APL★PLUS System

FOR THE VAX VMS ENVIRONMENT

User's Manual

Release 1
August 1987

A PLUS★WARE™ PRODUCT

STSC

APL★PLUS System

FOR THE VAX VMS ENVIRONMENT

User's Manual

Release 1
August 1987

A PLUS★WARE™ PRODUCT

STSC

10 9 8 7 6 5 4 3 2 1

ISBN 0-926683-32-2

Printed in the United States of America.

Contents

Introduction

Welcome to the APL ⋆PLUS System!

The APL⋆PLUS System for VAX/VMS is a version of the APL programming language developed especially for the VMS operating system.

To use the system and this manual most effectively, you should be familiar with the APL language. If you are not, read *APL Is Easy!* (STSC, 1987) when you get your APL⋆PLUS System running. *APL Is Easy!* is a tutorial that will teach you the basics of programming in APL. For "hands-on" practice, use the demonstration APL programs on the disks included with your package. For details on the many features available in the system, refer to the *APL ⋆PLUS System Reference Manual.*

Organization of the Manuals

This manual is tutorial in nature; it contains information on the capabilities and features of your APL⋆PLUS System. The information in it may be duplicated or reinforced in the *APL ⋆PLUS System Reference Manual*. The following paragraphs outline the contents of this manual.

Chapter 1 shows you how to set up and access your APL⋆PLUS System. It will make you familiar with the keyboard and show you how to move about in the APL environment.

Chapter 2 explains how to edit data and functions using the Session Manager supplied with the system.

Chapter 3 describes the file system. It explains the concepts behind APL component files and VMS native files and describes how to create and manipulate them.

Chapter 4 explains data formatting in the APL⋆PLUS System and how to use the system function ⎕*FMT* to format your data.

Chapter 5 describes the full-screen facilities of the system, how to handle the screen under program control , and how to use the programmable function keys.

Chapter 6 describes the communications capabilities of the system, how to communicate with personal computers, how to access the communications port, and how to transfer files and workspaces.

Chapter 7 shows how to interact with non-APL programs and VMS native files, and how to issue DCL commands from APL.

Chapter 8 describes the printing facilities of the system and explains how to use them.

Chapter 9 describes the workspaces supplied with your system and how to use them.

Chapter 10 contains tips on how to use your system more efficiently and how to avoid the more common errors experienced in APL.

This manual also contains several appendixes that describe system characteristics and limits, the system character set, error messages, the use of the termcap database to support other terminals, and the policy on use and distribution of the Kermit transfer program. A glossary and index complete the manual.

How to Read the Examples in This Manual

All items in `APL FONT` represent actual system output or information to be entered exactly as shown. Items in *lowercase italic* font are mnemonic representations for information that you supply or for system output that varies. For example, in the following expression:

'*filename*' ⎕FTIE *tieno*

`⎕FTIE` is the name of an APL system function. You would enter it exactly as written. The words *filename* and *tieno* are mnemonic representations for the file name and tie number. You supply this information. Single quotes are used around character vector arguments; they are not necessary if you use an APL variable as the argument.

All items in `Courier font` represent either output displayed by the operating system (VMS) or input entered into it. For example:

```
$apl

      APL*PLUS SERVICE

CLEAR WS
      )CMD
$dir

Directory $DISK1:[APL.RELB]

APL.COM;4    APLOTAB.;2   APLX.CMD;3  APLX.EXE;1
ASCIITAB.;5  ATERMCAP.;1  AVT.HLP;1   AVT.INIT;4
```

Whenever possible, examples of system behavior are formatted as the system would format them. User entries are indented six spaces to match the system prompt for user input. Where user entries or system responses are too wide or will not fit on the page, they are wrapped to the next line with no indentation. Unless otherwise stated, all examples assume index origin 1. The exceptions are generally ⎕*AV* indices, conventionally shown in origin 0.

What is APL?

APL (A Programming Language) was originally conceived in the late 1950s by Kenneth E. Iverson, then a professor of mathematics at Harvard University. It was initially used as a mathematical notation and not as a computer programming language. In the mid-1960s, the notation was implemented as a programming language for use by IBM's central research staff at the company's T.J. Watson Research Laboratory.

STSC, Inc., was organized in 1969 for the purpose of providing an improved interactive time sharing service based on the use of APL as the programming language. STSC's founders, active in the computer industry for many years, were convinced that APL offered significant productivity advantages when compared with more traditional languages such as COBOL, FORTRAN, BASIC, and PL/I.

