 1 and 4.

## Release a Previously Pre-read Variable: 0EH

| On Entry | Register Contents       |
|----------|-------------------------|
| AH       | 0EH                     |
| BX       | Variable identification |

| On Return | Register Contents   |
|-----------|---|
| CX        | Return code   |
| CF        | If set; operation failed, see CX for reason<br>If not set; successful |

### Description

This function will be used when the AP decides to abandon reading a previously pre-read variable.

See notes 1 and 4.

# Pass a Value to a Variable: 0FH

|                 |                                       |
|-----------------|---------------------------------------|
| <b>On Entry</b> | <b>Register Contents</b>              |
| <b>AH</b>       | <b>0FH</b>                            |
| <b>BX</b>       | <b>Variable identification</b>        |
| <b>DS</b>       | <b>Pointer to object to be passed</b> |

|                  |   |
|------------------|---|
| <b>On Return</b> | <b>Register Contents</b>  |
| <b>CX</b>        | <b>Return code</b>  |
| <b>CF</b>        | <b>If set; operation failed, see CX for reason<br/>If not set; successful</b> |

## Description

This call permits you to assign a new value to a shared variable.

See notes 1 and 4.

## Processor Sign-off: 10H

| On Entry | Register Contents                   |
|----------|-------------------------------------|
| AH       | 10H                                 |
| BX       | Processor number (used for sign-on) |

| On Return | Register Contents   |
|-----------|---|
| CX        | Return code   |
| CF        | If set; operation failed, see CX for reason<br>If not set; successful |

### Description

This function should normally not be used. When sign-on has failed, sign-off should NEVER be executed. It is provided for very special case, when an auxiliary processor may sign on and off dynamically from the SVP.

After request, execution is returned to the next instruction after the INT 0B6H

To make a formal termination you must use the \$RET macro supplied in \$AP.MAC.

See notes 1 and 3.

### SVP Reserved Function: 11H

Function AH = 11H is reserved for system use.

## Locate an Associated Variable: 12H

| On Entry | Register Contents                     |
|----------|---------------------------------------|
| AH       | 12H                                   |
| BX       | Variable identification               |
| CL       | Initial letter of associated variable |

| On Return | Register Contents   |
|-----------|---|
| BX        | Identification of associated variable                                 |
| CX        | Return code   |
| CF        | If set; operation failed, see CX for reason<br>If not set; successful |

### Description

Two variables are considered to be associated if their names, with the exception of the first letter, are the same, and they are shared with the same auxiliary processor.

See notes 1 and 4.



## Change the Keyboard / Screen Mode: 13H

| On Entry | Register Contents              |
|----------|--------------------------------|
| AH       | 13H                            |
| BX       | Desired mode (see table below) |

| On Return | Register Contents   |
|-----------|---|
| CX        | Return code   |
| CF        | If set; operation failed, see CX for reason<br>If not set; successful |

### Description

This call permits the BIOS monitor mode to be changed or the keyboard translation to be changed. A side effect of this call is that the screen is cleared if a screen mode change is requested (non-negative values of BX). Valid BX values:

- 0 - 25 × 40 Black and White Alphanumeric
- 1 - 25 × 40 Colour Alphanumeric
- 2 - 25 × 80 Black and White Alphanumeric
- 3 - 25 × 80 Colour Alphanumeric
- 4 - 200 × 320 Colour Graphics
- 5 - 200 × 320 Black and White Graphics
- 6 - 200 × 640 Black and White Graphics
- 7 - 25 × 80 Monochrome display
- ~1 - Set keyboard to APL Translation
- ~2 - Set keyboard to National Translation

See notes 1 and 4.

## Get Loop Count for Delay: 14H

|          |                   |
|----------|-------------------|
| On Entry | Register Contents |
| AH       | 14H               |

|           |                      |
|-----------|----------------------|
| On Return | Register Contents    |
| CX        | Loop count for delay |

This function returns in **CX** the appropriate value so that a subsequent **LOOP** instruction will introduce a delay of one hundredth of a second.

See notes 1 and 4.

## Change the Keyboard / Screen Mode Without Clearing Screen: 15H

| On Entry | Register Contents                       |
|----------|---|
| AH       | 13H                                     |
| BX       | Desired mode (see table on page 13-26). |

| On Return | Register Contents   |
|-----------|---|
| CX        | Return code   |
| CF        | If set; operation failed, see CX for reason<br>If not set; successful |

### Description

This call permits the BIOS monitor mode to be changed without unnecessary erasure of screen contents.

It is essentially the same as call 13H, but may be used to swap between the monochrome display and the colour display without affecting the text displayed on either. The colour display must already be in the correct mode otherwise unpredictable results will be given.

See notes 1 and 4.

## **Notes**

1. Register BP must retain its original value over all calls to interrupt 0B6H.
2. All register contents are lost.
3. Registers ES, SI, DI are saved. All others are lost.
4. Registers DS, ES, SI, DI are saved. All others are lost.
5. The SVP requests space for the value to be passed. It is the auxiliary processor's responsibility to release that space when no longer needed (through function AH=5, for instance).

## Return Codes (Returned in CX Register)

### Code Meaning

- 0 - Success
- 1 - Value error
- 5 - Already signed on
- 6 - Processor table full
- 7 - Invalid sequence
- 8 - Variable locked
- 9 - Variable not shared
- 11 - Not signed on
- 20 - No variable offered
- 21 - Another variable offered
- 22 - Not shared
- 23 - Shared
- 24 - Data variable referenced/specified
- 25 - Control variable referenced
- 26 - No space available
- 27 - Variable was retracted
- 28 - Invalid block of memory to be released
- 29 - Invalid object/value
- 30 - Screen mode not supported in this PC

## Sample Auxiliary Processors

Two sample auxiliary processors in source (assembly) code are supplied as part of the package. They are called AP11.ASM (a single variable AP), and AP12.ASM (a two variable AP). They can be used as examples on how to build your own auxiliary processor.

## **APL Interrupt Usage**

The following information is given to assist system programmers to integrate APL into a total PC environment.

During session initialisation APL stores the pre-session interrupt vectors at the location of interrupt 90H onward. (Int 0H through 8FH are copied to 90H through 011EH respectively).

Therefore, if you have installed a program as a DOS extension, which utilises Int 5H before the APL session, you may invoke this function by calling Int 95H, using AP103 during the APL session.

APL itself utilises:

Int 0B5H For internal purposes

Int 0B6H SVP Services described earlier

If the AP80 auxiliary processor has not been loaded then interrupt 5H will point to a dummy address to prevent misleading results from a print screen request.

The following are also taken over by APL:

| Int 2H Non-maskable

| Int 4H Overflow

Int 9H Keyboard

Int 24H Critical DOS error

Int 0H Divide overflow

| Int 1BH Keyboard Break

| Int D7H Overflow (AT)

If AP2 is loaded then also:

Int 21H DOS function request

## How to Debug Auxiliary Processors

To help in the debugging of your Auxiliary Processor, you may use the DOS DEBUG program. The procedure would be the following:

1. Select the point (in the AP) where you want to set the first DEBUG breakpoint. At that point, insert in your source the following assembly instruction:

```
INT 3
```

2. Assemble and link-edit your AP.
3. Invoke APL under DEBUG in the following way:

```
DEBUG APL.EXE APxxx
```

where APxxx is the name of your AP.

4. When DEBUG gives you control, execute the following DEBUG commands:

```
-g
```

5. APL will start normally. Execute now the proper APL instructions so that your AP will receive control. (Share variables etc.) DEBUG will intercept the execution at the place you inserted INT 3. You may now follow the execution of the AP using DEBUG. Note: The keyboard will maintain its APL definition during execution of debug.

## Exchange Assembly Programs

There is a second way of building user-tailored processors: developing an exchange-assembly program, capable of getting information from APL and passing back results, and called through the non-APL program interface auxiliary processor (AP2). This makes it possible to extend APL in an extremely free fashion. You may, in fact, perform the equivalent of adding new APL primitives to your own system.

A simple example of this power is demonstrated by the FINDST function in the EXCHG workspace, which also contains APL defined function FIND to perform the same task. Although the speed differences are marginal on small objects, when their arguments get very large the gain in speed by FINDST becomes very apparent.

To assist you in the understanding of this method a sample source program (HEXOBJ.ASM) has been included with the system. It works as a monadic function with explicit result. FINDST on the other hand acts as a dyadic function with explicit result.

On close examination of the HEXOBJ program you should note two things: first, the arguments of the function are passed from APL to the exchange program through AP2 shared variables D and E. The exchange program receives pointers to their values. This pointers are located in the program segment prefix, at addresses DS:80H and DS:82H, respectively.

Second, the exchange assembly program is also at liberty to use the services of the SVP via interrupt 0B6H, (This fact is also true for APL programs through the BIOS/DOS interrupt auxiliary processor, AP103. See the DOSFNS workspace).

Return of arguments is via the same method. AP2 expects the return values to be pointed from the same addresses in the segment prefix. They are passed then to APL through shared variables D and E.



Objects passed to or from the exchange assembly program have the same format as those described for an auxiliary processor.

If the exchange assembly program takes storage from the workspace, and if it is not sending it back as a result through addresses 80H or 82H in the segment prefix, it is its responsibility to free that storage, otherwise it will be lost during the remainder of this APL session.

If an object is received from APL, but not passed back, the exchange program must free its space. If no results are passed in one shared variable, the 80H or 82H pointer must be zeroed. Failure to do this will be signalled by a return from AP2 of either -1002 or -1003 which should alert you to this situation.

