Personal Computer
Computer Language
Series

APL

by IBM Madrid Scientific Center

1502219

Personal Computer
Computer Language
Series

APL

by IBM Madrid Scientific Center

First Edition (May 1983)

Preface

APL is a general-purpose language that can be used in many applications, such as commercial data processing, system design, mathematical and scientific computation, and the teaching of mathematics and other subjects. This book describes the IBM Personal Computer APL system.

The book consists of two parts. Part 1, "Operation Guide," has four chapters that describe the use of the system. Part 2, "APL Reference Guide," has eight chapters that explain, in detail, the APL language as implemented on the IBM Personal Computer. Part 1 assumes some familiarity with computing, as well as a basic knowledge of APL which can be obtained from Part 2.

Chapter 1, "Introduction," describes the use of the IBM Personal Computer devices under APL, and should be read by all who use the system. The chapter contains information about the use of diskette drives and fixed disks, the keyboard, display monitors, and the IBM Graphics Printer. This chapter also introduces the APL character set, and describes the full-screen APL input editor.

Chapter 2, "Application Workspaces," describes a number of functions provided with the system that allow a person with a basic understanding of APL to use the IBM Personal Computer's devices. A set of workspaces has been provided to help you with the following operations:

- Printer management
- Full-screen editing of APL defined functions
- Disk Operating System file management
- Asynchronous communications with an IBM System/370 host
- Music samples

Chapter 3, "Auxiliary Processors," describes, in detail, the auxiliary processors provided with the system, which allow a knowledgeable APL user to create specific application workspaces. The following auxiliary processors have been included to help you obtain greater control of the IBM Personal Computer's devices:

- AP80: IBM Graphics Printer Control
- AP100: BIOS/DOS interrupt handling
- AP205: Full-screen display management
- AP210: DOS file management
- AP232: Asynchronous communications
- AP440: Music generator

Chapter 4, "How to Build an Auxiliary Processor," describes auxiliary processors that can exchange information with APL. A person who has both a good knowledge of APL and the ability to program in the IBM Macro Assembler language can use this chapter to help produce his own auxiliary processor to perform a specific application not currently supported by the APL system.

Chapter 5, "Using APL," describes the major characteristics of the APL language.

Chapter 6, "Fundamentals," shows some typical APL statements, lists error messages with recommended corrective actions, describes the APL character set, and explains the terms used in APL.

Chapter 7, "Primitive Functions and Operators," describes the scalar and mixed primitive functions of APL and the operators that may be used with them.

Chapter 8, "System Functions and System Variables," describes the interaction between the APL language and the system implementing it.

Chapter 9, "Shared Variables," describes the interface that allows two independently operating processors to exchange information.

Chapter 10, "Function Definition," describes the various methods by which a defined function can be established in an APL workspace. Ambi-valent functions, localization of names, branching, labels, comments, and the *del* (∇) function editor are all discussed in this chapter.

Chapter 11, "Function Execution," describes the execution of defined functions, including the related topics of halted execution, the state indicator, stop control, trace control, locked functions, recursive functions, and console input and output.

Chapter 12, "System Commands," describes the various APL system commands that are used to control the work session and to manage workspaces.

Three appendixes, "Alt Codes and Associated Characters," "Printer Control Codes," and "Internal Representation of Displayed Characters" are also included.

Notes:

Contents

Part 2. APL Reference Guide

Notes:

IBM

Personal Computer Computer Language Series

Part 1 Operation Guide

Part 1. Operation Guide

Chapter 1. Introduction

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 printer or video display. 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 would have forced you to be concerned as much with the internal requirements of the machine as with your own statement of your problem. APL takes care of those internal considerations automatically.

A programming language needs 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 <u>one</u>, 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 Center. 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 Center. 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 a 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 all 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 APL object not currently 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 minimize 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 is 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 of 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 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.

- Multiple display monitors can be used, with dynamic switching between the following modes:

  — 40-Column Color/Graphics

  — 80-Column Color/Graphics

  — 80-Column Monochrome (non-APL characters)

- 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 the 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 or the IBM Personal Computer XT

- 128K of random access memory

- The IBM Personal Computer Math Co-Processor

- One diskette drive

- The IBM Color/Graphics Adapter, to generate both APL and non-APL characters

- The IBM Color Monitor or other monitor that functions with the Color/Graphics Adapter, or a television set and RF modulator. (Television sets and RF modulators are not sold by IBM.)

- Optional IBM 80 CPS Graphics Printer, with either the Parallel Printer Adapter or the Monochrome Display and Printer Adapter.

- Optional Monochrome Display and Printer Adapter with the Monochrome Display, to display ASCII (non-APL) characters

- Optional IBM Personal Computer Expansion Unit

- Optional IBM Asynchronous Communications Adapter

APL comes to you on a diskette, so you have to load it into memory before you can use it. You should read this entire chapter before trying to use the APL system.

Backing Up Your Diskette

Because you have only one copy of the APL system, you should *back it up* 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 diskette properly in a place where you can find them if you need to. Use the backup diskettes for everyday operations.

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

Before You Begin

You will need these diskettes:

- The diskette you want to back up–we're going to call this your *original* diskette. You may also see it called the *source* diskette.
- The diskette that will become the backup diskette. Other names for this diskette are *target* or *destination* diskette.
- 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 diskette. You may remove the tab when the backup diskette has 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.

   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)?
   ```

7. Type

   n

   You don't have to press the Enter key.

8. The DOS prompt, A>, is displayed. Remove the backup diskette from the drive. With a felt-tip pen, mark the label 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:

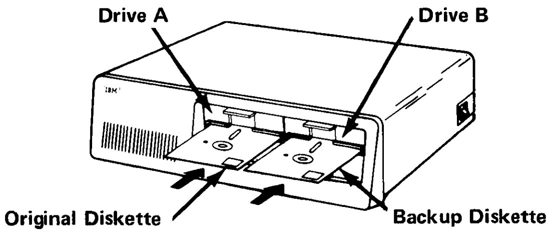
   and press the Enter key. The following message appears:

   ```
   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.

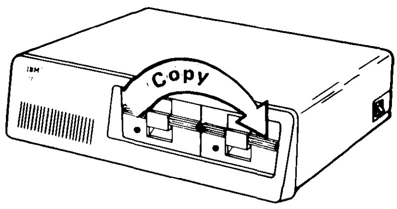


7. Press any key.

   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.

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)?
   ```

9. Remove your original diskette from drive A, and insert your DOS diskette.

10. Type

    n

    and press the Enter key. The DOS prompt, A>, appears.

11. Remove both diskettes.

    Use a felt-tip pen to label and date the backup diskette. You may also want to write "Backup" on the label as a reminder that this is a copy of another diskette.

Installing APL On Your Fixed Disk

If you have an IBM Personal Computer XT or an IBM Personal Computer Expansion Unit, 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 diskette in drive A.

   b. Type

      A:FDTRANS

      and press the Enter key.

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 main directory on your fixed disk to make it easy for you to start APL.

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 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 main directory to be invoked.

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

```
IBM PERSONAL COMPUTER   A P L
Version 1.00 (C) Copyright IBM Corp. 1983
Produced by IBM Madrid Scientific Center
```

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 [EXAPL] [APx] [APy] ... [APz] [APL system command]
```

where the maximum number of names given after APL, excluding the field, "APL system command," is six.

- EXAPL is the filename specification of the program that has the dyadic formats (numeric format and picture format). For more information, see Chapter 7.

- APx, APy, and APz 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 4). The following auxiliary processors are included with the system:

— AP80:	IBM Graphics Printer control
— AP100:	BIOS/DOS interrupt handling
— AP205:	Full-screen display management
— AP210:	DOS file management
— AP232:	Asynchronous communications
— AP440:	Music generator

- "APL system command" means that you can type 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 12.) 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 EXAPL or some auxiliary processor support, or 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 actively shared with an auxiliary processor will be automatically retracted upon exit from the APL system.

Examples:

```
APL AP210 AP100 )LOAD WORKSP
```

This starts APL and auxiliary processors AP210 and AP100. Also, the workspace called **WORKSP** is loaded. The Graphics Printer will not be available.

```
APL EXAPL AP80
```

This will start APL with the dyadic formats in **EXAPL**. The printer is available.

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 underlined.
- Numeric, which consists of the digits 0 through 9.
- Special APL characters (see Figure 3 in Chapter 6).
- 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 "Special Key Combinations" section.

The keyboard consists of three general areas:

- Function keys, labeled 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 currently supported by the APL system are:

- *Alt-F1*: switch to the Monochrome Display mode, with 80 characters per line.
- *Alt-F4*: switch to the Color Graphics mode, with 40 characters per line.
- *Alt-F8*: switch to the Color Graphics mode, with 80 characters per line.

Typewriter Keyboard

The middle area of the keyboard behaves much like a standard typewriter. Under APL, the capitalized Roman alphabet and the numbers 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 numbers 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 *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. The key also is used to halt a request for literal input from a defined function.

Pressing the Esc key twice generates a *strong interrupt* that will cause an execution within a statement to halt as soon as the interrupt is detected.

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.

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 without erasing what has been typed. Under DOS or BASIC, characters are erased during backspacing, but the APL backspace is *non-destructive*.

PrtSc (Print Screen): Just below the Enter key is a key labeled with *PrtSc* and *. If the National character set is active, pressing this key generates an asterisk; if the APL character set is active, a not-equal sign (≠) is generated. 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, 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 the IBM Graphics Printer attached to your system and you loaded the printer auxiliary processor, AP80, at the start of the APL work session.

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. 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 code for the desired character (see Appendix A).

The Ctrl key is similarly used to generate certain codes and characters not otherwise available from the keyboard. The Ctrl-Backspace combination is used to switch between the APL and National character-set mappings.

Numeric Keypad

This area of the keyboard is normally used in conjunction with the APL Input Editor, which is described later in this chapter. 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 operation keys that are normally used with the numeric keypad. When the National character set is active, these keys generate a + (representing addition), and a − (representing subtraction). With the APL character set active, however, these keys generate a ÷ (division) and a + (addition).

Special Key Combinations

You should be aware of the special functions of the folowing 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.

- *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:* This combination 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").

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

Decals with the APL character set for the IBM Personal Computer keyboard have been included with this book and are in the plastic sleeve inside the back cover. Place the decals on the keytops as shown in Figure 1. Notice that the alternate-shift characters ⌶, ⍒, ⍋, ⍋, ⌽, ⍉, ⊖, ⍟, ⍱, ⍲, !, and ⌹ go on the fronts of the keys along the top row.

Figure 2 shows the keyboard with the APL character set.

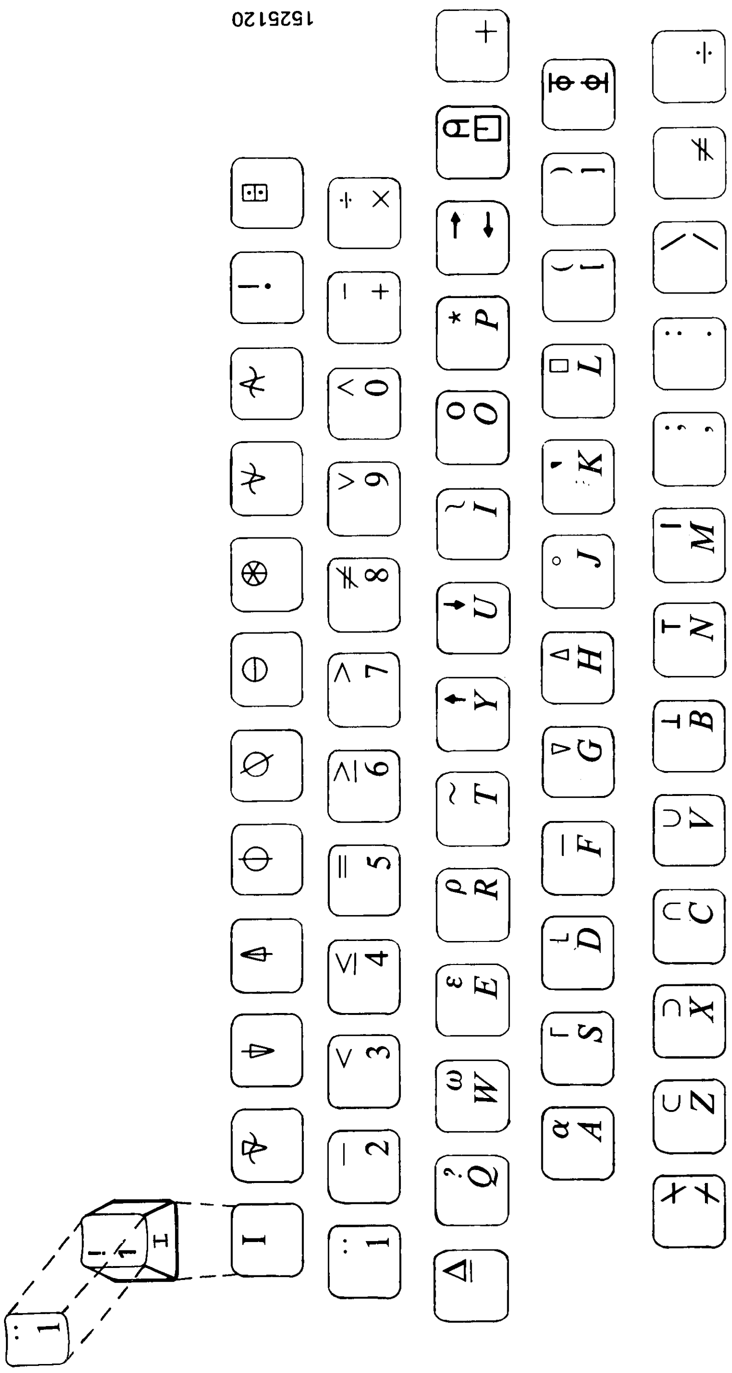


Figure 1. The APL Decals

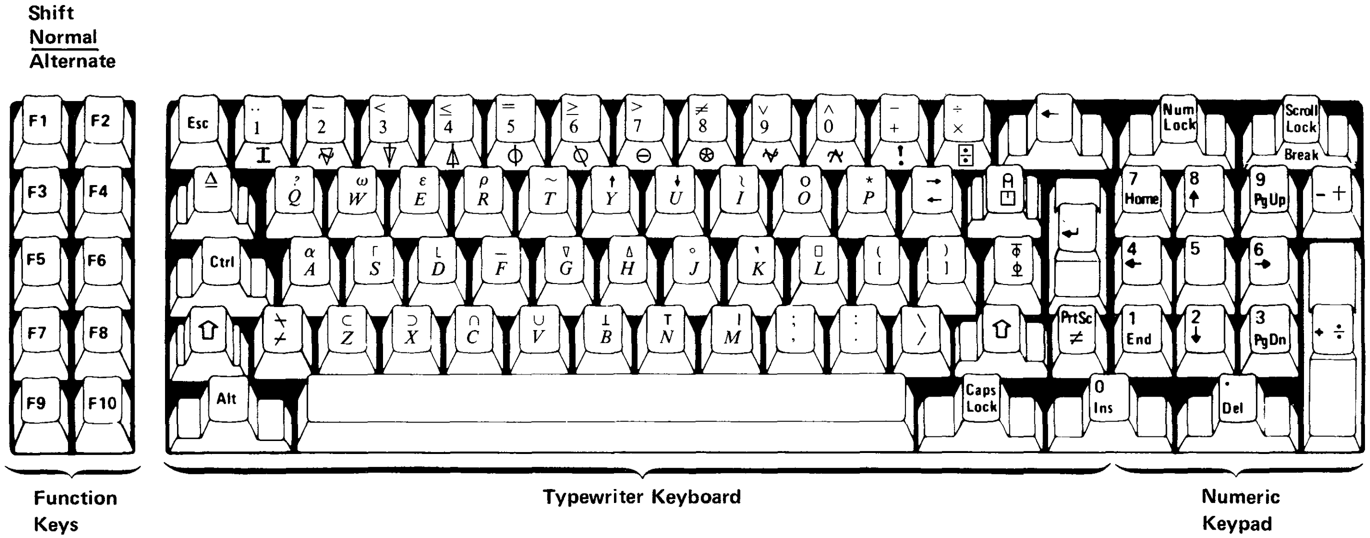


Figure 2. Keyboard with APL Character Set

Use of Displays

APL enables you to work, sequentially, in the three following modes during the same working session: Monochrome Display, Color/Graphics Adapter with 40 characters per line, and Color/Graphics Adapter with 80 characters per line. Only the Color/Graphics Adapter modes support APL characters. You may use the Monochrome Display mode; however, some APL characters will not be displayed. Instead, the corresponding IBM Personal Computer ASCII characters will be displayed.

> **Note:** "Sequentially" means that at any time during the work session, you can change modes without leaving APL.

At load time, the configuration you are in is maintained. If you have both a Monochrome and Printer Adapter, and a Color/Graphics Adapter, the Color Graphics mode with 40 characters per line is activated.

If you want to change to the Monochrome Display mode, press the Alt and F1 keys at the same time. To switch to the Color Graphics mode with 80 characters per line, press the Alt and F8 keys. To switch to the Color Graphics mode with 40 characters per line, press the Alt and F4 keys. APL does not allow you to switch to a monitor that is not available.

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

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

The mode you switch to when you press an Alt-F key combination, is as follows:

Your Configuration	Alt-F1	Alt-F4	Alt-F8
Color 80	Color 80	Color 40	Color 80
Color 40	Color 40	Color 40	Color 40
Monochrome	Monochrome	Monochrome	Monochrome
Monochrome & Color	Monochrome	Color 40	Color 80

Note: The APL system detects the IBM Personal Computer configuration reflected in the switch settings (see the *Technical Reference* manual). If your actual configuration is different (for example, you forgot to switch on your Color Monitor), the system may switch to a monitor that is not operating, and you will not be able to see anything you type, although it can be executed if you press the Enter key (more about this later). If this happens, the best action is to return to the active monitor 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 identification number:

Device	DOS Drive Spec.	APL Library
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 12) 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 3, and the **FILE** workspace, which is discussed in Chapter 2.

The Printer

The optional IBM Graphics Printer 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. 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. If you are using the IBM Monochrome Display, the untranslated ASCII characters displayed on the Screen will be printed as their APL equivalents.

- *Ctrl-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 3 discusses, in detail, the use of AP80 to control the printer with a shared variable, and Chapter 2 explains the use of the **PRINT** workspace. Control codes can be sent to the printer, but they will not affect the APL special characters.

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 underline or box 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, it is echoed below the last entered line and executed. The input editor also can be used within the *del* (∇) editor during function definition (see Chapter 10) to help create or modify programs.

A full-screen, defined-function editor is included with the EDIT workspace and is described in Chapter 2. 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.

- End (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 covers the character position on the Monochrome Display, or blinks twice as fast as normal on the Color Graphics monitor. 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 the 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, because the APL backspace is non-destructive.

- **Esc:** When pressed anywhere in a line, Esc causes the message **INTERRUPT** to be written, and the entire line is ignored. The line is not passed to APL for processing. If you press Esc once while a defined APL function is executing (see Chapter 11), the function is interrupted after the current line is executed. This is called a *weak interrupt*. If you press Esc more than once while a function is executing, the function stops executing as soon as the interrupt is detected. This is called a *strong interrupt*.

- ⇥ (Tab): Treated the same as a blank character.

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 the 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, Backspace, 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 Cursor-Right 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. 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. Use the Cursor-Right 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 Cursor-Right 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 the Enter key.

Erasing part of a Line: To end a line at the current cursor position, press the End key. Then you can continue typing.

Canceling 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 not passed to APL for processing.

Chapter 2. Application Workspaces

Notes:

The APL diskette has a number of *workspaces* in transfer form (extension .AIO). These workspaces contain functions that you can call from your programs to perform the following applications:

- Using the printer from APL programs (**PRINT**)
- Using the APL full-screen function editor (**EDIT**)
- Using DOS file management routines (**FILE**)
- Uploading and downloading files (**VM232** and **FILE**)
- Using samples for the music auxiliary processor (**MUSIC**)

These functions also can be used as examples of how to program with the corresponding auxiliary processors in the IBM Personal Computer APL system:

- **PRINT** uses AP80
- **EDIT** uses AP205
- **FILE** uses AP210
- VM232 uses FILE, AP232, and AP210
- **MUSIC** uses AP440

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. For example,

APL AP80

To copy the **PRINT** workspace, you must enter:

)IN PRINT

This command will load two functions (**PRINT** and **FONTS** into your active workspace.

The **PRINT** function 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.

PRINT may be called from any other APL-defined function, thus giving the program control of the printer.

The following examples show what is printed for various entries.

Entry	**Printed**
`PRINT 2+2`	4
`PRINT 'ABCabc'`	ABCabc
`PRINT ⍳10`	1 2 3 4 5 6 7 8 9 10
`PRINT 2 3⍴'ABCDEF'`	ABC 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 ⎕AV[⎕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 emphasized printing, large character sizes, and other special printing functions. Appendix B shows many of these control codes, and their functions.

The **FONTS** function contains several examples that use the printer's alphameric mode to send control commands to the printer.

If the first character in the string is not ⎕AV[⎕IO+255], the whole string is printed as it is. Therefore, a single character can have a dual function, depending on the selected printing mode.

The EDIT Workspace

The **EDIT** workspace provides an APL full-screen, defined-function, editor. It is used with AP205.

To use the **EDIT** functions from APL programs, you must include the full-screen auxiliary processor, AP205, as a parameter to the APL command at load time before you begin an APL work session. For example,

APL AP205

When APL becomes ready, you must copy the **EDIT** workspace into your active workspace by entering:

)IN EDIT

You can use the full-screen function editor in either 40- or 80-column mode.