# Appendix A. Backing up Diskettes

## Before You Begin

You will need these diskettes:

- The diskettes you want to back up - we're going to call these your *original* diskettes. You may also see them called *source* diskettes.
- The diskettes that will become the backup diskettes. Other names for these diskettes are *target* or *destination* diskettes.
- DOS diskette

## Protecting Your Original Diskette

Hint: It's a good idea to put a tab over the write-protect notch to make sure you don't accidentally write on your original diskettes. You may remove the tab when the backup diskettes have been made.

When the write-protect notch is covered, if the diskettes get mixed up, a message similar to the following appears:

```
Target diskette write protected
Correct, then strike any key
```

If you get this message:

1. Remove the original diskette from the drive.
2. Insert the backup diskette.
3. Press any key.

(You do not have to press the Enter key.)

## Backing Up Diskette with One Drive

If you have only one diskette drive, you must remove the original diskette and insert the backup. You may have to make this switch several times; the Disk Operating System (DOS) will tell you when.

The DISKCOPY command will give you the following messages:

**Insert source diskette in drive A:**

**Insert target diskette in drive A:**

So you should:

**INSERT:**

**WHEN:**

**Original diskette**

**"source" message appears**

**Backup diskette**

**"target" message appears**

**IMPORTANT:** Read all of the following steps *BEFORE* starting.

1. Make sure DOS is ready and A> is displayed.
2. Insert the DOS diskette in the drive, if it is not already there.
3. Type:

**diskcopy**

and press the Enter key. The following message appears:

**Insert source diskette in drive A:**

**Strike any key when ready**

**BEFORE YOU PRESS A KEY:**

- a. Remove the DOS diskette that is in the drive.

## IBM Internal Use Only

- b. Insert your original diskette.
- c. NOW press any key.
4. You will see the *in use* light come on while the original diskette is being read; then the following is displayed:

**Insert target diskette in drive A:**

**Strike any key when ready**

### BEFORE PRESSING A KEY:

- a. Remove your original diskette.
- b. Insert the backup diskette.
- c. NOW press any key to tell DOS the correct diskette has been inserted.
5. You will see the *in use* light come on while the backup diskette is being written. Then the message shown in Step 3 will appear again.

Hint: For this procedure, you can remember which diskette to insert if you remember "Original = Source".

Insert your original diskette when DISKCOPY asks for the source diskette.

6. Repeat Steps 3 and 4 until the following message appears:

**Copy complete**

**Copy another (Y/N)?**

Remove the backup diskette from the drive.

7. Type:

**y**

if there are more diskettes to backup. If so, repeat the process for the next diskette to be copied. Otherwise, type:

n

The DOS prompt, A> , is displayed.

(You don't have to press the Enter key.)

8. With a felt-tip pen, mark the label of each backup diskette with the contents, the date, and perhaps, the word "Backup".

## Backing Up Diskette with Two Drives

1. Make sure DOS is ready and A> is displayed.
2. Insert your DOS diskette in drive A.
3. Type:

```
diskcopy a: b:
```

and press the Enter key. The following message appears:

```
Insert source diskette in drive A:
```

```
Insert target diskette in drive B:
```

```
Strike any key when ready
```

4. Remove your DOS diskette from drive A.
5. Insert your original diskette in drive A.
6. Insert your backup diskette in drive B.
7. Press any key.

This tells DOS you are ready, and DOS starts copying the diskette.

If the diskette had not previously been formatted with the same format as the original diskette, a *formatting while copying* message will appear.

## IBM Internal Use Only

All information is now being copied from the diskette in drive A to the diskette in drive B.

You will see one *in use* light go on, then the other.

8. When the copy has been made, you will see a message similar to the following:

**Copy complete**

**Copy another (Y/N)?**

Remove the original diskette and backup diskettes from the drive.

9. Type:

**y**

if there are more diskettes to backup. If so, repeat the process for the next diskette to be copied. Otherwise, type:

**n**

The DOS prompt, A > , is displayed.

(You don't have to press the Enter key.)

10. Remove the last backup diskette from the drive.
11. With a felt-tip pen, mark the label of each backup diskette with the contents, the date, and perhaps, the word "Backup".

**Notes:**

## | Appendix B. APL/PC 1.0 Workspace Migration

| Workspaces generated by means of the IBM Personal Computer APL (Version 1.0) are not compatible with the IBM APL/Personal Computer Version 2.1. A migration procedure must be performed.

| Workspaces generated by means of the internally distributed APL/PC 1.1 and saved as “.APL” files are compatible, but workspaces saved as “.AIO” files are not and also require the use of the migration procedure.

| Workspaces generated by means of the IBM APL Personal Computer Version 2.0 product are fully compatible.

| **Warning:** before attempting to migrate workspaces, a back-up copy of each workspace to be migrated should be made. This will prevent any loss of valuable programs or data should any unexpected problems arise during the migration procedure.

### | Workspaces in APL format

| To be migrated, Version 1.0 workspaces must be in transfer form, i.e. they should have been generated in one of the following ways:

- | • With the *)OUT* command.
- | • With the *OUT* function in the *FILE* workspace.

| APL workspaces in transfer form are recognised because they have an extension of “.AIO”.



| APL 1.0 workspaces in APL format should be converted to  
| transfer form before being migrated to Version 2.1. This is  
| done in the following way:

- | 1. Activate Personal Computer APL Version 1.0.
- | 2. Perform the following APL operations for each  
| APL-format workspace to be converted:

```
|          )LOAD wsid
|          )RESET
|          )OUT wsid
```

| If you get a **SYSTEM LIMIT** error message while the **)OUT** is  
| being performed, it means there is a very large object in the  
| workspace. You will have to split it into smaller objects.  
| (This limitation is not in effect in APL/PC Version 2.1).

| If you get a **LIBRARY FULL** error message, the AIO  
| workspace does not fit in the disk(ette). Discard the partial  
| copy you will find there, and use a diskette with more available  
| space.

| **Warning:** If you **)LOAD** a Version 1.0 compatible workspace  
| under Version 2.1, the command will be rejected with the  
| message **INVALID WORKSPACE**. However, the message **NOT**  
| **FOUND** will appear if you try to **)LOAD** a Version 2.1  
| compatible workspace under APL/PC 1.0.

## | Workspaces in AIO format

| Workspaces in “.AIO” format (transfer file) may be directly  
| migrated to APL Version 2.1 in the following way:

- | 1. Activate APL/Personal Computer Version 2.1.
- | 2. Perform the following APL operation for each AIO-format  
| workspace to be converted:

```
|          )LOAD MIGRATE
```

## IBM Internal Use Only

You will be prompted for the name of the (.AIO) workspace to be migrated. The name should be given in the following way:

*[libn] filename*

i.e. an APL library number may be included, but the extension (.AIO) should not be given.

From this point, the process is completely automatic. The converted workspace will overwrite the old copy. This workspace will be compatible with Version 2.1, but not with Version 1.0.

**Warning:** If you *IN* a 1.0 compatible workspace under Version 2.1, the command will perform successfully, but if you try to execute a function, you are likely to get a *SYNTAX ERROR* message in an apparently correct line, with the caret pointing to an APL right or left arrow. The same will happen if you try to execute a 2.1 compatible workspace under APL 1.0. In this latter case, however, the arrows may appear as other symbols ( $\frac{1}{2}$  or the spanish peseta).

**Notes:**

## Appendix C. The APL Character Set and $\square AV$

The table in this appendix contains the APL character set in the order in which it is contained in the Atomic Vector ( $\square AV$ ). The third column in this table gives the Alt code needed to produce the character under the APL keyboard mapping.

The Atomic Vector is given as it would be printed or when shown on a display with the APL character ROM installed. Certain characters, below ASCII 128, cannot be produced without using this ROM. However, these characters are included for compatibility with the character set of the IBM mainframe program product APL2, and are not used in APL/PC Version 2.1.

| I  | ⌈AV[I] | Alt Code | I  | ⌈AV[I] | Alt Code |
|----|--------|----------|----|--------|----------|
| 0  | (NULL) | 000      | 32 | ⋈      | 251      |
| 1  | ←      | 158      | 33 | ⋉      | 252      |
| 2  | →      | 171      | 34 | ⋊      | 237      |
| 3  | ↑      | 024      | 35 | ⋋      | 232      |
| 4  | ↓      | 025      | 36 | ⋌      | 233      |
| 5  | ∈      | 238      | 37 | ⋍      | 146      |
| 6  | ∖      | 236      | 38 | ⋎      | 174      |
| 7  | ρ      | 230      | 39 | ⋏      | 175      |
| 8  | ,      | 044      | 40 | ⋐      | 159      |
| 9  | ?      | 063      | 41 | ⋑      | 157      |
| 10 | ~      | 126      | 42 | ⋒      | 152      |
| 11 | ○      | 234      | 43 | ⋓      | 240      |
| 12 | +      | 043      | 44 | ⋔      | 241      |
| 13 | -      | 045      | 45 | /      | 047      |
| 14 | ÷      | 246      | 46 | \      | 092      |
| 15 | ⊗      | 015      | 47 | ∩      | 239      |
| 16 | *      | 042      | 48 | ∪      | 225      |
| 17 | x      | 245      | 49 | ⊂      | 226      |
| 18 | !      | 033      | 50 | ⊃      | 227      |
| 19 |        | 124      | 51 | ∘      | 248      |
| 20 | ⌈      | 169      | 52 | α      | 224      |
| 21 | ⌋      | 028      | 53 | ω      | 249      |
| 22 | =      | 061      | 54 | Ç      | 128      |
| 23 | ≠      | 244      | 55 | ç      | 135      |
| 24 | >      | 062      | 56 | ;      | 059      |
| 25 | ≥      | 242      | 57 | ]      | 093      |
| 26 | <      | 060      | 58 | [      | 091      |
| 27 | ≤      | 243      | 59 | )      | 041      |
| 28 | ⋄      | 229      | 60 | (      | 040      |
| 29 | ⋅      | 231      | 61 | :      | 058      |
| 30 | ^      | 094      | 62 | A      | 065      |
| 31 | v      | 235      | 63 | B      | 066      |