If, for example, the name of the function you want to create or edit is **FN1**, and you have an 80-column display, enter the following line:

```
EDIT 'FN1'
```

If you have a 40-column display, enter:

```
40 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 charaacters (with the Ins key), delete characters (with the Del key), delete to the end of a line (with the End key), 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	Displays the top page of the function (**TOP**).
F2	Displays the bottom page of the function (**BOT**).
F3	Ends function definition. All your modifications to the function are kept (**END**).
F4	Clears the screen and displays only the current line. You can use this key to edit lines longer than the screen width. The maximum line length this method allows is 160 characters (**LIN**).
F5	Inserts five empty lines after the current line (**INS**).

F6	Copies a line: First move the cursor to the line you want to be copied, then press F6. An asterisk (*) will be displayed in the lowest line of the screen, by the F6 key indicator. The system is now in "copy" state. Then move the cursor to the line after which the preceding line is to be copied (possibly on another page). And finally, press F6 again. The asterisk is erased and the system is no longer in "copy" state (**COP**).
F7	Executes the current line (**XEC**).
F8	Brings the cursor to the end of the current line (**EOL**).
F20	(Shift-F10) Cancels function definition. No changes are kept. The function remains as it was at the beginning of the session (**CAN**).

All other function keys are ignored.

OTHER SPECIAL KEYS

Tab	Moves the cursor to beginning of the next line.
PgDn	Displays the next page.
PgUp	Displays the preceding page.
Esc	Restores the state the current page had when the last special key was pressed (special keys are Esc, Enter, PgUp, PgDn, or any F key). Esc also clears the "copy" state. All changes made after a special key was pressed are lost.
Enter	Executes APL statements (excluding system commands) that are typed in the topmost unnumbered line of the screen. If this line contains only a number (for example, 24), the page starting at that line will be displayed. If the Enter key is pressed while the cursor is on any other line, the cursor will move to the topmost line.

To delete a line, simply move the cursor to the beginning of that line and press the End key. 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 End key.

Locked or halted functions cannot be changed with this editor. This full-screen function editor can be used to create new defined functions and modify existing ones.

The FILE Workspace

The FILE workspace has been designed to help you work DOS files, and allows either sequential or random access. It uses the 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.

To use the *FILE* functions 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. For example,

```
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**, has the functions for manipulating DOS files, including:

- WOPEN
- OPEN
- CLOSE
- READ
- READV
- WRITE
- DELETE
- RENAME

Other functions in this file that are used for related purposes are:

- PATCH
- IN
- PIN
- OUT
- COMPARE
- TYPE
- TYPEV

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 necesarily the same number of bytes), but the header fills a part of that area. (See Chapter 4, "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. All records must be equal to the selected record length, with each character occupying one byte.

file_number 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:]filename[.ext]

Note: If the message, **I/O ERROR**, appears when you are trying to access a file, either the door of the drive is open, the incorrect diskette is inserted, or the diskette is write-protected. See Figure 17 in Chapter 12 for the recommended action.

WARNING: Changing diskettes during an input/output operation, or when you have open files, may damage your diskette.

WOPEN

This function opens a DOS data file for reading or writing, with sequential or random access. Up to four files may be open at one time. (See "READ" and "WRITE" for descriptions of access methods.)

The syntax of the function is:

file_number WOPEN 'filespec [,code]'

If no file by that name has been previously created a new file is created.

OPEN

This function opens a DOS data file for read-only, with sequential or random access. Up to four files may be open at one time. (See "READ" and "WRITE" for descriptions of access methods.)

file_number OPEN 'filespec [,code]'

If no file by that name has been previously created an error will result – error 255. (See "AP210: The File Auxiliary Processor" in Chapter 3 for a listing of all return codes.)

READ

This function reads a DOS data file, sequentially or randomly, that was opened using *(W)OPEN*. The syntax is:

READ file_number [record_number [record_size]]

$0 \leq$ record_number ≤ 32767

$0 <$ record_size ≤ 2048

file_number matches the number that you specified in *(W)OPEN*ing the file.

If no **record_number** 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* and *WRITE* functions work from the same access point, meaning that the access point is advanced sequentially to the next record each time either of these commands is issued.

Random access is designated by specifying a particular record. **Record_size** can only be specified when using random-access method. If the **record_size** is not specified, the default is the **record_size** specified in the previous operation. If the **record_size** is not specified on the first *READ* or *WRITE*, the default is 128 bytes.

READV

This function sequentially reads a variable-length record DOS character file that was previously opened using *(W)OPEN*. The syntax is:

READV file_number

The **file_number** matches the number that you defined in *(W)OPEN*ing the file.

WRITE

This function writes to a DOS data file, either sequentially or randomly, that was previously opened using *WOPEN*. (Trying to *WRITE* to an un*WOPEN*ed file will result in error 24; see "AP210: The File Auxiliary Processor" for a listing of all 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_number [rec_num [rec_size]] WRITE APL_obj

0 ≤ **rec_num** ≤ 32767

0 < **rec_** ≤ 2048

file_number matches the number that you arbitrarily defined in *WOPEN*ing the file.

If the **record_number** is not 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 *WOPEN*; the second record (record 1) will be accessed next, and so on. The *READ* and *WRITE* functions work from the same access point, meaning that the access point is advanced sequentially to the next record each time either of these commands is issued.

Random access is designated by specifying a particular record. If the **record_size** is not specified, the default is the **record_size** specified on the previous *READ* or *WRITE*.

If the **record_size** is not specified on the first *READ* or *WRITE*, the default is 128 bytes.

CLOSE

This function closes a file that was previously opened using *(W)OPEN*. The previously assigned file_number is available for reuse. (*(W)OPEN*ing a **file_number** without having closed the corresponding file will cause the file to be automatically closed and reopened.)

The syntax for *CLOSE* is:

CLOSE file_number

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. If a valid drive is specified in the left argument, the drive is ignored. The syntax is:

'new_filespec' RENAME 'old_filespec'

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 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)
```

This function imitates the *)IN* command (see Chapter 12) 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. 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 mentioned 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 3ρ'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 (with the restriction mentioned above), 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. 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 mentioned 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:

  ```
  VAR←7
  (2 3ρ'FUNVAR') PIN 'MYFILE'
  ```

  The left argument of the *PIN* 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* is 1 0.

OUT

This function imitates the *)OUT* command (see Chapter 12) 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. 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 mentioned 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 3⍴'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 imitates the DOS *TYPE* command. The syntax is:

record__size [n] TYPE 'filespec [,code]'

record__size is the length of the record, the *N* first characters of which are to be typed. The file with the indicated **filespec** is displayed at the terminal. If *N* is not given, the full **record__size** is typed.

TYPEV

This function imitates the DOS *TYPE* command for variable record length character files. The syntax is:

TYPEV 'filespec'

Examples of Use

Following are examples of using the various DOS file-handling functions.

`1 WOPEN 'FILE.EXT'`	Creates a new file. Records will contain APL objects with header (default code).
`1 WRITE ⍳10`	First record will be a vector of elements from 1 to 10 (origin 1). Default **record_number** is 0; default **record_size** is 128 bytes.
`1 WRITE 2 3⍴⍳6`	A matrix of two rows and three columns, of elements from 1 to 6, is written sequentially to the file (origin 1).
`CLOSE 1`	The file is closed.
`1 OPEN 'FILE.EXT'`	Open the same file for read-only operation.
`READ 1 1`	Read the second record first.

`1 2 3`
`4 5 6`
	Here is the matrix

`READ 1 0`	Now ask for the first record.
`1 2 3 4 5 6 7 8 9 10`	A vector of integers.
`CLOSE 1`	Close the file.
`DELETE 'FILE.EXT'`	Delete the file.

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 full 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. 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" later in this chapter).

- Menu items 2 and 3 are listed for future use.

- Menu item 4 ("Other") lets you specify pertinent parameters to define your own terminal (see "User Specified Terminal" later in this chapter).

- 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" later in this chapter. 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 your 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 default 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."

- **Parity:** See "VM/370 Terminal."

- **No. of stop bits:** Stop bits are sent by your IBM Personal Computer after each character to keep the line in synchronization. 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:** Recognized by the host computer as the "end-of-line" designator. To change this character, type 5 in the *PARAMETER CHANGE* menu and press the Enter key. For more information, see "VM/370 Terminals."

- **Delete chars:** When you are in communication with the host computer, the host may transmit characters you do not want displayed on your screen. Generally these are special ASCII characters knows 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. Pressing Enter returns you 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. Pressing Enter returns you 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 12 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 if using the commands:

```
)CLEAR
)IN  name
```

where **name** is the name you used when you saved the application with the *)OUT* command (see Chapter 12 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 twice 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 twice. 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, one of 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 twice 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 is passed to the host for execution. There is no transmission transparency yet: 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 (⌶) are executed by the IBM Personal Computer APL 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 (⌶) 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, 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 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 (⌶). The functions are:

- UPLOAD
- DOWNLOAD
- APLOUT
- APLIN

These functions assume that:

- Transmission has been established.
- The host VM/370 system contains the file, EDIT EXEC, as described in the "Auxiliary Files on the Host" section.
- The host VM/370 system contains an APL EXEC file to load VSAPL.
- The host VM/370 system contains the APL workspace, OUT, as described in the "Auxiliary Files on the Host" section.

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 carriage-return 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 the combination of carriage return 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 FILENAME**). 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 filename is requested (**ENTER TARGET FILE NAME**). It must be given in Conversational Monitor System (CMS) format: *filename filetype filemode*. If the target filename 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 as *UPLOAD*. It sends a file from a minidisk in the host to a disk(ette) in your IBM Personal Computer.

> **Note:** If you download a file that has the APL character "→", you will not be able to edit it with the standard DOS editors, because they interpret that symbol as an end-of-file character.

The transmission protocol does not allow a file to be downloaded if the name of the file includes any of the following characters: @, #, and $.

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 appropriate_size
)COPY OUT IN
''IN'filename'
)ERASE IN
)SAVE appropriate_name
```

APLIN

Performs the opposite operation as *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.

> **Note:** If you download a file that has the APL character " → ", you will not be able to edit it with the standard DOS editors, because they interpret that symbol as an end-of-file character.

The transmission protocol does not allow a file to be downloaded if the name of the file includes any of the following characters: @, #, and $.

The correspondence between the alphabetic characters on the IBM Personal Computer and the VM/370 system is as follows:

Function	Capitals	Lower Case	Caps Underlined
Upload to 370	Capitals	Lower Case	N/A
Download from 370	Capitals	Lower Case	Special Characters
APLIN from 370	Capitals	Lower Case	Lower Case
APLOUT to 370	Capitals	Caps Underlined	N/A

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],'→'
      ⍴A
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*, to establish the characteristics of the type of terminal your IBM Personal Computer will simulate, and the detailed features of that terminal.

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, 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 twice 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:
 ********
 HHHHHHHH
 SSSSSSSS
password
```

Connection messages are received here. You may now IPL CMS.

```
CMS

I UPLOAD                    ⍝ Request for the sending program
ENTER SOURCE FILE NAME      ⍝ Prompt from UPLOAD function
B:TEST                      ⍝ Our answer
ENTER TARGET FILE NAME      ⍝ Prompt fromUPLOAD function
TEST TEST A                 ⍝ Our answer
FILE EXISTS. DO YOU WANT TO REPLACE?
                            ⍝ Prompt from UPLOAD function
Y                           ⍝ Our answer
END OF TRANSMISSION         ⍝ From this point, the system
3 RECORDS SENT              ⍝ automatically generates a
APL                         ⍝ set of lines that assure
                            ⍝ transparency of the
         V S  A P L  4.0    ⍝ transmission.

CLEAR WS

)LOAD OUT

SAVED  10:13:47  02/01/83

CMSIN'TEST TEST A'

RO;

)OFF HOLD

R;

ERASE TEST HIO A

R;
```

At this point, uploading is complete and the terminal opens. You are again connected to CMS.

```
TYPE TEST TEST A             ⍝ CMS command typed by the user

FIRST LINE                   ⍝ System answer
SECOND LINE
LAST LINE

R;
⌶ DOWNLOAD                   ⍝ Request for a DOWNLOAD
ENTER TARGET FILE NAME       ⍝ Prompt from DOWNLOAD function
B:TEST1                      ⍝ Our answer
ENTER SOURCE FILE NAME       ⍝ Prompt from DOWNLOAD function
TEST TEST A                  ⍝ Our answer
                             ⍝ We are sending back the file
APL                          ⍝ The 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 are sent in blocks of 10.
                             ⍝ Therefore, the number given is
                             ⍝ rounded up to a multiple of 10.
                             ⍝ We are again under CMS.

⌶                            ⍝ A single i-beam followed by Enter
                             ⍝ is a request to return to IBM
                             ⍝ Personal Computer APL.
         TYPEV'B:TEST1'
FIRST LINE
SECOND LINE
LAST LINE
       TERMINAL              ⍝ We go back to terminal state.
                             ⍝ Startup messages are received here.
Q PRT                        ⍝ This is a CMS command.

NO PRT FILES
R;

LOG

CONNECT= 00:12:10 VIRTCPU= 000:02.56 TOTCPU= 000:11.05
LOGOFF AT 10:47:08 EST TUESDAY 02/01/83

VM/370 ONLINE
```

The VM/370 system is now in *receive* state. To return to IBM Personal Computer APL, you have to press Esc and then →.

The terminal opens now, and you are back in IBM Personal Computer APL.

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 underlined letters are replaced by lowercase letters.
APLIN	Like CMSIN, but lowercase letters are replaced by underlined letters.

- Auxiliary to *EXPORT/IMPORT:*

CIN	Used by CMSIN, APLIN
COUT	Used by CMSOUT, APLOUT
GASC	Used by CMSIN, APLIN, CMSOUT, APLOUT
XUL	Used by APLOUT
XUL1	Used by APLIN
CMS	Used by CMSIN, APLIN, CMSOUT, APLOUT

- *IN/OUT* Functions:

IN	Equivalent to the)IN command (see "The FILE Workspace").
OUT	Equivalent to the)OUT command (see "The FILE Workspace").

In the function listings on the following pages, some non-APL characters are included. These characters have been highlighted to indicate that they must be generated from a 327X terminal in "APL OFF" mode. The only functions containing these non-APL characters are: CIN, COUT, GASC, XUL, and XUL1.

Functions

APLIN	APLOUT	CIN	CMS	CMSIN	CMSOUT
COUT	GASC	IN	OUT	XUL	XUL1

(14)

```
    ∇ 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'
    ∇
```

(14)

```
    ∇ APLOUT X;A;B;SH;N;I;⎕IO;ASC;N1;N2;AA
[1]   ⎕IO←0
[2]   N←X,' AIO(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=⍴A←N)/0
[12]  SH←COUT XUL A
[13]  →L
[14] E:'ERROR'
    ∇
```

(6)

```
     ∇ Z←CIN X;⎕IO;I;J
[1]    ⎕IO←0
[2]    X←(I←Z∊'"_')/⍳⍴Z←X,((¯1↑X)∊'"_')/' '
[3]    X←(N1,N2)[((⍴N1)×Z[X]='_')+ASC⍳Z[X+1]]
[4]    Z←(~I∨J←¯1⌽I)/Z
[5]    Z←(~I←(~J)/I)\Z
[6]    Z[I/⍳⍴I]←X
     ∇
```

(4)

```
     ∇ CMS X;CP;I
[1]    CP←'CMS'
[2]    I←100 ⎕SVO 'CP'
[3]    CP←X
[4]    'R',(⍕CP),';'
     ∇
```

(14)

```
     ∇ CMSIN X;A;SH;N;I;⎕IO;ASC;N1;N2;AUX
[1]    ⎕IO←0
[2]    N←((X⍳' ')↑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=⍴A←N)/0
[12]   SH←CIN A
[13]   →L
[14] E:'ERROR'
     ∇
```

(14)

```
     ∇ CMSOUT X;A;B;SH;N;I;⎕IO;ASC;N1;N2;AA
[1]    ⎕IO←0
[2]    N←(X←(X⍳' ')↑X),((X⍳' ')↓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=⍴A←N)/0
[12]   SH←COUT A
[13]   →L
[14] E:'ERROR'
     ∇
```

(10)

```
     ∇ Z←COUT X;I;J;⎕IO
[1]    ⎕IO←0
[2]    Z←,X
[3]    X←~Z∊(~ASC∊⎕AV[23 30])/ASC
[4]    I←,⍉(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
     ∇
```

(11)

```
    ∇ GASC;⎕IO
[1]   ⎕IO←0
[2]   ASC←' ',⎕AV[23],'#$%&''()*+,-./0123456789:;<=>
      ?@ABCDEFGHIJKLMNOPQRSTUVWXYZ'
[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 22 24 25 228
       233 237 238 31]
[6]   N1←N1,⎕AV[15 14],'AΔ',⎕AV[(1+⍳8),(16+⍳4),21,
      (26+⍳4),(32+⍳3),131]
[7]   N1←N1,'W',⎕AV[30],'XYZΔ¯IO¨∇⍫αω∩∪⊂⊃⌶×÷○⊛⌈⌊|
      ∧∨⍲⍱≤≥'
[8]   N2←'≠!ρ⍳∊⊥⊤⌽⍉⊖⌿\⍀⌹⍋⍒↑↓→←⎕⍞⍎⍕∘[⍝',⎕TC[0 2 1]
[9]   N2←N2,'UMN',⎕AV[240 239],'QD',⎕AV[241],'FBGHJ',
      ⎕AV[2 0],'CK',⎕AV[219 220]
[10]  N2←N2,'L',⎕AV[222 223 242],'PRST',⎕AV[243],
      'V',⎕AV[(203+⍳4), 235 236]
[11]  N2←N2,⎕AV[(207+⍳12), 221 250 23 252 253 254 255
       132 13]
    ∇
```

(33)

```
    ∇ 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=⍴LS)/ZXL2
[7]   ZR←(1↑⍴LS)⍴0
[8]  ZXL2:ZXA←''
[9]  ZXL3:→(0=⍴ZX1←SH1)/0
[10]  ZXA←ZXA,1↓¯8 0[1+(1↑ZX1)∊'CE']↓ZX1
[11]  →((1↑ZX1)∊' C')/ZXL3
[12]  ZXM←(ZXA⍳' ')↓ZXA
[13]  ZXA←(ZXA⍳' ')↑ZXA
[14]  →(0=⍴LS)/ZXL1
[15]  →((⍴ZXA)>2+1↓⍴LS)/ZXL2
[16]  →(~1∊LS∧.=(1↓⍴LS)↑1↓ZXA)/ZXL2
[17] ZXL1:ZX1←⍎(ZXM⍳' ')↑ZXM
```

```
[18]  ZXM←(ZXM⍳' ')↓ZXM
[19]  ZX2←⍳0
[20] ZXL4:→(ZX1=0)/ZXL5
[21]  XZ2←ZX2,⍎(ZXM⍳' ')↑ZXM
[22]  ZXM←(ZXM⍳' ')↓ZXM
[23]  →ZXL4,ZX1←ZX1-1
[24] ZXL5:→('FC'=1↑ZXA)/ZXL6,ZXL7
[25]  ⍎(1↓ZXA),'←',(⍕ZX2),((0≠⍴ZX2)/'⍴'),ZXM,(0≠⍴ZX2)/' 0'
[26]  →ZXL8
[27] ZXL6:ZX1←' '=0\0⍴⎕FX ZX2⍴ZXM
[28]  →ZXL9
[29] ZXL7:⍎(1↓ZXA),'←',(⍕ZX2),((0≠⍴ZX2)/'⍴'),'ZXM'
[30] ZXL8:ZX1←1
[31] ZXL9:→(0∊⍴LS)/ZXL2
[32]  ZR[(LS∧.=(1↓⍴LS)↑1↓ZXA)⍳1]←ZX1
[33]  →ZXL2
    ∇
```

```
      ∇ ZR←ZXM OUT SH1;ZXA;ZX1;⎕PP;CMS
[1]    ⎕PP←16
[2]    →(~0∊ρZXM)/ZXL1
[3]    ZXM←(∧/ZXM∨.≠((¯1↑ρZXM),3)↑⍉ 3 3 ρ'OUTSH1ZXM')⌿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↑⍎'0',
       (2=⎕NC ZXM[⎕IO;])/',⍎''0=0\0ρ',ZXM[⎕IO;],'''']
[7]    ZX1←⍎(ZX1/'⎕CR'''),ZXM[⎕IO;],ZX1/''''
[8]    ZXA←ZXA,((' '≠ZXM[⎕IO;])/ZXM[⎕IO;]),' ',(⍕(ρρZX1),
       ρZX1),' ',,⍕ZX1
[9]    ZXA←(((¯1+1↑ρZXA)ρ' '),'X'),ZXA←(ZX1,79)↑((ZX1←⌈
       (ρZXA)÷71),71)ρZXA,71ρ' '
[10]   →(2=⎕SVO 'SH1')/ZXL2
[11]   CMS←'CMS'
[12]   ⎕WA←100 ⎕SVO 'CMS'
[13]   CMS←'ERASE ',SH1,' AIO'
[14]   'R',(⍕CMS),';'
[15]   SH1←SH1,' AIO(192 FIX'
[16]   ⎕WA←110 ⎕SVO 'SH1'
[17]   ⍎(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
    ∇
```

(2)

```
    ∇ R←XUL X;I;J
[1]     J←26≠I←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'ιR←,X
[2]     R[J/ιρR]←'abcdefghijklmnopqrstuvwxyz'[J/I]
    ∇
```

(2)

```
    ∇ R←XUL1 X;I;J
[1]     J←26≠I←'abcdefghijklmnopqrstuvwxyz'ιR←,X
[2]     R[J/ιρR]←'ABCDEFGHIJKLMNOPQRSTUVWXYZ'[J/I]
    ∇
```

Typical APL EXEC File

Your host system also needs a file called *APL EXEC* from which you can access APL. The content of this file is system-dependent. An example of a typical *APL EXEC* file follows.

```
&TRACE OFF
&IF .&FILEMODE EQ .S2 &SKIP 3
&IF .&FILEMODE EQ .Y2 &SKIP 2
&TYPE * APL DISK must be accessed as Y/S or Z/Y *
&EXIT 99
CONTYPE
&TERMCLA = &PIECE OF &RETCODE 1 2
&TERMTYP = &PIECE OF &RETCODE 3 2
&TERM = GRAF
&IF &TERMCLA EQ 80 &IF &TERMTYP NE 80
  &TERM = LINE
&IF &TERM = LINE CP TERM ATTN OFF
&IF &TERM = LINE CP SET LINEDIT OFF
CP TERM APL ON
* Remove FI for APLDUMP if dumps not wanted
*   You will get one every time if GDDM not installed
FI APLDUMP PRINTER
CP SET EMSG OFF
CP SET IMSG OFF
APL4 &1 &2 &3 &4 &5 &6 &7 &8 &9 &10 &11 &12
FI APLDUMP CLEAR
CP TERM APL OFF
&IF &TERM = LINE CP TERM ATTN ON
&IF &TERM = LINE CP SET LINEDIT ON
CP SET EMSG TEXT
CP SET IMSG ON
```

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 at 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. 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.

Sakura	*Pop*	*Stars*	*Blue*
Bug	*Humor*	*Forty*	*Hat*
Scales	*Dandy*	*March*	

To perform them, you have to execute the following:

PLAY name

where **name** is the title of the melody.

Notes:

Chapter 3. Auxiliary Processors

Notes:

The auxiliary processors discussed in this chapter are:

● AP80	IBM Graphics Printer control
● AP100	BIOS/DOS interrupt handling
● AP205	Full-screen display management
● AP210	DOS file management
● AP232	Asynchronous communications
● AP440	Music generator

Each auxiliary processor requires storage space in addition to that required for APL and the shared variable processor ($SVP). When you start APL with an 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 9.)

The following table gives approximate sizes for the auxiliary processors, APL, shared variable processor, and EXAPL (dyadic formats). Notice that EXAPL does not require the shared variable processor.

Module	**Approximate Size (K-bytes)**
APL	71.0
EXAPL	7.6
$SVP	1.6
AP80	1.2
AP100	0.8
AP205	8.3
AP210	3.8
AP232	5.0
AP440	1.5

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

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 shared variable processor. This auxiliary 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 alphameric mode. In this way, printer control codes can be included and executed. Appendix B shows many of these control codes. When a carriage-return 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 according to the current print mode. A carriage-return 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'              Offer the variable for
                                sharing
1
      ⎕SVO 'X'                  Is it accepted?
2                               Yes
      X←'ABCD',⎕TC[2]           The printer prints the
                                string, ABCD, followed
                                by a carriage return to
                                the beginning of the next
                                line (⎕TC[2]).
      X←⎕AV[256],'←E'           Set emphasized mode
                                (this is a printer control
                                code).
      X←'ABCD',⎕TC[2]           The printer prints
                                ABCD in emphasized
                                mode, followed by a
                                carriage return.
      ⎕SVR 'X'                  Retract the variable.
2
```

BIOS/DOS Interrupt Auxiliary Processor: AP100

The AP100 auxiliary processor provides an interface to generate BIOS and DOS interrupts or function calls.

To use this auxiliary processor, you must include AP100 as a parameter to the APL command at load time before you begin an APL work session. For example,

APL AP100

The following APL line must be executed before the auxiliary processor can be used.

```
100 ⎕SVO 'name'
```

where **name** is the name of the APL variable being shared with AP100.

The result of the previous line will be a 1 if the variable name is accepted by the shared variable processor. This auxiliary processor accepts only one variable.

The following line must be executed next.

```
⎕SVO '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.

Any 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.

In the following discussion, an *index origin of 0* will always be implied. Unless specifically stated, all characters will be considered equivalent to the one-byte integers that are equal to the position of each on the APL atomic vector (⎕AV).

Example:

If you have to send the number *22* to the auxiliary processor, send it in the following way:

```
⎕AV[22]
```

If the auxiliary processor returns the character " ∇ " its value will be determined by executing the expression:

```
⎕AV⍳'∇'
```

The elements of the vector must contain the following information:

- Element 0 gives the interrupt number desired.
- Elements 1 to 14 give the character contents of the machine registers that must be passed to the BIOS/ DOS interrupt service programs, in the following order: AL, AH, BL, BH, CL, CH, DL, DH, lower byte of SI, higher byte of SI, lower byte of DI, higher byte of DI, lower byte of BP, higher byte of BP.
- Element 15 should always to 0 to assure future compatibility.
- Element 16 should be either 0, 64, 128, or 192, and is used to control the translation of the contents of the AL register (see below).

Element 16 governs how all AL values are sent and returned. 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. (See the following diagram.)

If no more elements are given, the interrupt is executed only once. On the other hand, if the vector has more than 17 elements, all the remaining ones are considered as successive values for the AL register. The same interrupt may thus be executed several times in succession, with all the registers except AL containing the same values.

When the command has been executed, a character vector with the same number of elements as the command is returned to the shared variable, with the following information:

- Element 0 is the interrupt number just executed.
- Element 1 and those after element 16 (if any) will contain the successive values of register AL after each instance of the interrupt was executed. If element 16 of the command was 0 or 128, the values are passed as they are. However, if element 16 of the command was 64 or 192, the AL values are considered as ASCII characters, and are translated to their internal APL equivalents (see diagram below).
- Elements 2 to 14 are the new values of the machine registers, after the last execution of the interrupt, in the same order indicated above.
- Elements 15 and 16 contain the machine flags after the last execution of the interrupt, as described in the appendix, "Assembly Instruction Set Reference" of the *Technical Reference* manual. The low-order byte is passed in element 15, and the high-order byte in element 16.

Element 16	**0**	**64**	**128**	**192**
Output of AL	as is	as is	APL→ASCII	APL→ASCII
Return of AL	as is	ASCII→APL	as is	ASCII→APL

The types of errors that may be found in the command are:

- *rank error* (when a value is not a vector)
- *length error* (when there are less than 17 elements or too many for the buffer in the auxiliary processor)
- *domain error* (when a character vector is not given)

In those cases, an error return code (the integer scalar 2) is passed back to the shared variable.

Example of Use:

To read an ASCII character struck from the keyboard, using BIOS interrupt 16H and AP100, you can execute the following function (see the *Technical Reference* manual for information about BIOS interrupts):

```
      ∇ Z←INKEY;X;⎕IO

[1]     ⎕IO←0

[2]     Z←100 ⎕SVO 'X'

[3]     Z←⎕SVO 'X'

[4]     X←⎕AV[22,16⍴0]

[5]     Z←⎕AV⍳X[1 2]

      ∇
```

- Line 1 sets the origin to 0.
- Lines 2 and 3 share variable **X** with AP100.

- 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 hexadecimal 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 *Technical Reference* manual).

The Full-Screen Auxiliary Processor: AP205

The full-screen auxiliary processor, AP205, is used to manage the Monochrome or the Color/Graphics display screens. It can be used to:

- Divide the screen into rectangular areas
- Read from or write to the screen.

- Produce highlighting, reverse video, colors, etc.
- Allow you to modify the text or graphics displayed in certain preselected areas in an interactive way.

To use this auxiliary processor, you must include AP205 as a parameter to the APL command at load time before you begin an APL work session. For example:

APL AP205

Two shared variables are required to process the screen – a *control* variable and a *data* variable. They can be offered in any order. The name of the data variable must always begin with the letter "D"; the control variable must begin with the letter "C." The remaining characters in both names (possibly none) need not be the same for this auxiliary processor. This auxiliary processor accepts only one pair of shared variables.

The control variable is used to select the operation to perform and to control each input or output operation. The function of the data variable is to transfer actual data.

The following APL lines must be executed before the auxiliary processor can be used:

```
205 ⎕SVO 'Cname'

205 ⎕SVO 'Dname'
```

where **name** is the remainder of the name of each variable.

The preceding two instructions should give a result of 1. You then should test if the variables have been accepted by AP205 by entering:

```
⎕SVO 'Cname'

⎕SVO 'Dname'
```

Both entries must give a result of 2; otherwise, AP205 is not active or has already accepted other variable names that have not been retracted.

When this auxiliary processor is used from a defined function, the following steps are usually performed:

1. Offer the variables for sharing and test acceptance.

2. Specify data (if any) in the data variable.

3. Specify a request command and a field number (if any) in the control variable.

4. Check the return code from the control variable.

5. Get data (if any is returned) from the data variable.

6. Repeat Steps 2 through 5.

Screen Formatting

The screen is considered to be divided into rectangular areas called *fields*. It is only in these areas that data can be displayed or entered. Each field is defined by the following six elements:

SR
SC
HT
WD
T
A

where:

- **SR** and **SC** are the screen coordinates (row and column, respectively) of the upper-left corner (starting position) of the field relative to the upper-left corner of the screen.

- **HT** and **WD** are the numbers of rows (height) and columns (width) in the field

- T is the type of field
- A is the field's attribute.

If a field is defined beyond the right edge and/or lower end of the screen, an error return code will result, with one exception: if the defined field has only one row, it may reach beyond the right edge of the screen and follow on the next sequential lines of the screen. In this way you can define, for example, a field with 1 row and 2000 columns occupying the whole screen.

The following field types are supported:

- 0: Field is read/write.
- 2: Field is read-only (that is, you are not allowed to type new information in this field).

The attribute of the field is a positive integer, not greater than 255, that defines certain characteristics applicable to all characters displayed in the field, such as color, normal/reverse video, highlighting, blinking, underlining, etc. (See the *Technical Reference* manual for more information.)

Following is a list of several commonly-used attributes for the Monochrome Display, and the effect they produce.

Attribute	Effect
0	Invisible field. Characters are accepted but not displayed.
1	Underlined characters.
7	Normal characters.
9	Highlighted underlined characters.
15	Highlighted characters.
112	Reverse video.
120	Highlighted reverse video.
129	Blinking underlined characters.
135	Blinking normal characters.
137	Blinking highlighted underlined characters.
143	Blinking highlighted characters.
240	Blinking reverse video.
248	Blinking highlighted reverse video.

For the grapics mode with 80 characters per line, attributes are ignored.

When using a color/graphics display with 40 characters per line, the attribute of a field is selected by specifying an integer number *N*. For $N \geq 0$,, the attribute is given by the remainder of $N \div 4$ (or the residue $4|N$). The color/graphics attributes are:

0:	invisible
1:	cyan
2:	red
3:	white

All fields must be defined together. Their definitions are passed as an integer six-column matrix, each row corresponding to one field. The maximum number of different fields is 50. If fields overlap, unpredictable effects may arise.

Once the screen format has been defined, each field is identified by its position in the matrix, beginning at 1. Thus, the first row in the matrix defines field number 1, the second corresponds to field number 2, and so forth.

Example: The following matrix divides the screen into three rectangular fields:

```
 2   2  10  10   2    7
12  20   1   1   0  135
20   1   1  40   2   15
```

The first field is a square starting at row 2, column 2, and occupying 10 rows and columns. It is a *read-only* field (that is, you cannot type in it, but the program can), and characters are displayed normally.

The second field is a single character at row 12, column 20. It can be overwritten (type is 0), and will blink if the Monochrome Display is used.

Finally, the third field is a single line, 40 characters wide, at the beginning of row 20. It is a read-only field. If the Monochrome Display is used, characters written by the program are displayed with double intensity (that is, high-lighted).

All other portions of the screen not specifically defined are dark.

Control Commands

Once the control variable has been shared, each value you assign to it is considered to be a command that describes the operation to perform.

The following commands are accepted:

Command	Function
0	Clears the screen immediately
0,n	Dynamically switches screen modes and clears the new screen. n must be 1, 4, or 8 (monochrome, 40-column color, 80-column color).
1	Establishes new screen format as defined in the data variable.
1,fn	Redefines field, the number of which (fn) is given by its position in the matrix defining the screen format.
2,fn	Writes a character vector in the indicated field (fn). The actual information is passed in the data variable.
2,fn,1	Writes a boolean vector in the indicated field (fn). The actual information is passed in the data variable. Each element corresponds to a picture element on the screen. A 1 means that the element is visible, and a 0, invisible.

2,fn,2	Performs the logical *AND* operation between the contents of the indicated field (fn) and the boolean vector contained in the data variable. The result is displayed in the field.
2,fn,3	The same as the preceding command, but this one performs an *OR* operation.
2,fn,4	The same as the preceding, but this one performs an *EXCLUSIVE OR* operation.
3	Goes into interactive mode. You may now type in any of the allowed fields, as described below.
4,fn	Writes a character vector in the indicated field (fn). The actual information is passed in the data variable.
4,fn,1	Writes a boolean vector in the indicated field (fn). The actual information is passed in the data variable. Each element corresponds to a picture element on the screen. A **1** means that the element is visible, and a **0**, invisible.
4,fn,2	Performs the logical *AND* operation between the contents of the indicated field (fn) and the boolean vector contained in the data variable. The result is displayed in the field.
4,fn,3	The same as the preceding command, but this one performs an *OR* operation.
4,fn,4	The same as the preceding command, but this one performs an *EXCLUSIVE OR* operation.
5,fn	Reads the present contents of field fn as a character vector returned through the data variable.

5,fn,1	Reads the current contents of field **fn** as a boolean vector, and returns it in the data variable.
8	Goes into interactive mode. You may now type in any of the allowed fields, as described below.
9	Reads the current definition of all the fields.
9,fn	Reads the current definition of field **fn**.
12	Sets the cursor position on the screen. The desired position is passed in the data variable as a three-element numeric vector, giving the field number and the row/column coordinates of the desired point relative to the beginning of that field.

Interactive Use of the Screen

When commands 3 or 8 are requested, the auxiliary processor enters a *Wait* state, allowing you to type at the keyboard, thus changing the contents of one or more read/write fields. While in this mode, the following special keys may be used:

- The four arrow keys at the right of the keyboard: these move the cursor in the direction indicated by the arrow. Any point on the screen may be accessed.
- The Ins, Del, and End keys: these have the same function as described under "APL Input Editor Special Keys" in Chapter 1.

- The Tab key: positions the cursor at the beginning of a line within a read/write field according to the following priority:

  1. The next sequential line of the same field, or

  2. The first line of the next read/write field, or

  3. The first line of the first read/write field

- The Ctrl-Backspace keys: switch between the APL keyboard and the National keyboard.

You signal the end of the interactive use of the screen by pressing any of the following keys:

- A function key
- A function key in *Shift* mode
- A function key in *Ctrl* mode
- The Enter key
- The Esc key
- The PgUp key
- The PgDn key

When one of these keys is pressed, AP205 returns control to the APL. The data variable returns a vector of five numeric elements, which contain the following information:

KT, KN, CF, CR, CC

where KT, KN is a pair of numbers defining the key that was pressed to end the operation:

KT	KN	Key
0	2	Enter
1	2	Esc
1	3	PgUp
1	4	PgDn
2	n	Function

Normal function keys are assigned numbers from 1 to 10, *Shift* function keys from 11 to 20, and *Ctrl* function keys from 21 to 30. *Alt* function keys are reserved for internal system use.

CF, CR, and CC give the present position of the cursor. CF is the number of the field where the cursor is, and CR and CC are the row/column coordinates of the cursor's position relative to the beginning of that field.

Return Codes

The following is a list of the possible return codes for all the commands:

0: Successful
1: Command not recognized
2: Data variable erroneous
3: Data variable not shared
4: Buffer overflow
5: Attempt to use a non-existent display

Example of Use:

The following executable lines are assumed to be the lines of an APL-defined function. Results of individual lines are shown for clarity, but should not appear on the screen (for example, by assigning them to some variable). Otherwise, if the APL input editor is allowed to act, the full-screen application will not work correctly, because the screen is cleared whenever control is transferred between the APL Input Editor and the AP205 auxiliary processor.

```
      205 ⎕SVO 2 1⍴'CD'    The variables are offered
                           to AP205.
1 1
      ⎕SVO 2 1⍴'CD'        Are they accepted?
2 2                        Yes, they are.
```

```
      D←3 6⍴2 2 10 10 2 7 12 20 1 1 0 135 20 1 1 40 2 15
                                  The data variable is
                                  assigned the desired
                                  screen format (see above
                                  for an explanation of the
                                  format).
      C←1
      C                           Ask for the return code.
0                                 Successful operation.
      D←'FIELD 1'                 This text will be written
                                  in field 1.
      C←2 1                       This is the command to
                                  write.
      C                           Return code.
0                                 Success.
      C←3                         Allow the user to type in
                                  the predefined fields. To
                                  leave this state you have
                                  to press either Esc,
                                  Enter, PgUp, PgDn, or
                                  any F key.
      C                           Wait until a return code
                                  has been assigned to C.
0                                 Now.
      D                           Shows how the user
                                  ended.
1 2 2 1 1                         The user pressed the Esc
                                  key (1 2) and left the
                                  cursor in the first
                                  position (1 1) of field 2.
      C←5 2                       Let us read (5) the
                                  contents of field 2.
      C
0                                 Operation successful.
      D
A                                 The user has typed an A
                                  in that field.
      ⎕SVR 2 1⍴'CD'               The shared variables are
2 2                               retracted.
```

The File Auxiliary Processor: AP210

The file auxiliary processor, AP210, is used to read from or write to, fixed-length disk files under control of the DOS file system. The reading and writing can be either sequential or random.

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. 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 recognized by their name. Examples of valid pairs are: *C* and *D*, *C1* and *D1*, and *CXjj* and *DXjj*. The control variable is used to identify the file to be worked with, and the particular operation to be performed. It is also used to activate each input or output action. The data variable contains the information being read or written. Up to four pairs of variables may be shared at one time.

The following APL lines must be executed before the auxiliary processor can be used:

```
210 ⎕SVO 'Cname'

210 ⎕SVO 'Dname'
```

where **name** 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 'Cname'

⎕SVO 'Dname'
```

Both must give a result of 2. Otherwise, AP210 is not active or has already accepted four 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:

IR,*filespec*[,*code*]	Open for read-only
IW,*filespec*[,*code*]	Open for read/write
DL,*filespec*	Delete file
RN,*filespec*,*filename*[.*ext*]	Rename file

where *filespec* is the DOS file identification, of the form:

[d:] filename [.ext]

d: is a letter that identifies the drive (typically A, B, C, etc.). **filename** is a valid DOS file name (up to eight characters), and the extension of the name has no more than three characters (see your DOS manual).

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 4, "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.

> Note: If the **I/O ERROR** message appears when you are trying to access a file, either the door of the drive is open, the incorrect diskette is inserted, or the diskette is write-protected. See Figure 17 in Chapter 12 for the recommended action.

> **WARNING: Changing diskettes during an input/output operation, or when you have open files, may damage your 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. A valid drive specified in the second parameter is ignored. 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.

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:

[op [nr [rs]]]

where **op** is 0 (read operation) or 1 (write operation; this is not allowed if the subcommand state was entered through the IR command). **nr** is the record number to be read or written, where the first record in the file has a record number of 0. Finally, **rs** is the record length or size.

If **rs** is not specified, the value used in the previous operation applies. If such a previous operation does not exist (as in the first read/write subcommand after opening the file), a default record length of 128 bytes is used.

If **nr** is not specified, the value used in the preceding operation is increased by 1 (thus, sequential access is possible, as well as direct access). If not specifically stated, the first value of **nr** after opening a file is 0 (that is, the first record in the file).

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.

Return Codes

The following is a list of the possible return codes for all the different commands and subcommands.

Return Code	Interpretation
0	Successful.
1	Read: End of file reached. The data variable returns an empty vector in this case. Write: Disk full.
3	Read: The last record in the file is incomplete. In this case, its contents are passed anyway, padded by the characters, ⎕AV[⎕IO] up to the requested record size.
20	Command not recognized.
21	Subcommand not recognized.
22	Auxiliary processor buffer overflow; command too large.
23	Data variable not found.
24	Data type error. This can happen on input if code A has been requested and the record does not contain a valid APL object, in which case the data variable returns an empty vector to the APL processor. This can also happen on output if the data variable contains an object incompatible with the code selected in the command, or if the size of the record exceeds the specified record length.
255	IR: File not found. IW: File not found, or disk directory full. DL: File not found. RN: File not found, or duplicate name.

Examples of Use:

```
      210 ⎕SVO 2 2⍴'C1D1'Offer two variables (C1
                         and D1) to auxiliary
                         processor 210.
1 1                      SVP answer.
      ⎕SVO 2 2⍴ 'C1D1'   Test to see whether the
                         varibles have been
                         accepted.
2 2                      OK.
      C1←'IW,B:FILE.EXT' Creation of a new file.
                         Records will contain
                         APL objects with header
                         (default code).
      C1                 Return code.
0                        Success. The file is open.
                         You are now in
                         subcommand mode.
      D1←⍳10             First record will be a
                         vector of elements from 1
                         to 10.
      C1←1               Subcommand to write
                         the record in the file.
                         Default record number is
                         0, default record size is
                         128 bytes.
      C1                 Return code.
0                        Success.
      D1←2 3⍴⍳6          A matrix of two rows
                         and three columns of
                         elements from 1 to 6, is
                         assigned to the data
                         variable.
      C1←1               Write sequentially on the
                         file.
      C1                 Return code.
0                        Correct.
      C1←''              An empty vector closes
                         the file and puts the
                         control variable in
                         command mode.
      C1←'IR,B:FILE.EXT' Open the same file for
                         read-only operation.
      C1                          Return code.
0                                 Success.
      C1←0 1                      Read the second record
                                  first.
      C1                          Return code.
0                                 All right.
      D1                          Ask for the record
                                  contents.
1 2 3                             Here is the matrix.
4 5 6
      C1←0 0                      Ask now for the first
                                  record.
      C1                          Return code.
0                                 OK.
      D1                          Record contents is a
1 2 3 4 5 6 7 8 9 10              vector of integers.
      C1←⍳0                       Close the file and go into
                                  command state.
      C1←'RN,B:FILE.EXT,NEWFILE.XXX'
                                  Rename the file.
      C1                          Return code.
0                                 The file has been
                                  renamed.
      C1←'DL,B:NEWFILE.XXX'       Delete the file.
      C1                          Return code.
0                                 The file no longer exists.
      ⎕SVR 2 2⍴'C1D1'             Retract the shared
                                  variable.
2 2                               All done.
```

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.)

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. 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 shared variable processor. This auxiliary 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:

- Initialize (0)
- Transmit (1)
- Receive (2)
- Get port status (3)
- Set break (4)
- Get buffer size (5)

All commands are strings of a 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.

Initialize (0)

This command is used to initialize the port. It consists of a string of characters of the form:

C
N
B
P
S
X

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 of B	**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 of P	**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:

- −1: success

- 3: error

Transmit (1)

This command consists of a string of characters of the following form:

C
N
S

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, −1.

Receive (2)

The command consists of a string of characters of the form:

C
N
T
E
D

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).

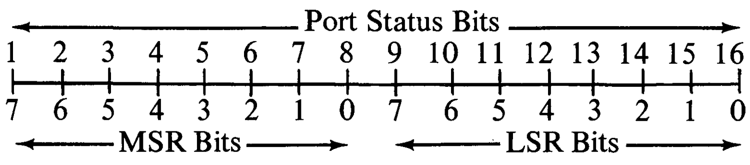
The command consists of a string of characters of the form:

C
N
where:

- 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 8 represent the content of the MSR, and bits 9 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:

C
N
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, −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:

C
N
O

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:

- **-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 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. 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 by the shared variable processor. 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*]}

Brackets indicate an optional parameter.

where:

[*tempo*]:	T*n*	*n* = 0 to 6; default 4
[*octave*]:	O*n*[{+ −}]	*n* = 0 to 6; default 3
[*mode*]:	M*n*	*n* = 0 to 2; default 1
[*length*]:	L*n*	*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, **L0** 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, **M1** 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, O3 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:

0	Largo (54 quarter notes per minute)
1	Largetto (66 per minute)
2	Adagio (78 per minute)
3	Andante (96 per minute)
4	Moderato (120 per minute)
5	Allegro (156 per minute)
6	Presto (198 per minute)

If no T*n* command has ever been given, T4 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 **name**.

For an example of how to use this auxiliary processor, examine the variables included in the MUSIC workspace.

Notes:

Chapter 4. How to Build an Auxiliary Processor

Notes:

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 data set. 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 has associated with it, 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 on the other, but neither can impose less on itself.

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.