**IBM Internal Use Only**

| I  | $\square AV[I]$ | Alt Code | I   | $\square AV[I]$      | Alt Code |
|----|-----------------|----------|-----|----------------------|----------|
| 64 | <i>C</i>        | 067      | 96  | <i>h</i>             | 104      |
| 65 | <i>D</i>        | 068      | 97  | <i>i</i>             | 105      |
| 66 | <i>E</i>        | 069      | 98  | <i>j</i>             | 106      |
| 67 | <i>F</i>        | 070      | 99  | <i>k</i>             | 107      |
| 68 | <i>G</i>        | 071      | 100 | <i>l</i>             | 108      |
| 69 | <i>H</i>        | 072      | 101 | <i>m</i>             | 109      |
| 70 | <i>I</i>        | 073      | 102 | <i>n</i>             | 110      |
| 71 | <i>J</i>        | 074      | 103 | <i>o</i>             | 111      |
| 72 | <i>K</i>        | 075      | 104 | <i>p</i>             | 112      |
| 73 | <i>L</i>        | 076      | 105 | <i>q</i>             | 113      |
| 74 | <i>M</i>        | 077      | 106 | <i>r</i>             | 114      |
| 75 | <i>N</i>        | 078      | 107 | <i>s</i>             | 115      |
| 76 | <i>O</i>        | 079      | 108 | <i>t</i>             | 116      |
| 77 | <i>P</i>        | 080      | 109 | <i>u</i>             | 117      |
| 78 | <i>Q</i>        | 081      | 110 | <i>v</i>             | 118      |
| 79 | <i>R</i>        | 082      | 111 | <i>w</i>             | 119      |
| 80 | <i>S</i>        | 083      | 112 | <i>x</i>             | 120      |
| 81 | <i>T</i>        | 084      | 113 | <i>y</i>             | 121      |
| 82 | <i>U</i>        | 085      | 114 | <i>z</i>             | 122      |
| 83 | <i>V</i>        | 086      | 115 | $\underline{\Delta}$ | 247      |
| 84 | <i>W</i>        | 087      | 116 | $\underline{\quad}$  | 095      |
| 85 | <i>X</i>        | 088      | 117 | $\overline{\quad}$   | 253      |
| 86 | <i>Y</i>        | 089      | 118 | 0                    | 048      |
| 87 | <i>Z</i>        | 090      | 119 | 1                    | 049      |
| 88 | $\Delta$        | 030      | 120 | 2                    | 050      |
| 89 | <i>a</i>        | 097      | 121 | 3                    | 051      |
| 90 | <i>b</i>        | 098      | 122 | 4                    | 052      |
| 91 | <i>c</i>        | 099      | 123 | 5                    | 053      |
| 92 | <i>d</i>        | 100      | 124 | 6                    | 054      |
| 93 | <i>e</i>        | 101      | 125 | 7                    | 055      |
| 94 | <i>f</i>        | 102      | 126 | 8                    | 056      |
| 95 | <i>g</i>        | 103      | 127 | 9                    | 057      |

| I   | AV[I] | Alt Code | I   | AV[I]                          | Alt Cod |
|-----|-------|----------|-----|--------------------------------|---------|
| 128 | .     | 046      | 160 | á                              | 160     |
| 129 | (CR)  | 013      | 161 | í                              | 161     |
| 130 | (NL)  | 010      | 162 | ó                              | 162     |
| 131 | (BS)  | 008      | 163 | ú                              | 163     |
| 132 | (SP)  | 032      | 164 | ñ                              | 164     |
| 133 | (TAB) | 009      | 165 | Ñ                              | 165     |
| 134 | □     | 144      | 166 | á                              | 166     |
| 135 | □     | 145      | 167 | ó                              | 167     |
| 136 | '     | 039      | 168 | í                              | 168     |
| 137 | ⊗     | 228      | 169 | é                              | 130     |
| 138 | ▽     | 031      | 170 | ¬                              | 170     |
| 139 | ⌘     | 250      | 171 | →                              | 026     |
| 140 | ..    | 254      | 172 | U                              | 172     |
| 141 | ì     | 141      | 173 | ì                              | 173     |
| 142 | Ä     | 142      | 174 | ä                              | 132     |
| 143 | Å     | 143      | 175 | ë                              | 137     |
| 144 | â     | 131      | 176 | <small>DDIS<br/>ON 1/4</small> | 176     |
| 145 | ê     | 136      | 177 | <small>DDIS<br/>ON 1/2</small> | 177     |
| 146 | î     | 140      | 178 | <small>DDIS<br/>ON 3/4</small> | 178     |
| 147 | ô     | 147      | 179 |                                | 179     |
| 148 | ö     | 148      | 180 | ¬                              | 180     |
| 149 | ò     | 149      | 181 | ≠                              | 181     |
| 150 | û     | 150      | 182 | ¬                              | 182     |
| 151 | ù     | 151      | 183 | ¬                              | 183     |
| 152 | è     | 138      | 184 | ≠                              | 184     |
| 153 | Ö     | 153      | 185 | ≠                              | 185     |
| 154 | Ü     | 154      | 186 |                                | 186     |
| 155 | ç     | 155      | 187 | ¬                              | 187     |
| 156 | £     | 156      | 188 | ≠                              | 188     |
| 157 | à     | 133      | 189 | ≠                              | 189     |
| 158 | ←     | 027      | 190 | ≠                              | 190     |
| 159 | â     | 134      | 191 | ¬                              | 191     |

**IBM Internal Use Only**

| I   | □AV[I] | Alt Code | I   | □AV[I] | Alt Code |
|-----|--------|----------|-----|--------|----------|
| 192 | L      | 192      | 224 | ☒      | 022      |
| 193 | ⊥      | 193      | 225 | ∈      | 001      |
| 194 | ⊤      | 194      | 226 | ⊥      | 002      |
| 195 | ⊢      | 195      | 227 | ♥      | 003      |
| 196 | —      | 196      | 228 | ♦      | 004      |
| 197 | +      | 197      | 229 | ♣      | 005      |
| 198 | ⊣      | 198      | 230 | ♠      | 006      |
| 199 | ⊢      | 199      | 231 | (BEL)  | 007      |
| 200 | ⊥      | 200      | 232 | ▶      | 016      |
| 201 | ⊣      | 201      | 233 | ◀      | 017      |
| 202 | ⊥      | 202      | 234 | ⊥      | 019      |
| 203 | ⊣      | 203      | 235 | ≡      | 011      |
| 204 | ⊣      | 204      | 236 | ♀      | 012      |
| 205 | ≡      | 205      | 237 | ♫      | 014      |
| 206 | ⊣      | 206      | 238 | ï      | 139      |
| 207 | ⊥      | 207      | 239 | ü      | 129      |
| 208 | ⊥      | 208      | 240 | ‘      | 096      |
| 209 | ⊣      | 209      | 241 | @      | 064      |
| 210 | ⊣      | 210      | 242 | "      | 034      |
| 211 | ⊥      | 211      | 243 | #      | 035      |
| 212 | ⊥      | 212      | 244 | \$     | 036      |
| 213 | ⊣      | 213      | 245 | %      | 037      |
| 214 | ⊣      | 214      | 246 | &      | 038      |
| 215 | ⊣      | 215      | 247 | ¶      | 020      |
| 216 | ⊣      | 216      | 248 | §      | 021      |
| 217 | ┘      | 217      | 249 | ☐      | 023      |
| 218 | ┘      | 218      | 250 | ∴      | 018      |
| 219 | ■      | 219      | 251 | {      | 123      |
| 220 | ▣      | 220      | 252 | —      | 029      |
| 221 | ▤      | 221      | 253 | }      | 125      |
| 222 | ▥      | 222      | 254 | ⏏      | 127      |
| 223 | ▦      | 223      | 255 |        | 255      |



**Notes:**

## **Appendix D. Internal Representation of Displayed Characters**

The National keyboard character set for the IBM Personal Computer has been modified to include some APL characters. The following table contains all the Alt codes (in decimal and hexadecimal) and the characters that they produce under the National mapping of the keyboard.