Shared Variable Processor Services and Return Codes

The following is a listing of all the SVP macros in the file $APMAC.ASM on the APL diskette, and the expected arguments and return codes. These macros use certain memory positions (*$P1*, *$P2*, *$P3*, *$P4*, *$IFLOT*, and *$IFLOT+2*) to pass information about the operation the auxiliary processor is requesting. The contents of these data objects should be changed only in the cases indicated below. They should never be used as intermediate positions for internal calculations. Since the definition of these objects depends on register **BP**, the contents of register **BP** should also remain unchanged.

Each auxiliary processor should have a buffer (the size of which depends on the application) where the values of the shared variables may be passed through, or received from, the SVP. This buffer and all the remaining data areas needed for each particular application should be in the same segment as the program itself (CS:), because the data segment register must point to the APL data area, where some of the required transfer positions are.

In all the following functions, *$P4* is assumed to contain the AP number.

$P3, in those functions that need it, should pass a pointer to the shared variable control block.

$P1 and *$P2* pass the buffer address when needed. *$P1* is the segment register, *$P2* the displacement.

$IFLOT and *$IFLOT+2* may pass additional information (described later in the chapter).

$IFLOT returns the result of the operation or the return code. *$SVSC* also returns *$IFLOT+2*.

The SVP macros are: *$SVSN*, *$SVSF*, *$SVSC*, *$SVSH*, *$SVRT*, *$SVSA*, *$SVGA*, *$SVPR*, *$SVSZ*, *$SVPW*, *$SVCP*, *$SVRD*, *$SVWR*, *$SVRL*, and *$SVWA*. They are included in a file called **$APMAC.ASM**, with all the macros needed to build an auxiliary processor.

All the SVP macros, except *$SVWA*, maintain the values of *$P1*, *$P2*, *$P3* and *$P4*. Registers *AX*, *BX*, and *BP* are also maintained. All others are lost. *$IFLOT* and *$IFLOT+2* are changed by SVP. *$SVWA* keeps only the value of *$P4*.

$SVSN (Sign On to the SVP)

Sign-on identifies an auxiliary processor to the SVP. It must be successfully carried out before any other service can be obtained. Each processor is identified by a number, between 2 and 32767. Two different processors cannot have the same number.

Input: *$P4*
Output: 0, −5, −6.

$SVSF (Sign Off to the SVP)

Sign-off disconnects an auxiliary processor from the SVP and retracts all the processor's shared variables. Processor sign-off is also automatic when you exit APL through the)OFF command.

Input: *$P4*
Output: 0, −11.

$SVSC (Scan for Offers)

This function scans the SVP tables for an offer to this processor. Only offers after a given chronology are scanned. The SVP maintains a counter, which is increased by 1 each time a new variable is offered. The value of this counter is stored in the variable control block and is called its chronology.

Input:	*$P4, $IFLOT+2* = initial chronology.
Output:	0 (no offer), −11, or the control block pointer. *$IFLOT+2* has the updated chronology.

$SVSH (Share a Variable)

This function offers a variable for sharing, or requests the present state of the variable, if previously shared.

Input:	*$P4, $P3*
Output:	0, rejected; 1, offered; 2, shared.

$SVRT (Retract a Variable)

Retraction of a shared variable ends an auxiliary processor's connection to that variable. If the partner has already retracted, the variable is no longer shared.

Input:	*$P4, $P3*
Output:	Previous condition of variable.

$SVSA (Set Access-Control Vector)

Set-access-control changes the auxiliary processor's setting of a shared variable's access-control vector. The request can be issued at any time after the variable has been offered. The first setting (0 0 0 0) is provided when a variable is offered.

Input:	*$P4, $P3, $IFLOT* = new access vector, in decimal (15 = 1 1 1 1)
Output:	Joint access vector, in decimal.

$SVGA (Get Access-Control Vector)

This function requests the present value of the access-control vector as previously set by both users.

Input:	*$P4, $P3*
Output:	Joint access vector, in decimal.

$SVPR (Pre-Read), $SVSZ (Seize), $SVPW (Pre-Write)

If the variable is not under the control of the partner, the access state and control vectors are examined to determine if a use is permissible. If so, the variable is registered in SVP as being under the control of the auxiliary processor. When successful, the amount of storage required for the value is also returned.

Input:	*$P4, $P3*
Output:	If positive, number of bytes.
Errors:	−1, −8, −9.

$SVCP (Copy), $SVRD (Read)

The value of the shared variable is transferred to the auxiliary processor's buffer. For *Copy*, the access state is not changed and control of the variable is not released. For *Read*, control is released and the access state is changed to show that the auxiliary processor has used the value.

Input:	*$P4, $P1, $P2, $P3*
Output:	0, −1, −7, −9

$SVWR (Write)

The value in the auxiliary processor's buffer is transferred to the SVP. The access state is changed to show that the auxiliary processor has set the variable. Control of the variable is released.

Input:	*$P4, $P1, $P2, $P3, $IFLOT* = number of bytes
Output:	0, −7, −9.

$SVRL (Release the Variable)

Control of the variable, if held by the auxiliary processor, is released. The access state is not changed.

Input:	*$P4, $P3*
Output:	0, −9

$SVWA (Wait)

Wait releases control and allows other auxiliary processors and the APL processor to get control. The wait state is left (that is, control is given back to this processor) when: an offer is extended to this processor; a shared variable is referenced, specified, or retracted by the partner; the access-control vector is changed by the partner.

Input: *$P4*
Output: None

SVP Macros Error Return Codes

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

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. If you pass invalid data to APL, unpredictable errors may occur. Each variable contains information that describes its data type, shape, and size.

An APL variable is one of four types:

- real (floating point)
- integer
- logical (boolean)
- literal (character)

Regardless of a variable's data type, its elements always occupy some number of words. 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 real variable are represented in long floating-point format, with eight bytes per element.

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 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 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.

When you receive numeric data from APL, you should be prepared to accept the data in any representation and to convert between different representations.

The rank of a variable defines its *shape*. 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 0 includes size information: as many dimensions as the value of its rank. Each dimension must be a two-byte integer, the value of which belongs to the interval (0, 32767). The maximum rank of a variable is 63.

Internal Structure of APL Variables

The variable is supposed to have been copied in the auxiliary processor's buffer, which we will call *$BUF*.

```
$PTR    EQU  WORD PTR $BUF    ; A pointer. Should
                              ; be ignored by the
                              ; auxiliary processor

$NB     EQU  WORD PTR $BUF+2  ; Total number of
                              ; bytes in this
                              ; APL object
                              ; Note: The number of
                              ; bytes of an APL object MUST
                              ; ALWAYS be rounded up to
                              ; EVEN.

$NELM   EQU  WORD PTR $BUF+4  ; Total number of
                              ; elements in this
                              ; APL object

$TYPE   EQU  BYTE PTR $BUF+6  ; APL object type
                              ; 0=Logical, 1=Integer,
                              ; 2=Real,    3=Character

$RANK   EQU  BYTE PTR $BUF+7  ; Rank of APL object
                              ; (between 0 and 63)

$DIM    EQU  WORD PTR $BUF+8  ; First dimension (if any).
                              ; As many dimensions as
                              ; value of $RANK follow.
             ; Immediately after dimensions,
             ; values themselves appear in
             ; row order, packed according
             ; to type of data object:
             ; Logical: one bit/element.
             ; Integer: one word/element.
             ; Real: eight bytes/element.
             ; Character: one byte/element.
```

Information About Shared Variables

The shared-variable control block is created and accessed by the SVP. Its address is passed to the auxiliary processors in variable *$P3*. Its contents are as follows:

```
$ACB    EQU  WORD PTR  [$P3]   ; Pointer to the symbol table
                               ; element of this variable.
                               ; Remaining bytes used only
                               ;  by the SVP.
```

APL contains additional information about a shared variable in the APL Symbol Table. The address of this information block is contained in the shared-variable control block (*$ACB*). Its contents are as follows:

; First seven bytes used only by APL.

```
$ONC    EQU  BYTE PTR [$ACB+7] ; Number of characters in
                               ; the name of the variable.
                               ; (Maximum number is 12.)

$ONA    EQU  BYTE PTR [$ACB+8] ; First character in the
                               ; name of the variable.
                               ; Remaining characters are
                               ; consecutively stored.
```

Auxiliary Processor Example With One Shared Variable

The auxiliary processor shown in this section illustrates the use of macros and segment registers. It can establish a single connection, using a shared variable.

```
DGROUP      GROUP     APxxx        ; xxx should be replaced by
                                   ; AP identification number
            INCLUDE   $APMAC.ASM   ; APL data segment
            NAME      APxxx
APxxx       SEGMENT   PUBLIC 'DGROUP'
            ASSUME    CS:DGROUP,DS:$SQ
                                   ; Assume seg registers
```

```
        $PAPL                   ; APL environment macro
$APxxx  PROC      FAR           ; All APs are considered
                                ; as FAR procedures
        JMP       $BEGIN        ; Jump over data area
$SHV    DW        0             ; Shared variable block
                                ; address stored here
                                ; Value of zero means
                                ; "not shared"
$BUF    DB        512 DUP(?)    ; The AP buffer
; Other aux processor data words should be included here
$PTR    EQU WORD PTR $BUF       ; APL data object
$NB     EQU WORD PTR $BUF+2     ; Total number of bytes
$NELM   EQU WORD PTR $BUF+4     ; Number of elements
$TYPE   EQU          $BUF+6     ; Data object type
$RANK   EQU          $BUF+7     ; Rank (0–63)
$SCALAR EQU          $BUF+8     ; Address of value if scalar
                                ; ($RANK = 0)
$BEGIN:  $SAVE                  ; APL SAVE macro
         MOV      $P4,xxx       ; Load AP identification
                                ; into $P4
         $SVSN                  ; Sign on to SVP
EW:      $SVWA                  ; Wait for requests
; Set initial chronology for SCAN
         MOV      $IFLOT+2,0
; Set $P3 equal to the shared variable block address
        MOV       AX,CS:$SHV
        MOV       $P3,AX
        CMP       AX,0          ; Is the variable shared?
        JNE       E0            ; Jump if so
EW0:    $SVSC                   ; Scan for offers
        MOV       AX,$IFLOT     ; $IFLOT has the offer
        CMP       AX,0          ; or a zero if no offers
        JE        EW            ; Wait again if no offers
        MOV       $P3,AX        ; There was an offer.
; Save its block address in $P3
        $SVSH                   ; Accept the offer
        CMP       $IFLOT,2      ; Was it successful?
        JNE       EW            ; Wait again if not
        MOV       AX,$P3        ; Otherwise save the block
        MOV       CS:$SHV,AX    ; address in $SHV
        MOV       $IFLOT,15     ; Prepare to set access
                                ; control vector
        $SVSA                   ; Set it
        JMP       E1            ; and go to read the value
```

```
; The variable was previously shared
E0:     $SVSH                   ; Question share state
        CMP     $IFLOT,2        ; Is it still shared?
        JE      E1              ; Go to read if so
E01:    $SVRT                   ; Retract the variable
        MOV     $SHV,0          ; Indicate variable is
                                ; no longer shared
        JMP     EW0             ; Go to scan for new offers
; Read the value of the variable
E1:     $SVPR                   ; Pre-read
        CMP     $IFLOT,-9       ; Retract variable if
        JE      E01             ; no longer shared
        CMP     $IFLOT,0        ; Positive return code
        JG      E3              ; means a value
        JMP     EW              ; Wait if not so
; the variable has a value we can read
E3:     PUSH    CS              ; $P1 must point to segment
        POP     $P1             ; containing buffer
        LEA     AX,$BUF         ; and $P2 to buffer itself
        MOV     $P2,AX          ; within the segment
        CMP     $IFLOT,512      ; Is value larger than
                                ; buffer?
        JBE     E2              ; Jump if not
        $SVRL                   ; Buffer overflow: release
                                ; variable
        JMP     ER              ; and go to prepare an
                                ; appropriate error code
E2:     $SVRD                   ; Read value of variable
                                ; into buffer
```

```
.............................................................
     THE ACTUAL OPERATION OF AUXILIARY
     PROCESSOR WILL BE INCLUDED HERE

        INC     $NB          ; Round up to even the
        AND     $NB,-2       ; number of bytes of the
                             ; result
.............................................................
EWRT:   $SVPW                ; Prepare to write return
                             ; value or return code
        MOV     AX,$NB       ; $IFLOT must contain
        MOV     $IFLOT,AX    ; total number of bytes
                             ; in value passed back
        $SVWR                ; Pass value to APL
        JMP     EW           ; Wait for a further event
ER:     MOV     $NB,10       ; Error return code
                             ; Number of bytes will be 10
        MOV     $NELM,1      ; Number of elements is 1
        MOV     $TYPE,0      ; Type is logical
        MOV     $RANK,0      ; Rank is 0 (scalar)
        MOV     $SCALAR,128  ; Value is 1 (First bit)
        JMP     EWRT         ; Go to write return code
$APxxx  ENDP
APxxx   ENDS
        END
```

Auxiliary Processor Example with Two Shared Variables

Our second example is an auxiliary processor designed to establish up to four connections of two shared variables each. The first one is a control variable that is used by APL to send commands to the AP, and by the AP to return the corresponding return code. The second variable of the pair is a data variable, used by both processors to exchange data objects.

```
DGROUP GROUP   APxxx         ; xxx should be replaced
                             ; by the AP id number
      INCLUDE $APMAC.ASM     ; APL data segment
      NAME    APxxx
APxxx SEGMENT PUBLIC 'DGROUP'
```

```
          ASSUME    CS:DGROUP,DS:$SQ    ; Assume segment registers
          $PAPL                         ; APL environment macro
$APxxx PROC         FAR                 ; All APs considered
                                        ; as FAR procedures
          JMP       $BEGIN              ; Jump over data area
$CTL      DB        0                   ; CTL-DAT toggle
$LBUF     EQU       2058                ; Length of buffer
$BUF      DB        $LBUF DUP(?)        ; Buffer of AP
          DB        ?                   ; One extra byte
$PTR      EQU       WORD PTR $BUF       ; APL data object in buffer
$NB       EQU       WORD PTR $BUF+2     ; Total number of bytes
$NELM     EQU       WORD PTR $BUF+4     ; Number of elements
$TYPE     EQU       BYTE PTR $BUF+6     ; Type
$RANK     EQU       BYTE PTR $BUF+7     ; Rank
$CR       EQU       WORD PTR $BUF+8     ; Value of return code
$LCB      EQU       62                  ; Size of block for each
                                        ; CTL variable
$LTCB     EQU       4*$LCB              ; There are 4 CTL variables
$LDB      EQU       4                   ; Size of block for DAT vars
$LTDB     EQU       4*$LDB              ; There are 4 DAT vars

; Do not change order of following four instructions

$DAT      DB        $LTDB DUP(0)        ; DAT block area
$DATE     LABEL     NEAR                ; End of DAT block area
$SHV      DB        $LTCB DUP(0)        ; CTL block area
$SHVE     LABEL     NEAR                ; End of CTL block area
$SVPT     EQU       WORD PTR  [BX]      ; First word of block
$STPT     EQU       WORD PTR  [BX+2]    ; Second word of block
$ONA      EQU       BYTE PTR  [SI+8]    ; Name of variable in APL
                                        ; symbol table
$BEGIN: $SAVE                           ; APL SAVE macro
          MOV       $P4,xxx             ; Load AP number into $P4
          $SVSN                         ; Sign on to SVP
EW:       $SVWA                         ; Wait for an event
          LEA       BX,$DAT             ; Point to DAT blocks
          MOV       $CTL,0              ; Toggle to DAT vars
E000:     MOV       $IFLOT+2,0          ; Start chronology
E01:      CMP       CS:$SVPT,0          ; Is variable shared?
          JE        E02                 ; Jump to E02 if not
          JMP       E2                  ; Otherwise go to E2
```

```
E02:  $SVSC                        ; Scan for offers
      MOV     AX,$IFLOT            ; Is there an offer?
      CMP     AX,0
      JNE     E1                   ; Jump if so
E03:  TEST    $CTL,1               ; Go to E030 if this
      JNZ     E030                 ; is a CTL variable
      ADD     BX,$LDB              ; Increment for DAT variables
      CMP     BX,OFFSET $DATE      ; End of DAT blocks?
      JNE     E01                  ; Retry if not
      MOV     $CTL,1               ; Next we will try CTL vars
      JMP     E000                 ; Jump back and retry
E030: ADD     BX,$LCB              ; Increment for CTL vars
      CMP     BX,OFFSET $SHVE      ; End of CTL blocks?
      JNE     E01                  ; Retry if not
      JMP     EW                   ; Wait for next event

; There was an offer

E1:   MOV     $P3,AX               ; Save offer in $P3
      MOV     SI,AX                ; Get pointer to this var
      MOV     SI,[SI]              ; in APL symbol table
      MOV     CS:$STPT,SI          ; and save it in block
      CMP     $ONA,64              ; Is first letter a "C"?
      JE      EQ1                  ; Jump to EQ1 if so
      CMP     $ONA,65              ; Is first letter a "D"?
      JNE     E02                  ; Reject variable if not
      TEST    $CTL,1               ; Reject it if CTL variable
      JNZ     E02
      JMP     SHORT EQ0            ; Otherwise accept it
EQ1:  TEST    $CTL,1               ; Reject it if DAT variable
      JZ      E02
EQ0:  $SVSH                        ; Share variable
      CMP     $IFLOT,2             ; Try next block
      JNE     E03                  ; if not success
      MOV     AX,$P3               ; Save $P3 in first
      MOV     CS:$SVPT,AX          ; word of block
      TEST    $CTL,1               ; If DAT variable,
      JZ      E03                  ; all done
      MOV     $IFLOT,15            ; CTL var must have access
      $SVSA                        ; vector equal to 1 1 1 1
      JMP     E03                  ; All done for this block
```

```
; This variable had been shared before

E2:     MOV     AX,CS:$SVPT         ; Load $P3
        MOV     $P3,AX
        $SVSH                       ; Question variable state
        CMP     $IFLOT,2            ; If still shared,
        JE      E20                 ; go to E20

; The variable is no longer shared

E3:     $SVRT                       ; Retract variable
        MOV     CS:$SVPT,0          ; Show it is not shared
        JMP     E02                 ; Try new offers

; The variable is still shared

E20:    TEST    $CTL,1              ; All done if DAT variable
        JNZ     E20A
E20B:   JMP     E03
E20A:   $SVPR
; Get control of variable to read
        CMP     $IFLOT,-9           ; Retract if no longer
        JE      E3                  ; shared
        CMP     $IFLOT,0            ; Number of bytes must
        JLE     E20B                ; be positive
        CALL    EAUX                ; Read value
        JNC     EQ2                 ; Successful read?
        JMP     ER2                 ; Else, error message
EQ2:
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

; Insert here the appropriate code for the AP
        INC     $NB                 ; Round up to even the
        AND     $NB,-2              ; number of bytes of result
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
ER2:    MOV     AL,22               ; Error return code=22
EFN4:   MOV     DX,CS:$SVPT         ; All return codes come here
        MOV     $P3,DX              ; Load $P3 for CTL variable
        XOR     AH,AH               ; Return code is in AL
        MOV     $NB,10              ; A scalar integer
                                    ; occupies 10 bytes
        MOV     $TYPE,1             ; Set type as integer
        CMP     AL,1           ; If AL = 0, 1, it is logical
        JA      EFN5
        MOV     $TYPE,0        ; Therefore, type is logical
        ROR     AL,1           ; Put byte in place
                               ; (Remember byte reversal
                               ; in two-byte words)
EFN5:   MOV     $RANK,0        ; Rank is 0 (scalar result)
        MOV     $NELM,1        ; Number of elements is 1
        MOV     $CR,AX         ; Put result in buffer
        $SVPW                  ; Prepare to write
        MOV     $IFLOT,10      ; Number of bytes is 10
        $SVWR                  ; Write the return code
        JMP     EW             ; and wait for next event
$APxxx ENDP
; This routine reads the value of a shared variable
EAUX    PROC    NEAR
        PUSH    CS             ; $P1 must point to segment
        POP     $P1            ; of buffer
        LEA     AX,$BUF        ; $P2 must point to $BUF
        MOV     $P2,AX         ; within the segment
        CMP     $IFLOT,$LBUF   ; Is buffer length sufficient?
        JA      EAU1           ; Jump if not
        $SVRD                  ; Read value into the buffer
        CLC                    ; Clear carry (success)
        RET                    ; and return to caller
EAU1:   $SVRL                  ; Release variable
        STC                    ; set carry (fail)
        RET                    ; and return to caller
EAUX    ENDP
APxxx   ENDS
        END
```

APL Data Segment and Macros for Auxiliary Processors

The macros and data segments for auxiliary processors are in the file, $APLMAC.ASM.

```
        IF1
$SVSN   MACRO
        MOV     $PPTR,0
        CALL    $SVP
        ENDM
$SVSF   MACRO
        MOV     $PPTR,1
        CALL    $SVP
        ENDM
$SVSC   MACRO
        MOV     $PPTR,2
        CALL    $SVP
        ENDM
$SVSH   MACRO
        MOV     $PPTR,3
        CALL    $SVP
        ENDM
$SVRT   MACRO
        MOV     $PPTR,4
        CALL    $SVP
        ENDM
$SVSA   MACRO
        MOV     $PPTR,5
        CALL    $SVP
        ENDM
$SVGA   MACRO
        MOV     $PPTR,6
        CALL    $SVP
        ENDM
$SVPR   MACRO
        MOV     $PPTR,7
        CALL    $SVP
        ENDM
$SVPW   MACRO
        MOV     $PPTR,8
        CALL    $SVP
        ENDM
$SVSZ   MACRO
        MOV     $PPTR,9
        CALL    $SVP
        ENDM
$SVRD   MACRO
        MOV     $PPTR,10
        CALL    $SVP
        ENDM
```

```
$SVCP     MACRO
          MOV       $PPTR,11
          CALL      $SVP
          ENDM
$SVWR     MACRO
          MOV       $PPTR,12
          CALL      $SVP
          ENDM
$SVRL     MACRO
          MOV       $PPTR,13
          CALL      $SVP
          ENDM
$SVWA     MACRO
          MOV       $PPTR,14
          CALL      $SVP
          ENDM
$SVSV     MACRO
          MOV       $PPTR,15
          CALL      $SVP
          ENDM
$PAPL     MACRO
$P1       EQU       WORD PTR[BP]
$P2       EQU       WORD PTR[BP+2]
$P3       EQU       WORD PTR[BP+4]
$P4       EQU       WORD PTR[BP+6]
$P5       EQU       WORD PTR[BP+8]
$P6       EQU       WORD PTR[BP+10]
$PPTR     EQU       WORD PTR[BP+8]
$PLKR     EQU       WORD PTR[BP+10]
          ENDM
;
$SAVE     MACRO
          PUSH      AX
          PUSH      BX
          PUSH      BP
          PUSH      $P6
          PUSH      $P5
          PUSH      $P4
          PUSH      $P3
          PUSH      $P2
          PUSH      $P1
          MOV       BP,SP
          ENDM
          ENDIF
$SQ       SEGMENT   COMMON
$WBEG1    LABEL     BYTE
$SVP      EQU       DWORD PTR $WBEG1+371H   ;SVP LINK
$APLOFF   EQU       $WBEG1+4ACH             ;ASCII TO APL T.TABLE
$APLON    EQU       $WBEG1+5ACH             ;APLON KEYBOARD TABLE
$ASCII    EQU       $WBEG1+6ACH             ;APL TO ASCII T.TABLE
$PMR      EQU       WORD PTR $WBEG1+8ACH    ;APLDFN START ADDRESS
$L        EQU       $PMR+52*TYPE $PMR       ;INTERPRETER WORK AREA
                                            ;DO NOT USE IT.
$IFLOT    EQU       $L+56*TYPE $L           ;RETURN CODES WORD
$SQ       ENDS
```

Notes:

IBM

Personal Computer
Computer Language
Series

Part 2
APL
Reference Guide

Part 2. APL Reference Guide

Chapter 5. Using APL

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.

Two Examples 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 (⎕PP) equal to 6) would look like this:

```
      ⎕PP←6
      1000×(1+.06÷1 4 12 365)*10×1 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 + × ÷ 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.

A Prepared Workspace

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, possible months or years. For this reason APL systems are organized 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 summarized 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(`⍴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 behavior 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∘.*YEARS for an interest table, and the *inner product*, a simple generalization 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: + - × ÷ ⋆ ⍟ ○ | ⌊ ⌈ ! ⌹

  – Boolean and Relational: ∨ ∧ ⍱ ⍲ ~ < ≤ = ≥ > ≠

  – Selection and Structural: / \ ⌿ ⍀ [;] ↑ ↓ ⍴ , ⌽ ⍉ ⊖

  – General: ∊ ⍳ ? ⊥ ⊤ ⍒ ⍋ ⍎ ⍕

Chapter 6. Fundamentals

Notes:

A typical statement in APL is of the form:

```
AREA←3×4
```

The effect of the statement is to assign to the name *AREA* the value of the expression 3×4 to the right of the specification arrow ←; 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 10) 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:

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

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

```
      X←2+⎕←3+Y←4
12
      X
14
      Y
4
```

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

```
       X←5
       3+(Y×X)
VALUE  ERROR
       3+(Y×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 localized and not been assigned a value.
INDEX	Index value out of range.
INTERRUPT	1. The input line being typed is ignored. Begin typing again. 2. The input/output operation attempted was not completed.

	3. 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 problem can be corrected by executing the following sequences of commands: *)OUT*, *)CLEAR*, *)IN* or *)OUT*, *)CLEAR*, *)SYMBOLS*, *)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 or 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 alphabetics comprise the roman alphabet in uppercase italic font, the same alphabet in lowercase, delta (∆), and delta underline (⍙). The complete set is shown in Figure 3 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 ∆
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 ⍙
0 1 2 3 4 5 6 7 8 9

¨	dieresis	⍺	alpha	⍱	nor
¯	overbar	⌈	upstile	⍲	nand
<	less	⌊	downstile	⍒	del stile
≤	not greater	_	underbar	⍋	delta stile
=	equal	∇	del	⌽	circle stile
≥	not less	∆	delta	⍉	circle slope
>	greater	∘	null	⊖	circle bar
≠	not equal	'	quote	⍟	log
∨	or	⎕	quad	⌶	I-beam
∧	and	(	left paren	⍫	del tilde
-	bar	)	right paren	⍎	base null
÷	divide	[	left bracket	⍕	top null
+	plus	]	right bracket	⍀	slope bar
×	times	⊂	left shoe	⌿	slash bar
?	query	⊃	right shoe	⍝	cap null
⍵	omega	∩	cap	⍞	quote quad
∊	epsilon	∪	cup	!	quote dot
⍴	rho	⊥	base	⌹	domino
~	tilde	⊤	top	|	stile
↑	up arrow	;	semicolon	*	star
↓	down arrow	:	colon	⍳	iota
→	right arrow	,	comma	\	slope
←	left arrow	.	dot	/	slash
○	circle		space	⍙	delta underbar

Figure 3. APL Character Set

The names suggested are for the symbols themselves and not necessarily for the functions they represent. For example, the downstile (⌊) 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 on the previous page) 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 "Primitive Functions and Operators"). For example, + and × are primitive functions as are +/ and ×/ (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 unparenthesized 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 Arctan X, which means 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 heirarchy among functions, such as multiplication being done before addition or subtraction.

An equivalent statement of this rule is that an unparenthesized expression is evaluated in order from right to left. For example, the expression `3×8⌈3*|5-7` is equivalent to `3×(8⌈(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÷3)×2` is 8 and `12÷3×2` is 2. However, redundant pairs of parentheses can be used to help improve readability. Thus, the expressions `12÷3×2` and `12÷(3×2)` 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 variables.

Arrays

Data is organized in ordered collections called *arrays*. Arrays are characterized 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-dimentional arrays or 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:

```
      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 ⍴.

```
      M←2 4⍴V                B←2 4⍴A
      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 random shape and rank may be produced by the same scheme. For example:

```
      T←2 3 4⍴'ABCDEFGHIJKLMNOPQRSTUVWX'
      T
ABCD
EFGH
IJKL

MNOP
QRST
UVWX
```

The shape of an array can be determined by the monadic function denoted by ⍴ .

```
      ⍴V              ⍴M                ⍴T
8               2 4               2 3 4
```

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

```
      M[2;3]                T[2;1;4]
17                    P
      M[2 1;2 3 4]          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 ⎕IO (see "System Functions and System Variables"). Thus:

```
      ⎕IO←1                 ⎕IO←0
      V[1 2 3]              V[0 1 2]
2 3 5                 2 3 5
      B[2;3]                B[1;2]
G                     G
```

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 1.44*E*2 is equivalent to 144.

Negative numbers are represented by an *overbar* immediately preceding the number; for example, ¯1.44 and ¯144*E*¯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 organizational 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 random names. They occupy space on disk and cannot be worked with directly. When required, copies of stored workspaces can be made active, or selected information can be transferred from them into an active workspace.

Workspaces and libraries are managed by *system commands*, as described under "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. 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 alphameric characters (see your DOS book).
- Lowercase letters, delta, and delta underlined 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 32767.
- The maximum number of elements in a variable is 32767.
- The maximum size of an APL object is 32767 bytes.
- The maximum number of lines in a function is 1000.
- The maximum size of the symbol table is 8K bytes.
- The maximum size of the stack is 8K bytes.

An APL workspace consists of two parts:

- The *main workspace*. It occupies a maximum of 64K bytes. It is in this part where all APL statements are executed, and where all APL objects are 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. All APL objects not part of an execution are moved to the elastic workspace if the space they occupy in the main workspace is needed for other purposes.

Chapter 7. Primitive Functions and Operators

Notes:

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 ⎕ (as in ⎕*IO* and ⎕*CT*). These system variables are discussed in more detail in "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 on all real numbers with two general exceptions: the five boolean functions are defined only on the numbers 0 and 1, and the functions = and ≠ are defined on characters as well as numbers. Specific exceptions (such as 4 ÷ 0) will be noted where appropriate.

The scalar functions are summarized in Figure 4 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*		**f**	**Dyadic form** *A* **f** *B*	
Definition or Example	**Name**		**Name**	**Definition or Example**
+*B* is *B*	Conjugate	+	Plus	2+3.2 is 5.2
-*B* is 0-*B*	Negative	−	Minus	2-3.2 is ¯1.2
×*B* is (*B*>0)+*B*<0	Signum	×	Times	2×3.2 is 6.4
÷*B* is 1÷*B*	Reciprocal	÷	Divide	2÷3.2 is 0.625
|¯3.14 is 3.14	Magnitude	|	Residue	*A*|*B* is *B*-*A*×⌊*B*÷*A*+*A*=0
B ⌊*B* ⌈*B* 3.14 3 4 ¯3.14 ¯4 ¯3	Floor Ceiling	⌊ ⌈	Minimum Maximum	3⌊7 is 3 3⌈7 is 7
?*B* is Random choice from ⍳*B*	Roll	?	Deal	A Mixed Function (see Figure 8)
B* is (2.71828..)B*	Exponential	*	Power	2*3 is 8
⍟**B* is *B* is *⍟*B*	Natural logarithm	⍟	General logarithm	*A*⍟*B* is Log *B* base *A* *A*⍟*B* is (⍟*B*)÷⍟*A*
○*B* is *B*×3.14159...	Pi times	○	Circular, Hyperbolic,	Pythagorean (see table at left)
!0 is 1 !*B* is *B*×!*B*-1 or !*B* is Gamma (*B*+1)	Factorial	!	Binomial	*A*!*B* is (!*B*)÷(!*A*)×!*B*-*A* 2!5 is 10 3!5 is 10
~1 is 0 ~0 is 1	Not	~		
		∧	And	
		∨	Or	
		⍲	Nand	
		⍱	Nor	
		<	Less	Relations
		≤	Not greater	Result is 1 if the relation holds,
		=	Equal	0 if it does not:
		≥	Not less	3≤7 is 1
		>	Greater	7≤3 is 0
		≠	Not Equal	

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

(-*A*)○*B*	*A*	*A*○*B*
(1-*B**2)*.5	0	(1-*B**2)*.5
Arcsin *B*	1	Sine *B*
Arcos *B*	2	Cosine *B*
Arctan *B*	3	Tangent *B*
(¯1+*B**2)*.5	4	(1+*B**2)*.5
Arsinh *B*	5	Sinh *B*
Arcosh *B*	6	Cosh *B*
Artanh *B*	7	Tanh *B*

Table of Dyadic ○ Functions

Figure 4. 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 `÷`, since `X ÷ 1` is `X`; **zero** is a left or right identity of `+`; one is a left or right identity of `×`, and the general logarithm function `⍟` 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 `×`). These matters are discussed further in the treatment of the reduction operator, which concerns such applications of dyadic functions over vectors.

Figure 5 lists the identity elements of the dyadic scalar functions. The relational functions `<`, `≤`, `=`, `≥`, `>`, and `≠` 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		L	
Not greater	≤	1	Apply for	L	
Equal	=	1	boolean	L	R
Not less	≥	1	arguments		R
Greater	>	0	only		R
Not equal	≠	0		L	R

Notes:

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

Figure 5. 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 ÷ 0` is defined to give the value 1. For `X≠0` , the expression `X÷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≠0` , then `A|B` lies between `A` and `0` (being permitted to equal `0` but not `A`), and is equal to `B-N×A` for some integer `N` .

For example:

```
      1|2.385               ¯3| ¯3 ¯2 ¯1 0 1 2 3
0.385                0 ¯2 ¯1 0 ¯2 ¯1 0
      0|5.8                  3|¯3 ¯2 ¯1 0 1 2 3
5.8                  0 1 2 0 1 2 0
```

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* `÷X` is equivalent to `1÷X`. For example, if `X←4 ¯5` , then:

```
      +X          -X            ÷X
4 ¯5        ¯4 5        0.25 ¯0.2
```

The result of the *signum function* ×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××X. For example:

```
      ×¯3 0 4         |¯3 0 4
¯1 0 1           3 0 4
```

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

The monadic function **NOT** yields the logical complement of its argument; that is ~0 is 1, and ~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:

```
      3 5<5 3           3 5 7 ≠ 7 5 3
1 0                1 0 1
```

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* ⎕*CT*. Two scalar quantities **A** and **B** are considered to be equal if the magnitude of their difference does not exceed the value of ⎕*CT* multiplied by the larger of the magnitudes of **A** and **B**; that is, if (|*A*-*B*) is less than or equal to ⎕*CT*×(|*A*)⌈ |*B* Similarly, *A*≥*B* is considered to be true of (*A*-*B*) is greater than or equal to -⎕*CT*×(|*A*)⌈(|*B*), and *A*>*B* is considered true if *A*≥*B* is true and *A*=*B* is not.

The comparison tolerance ⎕*CT* is typically set to the value 1*E*¯13. The setting ⎕*CT*←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←0.666666666666667
      X
0.666666666666667
      Y←3×X
      Y-2
2.22044604925031E¯15
      2=Y
0
      ⎕CT←1E¯13
      2=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≠Y* is the *exclusive–OR* of X and Y, and *X≤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.

A	*B*						*A*	f	*B*								
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 *~X* or *~Y* or 0 or 1, as appropriate.

Minimum and Maximum

The dyadic functions, *minimum* and *maximum*, denoted by ⌊ and ⌈, 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 ⌊, yields the integer part of its argument; that is, ⌊*X* yields the largest integer that does not exceed X. Similarly, the *ceiling* function denoted by ⌈*X*, yields the smallest integer that is not less than X. For example:

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

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

```
      X←3×0.666666666666667
      ⎕CT←1E¯13                 ⎕CT←0
      ⌊X                        ⌊X
2                         2
      ⌈X                        ⌈X
2                         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 ⍳6 that is the first six integers beginning with either 0 or 1 according to the value of the index origin ⎕*IO.* For example:

```
      ⎕IO←1
      ?6              ?6             ?6
1               5              3
      ?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
      ⎕IO←0
      ?6 6 6 6 6 6 6 6 6 6 6 6
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 ⎕RL. The result of the roll function is system-dependent, but typically for X<2*31 is equal to ⎕IO plus the integer part of X×⎕RL÷¯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 generalized 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*÷N is the Nth 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÷3 and ¯8*2÷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 ⍟*X* is the inverse of the exponential; that is, ⋆⍟*X* and ⍟⋆*X* both equal X. For example:

```
      ⍟1 2 3 4
0 0.6931471806 1.098612289 1.386294361
      ⋆⍟1 2 3 4
1 2 3 4
      ⍟⋆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*⍟*X* is defined as (⍟*X*)÷⍟*B* . It is inverse to the power function in the following sense: *B*⋆*B*⍟*X* and *B*⍟*B*⋆*X* both equal X. Limitations on the domain follow directly from the defining expression.

Circular, Hyperbolic, and Pythagorean Functions

The symbol ○ denotes a monadic function whose result equals *pi* times its argument. For example:

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

The symbol ○ is also used dyadically to denote a family of 15 related functions as follows: the expression *I*○*X* is defined for integer values of *I* from ¯7 to 7, and is in each case equivalent to one of the circular, hyperbolic, or pythagorean functions, as indicated in Figure 4.

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

```
      PI←○1
      1○PI÷2 3 4
1 0.8660254038 0.7071067812
```

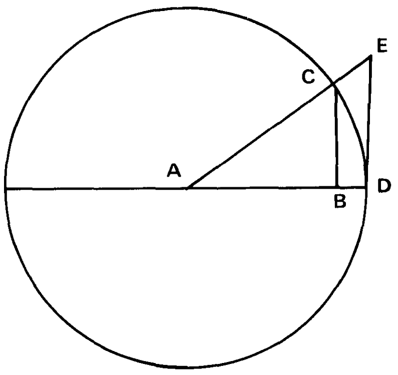
The *hyperbolic* functions, **SINH** and **COSH** (5○*X* and 6○*X*), are the odd and even components of the exponential function; that is, 5○*X* is odd, 6○*X* is even, and the sum (5○*X*)+(6○*X*) is equivalent to ⋆*X*. Consequently:

```
5○X equals .5×(⋆X)-(⋆-X)
6○X equals .5×(⋆X)+(⋆-X)
```

The definition of the hyperbolic tangent function, **TANH** (7○*X*), is similar to that of the tangent; that is 7○*X* equals (5○*X*)÷6○*X* .

The *pythagorean* functions 0○*X*, 4○*X* , and ¯4○*X* are defined as shown in Figure 4, and are related to the properties of a right triangle as indicated in Figure 6. They may also be defined as follows:

```
¯4○X equals 5○¯6○X
0○X equals 2○¯1○X  or 1○¯2○X
4○X equals 6○¯5○X
```



```
AC=1
AB=0○BC
BC=0○AB
AE=4○DE
DE=¯4○AE
```

Figure 6. The Pythagorean Functions

Each of the family of functions, `I○X`, has an inverse in the family; that is, `(-I)○X` is the inverse of `I○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←¯6○X   R≥0
       R←¯4○X   R≥0

Arctan R←¯3○X   (|R)≤○0.5

Arccos R←¯2○X   (R≥0)∧(R≤○1)

Arcsin R←¯1○X   (|R)≤○0.5
       R←0○X    R≥0
       R←4○X    R≥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×!N-1`, or equivalently, `!N-1` equals `(!N)÷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
      !¯.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 symbols `∘.` 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←1 2 3 4
      A∘.×A                           A∘.*A
1  2   3   4                1    1     1     1
2  4   6   8                2    4     8    16
3  6   9  12                3    9    27    81
4  8  12  16                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←1 2 3 4 5`, then `+/V` yields the sum of the items of `V`, and `×/V` yields their product:

```
      +/V               ×/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`:

```
      ⌈/V               1⌈2⌈3⌈4⌈5
5                 5
      -/V               1-2-3-4-5
3                 3
```

The last example emphasizes 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:

```
      A←1 ¯1 1 ¯1 1
      V×A
1 ¯2 3 ¯4 5
      +/V×A
3
      -/V
3
```

Similarly, `÷/V` yields the *alternating product*:

```
      V*A
1 0.5 3 0.25 5
      ×/V*A
1.875
      ÷/V
1.875
```

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 defined as the *identity element* of the function, if one exists, and as a *domain error* if one does not. Thus if *V* is an empty vector, +/*V* equals 0, and ∧/*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*←*P*,*Q* which is obtained by chaining them together. For example:

```
+/V equals (+/P)+(+/Q)
×/V equals (×/P)×(×/Q)
```

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

Scan

The *scan* operator is denoted by the symbol \ 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 *Kth* element of which is equal to the corresponding reduction over the first K elements of *V*. For example:

```
      +\1 2 3 4 5
1 3 6 10 15
      ×\1 2 3 4 5
1 2 6 24 120
      ∨\0 0 1 0 1
0 0 1 1 1
      ∧\1 1 0 1 0
1 1 0 0 0
      <\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 by 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 any array without the axis operator, the direction of application will be along the last axis. For example:

```
      ⎕←M←3 4⍴⍳12
 1    2    3    4
 5    6    7    8
 9   10   11   12
      +\[1]M                          +/[1]M
 1    2    3    4           15   18   21   24
 6    8   10   12
15   18   21   24
      +\[2]M                          +/[2]M
 1    3    6   10           10   26   42
 5   11   18   26
 9   19   30   42
      +\M                             +/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, ⎕IO. With ⎕IO←1, the leftmost or first axis has an index value of 1. The symbols ⌿ and ⍀ 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×Q` has a variety of useful interpretations. For example, if *P* is a list of prices and *Q* a list of corresponding order quantities, then `+/P×Q` is the total cost. Expressions of the same form using functions other than + and × are equally useful, as suggested by the following examples (where **B** is used to denote a boolean vector):

`∧/P=Q`	Comparison of *P* and *Q*
`+/P=Q`	Count of agreements between *P* and *Q*
`⌊/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×B`	Sum over a subset of *P* specified by *B*
`×/P*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+.×Q` is equivalent to `+/P×Q`, and `P×.*B` is equivalent to `×/P*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 7.

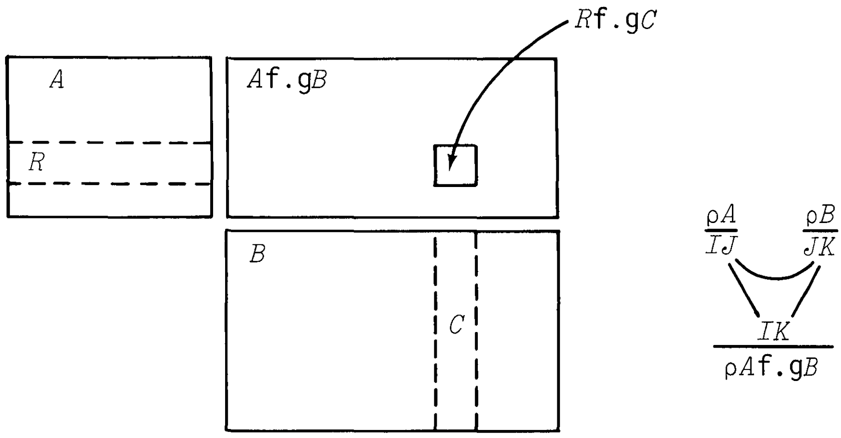


Figure 7. Inner Product

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

```
 ρA     ρB       ρC    ρD    ρE    ρF     ρG      ρH
---   -----     ---    --    --    ---    --      --
3 5    5 2 7    7 9    9     8     8 6     7       7
|  -----  | |   |  -----     ------  |     -------
|         | |   |                    |
3         2 7   7                    6     scalar
  ρAf.gB        ρCf.gD     ρEf.gF         ρGf.gH
```

Formally, ρAf.gB equals (¯1↓ρA),1↓ρB .

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

```
        P←2 3 5 7
        M←(⍳4)∘.≤⍳4

        M                   M∧.=M
1 1 1 1             0 0 0 1
0 1 1 1             0 0 0 0
0 0 1 1             0 0 0 0
0 0 0 1             0 0 0 0
        M+.×M               M-.×M
1 2 3 4              1  0  1  0
0 1 2 3              0 ¯1  0 ¯1
0 0 1 2              0  0  1  0
0 0 0 1              0  0  0 ¯1
        M+.×P               P×.*M
17 15 12 7          2 6 30 210
        P+.×M               M∧.=0 0 1 1
2 5 10 17           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+.×1` is equivalent to `+/A` , and `1+.×A` is equivalent to `+/A` .

Outer Product

The *outer product* operator, denoted by the symbols `∘.` 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←1 2 3` and `B←1 2 3 4 5` , then:

```
        A∘.×B                A∘.<B
1  2  3  4  5           0 1 1 1 1
2  4  6  8 10           0 0 1 1 1
3  6  9 12 15           0 0 0 1 1
```

Such tables may be better understood if they are labeled 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:

		B				
	×	1	2	3	4	5
	1	1	2	3	4	5
A	2	2	4	6	8	10
	3	3	6	9	12	15

		B				
	<	1	2	3	4	5
	1	0	1	1	1	1
A	2	0	0	1	1	1
	3	0	0	0	1	1

In the preceding example, the shape of the result `A∘.×B` is clearly equal to `(ρA),(ρB)`. This expression yields the shape for any arguments `A` and `B`. Thus, if `R←A∘.+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 8, 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 ⊖, ⍀, and ⌿, the first axis.

Figure 8 summarizes the restrictions on the ranks of arguments that may be used with each mixed function.