*Note:* Some alternate codes are reserved for system control functions, and will not generate a displayable character. The reserved codes are:

| <b>Alt Code</b> | <b>Control Function</b> |
|-----------------|-------------------------|
| 007             | Beep                    |
| 008             | Backspace               |
| 009             | Tab                     |
| 010             | Line feed               |
| 027             | Escape (Interrupt)      |
| 127             | National to APL         |

| DECIMAL VALUE | ◆                  | 0    | 16 | 32 | 48 | 64 | 80 | 96 | 112 |
|---------------|--------------------|------|----|----|----|----|----|----|-----|
| ◆             | HEXA DECIMAL VALUE | 0    | 1  | 2  | 3  | 4  | 5  | 6  | 7   |
| 0             | 0                  | NULL | ▶  | SP | 0  | @  | P  | '  | p   |
| 1             | 1                  | ≡    | ◀  | !  | 1  | A  | Q  | a  | q   |
| 2             | 2                  | ℓ    | ∴  | "  | 2  | B  | R  | b  | r   |
| 3             | 3                  | ♥    | ∥  | #  | 3  | C  | S  | c  | s   |
| 4             | 4                  | ◆    | ∏  | \$ | 4  | D  | T  | d  | t   |
| 5             | 5                  | ♣    | §  | %  | 5  | E  | U  | e  | u   |
| 6             | 6                  | ♠    | ∅  | &  | 6  | F  | V  | f  | v   |
| 7             | 7                  | BEL  | ☐  | '  | 7  | G  | W  | g  | w   |
| 8             | 8                  | BS   | ↑  | (  | 8  | H  | X  | h  | x   |
| 9             | 9                  | TAB  | ↓  | )  | 9  | I  | Y  | i  | y   |
| 10            | A                  | NL   | →  | *  | :  | J  | Z  | j  | z   |
| 11            | B                  | ≡    | ←  | +  | ;  | K  | l  | k  | {   |
| 12            | C                  | ♀    | L  | ,  | <  | L  | \  | l  | l   |
| 13            | D                  | CR   | ↔  | -  | =  | M  | l  | m  | }   |
| 14            | E                  | ♪    | Δ  | .  | >  | N  | ^  | n  | ~   |
| 15            | F                  | ⊕    | ∇  | /  | ?  | O  | _  | o  | ⊔   |

Figure D-1 (Part 1 of 2). Internal Representation of Displayed Characters

| DECIMAL VALUE | HEX A<br>DECIMAL VALUE | 128 | 144 | 160 | 176         | 192 | 208 | 224 | 240 |
|---------------|------------------------|-----|-----|-----|-------------|-----|-----|-----|-----|
| 0             | 0                      | Ç   | □   | á   | DOTS ON 1/4 |     |     | α   | ∕   |
| 1             | 1                      | ü   | ▣   | í   | DOTS ON 1/2 |     |     | β   | ∕   |
| 2             | 2                      | é   | ▤   | ó   | DOTS ON 3/4 |     |     | γ   | ∞   |
| 3             | 3                      | â   | ô   | ú   |             |     |     | δ   | ∞   |
| 4             | 4                      | ä   | ö   | ñ   |             |     |     | ρ   | ≠   |
| 5             | 5                      | à   | ò   | Ñ   |             |     |     | ∞   | x   |
| 6             | 6                      | å   | û   | ä   |             |     |     | ρ   | ÷   |
| 7             | 7                      | ç   | ù   | o   |             |     |     | ∞   | Δ   |
| 8             | 8                      | ê   | τ   | ı   |             |     |     | Φ   | ○   |
| 9             | 9                      | ë   | Ö   | Γ   |             |     |     | Θ   | ω   |
| 10            | A                      | è   | Ü   | ┘   |             |     |     | ○   | ∞   |
| 11            | B                      | ï   | ϕ   | →   |             |     |     | v   | ⋈   |
| 12            | C                      | î   | £   | U   |             |     |     | l   | ∇   |
| 13            | D                      | ì   | ⊥   | ı   |             |     |     | ∅   | —   |
| 14            | E                      | Ä   | ←   | Φ   |             |     |     | €   | ∴   |
| 15            | F                      | Å   | I   | Φ   |             |     |     | ∩   |     |

Figure D-1 (Part 2 of 2). Internal Representation of Displayed Characters

**Notes:**

## Appendix E. APL Keyboard Redefinition

The following is a map of the APL keyboard, which is software defined. It consists of a country flag (indicating whether the APL-on or the APL-off keyboard is in effect), and four different tables, each table defining the keys in scan code order, in the four possible combinations:

1. Base case.
2. Shift mode.
3. Alt mode.
4. Ctrl mode.

Undefined key combinations in the tables are denoted by a value of -1.

Country\_Flag DB 0FFH

Base\_Case LABEL BYTE  
 DB 01BH, '1234567890+', 0F5H, 08H  
 DB 09H, 'QWERTYUIOP', 9EH, 91H, 0DH  
 DB -1, 'ASDFGHJKL[]', 0AFH, -1, 0F0H  
 DB 'ZXCVBNM, ./', -1, '×', -1, ' ', -1

Shift\_Case LABEL BYTE  
 DB 27, 0FEH, 0FDH, '<', 0F3H, '=' , 0F2H  
 DB '>', 0F4H, 0EBH, 5EH, '-' , 0F6H, 08H  
 DB 0, '?' , 0F9H, 0EEH, 0E6H, 7EH, 18H  
 DB 19H, 0ECH, 0EAH, '×' , 0ABH, 0E4H, 0DH  
 DB -1, 0E0H, 0A9H, 1CH, '\_' , 1FH, 1EH  
 DB 0F8H, 27H, 90H, '(' )' , 0AEH, -1, 0F1H  
 DB 0E2H, 0E3H, 0EFH, 0ACH, 9DH, 98H  
 DB 7CH, ';' :' , 5CH, -1, 0, -1, ' ', -1

Alt\_Case LABEL BYTE  
 DB 27, 9FH, 0FAH, 0FCH, 0FBH, 0E8H, 0EDH  
 DB 0E9H, 0FH, 0E7H, 0E5H, '! ' , 92H, 08H  
 DB -1, 'qwertyuiop' , 91H, 0F7H, 0DH, -1  
 DB 'asdfghjkl' , 0AFH, 0AEH, -1, -1, -1  
 DB 'zxcvbnm' , 0E4H, 0F1H, 0F0H, -1, -1  
 DB -1, ' ', -1

Ctrl\_Case LABEL BYTE  
 DB 27, -1, -1, -1, -1, -1, -1, -1, -1  
 DB -1, -1, -1, -1, -1, 0DAH, 0C2H, 0BFH  
 DB -1, -1, -1, -1, -1, -1, 0C9H, 0CBH  
 DB 0BBH, 10, -1, 0C3H, 0C5H, 0B4H, -1, 7  
 DB 0C4H, 0CDH, -1, 0CCH, 0CEH, 0B9H, -1  
 DB -1, -1, 0C0H, 0C1H, 0D9H, 0B3H, 0BAH  
 DB -1, -1, 0C8H, 0CAH, 0BCH, -1, -1, -1  
 DB ' ', -1

The address of these tables may be obtained by means of the *KEYB* function included with the UTIL workspace.

The country flag may be assigned two different values:

- 000H (all bits zero) indicates “national keyboard” is in effect.
- 0FFH (all bits one) indicates “APL keyboard” is in effect.

Each of the four keyboard tables maps the central part of the PC keyboard, starting at the top left key (the ESC key), advancing in left-to-right direction until the “backspace” key,

and passing then to the next row, also in left-to-right direction. The last key in each table is the “Caps Lock” key, which is undefined in all four tables.

One or more characters in the tables may be redefined by the user by means of the APL `⊞PK` system function. The redefined keyboard definition will only be retained for the current APL session. However, you may include a keyboard redefinition function in your `PROFILE` workspace, so that it will be automatically performed whenever APL is invoked.

Example: To redefine the Q key to an A the following should be done (the `KEYB` function is assumed to be included in the active workspace):

```
⊞AV[⊞IO+65] ⊞PK 0,KEYB+16
```

In the preceding, `KEYB` gives us the starting position of the keyboard complex (the address of the country flag). Therefore, `KEYB+16` is the address of the byte defining the key marked as a “Q” in the normal APL keyboard, in base case.

The ASCII representation of the character to be replaced (“A”) is 65. This may be converted to APL character form by `⊞AV[⊞IO+65]`.

Finally, the new representation will be poked into the appropriate position in the table by `⊞PK`.

Poking a 255 (-1) in some position in the table will disable the corresponding character key.

A function to simplify the defining of keyboard layouts is supplied in the `UTIL` workspace. See the `KEYBOARD` function described in “The `UTIL` Workspace” on page 11-92.



**Notes:**

## Appendix F. The GRAPHPAK Workspaces - Functions

This appendix contains a list of the functions of the GRAPHPAK workspaces. It is intended as a quick reference guide for experienced GRAPHPAK users.

**Warning:** GRAPHPAK is an powerful general purpose graphical library. It is included in APL/PC 2.1 for the convenience of mainframe programmers already familiar with its functions. Users who do not require applications to run on both Personal and mainframe computers are advised to use the functions provided in AP206. For a full description of GRAPHPAK refer to the *APL2 GRAPHPAK Users Guide and Reference, SH20-9230*.

Users should be aware of the following differences between the APL/PC version of GRAPHPAK and GRAPHPAK as implemented under APL2.

- Only four colours are available;
- Transparent fill is not available (*sf*←0);
- Line style is not implemented (the function *STYLE* is included for compatibility, but has no effect on the display);

### GPBASE

This workspace contains the fundamental drawing and writing functions, and is required by all other GRAPHPAK workspaces. It contains the following functions:

|                 |   |
|-----------------|---|
| <i>COLOR</i>    | changes the current colour;   |
| <i>COPY</i>     | produces hard copy of the current graphic display on an IBM Graphics Printer;                   |
| <i>DRAW</i>     | draws lines between points;   |
| <i>ERASE</i>    | clears the screen;  |
| <i>FILL</i>     | fills a polygon; (Note: transparent fill is not implemented)                                    |
| <i>FIXVP</i>    | sets the viewport;  |
| <i>INTO</i>     | used in co-ordinate transformations;  |
| <i>MODE</i>     | changes the current line mode; (Note: not functional, included for compatibility only)          |
| <i>READ</i>     | reads data items from the screen;   |
| <i>STYLE</i>    | changes the current fill or line style; (Note: not functional, included for compatibility only) |
| <i>USE</i>      | permanently changes the current attributes (colour and width);                                  |
| <i>USING</i>    | temporarily changes the current attributes (colour and width);                                  |
| <i>VIEW</i>     | displays the current contents of the graphics field;  |
| <i>VIEWPORT</i> | returns the co-ordinates of the corners of the current clipping viewport;                       |
| <i>WIDTH</i>    | changes the current line mode;  |
| <i>WRITE</i>    | writes text on the current graphics field;  |
| <i>XFM</i>      | used in co-ordinate transformations.  |

## GPCHT

This workspace contains functions for drawing charts. It requires GPBASE and GPLOT, and the following functions are available:

|                 |   |
|-----------------|---|
| <i>CHART</i>    | draws a bar or column chart;  |
| <i>FREQ</i>     | plots a frequency chart;  |
| <i>HCHART</i>   | plots a hierarchical chart;   |
| <i>PIECHART</i> | draws a pie chart;  |
| <i>PIELABEL</i> | labels a pie chart;   |
| <i>SAXES</i>    | plots all three default axes on a surface or skyscraper chart;                          |
| <i>SAXISX</i>   | plots the X axis of a surface or skyscraper chart with definable labels and annotation; |
| <i>SAXISY</i>   | plots the Y axis of a surface or skyscraper chart with definable labels and annotation; |
| <i>SAXISZ</i>   | plots the Z axis of a surface or skyscraper chart with definable labels and annotation; |
| <i>SLABEL</i>   | writes the default labels the axes of a surface or skyscraper chart.                    |
| <i>SLBLX</i>    | places definable labels on the X axis of a surface or skyscraper chart;                 |
| <i>SLBLY</i>    | places definable labels on the Y axis of a surface or skyscraper chart;                 |
| <i>SLBLZ</i>    | places definable labels on the Z axis of a surface or skyscraper chart;                 |

|                |   |
|----------------|---|
| <i>SS</i>      | plots a skyscraper chart;                                 |
| <i>STEP</i>    | plots a step chart;                                       |
| <i>STITLE</i>  | adds a title to a surface or skyscraper chart;            |
| <i>SURFACE</i> | plots a surface chart;                                    |
| <i>SXFM</i>    | maps three dimensional co-ordinates into a screen window; |
| <i>WITH</i>    | formats data for use with <i>PIECHART</i> .               |

## GPCONT

This workspace contains functions for drawing contour maps. It requires GPBASE and GPLOT, and contains the following functions:

|                |   |
|----------------|---|
| <i>BY</i>      | used to structure the input to <i>CONTOUR</i> ; |
| <i>CONTOUR</i> | draws a contour map;                            |
| <i>OF</i>      | used to structure the input to <i>CONTOUR</i> . |

## GPDEMO

This workspace contains functions illustrating many aspects of the APL/PC version of GRAPHPAK. Loading all the GRAPHPAK workspaces and invoking the function DEMO will cycle through the complete set of demonstrations.

The individual demonstration functions contained in this workspace are:

## IBM Internal Use Only

|                   |   |
|-------------------|---|
| <i>APPLE</i>      | shows the use of <i>DRAW</i> to create a picture;               |
| <i>ATTRIBUTES</i> | shows the various line type and fill options available;         |
| <i>BLI</i>        | shows the use of <i>DRAW</i> to create a picture;               |
| <i>CAYUCAPLOT</i> | shows the use of <i>PLOT</i> to create line graphs;             |
| <i>DEMO</i>       | cycles through all the demo functions in the workspace;         |
| <i>FLAG</i>       | shows the use of <i>DRAW</i> to create a picture;               |
| <i>FSTAR</i>      | shows the use of <i>DRAW</i> to create a picture;               |
| <i>HEALTH</i>     | shows the use of <i>CHART</i> to produce a column chart;        |
| <i>IBMF</i>       | draws the IBM logo;   |
| <i>MILERUN</i>    | shows the use of <i>CHART</i> to produce a bar chart;           |
| <i>NHIST</i>      | shows the use of <i>CHART</i> to produce a step chart;          |
| <i>PIES</i>       | shows the use of <i>CHART</i> to produce a pie chart;           |
| <i>REVB</i>       | shows the use of <i>CHART</i> to produce a simple bar chart;    |
| <i>REVC</i>       | shows the use of <i>CHART</i> to produce a simple column chart; |
| <i>REVENUES</i>   | shows the use of <i>PLOT</i> to create line graphs;             |

|                   |   |
|-------------------|---|
| <b>SKYSCRAPER</b> | shows the use of <i>SS</i> to produce a skyscraper chart;                                     |
| <b>SPIRAL</b>     | shows the use of <i>SKETCH</i> , <i>THREEVIEWS</i> and <i>PERSPECTIVE</i> to create pictures; |
| <b>WAVEGUIDE</b>  | shows the use of <i>SURFACE</i> to produce a surface chart;                                   |
| <b>WGCONT</b>     | shows the use of <i>SURFACE</i> to produce a contour chart.                                   |

## GPFIT

This workspace contains functions for curve fitting. It requires GPBASE and GPLOT, and contains the following functions:

|               |   |
|---------------|---|
| <b>AVG</b>    | prepares data for <i>FIT</i> to plot the average Y value;   |
| <b>CLEAR</b>  | clears the display screen;  |
| <b>FIT</b>    | draws a line graph through data points prepared by <i>AVG</i> , <i>SL</i> , <i>POLY</i> , <i>EXP</i> , <i>LOG</i> , <i>POWER</i> , <i>LOGLOG</i> or <i>SPLINE</i> |
| <b>EXP</b>    | prepares data for <i>FIT</i> to plot the best log-linear fit on linear axes;  |
| <b>FITFUN</b> | executes the last function plotted;   |
| <b>LOG</b>    | prepares data for <i>FIT</i> to plot the best log-linear fit on log-linear axes;  |
| <b>LOGLOG</b> | prepares data for <i>FIT</i> to plot the best log-log fit on log-log axes;  |

## IBM Internal Use Only

|                |   |
|----------------|---|
| <i>POLY</i>    | prepares data for <i>FIT</i> to plot the best polynomial of specified degree; |
| <i>POWER</i>   | prepares data for <i>FIT</i> to plot the best log-log fit on linear axes;     |
| <i>SCRATCH</i> | erases points from a data array;  |
| <i>SL</i>      | prepares data for <i>FIT</i> to plot the best straight line fit;              |
| <i>SPLINE</i>  | prepares data for <i>FIT</i> to plot a cubic spline specified points.         |

## GPGEOM

This workspace contains descriptive geometry functions. It requires GPBASE and GPLOT, and contains the following functions:

|                    |   |
|--------------------|---|
| <i>ISOMETRIC</i>   | restructures data so that <i>SKETCH</i> produces an isometric projection;   |
| <i>MAGNIFY</i>     | transforms a data array so that the object represented is magnified in size;                                      |
| <i>OBLIQUE</i>     | restructures data so that <i>SKETCH</i> produces an oblique projection;   |
| <i>PERSPECTIVE</i> | restructures data so that <i>SKETCH</i> produces a perspective drawing;   |
| <i>RETICLE</i>     | draws an outline of the clipping viewport and a pair of axes showing the extent of the window into problem space; |
| <i>ROTATE</i>      | transforms a data array so that the object represented is rotated;  |



|                   |   |
|-------------------|---|
| <b>SCALE</b>      | scales all elements of a data array so that the largest lies within set bounds; |
| <b>SKETCH</b>     | produces an orthogonal projection;  |
| <b>STEREO</b>     | produces a stereo pair of images of an object;                                  |
| <b>THREEVIEWS</b> | produces three views of an object, projected into the planes of the axes;       |
| <b>TRANSLATE</b>  | transforms a data array so that the object represented is moved.                |

## GPLOT

This workspace contains plotting functions. It is required by GPFIT, GPCONT and GPCHT, and requires GPBASE. It contains the following functions:

|              |  |
|--------------|--|
| <b>AND</b>   | used in formatting input for <i>PLOT</i> or <i>SPLOT</i> ;         |
| <b>ANNX</b>  | draws an annotated horizontal axis with definable label positions; |
| <b>ANNY</b>  | draws an annotated vertical axis with definable label positions;   |
| <b>AXES</b>  | draws default axes;  |
| <b>AXIS</b>  | draws definable axes;  |
| <b>HOR</b>   | draws an annotated horizontal axis with default label positions;   |
| <b>LABEL</b> | produces the default labels for the X and Y axes;                  |

## IBM Internal Use Only

|                |  |
|----------------|--|
| <i>LBLX</i>    | produces definable labels for the X axis;  |
| <i>LPLY</i>    | produces definable labels for the Y axis;  |
| <i>PLOT</i>    | plots a line graph;  |
| <i>RESTORE</i> | restores all attribute and plotting variables to their default values;                 |
| <i>SPLOT</i>   | similar to <i>PLOT</i> , but allowing more user control over plotting characteristics; |
| <i>TITLE</i>   | adds a title to the graph;   |
| <i>VER</i>     | draws an annotated vertical axis with default label positions;                         |
| <i>VS</i>      | used in formatting input for <i>PLOT</i> or <i>SPLOT</i> .                             |

**Notes:**

## | Appendix G. Hardware Modification for | IBM 4860 PCjr

| A minor hardware modification to the PCjr must be performed  
| before APL may be run on a PCjr.

| The hardware modification is required because the design of  
| the PCjr allows the Test pin of the 8088 processor to float as it  
| is not needed for normal operation. However, the correct  
| operation of the 8087 emulation software depends on this line  
| being held high.

| To achieve this, solder a 4.7K ohm resistor between the Vcc  
| pin (pin 40) and the Test pin (pin 23) of the 8088 processor  
| chip.

| **Warning:** This hardware modification should only be  
| attempted by someone with the appropriate expertise in  
| handling a soldering iron!