Name	**Sign (1)**	**Definition or Example (2)**
Functions Concerning the Structure of Arrays		
Shape	⍴*A*	⍴*P* is 4
		⍴*E* is 3 4
		⍴5 is ⍳0
Reshape	*V*⍴*A*	Reshape A to dimension V
		3 4⍴⍳12 is *E*
		12⍴*E* is ⍳12
		0⍴*E* is ⍳0
Ravel	,*A*	,*A* is (×/⍴*A*)⍴*A*
		,*E* is ⍳12
		⍴,5 is 1
Reverse	⌽*A*	*DCBA*
(3)		⌽*X* is *HGFE*
		LKJI
		IJKL
		⌽[1]*X* is ⊖*X* is *EFGH*
		ABCD
		⌽*P* is 7 5 3 2
Rotate	*A*⌽*A*	3⌽*P* is 7 2 3 5 is ¯1⌽*P*
(3)		*BCDA*
		1 0 ¯1⌽*X* is *EFGH*
		LIJK

Figure 8 (Part 1 of 4). Primitive Mixed Functions
See notes on page 7–29.

Name	Sign (1)	Definition or Example (2)
Functions Concerning the Function of Arrays (cont)		
Catenate, Laminate	*A*,*A*	*P*,⍳2 is 2 3 5 7 1 2 '*T*','*HIS*' is '*THIS*' *P*,[.5]*P* is 2 3 5 7 2 3 5 7
Transpose (4)	*V*⍉*A*	Coordinate I of A becomes coordinate *V*[*I*] of result *AEI* 2 1⍉*X* is *BFJ* *CGK* *DHL* 1 1⍉*E* is 1 6 11
	⍉*A*	Reverse order of coordinates ⍉*E* is 2 1⍉*E*
Functions Concerning Selection from Arrays		
Take	*V*↑*A*	2 3↑*X* is *ABC* ¯2↑*P* is 5 7 *EFG* Take or drop |*V*[*I*] first (*V*[*I*]≥0) or last (*V*[*I*]<0) elements of coordinate *I*
Drop	*V*↓*A*	2 3↓*X* is *L* ¯2↓*P* is 2 3
Compress (3)	*V*/*A*	1 0 1 0/*P* is 2 5 1 0 1 0/*E* is 1 3 5 7 9 11 1 0 1/[1]*E* is 1 2 3 4 is 1 0 1⌿*E* 9 10 11 12
Expand (3)	*V*\A	1 0 1\\⍳2 is 1 0 2 *A BCD* 1 0 1 1 1\X is *E FGH* *I JKL*
Indexing (4, 5)	*V*[*A*]	*P*[2] is 3 *P*[4 3 2 1] is 7 5 3 2
	M[*A*;*A*]	*E*[1 3;3 2 1] is 3 2 1 11 10 9
	A[*A*;.. ..;*A*]	*E*[1;] is 1 2 3 4 *E*[;1] is 1 5 9 *ABCD* '*ABCDEFGHIJKL*'[*E*] is *EFGH* *IJKL*

Figure 8 (Part 2 of 4). Primitive Mixed Functions
See notes on page 7—29.

Name	Sign (1)	Definition or Example (2)
Functions That Generate Selector Information		
Index Generator (4)	*ιS*	First S integers ι4 is 1 2 3 4 ι0 is an empty vector
Index of (4)	*VιA*	Least index of A in V, or 1+ρ*V* *P*ι3 is 2 5 1 2 5 *P*ι*E* is 3 5 4 5 5 5 5 5 4 4ι4 is 1
Membership	*AϵA*	ρ*W*ϵ*Y* is ρ*W* *P*ϵι4 is 1 1 0 0 0 1 1 0 *E*ϵ*P* is 1 0 1 0 0 0 0 0
Grade up (4)	⍋*V*	⍋3 5 3 2 is 4 1 3 2 The permutation that would order V (ascending or descending)
Grade down (4)	⍒*V*	⍒3 5 3 2 is 2 1 3 4
Grade up (dyadic) (4)	*A*⍋*A*	'*ABCDE*'⍋'*DEAL*' is 3 1 2 4
Grade down (dyadic) (4)	*A*⍒*A*	'*ABCDE*'⍒'*DEAL*' is 4 2 1 3
Deal (4)	*S?S*	*W?Y* is Random deal of W elements from ι*Y*
Functions That Involve Numeric Calculations		
Matrix inverse	⌹*M*	⌹2 2ρ1 1 0 1 is 1 ¯1 0 1 Arguments may be scalars, vectors, or matrices
Matrix division	*M*⌹*M*	(2 2ρ*P*)⌹2 2ρ1 1 0 1 is ¯3 ¯4 5 7
Decode	*A*⊥*A*	10⊥1 7 7 6 is 1776 24 60 60⊥1 2 3 is 3723
Encode	*A*⊤*A*	24 60 60⊤3723 is 1 2 3 60 60⊤3723 is 2 3

Figure 8 (Part 3 of 4). Primitive Mixed Functions
See notes on page 7–29.

Name	Sign (1)	Definition or Example (2)
Functions That Involve Data Transformation		
Execute	⍎*V*	⍎'1+2' is 3
		⍎'*P*' is 2 3 5 7
Format (Monadic)	⍕*A*	'¯1.5'∧.=⍕¯1.5 is 1
		ρ⍕*E* is 3 12
		X is ⍕*X*
Format (Dyadic)	*V*⍕*A*	4 1⍕*P* is 2.0 3.0 5.0 7.0
		8 ¯1⍕*P* is 2*E*000 3*E*000 5*E*000 7*E*000
		'0,55'⍕*P* is 0,020,030,050,07

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 9).

   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 ⌿ , ⍀ , and ⊖ are equivalent to / , \ , and ⌽ , 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 8 (Part 4 of 4). Primitive Mixed Functions

Figure 9 shows for what mixed functions and under what conditions scalar and vector arguments may be substituted for each other.

1. A scalar may be used in place of a one-element vector.
   a. as left argument of:

reshape	`3ρ4`	←→	`(,3)ρ4`
take	`3↑⍳5`	←→	`(,3)↑⍳5`
drop	`3↓⍳5`	←→	`(,3)↓⍳5`
expand	`1\,5`	←→	`(,1)\,5`
tranpose	`1⍉,5`	←→	`(,1)⍉,5`
format	`5⍕3.2`	←→	`(,5)⍕3.2` ←→ `0 5 ⍕3.2`

   b. as right argument of:

execute	`⍎'P'`	←→	`⍎,'P'`
branch	`→4`	←→	`→,4`

2. A scalar is extended to conform as necessary:
   a. as left argument of:

compress	`1/ ⍳3`	←→	`1 1 1 / ⍳3`
rotate	`1⌽2 2ρ⍳4`	←→	`1 1 ⌽ 2 2ρ⍳4`

   b. as right argument of:

compress	`1 0 1 / 2`	←→	`1 0 1 / 2 2 2`
expand	`1 0 1 \ 2`	←→	`1 0 1 \ 2 2`
take	`2 3 ↑3`	←→	`2 3 ↑ 1 1ρ3`

3. A one-element vector is permitted in place of a scalar.
   a. as left argument of:

compress	`(,1)/⍳3`	←→	`1/⍳3`
deal	`(,3)?5`	←→	`3?5`
rotate	`(,2)⌽2 3 5 7`	←→	`2⌽ 2 3 5 7`

   b. as right argument of:

index generator	`⍳,5`	←→	`⍳5`
deal	`3?,5`	←→	`3?5`

Figure 9. 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 two cases (compression and expansion), is further restricted to be boolean.

Shape, Reshape, and Ravel

The *shape* function is the monadic function ρ. 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 ρ*A* is the vector `3 4`.

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

Type of Array	ρ*A*	ρρ*A*
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←,A` then *V* is a 12-element vector containing the integers `2 4 6 8 10...24`. If *A* is a vector, then ,*A* is equivalent to *A*; if *A* is a scalar, then ,*A* is a vector of length 1.

The *reshape* function is the dyadic function `ρ`, which reshapes its right argument to the shape specified by its left argument. If `M←DρV`, then *M* is an array of dimension *D* whose elements are the elements of *V*. For example, `2 3ρ1 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 `DρV`, is equal to the dimension of the vector *V*, then the ravel of `DρV` is equal to *V*. If *N* is less than `ρV`, then only the first *N* elements of *V* are used; if *N* is greater than `ρV`, 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*⍴*A* is equivalent to *D*⍴,*A*. For example:

If:

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

Then:

```
      3 5⍴A
1 2 3 4 5
6 1 2 3 4
5 6 1 2 3
```

The expressions 0⍴*X* and 0 3⍴*X* and 0 0⍴*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*⍴*A* is a scalar.

Reverse and Rotate

The monadic function *reverse* is denoted by ⌽; if *X* is a vector and *K*←⌽*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:

```
      A              ⌽[1]A               ⌽[2]A
1 2 3          4 5 6               3 2 1
4 5 6          1 2 3               6 5 4
```

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

The dyadic function *rotate* is also denoted by ⌽. If K is a scalar or one-element vector, and X is a vector, then K⌽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⌽X is defined as X[1+(ρX)|¯1+K+ιρX]. For example, if X←2 3 5 7 11, then 2⌽X is equal to 5 7 11 2 3, and ¯2⌽X is equal to 7 11 2 3 5. In zero-origin indexing, the definition for K⌽X becomes X[(ρX)|K+ιρX].

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←K⌽[J]X. Moreover, the shape of K must equal the remaining dimensions of X, and each vector along the ***Jth*** 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 ρX is 3 4, and J is 2, the shape of K must be 3, and Z[I;] is equal to K[I]⌽X[I;]. If J is 1, ρK must be 4, and Z[;I] is equal to K[I]⌽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 ⌽[1]M                    1 2 3 ⌽[2]M
1  6 11  4                     2  3  4  1
5 10  3  8                     7  8  5  6
9  2  7 12                    12  9 10 11
```

The expression K⊖X denotes rotation along the first axis of X. The axis operator applies to ⊖, and K⊖[J]X is equal to K⌽[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←3 3ρ'ABCDEFGHI'
      M
ABC
DEF
GHI
      M,[1]M              M,[2]M                 M,M
ABC            ABCABC                ABCABC
DEF            DEFDEF                DEFDEF
GHI            GHIGHI                GHIGHI
ABC
DEF
GHI
```

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 *Ith* dimension. For example:

```
      V←'PQR'
      M,[1]V          M,[2]V          M,V
ABC         ABCP            ABCP
DEF         DEFQ            DEFQ
GHI         GHIR            GHIR
PQR
```

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

```
      C←'⌹'
      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 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                 M,[2.5]N
ABC                     A1
DEF                     B2
GHI                     C3

123                     D4
456                     E5
789                     F6
      M,[1.5]N
ABC                     G7
123                     H8
                        I9
DEF
456

GHI
789
```

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 2⍴'1234'
      B,[2.5]'×'
1×
2×

3×
4×

      ,B,[2.5]'×'
1×2×3×4×
```

Transpose

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

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

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

```
      A←2 3 5 7⍴⍳210
      ⍴A
2 3 5 7
      P←2 3 4 1
      ⍴P⍉A
7 2 3 5
```

More generally, `Q⍉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 `⍴⍴A` is 3, then `1 1 2` and `2 1 1` and `1 1 1` are suitable values for `Q`, but `1 3 1` is not. Just as for `P⍉A`, where `P` is a permutation, the *Ith* axis becomes the `Q[I]`th axis of `Q⍉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←3 3⍴⍳9                         B←3 5⍴⍳15
      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 1⍉A                            1 1⍉B
1 5 9                          1 7 13
```

The *monadic transpose* `⍉A` reverses the order of the axes of its argument. Formally, `⍉A` is equivalent to `(⌽⍳⍴⍴A)⍉A`. In particular, for a matrix `A`, this reduces to `2 1⍉A` 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 and compress, is further restricted to boolean.

Take and Drop

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

```
      3↑2 3 5 7                 7↑2 3 5 7
2 3 5                     2 3 5 7 0 0 0
      3↑'ABCDE'                 (7↑'ABCDE'),'⌹'
ABC                       ABCDE  ⌹
```

If *S* is a negative integer, then *S*↑*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:

```
      ¯3↑2 3 5 7                ¯7↑2 3 5 7
3 5 7                       0 0 0 2 3 5 7
```

If *A* is any array, then *W*↑*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 *Ith* axis of *A* . For example:

```
     A←3 4⍴⍳12
     A                           2 ¯3↑A
1  2  3  4                 2  3  4
5  6  7  8                 6  7  8
9 10 11 12
     2 3↑A                       ¯2 6↑A
1 2 3                     5  6  7  8  0  0
5 6 7                     9 10 11 12  0  0
```

The function *drop* (↓) is defined similarly, except that the indicated number of elements is dropped rather than taken. For example, ¯1 1↓*A* is the same matrix as the result of 2 ¯3↑*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 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←2 3 5 7 11` and `U←1 0 1 1 0`, then `U/X` is `2 5 7`, and `(~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` .

Expansion is the converse of compression and is denoted by `U\X`. If `Y←U\X`, then `U/Y` is equal to `X` and `(~U)/Y` is an array of zeros or spaces, depending on whether `X` is numeric or character. In other words, `U\X` expands `X` to the format indicated by the ones in `U` and fills in zeros or spaces. To be conformable, `+/U` must equal `ρX` . Continuing our previous example:

```
      D\C1
THIS IS AN EXAMPLE
```

The axis operator applies to both compress and expand and determines the axis along which they apply. If the axis operator is omitted, the last axis is used. The symbols ⌿ and ⍀ also denote compression and expansion, but when used without an axis operator, apply along the first axis.

For example:

```
      Q←3 4ρ'ABCDEFGHIJKL'
      Q
ABCD
EFGH
IJKL
      1 1 0 1\[1]Q              0 1 1 0/Q
ABCD                 BC
EFGH                 FG
                     JK
IJKL                             1 0 1/[1]Q
      1 1 0 1⍀Q      ABCD
ABCD                 IJKL
EFGH                             1 0 1⌿Q
                     ABCD
IJKL                 IJKL
```

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

Indexing

Indexing may be either 0-origin or 1-origin, as discussed in "Arrays." The following discussion assumes 1-origin. If X is a vector and I is a scalar, then X[I] denotes the *Ith* element of X. For example, if X←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 X 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 ⍴X[I] equals ⍴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⍴3 1 4 2 1 4 4 1 2 4 1 4
      I                   A[I]
3 1 4           CAD
2 1 4           BAD
4 1 2           DAB
4 1 4           DAD
```

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

In general, ⍴M[I;J] is equal to (⍴I),⍴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←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←2 4⍴3 1 4 2 1 4 4 1
      M                             M[;M]
3 1 4 2                       4 3 2 1
1 4 4 1                       3 2 2 3

                              4 1 1 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=×/⍴R` or `(1≠⍴L)/⍴L` must equal `(1≠⍴R)/⍴R`. For example:

```
      X←2 3 5 7 11
      X[1 3]←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 (`⍋`) is commonly used to produce indexes needed to reorder a vector into ascending order (as in `X[⍋X]`), but may also be used in the treatment of permutations as the inverse function; that is, `⍋P` yields the permutation inverse to `P` . Similarly, `⍳N` generates a vector of `N` successive indexes, but `.1×⍳N` generates a grid of values with an interval of `.1` .

Index Generator and Index Of

The *index generator* `ι` applies to a non-negative 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 `⎕IO`. For example, `ι5` yields `1 2 3 4 5` (in 1-origin) or `0 1 2 3 4` (in 0-origin), and `ι0` 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 `VιS` 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 `ιS` yields the first index outside the range of V; that is, `⎕IO+ρV`.

If S is any array, then `VιS` 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←Aι'HEAD CHIEF'
      J
8 5 1 4 27 3 8 9 5 6
      A[J]                      M←2 5ρ'HEAD CHIEF'
HEAD CHIEF                      M
      A[⌽J]         HEAD
FEIHC DAEH          CHIEF
      Aι'VAR3'                  AιM
22 1 18 28             8   5   1   4  27
                       3   8   9   5   6
```

Membership

The *membership* function, `X∊Y` , yields a boolean array of the same shape as X. Any particular element of `X∊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, `(ι7)∊3 5` is equal to `0 0 1 0 1 0 0` and `'ABCDEFGH'∊'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, `⍋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[⍋V]` yields the elements of V in ascending order. For example, if `V←8 3 7 5`, then `⍋V` is `2 4 3 1`, and `V[⍋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, yields `3 6 2 4 1 5`.

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

```
      A←7 2 5 11 3                 B←4 3 1 3 4 2
      ⍋A                           ⍋B
2 5 3 1 4                    3 6 2 4 1 5
      ⍒A                           ⍒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←L⍒R`

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

Collation 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' ⍒ 'DEAL'
4 2 1 3
      R ← 5 4⍴'DEALLEADDEADDEEDDALE'
      R
DEAL
LEAD
DEAD
DEED
DALE
     'ABCDE' ⍒ R
2 4 1 3 5
```

The last axis of L is the most significant for collating, 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 ← L ⍋ R

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

Collation 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' ⍋ 'DEAL'
3 1 2 4
     R ← 5 4⍴'DEALLEADDEADDEEDDALE'
     R
DEAL
LEAD
DEAD
DEED
DALE
     'ABCDE' ⍋ R
5 3 1 4 2
```

The last axis of L is the most significant for collating, 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 4⍴'dealDealdeadDeadDEED'
      R
deal
Deal
dead
Dead
DEED
      L ← 2 5⍴'abcdeABCDE'
      L
abcde
ABCDE
      Z ← L ⍋ 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 . 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 ⍳*N* . The expression, *P*[*M?*⍴*P*], selects M distinct elements from the population defined by the items of a vector **P**. For example:

```
          )CLEAR
          P←'ABCDEFGH'
          P[3?⍴P]                    P[(⍴P)?⍴P]
BGD                          EBAFHDGC
```

Numeric Functions

The numeric mixed functions apply only to numeric arguments and produce numeric results.

Matrix Inverse and Matrix Divide

The *domino* (⌹) 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, ⌹*A* (*matrix inverse*), yields the inverse of A, and *X*←*B*⌹*A* (*matrix divide*) yields a value of X that satisfies the relation ∧/ ,*B*=*A*+.×*X* and is therefore the solution of the system of linear equations conventionally represented as **AX=B**.

For example:

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

      B←1 3 6 10
      X←B⌹A                      A+.×X
      B                    1 3 6 10
1 3 6 10                         (⌹A)+.×B
      X                    1 2 3 4
1 2 3 4
      C←4 2⍴1 2 3 5 6 9 10 14
      Y←C⌹A
      C
 1  2
 3  5
 6  9
10 14
      Y               A+.×Y             (⌹A)+.×C
 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*⌹*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 ⌹*A* is equivalent to the matrix divide *I*⌹*A* . More generally, for any matrix **P**, the expression ⌹*P* is equivalent to the expression ((⍳*R*)∘.=⍳*R*)⌹*P*, where **R** is the number of rows in **P**.

The domino functions apply more generally to 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*←*B*⌹*A* is executed only if:

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

2. The columns of A are linearly independent.

If `X←B⌹A` is executable, then `⍴X` is equal to `(1↓⍴A),1↓⍴B`, and X is determined so as to minimize the value of `+/,(B-A+.×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 `⌹Y` is equivalent to `÷Y` and, except that it yields a domain error for `0⌹0` , the expression, `X⌹Y` , is equivalent to `X÷Y` .

The use of `⌹` for a non-square right argument can be illustrated as follows: if X is a vector, and `Y←F X`, then `Y⌹X∘.*0,⍳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⌹A` has certain useful geometric interpretations. If B is a vector, and A is a matrix, then saying that `+/(B-A+.×B⌹A)*2` is a minimum, is equivalent to saying that the length of vector `B-A+.×B⌹A` is a minimum. But `A+.×B⌹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←A+.×B⌹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)+.×A` is a zero vector.

If A and B are single-column matrices, then `B⌹A` is a 1-by-1 matrix, and `A+.×B⌹A` is equivalent to `A×S` , where S is the scalar `''⍴B⌹A` . If A and B are vectors, then `B⌹A` is a scalar, and the projection of B on A is therefore given by the simpler expression, `A×B⌹A` . For example:

```
A←4.5 1.7
B←2 5
      P←A×B⌹A
      P
3.403197926 1.28565255
      N←B-P
      N
¯1.403197926 3.71434745
      N+.×A
3.552713679E¯15
```

Similar analysis shows that if A is a vector, then `⌹A` is a vector in the direction of A; that is, `⌹A` is equal to `S×A` for some scalar S. Moreover, `A+.×⌹A` is equal to 1. In other words, `⌹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⊥X` yields the value of the vector X evaluated in a number system with radices `R[1]`,`R[2]`,...,`R[⍴R]`. For example, if `R←24 60 60` and `X←1 2 3` is a vector of elapsed time in hours, minutes, and seconds, then `R⊥X` has the value **3723**, and is the corresponding elapsed time in seconds. Similarly, `10 10 10 10⊥1 7 7 6` is equal to **1776**, and `2 2 2 ⊥1 0 1` is equal to **5**. Formally, `R⊥X` is equal to `+/W×X`, where **W** is the weighting vector determined as follows: `W[⍴W]` is equal to 1 and `W[I-1]` is equal to `R[I]×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⊥1 7 7 6` yields **1776**. The arguments are not restricted to integers; for example, if **X** is a scalar, then `X⊥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⊥B` is determined as the shape of the inner product, namely `(¯1↓⍴A),1↓⍴B`.

The *encode* or *representation* function `R⊤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⊥(R⊤X)` equals `(×/R)|X` rather than X. For example:

```
      10 10 10 10⊤123           10 10 10⊤123
0 1 2 3                     1 2 3
      10 10⊤123                 10⊤123
2 3                         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, E, whose representation is shown at the left below:

```
Z←A E B                         2 2 2⊤13
Z←0×A                   1 0 1
I←ρA                            ¯2 ¯2 ¯2⊤13
L:→(I=0)/0              ¯1 ¯1 ¯1
Z[I]←A[I]|B                     2 0 2⊤13
→(A[I]=0)/0             0 6 1
B←(B-Z[I])÷A[I]                 2 2 2⊤¯13
I←I-1                   0 1 1
→L                              ¯2 2 ¯2⊤13
                        0 1 ¯1
```

The basic definition of *R*⊤*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 10⊤215 486 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
      R⊤123
 1 1
 2 7
 3 3
```

The expression for the shape of the result of *R*⊤*X* is the same as for the shape of the outer product, namely (⍴*R*),⍴*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 ⍎ (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 → (branch symbol) or ⍝ (comment symbol), produces no explicit result, and therefore can be used only on the extreme left. For example:

```
      ⍎''
      Z←⍎''
VALUE ERROR
      Z←⍎''
      ∧
```

The domain of ⍎ is any character array of rank less than 2, and **RANK** and **DOMAIN** errors are reported in the usual way:

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

```
      ⍎3 4
DOMAIN ERROR
      ⍎3 4
      ∧
```

An error can also occur in the attempted execution of the APL expression represented by the argument of ⍎; such an indirect error is reported by the error type prefaced by the symbol ⍎ 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
      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×⍳4]               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←L⍕R`, the argument L may be a numeric or character vector.