**Notes:**

## Appendix H. Patch to Restore BIOS Keyboard Handler

New versions of PCs and ATs have a new BIOS that includes a keyboard handler which screens out certain characters that are required by APL. APL replaces this keyboard handler with an equivalent handler that does not screen out these characters. Older versions of PCs and ATs do not need this replacement code and the original handler may be restored by the patch given below.

The patch to restore the use of the BIOS interrupt X'16' keyboard handler is as follows:

From DOS bring up APL with AP210.

```
      APL AP210
      )IN FILE
      PATCH '$SCR.COM'
GIVE ADDRESS: 13
IS 26
GIVE NEW VALUE OR EMPTY LINE TO CANCEL PATCH
: EB
GIVE ADDRESS: 14
IS 80
GIVE NEW VALUE OR EMPTY LINE TO CANCEL PATCH
: 2B
GIVE ADDRESS:
```

**Notes:**

# Index

## Special Characters

\$AP.MAC 13-7  
 )CLEAR command 2-23, 10-9  
 )COPY command emulation 11-92  
 )DROP command 10-19  
 )ERASE command 10-11  
 )FNS command 10-13  
 )IN command 10-11  
 )LIB command 10-20  
 )LOAD command 2-22, 10-17  
 )OFF command 10-21  
 )OUT command 10-16  
 )RESET command 2-22, 10-12  
 )SAVE command 2-21, 10-15  
 )SI command 10-13  
 )SINL command 10-14  
 )STACK command 10-10, 10-13  
 )SYMBOLS command 10-10, 10-13  
 )VARS command 2-22, 10-13  
 )WSID command 10-15, 10-20  
 ]-- IMPLICIT ERROR  
   message 4-4  
 ]AI, account information 2-42, 6-16  
 ]AV, atomic vector 6-16, C-1  
 ]CR, canonical representation 6-5  
 ]CT, comparison tolerance 6-17  
 ]DL, delay function 6-6  
 ]EA, execute alternate 2-32, 6-6  
 ]EX, expunge 6-7  
 ]FC, format control 6-17  
 ]FX, function establishment 6-7  
 ]FX, function fix 2-46  
 ]HT, horizontal tabs 6-18  
 ]IO, index origin 2-43, 6-18  
 ]LC, line counter 2-31, 6-19  
 ]LX, latent expression 6-18  
 ]NC, name class 6-8  
 ]NL, name list 6-9  
 ]PK, peek/poke 6-9

]PP, printing precision 6-19  
 ]PW, printing width 6-20  
 ]RL, random link 6-20  
 ]SVC, shared variable control 7-7  
 ]SVO, shared variable 7-6  
 ]SVQ, shared variable query 7-12  
 ]SVR, shared variable  
   retraction 7-11  
 ]TC, terminal control 2-27, 6-20  
 ]TF, transfer form 6-11  
 ]TS, time stamp 6-20  
 ]TT, terminal type 6-20  
 ]UL, user load 6-20  
 ]WA, workspace available 6-20

## A

access control 7-7  
   shared variables 13-4  
 access control matrix (ACM) 7-8  
 access control vector 7-9, 13-4  
 access sequence disciplines 7-3  
 access state of shared variable 7-8, 13-4  
 accessing a file 13-3  
 account information 6-14, 6-16  
 active workspace 4-13, 10-9, 11-67, 11-69  
   copying to 10-11  
   inquiry commands 10-13  
   list of functions in 10-13  
   list of variables in 10-13  
   settings of state indicator 10-9  
   transfer form of objects  
     in 10-16  
 activities in suspended state 9-4  
 adding a statement 8-10  
 adding characters 1-32  
 adding to a header 8-12  
 adding to a statement 8-12  
 alphabetic character set 1-18, 4-6



- Alt codes 1-23, C-1
- Alt key 1-23
- alternating product 5-18
- alternating sum 5-18
- ambi-valent functions 8-5
- AND, boolean function 5-9
- APL
  - applications 1-3
  - as a computing system 1-3
  - character set 1-18, 1-19, 4-6
  - character set ROM 1-8
  - classes of instructions
    - statements 4-3
    - system commands 10-3
  - command format 1-16
  - data representation 13-5
  - data used in 4-10
  - environment 4-14, 10-9
  - examples of use 3-3
  - fundamentals 4-3
  - header 12-55
  - input editor 1-29
  - internal code 12-56
  - libraries 12-7, 12-14, 12-55
  - library numbers 1-35
  - major characteristics of 3-5
  - objects 12-55
  - package 1-8
- APLFILE workspace 11-42
- application workspaces 11-3
- AP101 stack and profile auxiliary processor 12-14
- AP103 BIOS/DOS auxiliary processor 12-18
- AP124 full screen auxiliary processor 12-24
- AP124 workspace 11-8
- AP190 host communications auxiliary processor 12-34
- AP190 workspace 11-19
- AP2 non-APL program interface auxiliary processor 12-4
- AP2 workspace 11-5
- AP205 full-screen auxiliary processor 12-38
- AP205 workspace 11-21
- AP206 graphic auxiliary processor 12-39
- AP206 workspace 11-21
- AP210 file auxiliary processor 12-53
- AP232 asynchronous communications auxiliary processor 12-62
- AP232X extended asynchronous communications auxiliary processor 12-69
- AP232X workspace 11-26
- AP440 music auxiliary processor 12-76
- AP488 GPIB support auxiliary processor 12-79
- AP488 workspace 11-28
- AP80 printer auxiliary processor 2-26, 12-10
- arguments 8-5
  - boolean 5-9, 5-10
  - character 5-30
  - functions 4-7, 4-8
  - left and right 4-8, 5-29
  - matrix 5-25, 5-30
  - rank of 5-25, 5-30
  - scalar 5-30
  - shape 5-30
  - vector 5-28
- arithmetic symbols 3-6, 4-6
- arrays
  - conformable 5-34, 5-38
  - determining shape of 4-11
  - elements 4-10
  - empty 4-10, 5-32
  - indexing of 5-44
  - multi-dimensional 4-10
  - number of dimensions 4-10
  - selecting elements 4-11, 5-25
  - shape 4-11
  - size of 5-30
  - structure of 5-25
  - vector 4-10
- ASCII codes 1-23, 12-56
- ASCII translation table 11-93
- assignment statement 4-3
- atomic vector 6-14, 6-16, C-1
- attention signal 9-4
- auxiliary files on the host 11-112
- auxiliary processor
  - AP101 12-14
  - AP103 12-18
  - AP124 12-24
  - AP190 12-34
  - AP2 12-4
  - AP205 12-38
  - AP206 12-39

## IBM Internal Use Only

AP210 12-53  
AP232 12-62  
AP232X 12-69  
AP440 12-76  
AP488 12-79  
AP80 12-10  
  debugging 13-32  
auxiliary processors 1-16  
axis indexing 5-41  
axis operator 5-19  
axis, permutation of 5-36

**B**

Back Tab key 1-31  
backing up diskette  
  with one drive A-2  
  with two drives A-4  
Backspace key 1-22  
bare output 9-13  
baud rate 11-98  
bilateral sharing of variables 7-3  
binomial  
  domain of 5-16  
  function 5-16  
BIOS function call 12-18  
blank character 4-6  
boolean and relational symbols 3-6  
boolean functions 5-9  
boolean variable 13-5  
branch 2-7, 4-3, 8-7  
branch statement 8-7, 9-4  
branch symbol 8-7

**C**

calculation, isolated 3-3  
cancelling a line 1-33  
canonical representation 6-5, 8-3  
Caps Lock key 1-22  
catenate function 5-33  
changing display modes 1-34  
changing incorrect characters 1-32  
changing keyboard between APL  
  and National character set 1-25  
character data 2-4, 2-24  
character input 9-12

character representation of  
  function 8-3  
character set C-1  
  APL 1-19, 4-6  
  classes 1-18, 4-6  
  National 1-19  
character variable 13-5  
circular functions 2-13, 5-14  
classes of instructions  
  statements 4-3  
  system commands 10-3  
clear workspace  
  command 2-23, 10-9  
  environment 10-9  
clearing the screen 1-34  
closing a file 11-56  
collating sequence 5-45  
colour graphics adapter mode 1-34  
COMMAND ERROR  
  message 2-22, 10-6, 10-10, 10-11  
commands 4-13, 4-14  
comment symbol 5-54, 8-9  
communication with  
  VM/370 11-95  
comparison tolerance 5-9, 6-14,  
  6-17  
compress function 5-38, 5-40  
conditional execution 2-31  
configuration minimum  
  requirements 1-6  
conformable arrays 5-34, 5-38  
conjugate function 5-8  
connection with host 11-103  
constants 4-12  
contents of machine registers 12-19  
control characters 5-59  
control structure 2-7  
controlled decorators 5-59  
conventional decorators 5-59  
conventional notation 4-9  
converting numeric data 13-6  
copy command emulation 11-92  
copying to active workspace 10-11,  
  11-67  
cosine 5-14  
coupling of name 7-6  
creating defined function 8-3  
Ctrl key 1-23  
Ctrl-End key 1-30  
Ctrl-Home key 1-30  
cursor 1-21, 1-29  
cyclic rotation 5-32