> **Note:** You will get a **SYNTAX ERROR** message if you try to execute dyadic format and have not previously loaded **EXAPL** (see Chapter 1).

If numeric, **L** may be a single number, a pair of numbers, or a vector of length 2×¯1↑1,ρ*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 3⍕A
      12.340          ¯34.567
        .000           12.000
       ¯.260          ¯123.450
3 24
      R←9 2⍕A
      S←9 ¯2⍕A
      ρ⍞←R                        ρ⍞←6 0⍕A
    12.34   ¯34.57               12    ¯35
      .00    12.00                0     12
     ¯.26  ¯123.45                0   ¯123
3 18                     3 12
      ρ⍞←S                        ρ⍞←7 ¯1⍕A
 1.2E001 ¯3.5E001          1E001   3E001
 0.0E000  1.2E001          0E000   1E001
¯2.6E¯001¯1.2E002        ¯3E¯001¯1E002
3 18                     3 14
```

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:

```
      ρ⎕←2⍕A                    ρ⎕←¯2⍕A
  12.34  ¯34.57             1.2E001  ¯3.5E001
    .00   12.00             0.0E000   1.2E001
   ¯.26 ¯123.45            ¯2.6E¯001 ¯1.2E002
3 15                      3 20
      ρ⎕←0  2⍕A                 ρ⎕←0 ¯2⍕A
  12.34  ¯34.57             1.2E001  ¯3.5E001
    .00   12.00             0.0E000   1.2E001
   ¯.26 ¯123.45            ¯2.6E¯001 ¯1.2E002
3 15                      3 20
```

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

```
      ρ⎕←0 2 0 2⍕A              ρ⎕←8 3 0 2⍕A
  12.34  ¯34.57             12.340  ¯34.57
    .00   12.00               .000   12.00
   ¯.26 ¯123.45              ¯.260 ¯123.45
3 15                      3 16
      ρ⎕←6 2 12 ¯3⍕A            ρ⎕←8 0 0 ¯2⍕A
 12.34    3.46E001              12 ¯3.5E001
   .00    1.20E001               0  1.2E001
  ¯.26   ¯1.23E002               0 ¯1.2E002
3 18                      3 18
      6 2 8 3 3 0 4 0 5 0 12 4⍕,A
 12.34 ¯34.567  0  12    0   ¯123.4500
```

The format function applied to an array of rank greater than 2 applies to each of the planes defined by the last two axes. For example:

```
      )CLEAR
      L←2=?2 2 5ρ2
      L                    4 1⍕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
```

Tabular 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
      ROWHDS←4 3ρ'JANAPRJULOCT'
      YEARS←75+⍳4
      TBL←.001×¯4E5+?4 4ρ8E5
      (' ',[1]ROWHDS),(2⌽9 0⍕YEARS),[1]9 2⍕TBL
        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
      1 0⍕2=?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←(∨/R<0)+(~P∊0 ¯1)+(|P)
  +(5,⌈/0,(R≠0)+⌊10⍟|R+R=0)[1+P≥0]
  ```

  where **R** is the rounded value of **C** given by

  ```
  R←(⌊.5+C×10*|P)÷10*|P
  ```

- The expressions, `(M⍕A),N⍕B` and `(M,N)⍕A,B` are equivalent if **M** and **N** are full control vectors; that is, if `((ρM)=2×¯1↑ρA)∧(ρN)=2×¯1↑ρB`. If `2=ρM`, then `(M⍕A),M⍕B` and `M⍕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, `¯1↑⍴Z` will equal `K×⍴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 bordered 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 `⎕FC[3]` (* for "check-protection") when otherwise blank
9	Pad zeros outward to this position if non-zero

If more than one numeric control character appears in a field, each one controls the side of the field that it 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
      Z
28
```

If there is more than one field, there must be one for every column of numbers in R:

```
      Z ← ' 5 5.5 5.55' ⍕ 1.12 2.12 3.12
      Z
 1 2.1 3.12
      ⍴Z
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
      ⍴Z
15
```

Embedded decorators may be included:

```
      Z ← 'HERE: 5 5.5 ;THERE: 5.55' ⍕ 1.12 2.12 3.12
      Z
HERE: 1 2.1 ;THERE: 3.12
      ⍴Z
24
```

A single field may have embedded decorators:

```
      Z ← '05/05/05' ⍕ 70481
      Z
07/04/81
      ⍴Z
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 $100.00
      Z
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, or comma to end a field.

```
      Z ← '06/06/06' ⍕ 7 4 81
      Z
07/04/81
      ⍴Z
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
      ⍴Z
7
```

An 8 can be used in the field to have otherwise blank positions in the result filled with ⎕FC[3]:

```
      Z ← ' 8555.50' ⍕ 1 0 10 100
      Z
***1.00 ****.00 **10.00 *100.00
      ⍴Z
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
      ⍴Z
21
```

If ⎕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
      ⍴Z
21
```

For more examples, refer to the Format Control system variable (⎕FC).

Notes:

Chapter 8. System Functions and System Variables

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 or the environment of the system in which APL operates. This can be done within the language by identifying certain variables as elements of the interface between APL and its host system, and using these variables for 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* (⎕). 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 localized (see Chapter 10, "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, 17 system functions are provided. Six of these are for managing the shared-variable facility and are described in Chapter 9, "Shared Variables." The other 11 are shown in Figure 10 and are described after the figure.

	Requirements			
Function	**Rank**	**Domain**	**Effect on Environment**	**Explicit Result**
⎕*CR A*	1≥ρρ*A*	Array of characters	None	Canonical representation of object named by A. The result of anything other than an unlocked defined function is size 0 0.
⎕*DL S*	0=ρρ*S*	Numeric value	None, but requires S secs. to complete.	Scalar value of actual delay.
⎕*EX A*	2≥ρρ*A*	Array of characters	Erase objects named by rows of A, except labels or halted functions.	A boolean vector whose *Ith* element is 1 if the *Ith* name is now free.
⎕*FX M*	2=ρρ*M*	Matrix of characters	Fix definition of function represented by M, 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.
⎕*NC A*	2≥ρρ*A*	Array of characters	None	Vector giving the usage of the name in each row of A: **0** name available **1** label **2** variable **3** function **4** other
A ⎕*NL N*	1≥ρρ*N*	∧/*N*∊1 2 3 Elements of A must be alphabetic.	None	Same as monadic form, except only names starting with letters in A will be included.
⎕*NL N*	1≥ρρ*N*	∧/*N*∊1 2 3	None	Matrix of rows (in accidental order) that represents names of designated kinds in dynamic environment: 1, 2, 3 for labels, variables, and functions.

Figure 10 (Part 1 of 2). System Functions

	Requirements			
Function	Rank	Domain	Effect on Environment	Explicit Result
A ⎕*EA* *B*	1≥ρρ*B* 1≥ρρ*A*	Characters	None	Executes B. For error, executes A.
N ⎕*PK* *A*	0=ρρ*N*	N is a scalar positive integer.	None	Peek memory contents. Result is character vector of elements of ⎕*AV*.
	1=ρρ*A*	A is a numeric vector of two elements.		
	1≥ρρ*N*	Character scalar/ vector.	Changes memory	Poke memory contents. Result is character vector. with previous contents.
	1=ρρ*A*	Numeric vector of 2 elements.		
⎕*PK* *A*	1=ρρ*A*	See dyadic ⎕*PK*	Depends on user programs.	Executes machine language program. Returns register contents and flags.
⎕*TF* *A*	1≥ρρ*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 10 (Part 2 of 2). System Functions

Canonical Representation – ⎕CR

The *canonical representation* of a defined function, as defined in Chapter 10, is obtained by applying the system function ⎕*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 ⎕*CR* are **RANK** error, if the argument is not a vector or a scalar, or **DOMAIN** error if the argument is not a character array. The use of ⎕*CR* is further described in Chapter10.

Delay – ⎕DL

The *delay* function, denoted by ⎕*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 ⎕*DL* is not a scalar with a numeric value, a **RANK** or **DOMAIN** error will be reported.

The delay function can 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 interactions between users and programs.

Execute Alternate: – ⎕EA

If you execute the statement

```
Z←L ⎕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. In that case, **Z** is the value of the APL expression in L. If the expression has no value, then *L* ⎕*EA R* has no value. Execution of L is subject to normal error handling.

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 the expression has no value, then *L* ⎕*EA R* has no value.

Examples:

```
      'ι2' ⎕EA 'ι4'
1 2 3 4
      'ι2' ⎕EA 'ι4.5'
1 2
      '→' ⎕EA 'ι4.5'
      'ι2.3' ⎕EA 'ι4.5
DOMAIN ERROR
      ι2.3
      ∧
```

If R calls a defined function F, then the statements executed by F are also under control of the error trap. In particular, R could call a long running function, and L could be an error recovery function.

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 defintion of a function with the same name.

An expression of the form *⎕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 "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 *⎕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 – ⎕NC

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

The result of ⎕*NL* is a suitable argument for ⎕*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 ⎕*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 – ⎕NL

The dyadic function ⎕*NL* yields a character matrix, each row of which represents the name of an object in the dynamic environment. The right argument is an integer scalar or vector 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 names produced to those with an initial letter occurring in the argument. The ordering of the rows of the result is random.

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

The uses of ⎕*NL* include the following:

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

Peek/Poke – ⎕PK

This function has three different uses that may be requested as follows:

1. Peek the memory contents.

   ```
   R←N ⎕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 workspace origin.

   R is a character vector with the contents of the selected memory positions as elements of ⎕*AV* (that is, if the contents of a byte is 120, the corresponding result will be the 120th element of ⎕*AV* in zero origin).

2. Poke the memory contents.

   ```
   R←V ⎕PK AR,ADDR
   ```

   where:

   V is a character vector with the values to be inserted in memory as elements of ⎕*AV*.

   AR and **ADDR** are interpreted as in 1 above.

   R is the previous contents of the changed memory.

3. Execute memory.

```
R←⎕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 language instruction (it is considered as a far procedure). The program must return the stack as it was found on entry.

Transfer Form – ⎕TF

In the expression:

```
Z←⎕TF R
```

if **R** is the name of a variable or a defined function, then **Z** is a character vector, which 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 if 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 *migration 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 ⎕*TF R*, where **R** is the name of the object.

The migration transfer form vector consists of 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.
3. A character representation of the rank and shape of the array, followed by a blank.
4. A character representation of the array elements in row major order (any numeric conversions are done to 15 digits).

A defined function is treated as the character matrix of its canonical form.

Examples:

```
      THIS ← 2 3ρι6
      Z ← ⎕TF 'THIS'
      Z
NTHIS 2 2 3  1 2 3 4 5 6
      ρZ
24
      THAT ← 3 4ρ'ABCDEFGHIJKL'
      Z ← ⎕TF 'THAT'
      Z
CTHAT 2 3 4 ABCDEFGHIJKL
      ρZ
24

    ∇ Z←L PLUS R
[1]   Z←L+R
    ∇

      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←ιN
[3]   V←(1=0+.=(1↓M)∘.|M)/M
    ∇

      Z ← ⎕TF 'PRIMES'
      Z
FPRIMES 2 4 21 V←PRIMES N;⎕IO;M      ⎕IO←1
        M←ιN                        V←(1=0+.=(1↓M)∘.|M)/M
      ρZ
99
```

System Variables

System variables are instances of *shared variables* (see "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 localized in a function, each time the function is used.

Figure 11 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, random link, printing precision, and printing width. In these cases, the value you specify (or that is 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 localization, 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 work area. In these cases, localization 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
⎕*CT*	1*E*¯13	0-1*E*¯4	Comparison tolerance used in monadic ⌈⌊ dyadic <≤=≥>≠∊⍳
⎕*FC*	.,*0_		Format control used in dyadic ⍕ (picture format)
⎕*HT*	⍳0		This variable is ignored by the system.
⎕*IO*	1	0 1	Index origin: used in indexing and in ? ⍳ ⍋ ⍒ ⍉ ⎕*FX*
⎕*LX*	''	characters	Latent expression executed on activation of workspace
⎕*PP*	10	⍳15	Printing precision: affects numeric output and monadic ⍕
⎕*PW*	79	30-80	Printing width: affects all but bare output and error reports
⎕*RL*	7*5	⍳¯1+2*31	Random link: used in ?
⎕*AI*			Account information: identification, computer time, connect time, keying time (all times in milliseconds and cumulative during sessions)
⎕*AV*			Atomic vector
⎕*LC*	⍳0		Line counter: statement numbers of functions in execution or halted, most recently activated first
⎕*TS*			Time stamp: year, month, day (of month), hour (on 24-hour clock), minute, second, millisecond

Figure 11 (Part 1 of 2). System Variables

Name	Value in Clear WS	Meaningful Range	Purpose
⎕*TC*			Terminal control: a three-element vector containing the backspace, new line, and line-feed characters, in that order
⎕*TT*			Terminal type: always zero
⎕*UL*			User load: always 1
⎕*WA*			Working area available (in bytes): main workspace size plus elastic workspace size

Figure 11 (Part 2 of 2). System Variables

Latent Expression – ⎕LX

The APL statement represented by the latent expression is automatically executed whenever the workspace is activated.

Formally, ⎕*LX* is used as an argument to the execute function ⍎⎕*LX*, and any error message, will be appropriate to the use of that function.

Common uses of the latent expression include the form ⎕*LX*←'*G*', used to invoke an arbitrary function **G**; the form,

```
LX←''' FOR CHANGES IN THIS WS ENTER:  NEW'''
```

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

```
      ⎕LX←'F'
    ∇ F;⎕LX
[1]   ⎕LX←'→⎕LC,ρ⍞←'' RESUME LESSON'''
[2]   'WE NOW BEGIN LESSON 2'
[3]   DRILLFUNCTION
    ∇
      )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.

Atomic Vector – ⎕AV

The *atomic vector* ⎕*AV* is a 256-element character vector, containing all possible characters. Certain elements of ⎕*AV* may be screen-control characters, such as carriage return or line feed. The indexes of any known characters can be determined by an expression such as ⎕*AV*⍳'*ABC*abc'.

Format Control – ⎕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:

⎕*FC*[1]	Use for decimal point
⎕*FC*[2]	Use for comma
⎕*FC*[3]	Fill when otherwise blank for digit 8
⎕*FC*[4]	Fill when otherwise **DOMAIN ERROR** for overflow
⎕*FC*[5]	Print as blank (may not be , .0123456789)

Elements of ⎕*FC* beyond 5 are not defined.

⎕*FC*[1] is used wherever a decimal point is needed in picture format:

```
      ⎕FC[1] ← ','
      '5.5555' ⍕ 3.1415
3,1415
```

⎕*FC*[2] is used wherever a comma is needed in picture format:

```
      ⎕FC[2] ← '.'
      '555,555,555' ⍕ 123456789
123.456.789
```

⎕*FC*[3] is used where a field containing an 8 would otherwise be blank in picture format:

```
      ⎕FC[3] ← '⎕'
      '855555' ⍕ 1234
⎕⎕1234
```

⎕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 ⎕FC[4] is '0' (which is the default), then a field that is too small will result in **DOMAIN** error.

⎕FC[5] is replaced by a blank without ending a field wherever it is used in picture format:

```
      ⎕FC[5] ← '⊖'
      '$⊖355' ⍕ 12
$ 12
```

Horizontal Tabs – ⎕HT

The value of ⎕HT is ignored by the IBM Personal Computer APL system.

Notes:

Chapter 9. Shared Variables

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 appeear 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 “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. Six system functions are provided for these purposes – three for the actual management, and three to provide related information. These are summarized in Figure 12.

	Requirements [1]				
Function	**Rank**	**Length**	**Domain**	**Effect on Environment**	**Explicit Result**
P ⎕*SVO N*	2≥⍴⍴*N*	(×/⍴*P*)∊1,¯1↓⍴*N*	*P*∊1↓⍳2*15 [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. Dimension: ,×/¯1↓⍴*N*.
⎕*SVO N*	2≥⍴⍴*N*	None	[2]	None	Degree of coupling now in effect for the name pair. Dimension: ,×/¯1↓⍴*N*.
C ⎕*SVC N*	2≥⍴⍴*N* 2≥⍴⍴*C*	(1≥⍴⍴*C*)∧1=×/⍴*C* or (⍴*C*)=(¯1↓⍴*N*),4	∧/*C*∊0 1 [2]	Sets access control.	New setting of access control. Dimension: (¯1↓⍴*N*),4.
⎕*SVC N*	2≥⍴⍴*N*	None	[2]	None	Existing access control.

Figure 12 (Part 1 of 2). System Functions

Function	Requirements [1]			Effect on Environment	Explicit Result
	Rank	Length	Domain		
⎕SVR P	2≥⍴⍴N	None	[2]	Retracts offer (ends sharing)	Degree of coupling before retraction. Dimension: ,×/¯1↓⍴N
⎕SVQ N	1≥⍴⍴P	1≥⍴,P	P∊1↓⍳¯1+2*15	None	If 0=⍴P: Vector of IDs of processors making offers to this user. If 1=×/⍴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≥⍴⍴N) must represent a name or pair of names. If a pair of names is used for an offer (dyadic ⎕SVO), either the pair, or the first name only, can be used for the other functions.

Figure 12 (Part 2 of 2). System Functions

Offers

A single offer to share is of the form *P* ⎕SVO *N*, where **P** is the identification of another processor, and **N** is a character 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 substitute name. The name of the variable may be its own substitute, in which case only the one name need be used, rather than two.

The substitute names have no effect.

The explicit result of the expression *P* ⎕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) increases 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 ⎕*SVO* does not affect the coupling of the name represented by its argument, but does not report the degree of coupling as its explicit result. If the degree of coupling is 1 or 2, a repeated offer has no further implicit result, and either monadic or dyadic ⎕*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**:

```
Z←OFFER P;Q
⍞←'NAME:'
→('  '∧.=Q←⍞)/Z←0
Z←P ⎕SVO Q
→(2=Z←⎕SVO Q)/L
'NO DEAL'
→0
L:'ACCEPTED'
```

If the arguments of ⎕*SVO* fail to meet any of the basic requirements listed in the following figure, 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, that value obtains; if neither had, the variable has no value. Names used in sharing are subject to the usual rules of localization.

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 which rows represents a name or a name pair. 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.

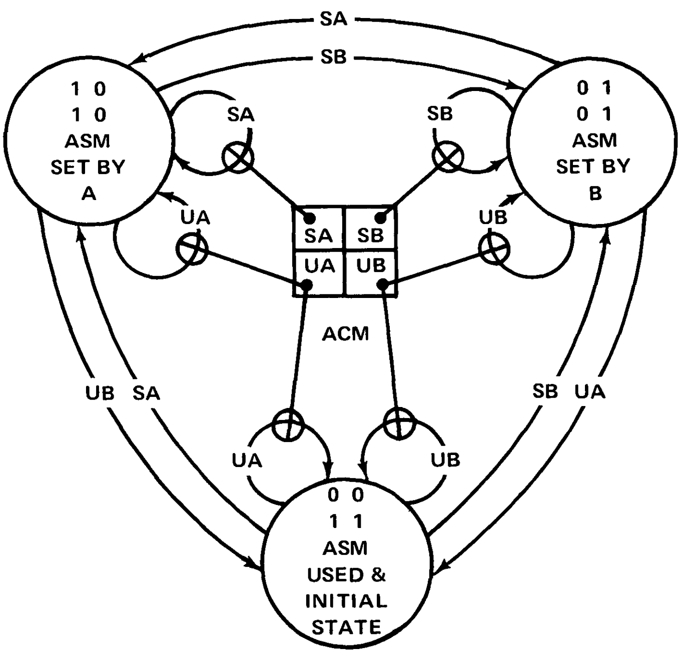
Access Control

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 `⎕SVC`. If one user had followed his offer to share V by the expression `1 1 1 1 ⎕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 13 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∧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.



Legend:

SA SB UA UB: Denote *set* or *use* by A or B.
ACM: Access Control Matrix
ASM: Access State Matrix

A one in an element of ACM inhibits the associated access. Allowable accesses are given by the zeros in $ACM \wedge ASM$. Access control vectors as seen by A and B, respectively, are $,ACM$ and $,\phi ACM$.

The access state matrix represents the last access: ones occurs in the last row if it is not a set, and in a column if it is, the first column if set by A and the last if set by B.

Figure 13. 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 ⎕*SVC*. For a shared variable V, the result of the expression ⎕*SVC* '*V*' executed by user A is the *access control vector* ,*ACM* (the four-element ravel of ACM). However, if user B executed the same expression, he would obtain the result ,⊖*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* ⎕*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←(2 2ρQA)∨⊖2 2ρQB
```

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 ⎕*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 14 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 13 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 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 14. Some Useful Settings for the Access Control Vector

A group of N access control matrices can be set at once by applying the function ⎕*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 (ravels 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)⍴*L*.

Retraction

Sharing offers can be *retracted* by the monadic function ⎕*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. This requirement obtains 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 `⎕SVO` and `⎕SVC`, already discussed, and the function `⎕SVQ`. A user who applies the `⎕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≠⎕SVO M)/[1] M←⎕NL 2` can be used to produce a character matrix whose rows represent the names of all shared variables in the dynamic environment.

Notes:

Chapter 10. Function Definition

Notes:

A defined function can be established in an APL workspace in three ways:

1. It can be copied from a stored workspace using a system command, as described in "System Commands."

2. It can be established in execution mode, using the system function ⎕*FX*, 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." 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
      ⍴DEF
3 21
```

The function ⎕*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 ⎕*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 ⎕*FX*:

```
      ⎕FX DEF
OVERTIME
```

The function ⎕*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 ⎕*FX M* will establish a function if the conditions described under "Function Establishment" 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:

Type	Valence	Result	No Result
Dyadic	2	*R←A F B*	*A F B*
Monadic	1	*R←F B*	*F B*
Niladic	0	*R←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 Nth root of its right argument. If no left argument is provided when the function is called, a default value is supplied:

```
    ∇ 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 11, "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. However, system commands and the *del* form of function definition (see below) continue to reference global objects under these circumstances.

The localization 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 localized 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 localized in an outer function has the significance assigned to it in that function when used without further localization in an inner function. The same name localized in a sequence of nested functions has the significance assigned to it at the inner-most level of execution. The *shadowing* of a name by localization is complete, in the sense that once a name has been localized, its global and outer values are nullified, even if no significance is assigned to it during execution of the function in which it is localized.

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 →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 →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:

```
Z←M GCD N
E:Z←M
M←M|N
N←Z
→(0≠M)/E
```

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 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 ⍝ (the *cap-null*) signifies that what follows is a comment for illumination only and is not to be executed; it may occur only as the first character in a statement, or as the first character following a label and colon.

Function Editing – The ∇ Form

The functions ⎕*CR* and ⎕*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 (∇) 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 ∇. 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, `[∆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 `[∆2 3 5]`.

Adding to 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⎕0]`. 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⎕0]`.

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. Labeled lines and comments are offset one space to the left.

2. Statement numbers in brackets are appended to the left of the statements.

3. A *del* character (∇) is prefixed to the header, separated by one space.

4. A final line is added, consisting of spaces and a *del* character, aligned with the *del* character which prefixes the header.

Figure 15 shows the canonical representation and function display of a function for computing the determinant of a matrix.

```
      ⎕CR'DET'                                   ∇DET[⎕]∇
Z←DET A;B;P;I                               ∇ Z←DET A;B;P;I
I←⎕IO                                 [1]    I←⎕IO
Z←1                                   [2]    Z←1
L:P←(|A[;I])⍳⌈/|A[;I]                 [3]   L:P←(|A[;I])⍳⌈/|A[;I]
→(P=I)/LL                             [4]    →(P=I)/LL
A[I,P;]←A[P,I;]                       [5]    A[I,P;]←A[P,I;]
Z←-Z                                  [6]    Z←-Z
LL:Z←Z×B←A[I;I]                       [7]   LL:Z←Z×B←A[I;I]
→(0  1  ∨.=Z,1↑⍴A)/0                  [8]    →(0 1 ∨.=Z,1↑⍴A)/0
A←1 1 ↓A-(A[;I]÷B)∘.×A[I;]            [9]    A←1 1 ↓A-(A[;I]÷B)∘.×A[I;]
→L                                    [10]  →L
⍝EVALUATES A DETERMINANT              [11]  ⍝EVALUATES A DETERMINANT
                                            ∇
```

Figure 15. Canonical Representation and Function Display

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 by typing a ∇ on a line by itself, or as the last character on any entry. 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 11.

Notes:

Chapter 11. Function Execution

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**.

```
PAY←R OVERTIME H;TIME
TIME←0⌈H-40
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 treated 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 ⎕*LC*. Entering →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. Pendent functions can be edited with the del (∇) 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, ⎕*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 (→). This behavior is illustrated by continuing the foregoing example as follows:

```
      →
      )SI
* H[7]
  G[2]
  F[3]
      ⎕LC
7 2 3
```

Repeated use of → will clear the state indicator completely and restore ⎕LC to an empty vector. The cleared state indicator displays as if a blank line had been enetered. The same effect can be obtained with commands)SI CLEAR or)RESET.

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 ¯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 analyzing the behavior of a defined function, particularly during its design.

The tracing of a function **PROFIT** is controlled by the trace control for PROFIT, denoted by *T∆PROFIT*. If one sets *T∆PROFIT*←2 3 5, then statements 2, 3, and 5 will be traced in any later execution of PROFIT. *T∆PROFIT*←⍳0 discontinues tracing of PROFIT. A complete trace of PROFIT is obtained by *T∆PROFIT*←⍳*N*, where **N** is the number of statements in PROFIT. In general, the trace control for any function is designated by prefixing *T∆* 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 analyzing 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 *S∆PROFIT*←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 ⍫ (called *del-tilde*) is used instead of ∇ 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 behavior 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)/0
R←(A|B)GCDR A

      18 GCDR 45
9
```

This can be compared to the equivalent function GCD defined iteratively in Chapter 10.

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.

Console Input and Output

In many significant applications, such as text processing, for example, it is necessary that you supply information as the execution of the application programs 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**:

```
      ⎕CR'CI'
A←R CI T
A←(1+R)*T
```

For example, the value of 1000 dollars at 5 percent for 7 years, compounded quarterly, might be found by:

```
      1000 × (.05÷4) CI 7×4
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 DOLLARS
⎕:
      1000
ENTER NUMBER OF TIMES COMPOUNDED IN ONE 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 DOLLARS'
C←⎕
'ENTER NUMBER OF TIMES COMPOUNDED IN ONE 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 ⎕ input is executed, but the system's response to the command is not treated as a response to ⎕ . 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 ⍞ (that is, a *quad* overstruck with 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←⍞
CAN'T               (Quote-quad input, not indented)
      X
CAN'T
```

Interrupting Execution During Input

The response → entered in response to ⎕ abandons execution of the function and any pendent functions leading up to it.

A request for ⍞ input can be interrupted by pressing the Esc key.

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 ⎕PP and the printing width ⎕PW). Hence, ⎕←X is equivalent to the statement X. The longer form ⎕←X is useful when employing multiple specification. For example,⎕←Q←X*2 assigns to Q the value X*2, then prints the value of X*2.

The maximum length of a line or 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 ⍞←*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.

Notes:

Chapter 12. System Commands

Notes:

An APL system recognizes 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 can be invoked only by individual entries from the keyboard and cannot be executed dynamically as part of a defined function. They are prefixed by a right (closing) parenthesis.

The system commands are summarized in Figure 16, 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
Active Workspace — Action Commands			
)*CLEAR*	Activate a clear WS	*CLEAR WS*	4
)*SYMBOLS* pi	Set size of symbol table	*WAS* number *CLEAR WS*	4
)*STACK* pi	Set size of execution stack	*WAS* number *CLEAR WS*	4
)*ERASE* nms	Erase objects from active WS		4, 6, 8
)*IN* wsid	Copy all objects from WS to active WS	*SAVED* time date	1, 2, 4, 9, 10, 12
)*IN* wsid nms	Copy named objects from WS to active WS	*SAVED* time date	1, 2, 4, 5, 9, 10, 12
)*SI CLEAR*	Clear the state indicator		4
)*RESET*	Clear the state indicator		4
Active Workspace — Inquiry Commands			
)*SYMBOLS*	Give size of symbol table and available space in bytes	number number	4
)*STACK*	Give size of execution stack	number	4
)*FNS*	List defined functions	(names)	4
)*VARS*	List variables	(names)	4
)*SI*	List halted functions	state indicator	4
)*SINL*	List halted functions and names	state indicator and names	4
Workspace Storage and Retrieval — Action Commands			
)*WSID* wsid	Change ID of active workspace	*WAS* wsid	4
)*SAVE* wsid	Replace named WS with copy of active WS	time date	3, 4, 7, 12
)*SAVE*	Place copy of active workspace in library	time date	3, 4, 7, 12
)*LOAD* wsid	Activate copy of named workspace	time date	1, 4, 11

Figure 16 (Part 1 of 2). System Commands

Form	Purpose	Normal Response	Trouble Reports
)*OUT* wsid	Generate a file in transfer form with all objects in the active workspace	time date	2, 3, 4
)*OUT* wsid nms	Generate a file in transfer form with the objects in nms	time date	2, 3, 4, 5
)*DROP*	Drop workspace or file from library		1, 4
Workspace Storage and Retrieval — Inquiry Commands			
)*WSID*	Give identification of active workspace	(number) name	4
)*LIB*	List workspaces or files in desired library	(names)	4, 12
Access to the System			
)*OFF*	End use of APL		4

Notes:

1. Items in parentheses are optional.
2. Abbreviations and Meanings:
   - **WS:** workspace
   - **wsid:** a workspace 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 16 (Part 2 of 2). System Commands

A system command that is not recognizable, 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 summarized in Figure 17.

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.

No.	Message	Meaning	Remedy
1	*NOT FOUND*	No stored workspace with given ID	
2	*NOT WITH UNCLEAR SI*	)IN and)OUT cannot be executed if the SI is not clear	Clear the SI
3	*LIB FULL*	No room on the disk	Change the disk or drop unneeded WS
4	*COMMAND ERROR*		
5	nms *NOT FOUND*	Workspace does not contain objects with purported names	
6	nms *NOT ERASED*	Purported names could not be erased	
7	*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	Give a name to the workspace OR Rename active workspace then store
8	*SI DAMAGE*	State indicator damaged by)ERASE	Clear SI
9	*SYMBOL TABLE FULL*	Too many names used	Erase objects not needed)*OUT* wsid)*CLEAR*)*SYMBOLS* n)*IN* wsid)*WSID* wsid
10	*WS FULL*	Workspace full	1. Erase unneeded objects 2. Clear SI
11	*WS TOO LONG*	Workspace does not fit in main storage	
12	*I/O ERROR*	The door of the drive you want to access is open, or diskette is not inserted correctly, or diskette is write-protected.	**INTERRUPT** message will appear and **STRONG** interrupt will occur. No I/O operation will be performed.

WARNING: Changing diskettes during an input/output operation, or when you have open files, may damage your diskette.

Figure 17. Trouble Reports

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 18. In use, the appropriate names or numbers should, of course, be substituted.

A	A letter of the alphabet
LIBNO	A library number (that is, the number of a disk drive). If this field is not given, the default drive is assumed.
WSNAME	A workspace name
FILENAME	A DOS file name
EXT	A DOS file extension
NAME	A string formed by numbers and uppercase letters, starting with a letter
OBJ	The name of an object within a workspace (that is, a function or a variable)
( )	Items enclosed in parentheses may, in some cases, be omitted.
< >	Items enclosed in angles may, in some cases, be omitted; when items are omitted, the system supplies default values.

Figure 18. 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 11) and other elements of the computing environment, mediated by several of the system variables (discussed in Chapter 8, "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 19). At sign-on, you receive a clear workspace characterized by these same initial values.

Symbol table size	2000 bytes
Horizontal tabs, ⎕*HT*	Empty
Index origin, ⎕*IO*	1
Latent expression, ⎕*LX*	Empty
Line counter, ⎕*LC*	Empty
State indicator	Cleared
Workspace name	None (CLEAR WS)
Printing precision, ⎕*PP*	10
Printing width, ⎕*PW*	79
Comparison tolerance, ⎕*CT*	1*E*¯13
Random link, ⎕*RL*	16807
Format control, ⎕*FC*	.,*0_
Work area available, ⎕*WA*	Depends on PC memory size

Figure 19. Environment within a Clear Workspace

```
)SYMBOLS N OR )STACK N
```

Sets the size of the symbol table or the execution stack, in bytes. New values of the maximum may be set only in a clear workspace. 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 report **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> WSNAME <OBJ1 (OBJ2...)>
```

The indicated objects or system variables are copied from the indicated transfer file (**WSNAME**) into the active workspace. The system reports the date and time at which the transfer file was last saved.

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 a global 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. This command can only be executed if no function is pending in the state indicator. The message **NOT WITH UNCLEAR SI** will appear if that is not the case, and the command is abandoned.

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.

- I/O ERROR

  — The door of the drive you want to access is open, or

  — The inserted diskette is not the correct one.

`)RESET` or `)SI CLEAR`

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 bytes.

)FNS

Reports a list of the functions in the active workspace, in ⎕*NL* order.

)VARS

Reports a list of the variables in the active workspace, in ⎕*NL* order.

)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 "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). Library identifications are usually consecutive numbers. You must be careful not to use numbers corresponding to nonexistent drives, because the action of the system is unpredictable. (Usually it will try to perform the requested operation in one of the existing drives.)

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 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.

Although saving does not affect the state of sharing in the active workspace, current values of the shared variables are saved in the stored copy.

Saving is acknowledged by a report showing the date and time at which the workspace was 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 accidently 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.

- **I/O ERROR**

  — The door of the drive you want to access is open, or

  — The inserted diskette is not the correct one or is write-protected.

)OUT <LIBNO> WSNAME <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.

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.

- **I/O ERROR**

  — The door of the drive you want to access is open, or

  — The inserted diskette is not the correct one or is write-protected.

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 (*⎕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 *)SAVE* command cannot be *)LOAD*ed, even though its extension is **.APL**.

- **I/O ERROR**

  — The door of the drive you want to access is open, or

  — The inserted diskette is not the correct one.

)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.

- **I/O ERROR**

  — The door of the drive you want to access is open, or

  — The inserted diskette is not the correct one or is write-protected.

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:

`)LIB 1`	Lists all files in the drive 1
`)LIB 1 APL`	Lists all files with name starting with "APL"
`)LIB 1 .APL`	Lists all saved workspaces
`)LIB 1 .AIO`	Lists all workspaces in transfer form
`)LIB 1 .EXE`	Lists all EXE files in the drive
`)LIB 1 .`	Lists all files with a blank extension
`)LIB AP .EXE`	Lists all files on the default drive with names starting with "AP" and having an extension of EXE

You will get the error message `I/O ERROR` if you try to access a drive that is not currently available.

Sign-Off

```
)OFF
```

Gets out of APL and gives control back to the Disk Operating System. The active workspace is lost.

Notes:

Appendix A. Alt Codes and Associated Characters

The following table lists all the Alt codes (in decimal) and the characters that they produce, under the APL mapping of the keyboard. (Alt codes and characters produced for the National keyboard mapping are given in Appendix C.)

000	001	002	003	004
NULL	☺	☻	♥	♦
005	006	007	008	009
♣	♠	BEEP	BACK SPACE	TAB
010	011	012	013	014
LINE FEED	♂	♀	CARR. RETURN	♪
015	016	017	018	019
ʿ	►	◄	↕	‼
020	021	022	023	024
¶	§	▬	↨	″
025	026	027	028	029
#	$	ESCAPE	%	↔

Figure 20 (Part 1 of 6). Alt Codes and Associated Characters

030	031	032	033	034
&	@	SPACE	¨	)
035	036	037	038	039
<	≤	=	>	]
040	041	042	043	044
∨	∧	≠	÷	,
045	046	047	048	049
+	.	/	0	1
050	051	052	053	054
2	3	4	5	6
055	056	057	058	059
7	8	9	(	[
060	061	062	063	064
;	×	:	\	¯
065	066	067	068	069
∝	⊥	∩	⌊	∈
070	071	072	073	074
_	▼	▲	⍳	∘
075	076	077	078	079
'	⎕	¦	⊤	○

Figure 20 (Part 2 of 6). Alt Codes and Associated Characters

080	081	082	083	084
*	?	ρ	⌈	∼
085	086	087	088	089
↓	∪	ω	⊃	↑
090	091	092	093	094
⊂	←	⌿	⍞	≥
095	096	097	098	099
—	⍉̲	A	B	C
100	101	102	103	104
D	E	F	G	H
105	106	107	108	109
I	J	K	L	M
110	111	112	113	114
N	O	P	Q	R
115	116	117	118	119
S	T	U	V	W
120	121	122	123	124
X	Y	Z	→	⍀
125	126	127	128	129
⍝	⍕	APL/ NAT.	Ç	ü

Figure 20 (Part 3 of 6). Alt Codes and Associated Characters

130	131	132	133	134
é	â	ä	à	å
135	136	137	138	139
$\underline{\Delta}$	q	w	e	r
140	141	142	143	144
t	y	u	i	o
145	146	147	148	149
p	¿	ô	ö	ò
150	151	152	153	154
a	s	d	f	g
155	156	157	158	159
h	j	k	l	¬
160	161	162	163	164
á	í	ó	ú	z
165	166	167	168	169
x	c	v	b	n
170	171	172	173	174
m	½	β	¡	{
175	176	177	178	179
}	DOTS ON 1/4	DOTS ON 1/2	DOTS ON 3/4	│

Figure 20 (Part 4 of 6). Alt Codes and Associated Characters

180	181	182	183	184
┤	╡	╢	╖	╕
185	186	187	188	189
╣	║	╗	╝	╜
190	191	192	193	194
╛	┐	└	┴	┬
195	196	197	198	199
├	─	┼	╞	╟
200	201	202	203	204
╚	╔	╩	╦	╠
205	206	207	208	209
═	╬	╧	╨	╤
210	211	212	213	214
╥	╙	╘	╒	╓
215	216	217	218	219
╫	╪	┘	┌	█
220	221	222	223	224
▄	▌	▐	▀	Pts
225	226	227	228	229
£	¢	Ü	Ö	ù

Figure 20 (Part 5 of 6). Alt Codes and Associated Characters

230	231	232	233	234
û	ç	ê	ë	è
235	236	237	238	239
ï	î	ì	Ä	Å
240	241	242	243	244
⌶	⍫	⍒	⍋	⌽
245	246	247	248	249
⍉	⊖	☼	⍱	⍲
250	251	252	253	254
!	⍠	ñ	Ñ	ª
255				
º				

Figure 20 (Part 6 of 6). Alt Codes and Associated Characters

Appendix B. Printer Control Codes

The following is a subset of the printer control codes.

Code		Printer Function
Bell	(⎕AV[232])	Sounds the printer's buzzer
♪	(ALT 014)	Sets double width on
☼	(ALT 247)	Sets compressed mode on
↕	(ALT 018)	Turns off compressed mode
¶	(ALT 020)	Turns off double width
← – 1		Sets underline on
← – 0		Turns off underline
← 0		Sets paper feeding to 1/8 inch
← 1		Sets paper feeding to 7/72 inch
← 2		Sets paper feeding to 1/6 inch
← 3n		Sets paper feeding to m/216 inch, where n is any ASCII character and m is its equivalent decimal ALT code
← E		Sets emphasized on
← F		Turns off emphasized
← G		Sets double strike on
← H		Turns off double strike
← S1		Sets subscript mode on
← S0		Sets superscript mode on
← T		Turns off subscript/superscript mode

Notes:

Appendix C. 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 hexa-decimal) and the characters that they produce, under the National mapping of the keyboard. (Alt codes and characters produced by the APL keyboard mapping are given in Appendix A.)

Note: Some alternate codes are reserved for system control functions, and will not generate a displayable character. The reserved codes are:

Alt Code	Control Function
007	Beep
008	Backspace
009	Tab
010	Line Feed
013	Carriage Return
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	►	SPACE	0	@	P	‘	p
1	1	☺	◄	!	1	A	Q	a	q
2	2	☻	↕	"	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	•	↨	'	7	G	W	g	w
8	8	◘	↑	(	8	H	X	h	x
9	9	○	↓	)	9	I	Y	i	y
10	A	◙	→	*	:	J	Z	j	z
11	B	♂	←	+	;	K	[	k	{
12	C	♀	∟	,	<	L	\	l	¦
13	D	♪	↔	—	=	M	]	m	}
14	E	♫	▲	.	>	N	^	n	~
15	F	☼	▼	/	?	O	_	o	△

Internal Representation of Displayed Characters

DECIMAL VALUE	➡	128	144	160	176	192	208	224	240
⬇	HEXA-DECIMAL VALUE	8	9	A	B	D	C	E	F
0	0	Ç	⎕	á	DOTS ON 1/4	└	╨	∝	⌿
1	1	ü	⍞	í	DOTS ON 1/2	┴	╤	β	⍀
2	2	é	⍠	ó	DOTS ON 3/4	┬	╥	⊂	≥
3	3	â	ô	ú	│	├	╙	⊃	≤
4	4	ä	ö	ñ	┤	─	╘	ɑ	≠
5	5	à	ò	Ñ	╡	┼	╒	⍲	×
6	6	å	û	a̲	╢	╞	╓	ρ	÷
7	7	ç	ù	o̲	╖	╟	╫	⍱	⍙
8	8	ê	⊤	¿	╕	╚	╪	Φ	°
9	9	ë	Ö	⌐	╣	╔	┘	⊖	⍵
10	A	è	Ü	¬	║	╩	┌	∘	⍫
11	B	ï	¢	½	╗	╦	█	∨	⍋
12	C	î	£	∪	╝	╠	▄	ɩ	⍒
13	D	ì	⊥	¡	╜	═	▌	⍉	¯
14	E	Ä	Pts	⍕	╛	╬	▐	∈	¨
15	F	Å	⌶	⍎	┐	╧	▀	∩	FORM FEED

Internal Representation of Displayed Characters

Notes:

INDEX

Reader's Comment Form

APL **1502219**

Your comments assist us in improving the usefulness of our publication; they are an important part of the input used for revisions.

IBM may use and distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Please do not use this form for technical questions regarding the IBM Personal Computer or programs for the IBM Personal Computer, or for requests for additional publications; this only delays the response. Instead, direct your inquiries or request to your Authorized IBM Personal Computer Dealer.

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 321 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Tape Please do not staple

Reader's Comment Form

APL **1502219**

Comments:

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 321 BOCA RATON, FLORIDA 33432

POSTAGE WILL BE PAID BY ADDRESSEE

IBM PERSONAL COMPUTER
SALES & SERVICE
P.O. BOX 1328-C
BOCA RATON, FLORIDA 33432

Fold here

Tape Please do not staple Tape

Continued from inside front cover

SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO YOU. THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

IBM does not warrant that the functions contained in the program will meet your requirements or that the operation of the program will be uninterrupted or error free.

However, IBM warrants the diskette(s) or cassette(s) on which the program is furnished, to be free from defects in materials and workmanship under normal use for a period of ninety (90) days from the date of delivery to you as evidenced by a copy of your receipt.

LIMITATIONS OF REMEDIES

IBM's entire liability and your exclusive remedy shall be:

1. the replacement of any diskette(s) or cassette(s) not meeting IBM's "Limited Warranty" and which is returned to IBM or an authorized IBM PERSONAL COMPUTER dealer with a copy of your receipt, or
2. if IBM or the dealer is unable to deliver a replacement diskette(s) or cassette(s) which is free of defects in materials or workmanship, you may terminate this Agreement by returning the program and your money will be refunded.

IN NO EVENT WILL IBM BE LIABLE TO YOU FOR ANY DAMAGES, INCLUDING ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE SUCH PROGRAM EVEN IF IBM OR AN AUTHORIZED IBM PERSONAL COMPUTER DEALER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO YOU.

GENERAL

You may not sublicense, assign or transfer the license or the program except as expressly provided in this Agreement. Any attempt otherwise to sublicense, assign or transfer any of the rights, duties or obligations hereunder is void.

This Agreement will be governed by the laws of the State of Florida.

Should you have any questions concerning this Agreement, you may contact IBM by writing to IBM Personal Computer, Sales and Service, P.O. Box 1328–W, Boca Raton, Florida 33432.

YOU ACKNOWLEDGE THAT YOU HAVE READ THIS AGREEMENT, UNDERSTAND IT AND AGREE TO BE BOUND BY ITS TERMS AND CONDITIONS. YOU FURTHER AGREE THAT IT IS THE COMPLETE AND EXCLUSIVE STATEMENT OF THE AGREEMENT BETWEEN US WHICH SUPERSEDES ANY PROPOSAL OR PRIOR AGREEMENT, ORAL OR WRITTEN, AND ANY OTHER COMMUNICATIONS BETWEEN US RELATING TO THE SUBJECT MATTER OF THIS AGREEMENT.

International Business Machines Corporation

P.O. Box 1328-W
Boca Raton, Florida 33432

1502219

Printed in United States of America