**D**

data 4-10  
 data transformation 5-53  
 deal function 5-48  
 debugging  
   APL code 2-28  
   auxiliary processors 13-32  
 decode function 5-51  
 decorators 5-59  
 defined function 3-3, 8-3  
   ambi-valent 8-5  
   execution 9-3  
   full-screen editor 11-51  
   names 4-13  
 defining function keys 12-15  
 defining libraries 12-16  
 definition mode 8-3  
 DEFN ERROR message 4-4  
 del editing 8-10  
 Del key 1-31, 1-32  
 delay function 6-6  
 delete control characters 11-101  
 deleting a statement 8-11  
 deleting characters 1-31  
 delta 2-19, 4-14  
 delta symbol 4-6  
 delta underbar 2-19  
 DEMO124 workspace 11-47  
 DEMO206 workspace 11-48  
 dimensions 4-10  
 disabling attention 6-11, 11-93  
 disabling output 11-94  
 diskettes 1-8  
 display modes 1-34  
 displaying internal  
   representation 11-55  
 DOMAIN ERROR message 2-13,  
   4-4  
 domino function 5-48  
 DOS command emulation 11-48  
 DOS file system 12-53  
 DOS function call 12-18  
 DOSFNS workspace 11-48  
 double field 5-62  
 down arrow key 1-30  
 downloading files 11-107  
 drop function 5-38  
 dropping a workspace 10-19  
 dyadic format 5-56

numeric 5-56  
 picture 5-59  
 dyadic function 2-12, 4-8, 8-5

**E**

e 2-12, 5-13  
 EDIT workspace 11-51  
 editing functions 2-29, 8-10  
 element, identity 5-7  
 elements 4-15  
 embedded decorators 5-61  
 empty array 4-10, 5-32  
 empty input 9-12  
 empty vector 4-10, 5-32  
 enabling attention 6-11, 11-93  
 enabling output 11-95  
 encode function 5-52  
 End key 1-30  
 end of line character 11-102  
 Enhanced Graphics Adapter  
   displaying APL characters 1-12  
 Enter key 1-21, 1-29  
 entering a line on the screen 1-29  
 environment in clear  
   workspace 10-9  
 erasing characters 1-32  
 erasing part of a line 1-33  
 error messages 4-3  
   □-- IMPLICIT 4-4  
   DEFN 4-4  
   DOMAIN 4-4  
   IMPLICIT 4-4  
   INDEX 4-4  
   INTERRUPT 4-4  
   LENGTH 4-4  
   RANK 4-5  
   SI DAMAGE 4-5  
   STACK FULL 4-5  
   SYMBOL TABLE FULL 4-5  
   SYNTAX 4-5  
   SYSTEM 4-5  
   SYSTEM LIMIT 4-5  
   VALUE 4-5  
   WORKSPACE FULL 4-6  
 error report 9-4  
 error stop 9-4  
 error trap 6-6  
 Esc key 1-21, 1-31, 1-33

# IBM Internal Use Only

establishing functions 8-10  
evaluated input 9-11  
EXAPL 12-3  
exchange assembly programs 13-33  
EXCHG workspace 11-55  
exclusive-OR 5-10  
execute alternate 2-31, 6-6  
execute function 2-43, 5-53, 5-54  
execution  
    mode 8-3  
    of defined function 9-3  
    order of 4-9  
execution stack 4-15, 10-10, 10-13  
expand 5-39, 5-40  
exponential function 5-13  
expunge system function 6-7

## F

factorial function 5-15  
file control functions 11-57  
file transfer functions 11-105  
FILE workspace 11-56  
FINDST 13-33  
fixed-length record disk files 12-53  
floating point variable 13-5  
floor function 5-11  
FOIL workspace 11-73  
format control 6-17  
format of APL command 1-16  
format of shared data 13-5  
format, dyadic  
    numeric 5-56  
    picture 5-59  
format, monadic 2-17, 5-55, 11-93  
format, scaled 4-12, 5-62  
forms of headers 8-5  
forms of numbers  
    conventional 4-12  
    scaled 4-12  
FORTRAN subroutines 11-74  
FORTRAN workspace 11-74  
French keyboard layout 11-92  
function  
    ambi-valent 8-5  
    boolean 5-9  
    catenate 5-33  
    character representation 8-3  
    circular 5-14

deal 5-48  
decode 5-51  
defined 8-3  
domino 5-48  
drop 5-38  
dyadic 4-8, 8-5, 9-3  
encode 5-52  
establishing 8-10  
execute 5-53, 5-54  
executing 9-3  
exponential 5-13  
factorial 5-15  
floor 4-7, 5-11  
format 5-55, 5-56  
general logarithm 5-13  
grade down 5-45  
grade up 5-46  
halted 9-5  
hyperbolic 5-14  
index of 5-43  
inverse 5-42  
membership 5-44  
minus 5-7  
mixed 5-25  
monadic 4-8  
names of 4-14  
natural logarithm 5-13  
negative 5-8  
niladic 4-8  
pendent 9-4, 10-14  
pendent execution 9-5  
plus 5-7  
power 5-12  
primitive 9-3  
pythagorean 5-14  
reciprocal 5-8  
relational 5-9  
replicate 5-39  
reshape 4-11  
residue 5-7  
reverse 5-32  
revising 8-10  
roll 5-12  
rotate 5-32  
shape 5-30  
signum 5-8  
stopping execution 9-4  
structural 5-30  
suspended execution 9-4  
system 6-3  
take 5-37  
times 5-7

transpose 5-36  
 unlocked defined 6-5, 8-13, 9-8  
 valence of 8-5  
 function body 8-3  
 function definition mode 8-3  
 function display 8-12  
 function editing 8-10  
 function establishment 6-7  
 function header 8-3  
 function keys 1-19, 12-14  
   defining 12-15  
 functions for manipulating DOS  
 files 11-57

## G

games control adapter 11-94, 12-23  
 GEDIT workspace 11-82  
 general logarithm function 5-13  
 general symbols 3-6, 4-6  
 generating selector  
   information 5-25  
 German keyboard layout 11-92  
 global names 8-6  
 global shared variable 7-3  
 go to 2-7  
 GPBASE workspace 11-84, F-1  
 GPCHT workspace 11-84, F-3  
 GPCONT workspace 11-85, F-4  
 GPDEMO workspace 11-85, F-4  
 GPFIT workspace 11-85, F-6  
 GPGEOM workspace 11-85, F-7  
 GPLOT workspace 11-85, F-8  
 grade down function 5-45  
 grade up function 2-41, 2-46, 5-46  
 graphic input 11-83  
 GRAPHPAK workspaces 11-84,  
 F-1

## H

halted execution 9-4  
 halting printing temporarily 1-26  
 header forms 8-5  
 hexadecimal conversion 11-55,  
 11-94  
 hexadecimal patches 11-65

Home key 1-30  
 horizontal tabs 6-18  
 hyperbolic functions 5-14

## I

I/O ERROR message 10-12,  
 10-16, 10-17, 10-19, 10-20  
 identity elements 5-7  
 implementation limits 4-15  
 IMPLICIT ERROR message 4-4,  
 6-14  
 inactive workspace 4-13  
 INDEX ERROR message 4-4  
 index generator 5-43  
 index of function 5-43  
 index origin 4-12, 6-14, 6-18  
 indexing 4-10, 5-41  
   array elements 4-11  
   one-origin 4-12  
   zero-origin 4-12  
 information exchange 13-3  
 inhibiting output 6-11  
 INKEY function 12-20  
 inner product operator 2-6, 5-21  
 input and output 9-10  
 input editor  
   special keys 1-29  
 input line stack 12-14  
 input state 1-32  
 inquiry commands, active  
   workspace 10-13  
 Ins key 1-30  
 insert mode 1-30  
 inserting a statement 8-11  
 inserting characters 1-30  
 installing APL on fixed disk 1-13  
 integer variable 13-5  
 interactive input mode 11-83  
 internal APL code 12-56  
 internal representation 11-55, 13-5  
 interpretations of file data 12-55  
 interrupt  
   strong 1-25  
   weak 1-21  
 INTERRUPT message 4-4, 9-8  
 interrupt number 12-19  
 interrupting execution during  
 input 9-12

INVALID WS message 10-18  
inverse function 5-42  
inverse transfer form 6-12  
isolated calculation 3-3

**J**

joystick 11-94, 12-23

**K**

key combinations 1-25  
keyboard 1-19, 1-28  
keyboard buffer 11-94  
keyboard keycaps 1-26  
keyboard layout 11-94  
keyboard template 1-27  
keycaps 1-26

**L**

labels 8-9  
laminar function 5-34  
lamp symbol 8-9  
latent expression 6-14, 6-18  
left argument 4-8  
left arrow key 1-30  
left identity elements 5-5  
LENGTH ERROR message 2-16,  
4-4  
LIBRARY FULL message 10-17  
library identification 1-35, 10-14,  
12-7, 12-14, 12-55  
defining 12-16  
limits, implementation 4-15  
line counter 6-14, 6-19  
line editor 1-32  
line parameter definition 11-96  
line signal 9-14  
literal (character) variable 13-5  
local names 8-5  
local shared variable 7-3  
locked function 9-8  
logarithm functions 2-12, 5-13

logical (boolean) variable 13-5  
loops 2-7, 2-33  
lowercase characters 4-6  
lowercase letters 4-14

**M**

machine language program 6-10  
machine registers 12-18  
magnitude function 5-8  
making corrections to current  
line 1-32  
managing resources 6-3  
matrix  
access control 7-8  
axes 4-11  
character 8-3  
matrix divide 5-48, 5-49  
matrix inverse 5-48, 5-49  
matrix product 5-22  
matrix transposition 5-36  
maximum function 5-11  
maximum limits 4-15  
membership function 5-44  
messages, error 4-4  
minimum function 5-11  
minus function 5-7  
minus sign 2-9  
mixed functions 5-25  
module sizes 12-3  
monadic function 2-12, 4-8, 8-5  
monochrome display mode 1-34  
multi-dimensional arrays 4-10  
MUSIC workspace 11-86

**N**

name assignment statement 4-3  
name class 6-8  
name coupling 7-6  
name list 6-9  
names  
as label 8-9  
defined functions 4-13, 4-14  
localisation 8-6  
rules for 2-19  
system functions 6-3

system variables 6-3  
 variable 4-13, 4-14  
 workspace 4-14

NAND, boolean function 5-9  
 national character set 1-19, 1-21  
 natural logarithm function 5-13  
 negative function 5-8  
 negative numbers 4-12  
 niladic function 4-8  
 non-integral index 5-35  
 NOR, boolean function 5-9  
 normal output 9-12  
 NOT ERASED message 10-7  
 NOT FOUND message 10-7  
 NOT function 5-9  
 NOT READY message 10-17,  
 10-19, 10-20  
 NOT READY report 10-12, 10-16  
 NOT SAVED message 10-7  
 Num Lock 1-24  
 numeric character set 1-18, 4-6  
 numeric constant 4-12  
 numeric data, converting 13-6  
 numeric format 5-56  
 numeric functions 5-48  
 numeric keypad 1-24

## O

object size limit 11-74  
 objects 4-7, 4-15  
 offers  
 to share 7-6  
 opening a file 11-56  
 operator information area 12-37  
 operators 2-6, 5-17  
 options in APL command 1-16  
 OR, boolean function 5-9  
 order of execution 2-10, 2-30, 5-17  
 outer product operator 2-18, 2-36,  
 2-40, 3-6, 5-23  
 overbar 2-9, 2-19, 4-12

## P

parentheses 2-10, 4-9  
 parity 11-98  
 pause state 1-26  
 PC Storyboard 11-95  
 PC3278 12-34  
 peek/poke 6-9  
 pendent function 9-4, 10-14  
 permutation of axes 5-36  
 Personal Editor 11-54  
 PFORTPAR 11-79  
 pi 2-14, 5-13  
 picture format 5-59, 6-17  
 playing music 11-86  
 PLOT workspace 11-87  
 plus function 5-7  
 power function 2-11, 5-12  
 precedence 2-10  
 precision indicator 5-56  
 primitive function 3-5, 9-3  
 classes of 5-3  
 primitive operator 9-3  
 primitive types 4-3  
 PRINT workspace 11-89  
 printer 1-36, 2-26  
 printer as system log 1-26, 1-36,  
 6-11, 11-94  
 printer control codes 11-90  
 printing copy of screen 1-22, 1-36  
 printing precision 6-14, 6-19  
 printing width 6-14, 6-20, 9-13  
 processor, auxiliary 1-16  
 Professional Editor 11-54  
 Professional Graphics Adapter  
 displaying APL characters 1-12  
 PROFILE workspace 11-91  
 program execution 4-13, 9-3  
 program, machine language 6-10  
 programs 3-3  
 PROTECTED message 10-17,  
 10-19  
 PROTECTED report 10-16  
 PrtSc key 1-22  
 pythagorean functions 5-14

**Q**

quad output 2-29  
 quad symbol 4-6, 6-3, 9-11  
 quiet option 1-17  
 quote-quad symbol 4-6, 9-12

**R**

radians 5-14  
 random link 5-12, 6-14, 6-20  
 RANK ERROR message 4-5  
 rank of a scalar 13-5  
 ravel 5-30  
 reading records from a file 11-60  
 real (floating point) variable 13-5  
 reciprocal function 5-8  
 recursive  
     See recursive  
 recursive functions 9-9  
 reduction operator 2-6, 2-34, 5-17  
 relational functions 5-9  
 replacing a statement 8-11  
 replacing header 8-11  
 replicate function 5-39  
 request for character input 9-12  
 reshape function 4-11, 5-31  
 residue function 5-7  
 retraction of sharing 7-11  
 return codes  
     AP101 12-17  
     AP124 12-33  
     AP2 12-9  
     AP206 12-53  
     AP210 12-59  
     AP232X 12-75  
 reverse function 5-32  
 revising functions 8-10  
 right argument 4-8  
 right arrow key 1-30  
 right identity elements 5-5  
 roll function 5-12  
 rotate function 5-32  
 rules, semantics 3-5

**S**

sample auxiliary processors 13-30  
 saving a file 11-56  
 saving line parameter  
     definition 11-102  
 saving workspace 2-21, 10-14  
 scalar 5-3  
     rank 13-5  
 scaled formatting 5-62  
 scan operator 5-19  
 screen control characters 6-16  
 screen fields 12-25  
 searching  
     functions 11-56, 11-95  
     string 11-56  
 selecting from arrays 4-11  
 selection and structural  
     symbols 3-6  
 selection from arrays 5-25, 5-42  
 selection functions 5-37  
 semantic rules 3-5  
 sending display output to the  
     printer 1-36  
 sequence control 3-5  
 services provided by auxiliary  
     processor 13-3  
 setting size of execution  
     stack 10-10  
 setting size of symbol table 10-10  
 shape function 5-30  
 shape of array 4-11  
 shared data 13-5  
 shared variable 2-25  
 shared variable offer 2-26, 7-6  
 shared variable processor 13-3  
     return codes 13-30  
     services 13-8  
 shared variables 7-3, 13-3  
     access control 13-4  
     access control vector 13-4  
     access states 13-4  
     managing 13-3  
 Shift keys 1-21, 1-23  
 SI DAMAGE message 4-5, 9-6  
 signing off 10-21  
 signum function 5-8  
 sine 5-14  
 size of a file 11-60  
 sizes of APL modules 12-3



special APL characters 1-18, 4-6  
 special key combinations 1-25,  
 1-26  
 specification arrow 4-3  
 square root 2-11  
 STACK FULL message 4-5, 9-9  
 stack input 12-14  
 stack, execution 4-15, 10-10, 10-13  
 stacking APL input 12-15  
 starting APL 1-15  
 state indicator 9-4, 10-14  
   clearing 9-6  
   damage 9-6  
   list 9-6  
   settings in active  
     workspace 10-9  
 statement 4-3  
   entering of 1-3  
   typical form of 4-3  
 stop bits 11-100  
 stop control 2-31, 9-7  
 stopping execution of function 9-4  
 Storyboard 11-95  
 string searching 11-56  
 strong interrupt 1-25  
 structural functions 2-5, 5-30  
 structure of arrays 5-25  
 subcommand state 12-57  
 suspended function 9-4, 10-14  
 suspended state 9-4  
 SVP 13-3, 13-8  
 switching displays 1-34  
 symbol table 4-15, 10-10, 10-13  
 SYMBOL TABLE FULL  
   message 4-5  
   symbols 3-6  
   syntax 3-5  
 SYNTAX ERROR message 4-5  
 system commands 10-3  
 SYSTEM ERROR message 4-5  
 system functions 6-3  
 SYSTEM LIMIT message 4-5  
 system log 1-26  
 system reset 1-25  
 system variables 6-14, 7-3

T

Tab key 1-21, 1-31  
 take function 5-37  
 tangent 5-14  
 template 1-27  
 temporarily halting printing 1-26  
 terminal control 6-14, 6-20  
 terminal input and output 9-10  
 terminal selection 11-96  
 terminal type 6-14, 6-20  
 terminating work session 10-21  
 time stamp 6-14, 6-20  
 times function 5-7  
 tolerance 6-14  
 TopView 11-66  
 trace control 2-30, 9-6  
 transfer file 10-11, 10-16  
 transfer form 6-11  
 transfer form of objects 10-16  
 transfer form vector 6-12  
 transformation of data 5-53  
 transpose function 5-36  
 trigonometric functions 2-13  
 trouble reports 10-7  
 turnaround local character 11-99  
 typematic keys 1-19  
 typewriter keyboard 1-21

U

undefined  
 underbar 2-19, 2-21  
 unlocked defined function 6-5,  
 8-13, 9-8  
 up arrow key 1-29  
 uploading files 11-106  
 user load 6-14, 6-20  
 using the printer 11-89  
 UTIL workspace 11-92

V

valence of function 8-5  
VALUE ERROR message 2-20,  
4-5  
variable 2-20  
variable-length record disk  
files 12-53  
variables  
names 2-19, 4-13, 4-14  
system 6-14, 7-3  
vector  
access-control 7-9  
elements of 4-10  
empty 4-10, 5-32  
forming 4-10  
VM232 workspace 11-95

W

WAS report 10-15  
weak interrupt 1-21, 1-31, 9-8  
width indicator 5-57  
width of number field 5-56  
work session  
initiation 1-15  
termination 10-21

workspace  
active 4-13, 10-9, 11-67, 11-69  
copying to 10-11  
inquiry commands 10-13  
list of functions in 10-13  
list of variables in 10-13  
transfer form of objects  
in 10-16  
determining name of 11-95  
dropping of 10-19  
for storing functions and  
data 3-4  
inactive 4-13  
management of 4-13  
names of 4-14  
retrieval 10-14, 10-20  
saving copies 10-14  
state indicator 4-13  
storage 2-21, 10-14  
workspace available 6-14, 6-20  
WORKSPACE FULL  
message 2-36, 4-6  
write-protect notch A-1  
writing data into a file 11-56

X

XOR 5-10

**Notes:**

IBM

ZZ33-0523-0  
Printed in U.S.A.

ZZ33-0523-00