Since 1969, STSC has been committed to an intensive research and development effort to enhance and extend the usefulness of APL. The result is a generalized application development system we call our APL⋆PLUS System. Until 1981, the APL⋆PLUS System was available only as a commercial time-sharing service. Now, however, the system is available to run on a variety of computers from large mainframes to personal computers.

Where to Start

Start by reading Chapter 1 of this manual. It contains the essential information you need to begin using the APL⋆PLUS System.

If you have never programmed in APL, you should read *APL Is Easy!*. *APL Is Easy!* is an introduction to APL that was developed specifically for people beginning to use APL.

Almost all APL language features of the APL⋆PLUS System for VAX/VMS are identical to those on other APL⋆PLUS Systems; however, some support features vary slightly because of the constraints imposed by different computers. Also, because of the unique environment of your computer, some useful features have been added to the APL⋆PLUS System for VAX/VMS that do not appear in other APL⋆PLUS Systems. These unique features are noted as system dependent or experimental in the *APL⋆PLUS System Reference Manual*.

How to Get Help with Your System

STSC wants you to get the most benefit from the APL⋆PLUS System. These guidelines will help you to obtain the best support from STSC.

Register Your System

STSC provides support and assistance only to registered software licensees. Use the Registration card to register your system. Registered system owners will receive information about updates to the system including news of new versions, bug fixes, new features, and other information of interest to APL⋆PLUS System users.

The Help Line

STSC's Help Line has been established to provide timely assistance to registered users of the APL⋆PLUS System. If you need assistance with the product or want to report a problem, write to the following address:

APL⋆PLUS VAX/VMS Help Line
STSC, Inc.
2115 East Jefferson Street
Rockville, Maryland 20852

You also can call the Help Line number below. Since this phone is answered for a limited numbers of hours, you may hear a recording that tells you when you can reach us. Call:

(800) 638-6660

or in Maryland or outside the U.S. call:

(301) 984-5140

Please be ready to provide the following information:

- your name
- your organization
- the name of the registered owner of the system
- the serial number of your system (on the diskette)
- the version of the software you are using (enter *⎕SYSVER*).

Report Apparent Errors in the Software to STSC

STSC is eager to learn of suspected problems with the system and you are invited to report apparent bugs to STSC. We want to resolve problems in future versions of the system so that all users benefit. We may receive multiple reports of a single problem, so if the difficulty is not interfering with your use of the system to an important degree, please write. If the problem is causing you considerable difficulty, you should call the Help Line.

You can help us find and fix a bug by providing specific information about the cause of the problem and the result. If you can narrow down the problem to a few lines of input in a clear workspace, it will make it much easier for us to find and fix the problem quickly.

STSC Offers APL Instruction and Consulting

STSC offers all levels of APL instruction, application design consulting, and full application implementation at your facility or at STSC headquarters. Contact STSC for more information about class schedules and tuition fees, and consulting rates.

Can I use the APL⋆PLUS System on Other Computers or Other Operating Systems Similar to the VMS Operating System?

The APL⋆PLUS System is designed for the VMS operating systems or close derivatives. A different STSC product is required to run the APL⋆PLUS System on a non-VAX computer or to run on a VAX under UNIX or ULTRIX.

Acknowledgments

Many people contributed to the design and production of the APL⋆PLUS System for VAX/VMS and the documentation. The editors are grateful to them all, especially:

Larry Goodwin	Richard Renich
William Lewis	Stuart Ritter
Edward Myers	Laurie Russell
Mark Osborne	William Rutiser
Marvin Renich	Mary Wise

Your Comments Are Welcome

We appreciate suggestions of new features that might increase your productivity. We particularly like to know what features you would like to see enhanced and what new features you would find useful. We design future releases based on suggestions we receive. Send your suggestions to:

APL⋆PLUS VAX/VMS Product Manager
STSC, Inc.
2115 East Jefferson Street
Rockville, Maryland 20852

Comments about this manual are also welcome. Please send your comments to:

Technical Documentation Manager
STSC, Inc.
2115 East Jefferson Street
Rockville, Maryland 20852

GETTING STARTED

Chapter One *Getting Started*

The Installation Guide describes the procedure for installing the APL⋆PLUS System on your computer. The material in this chapter assumes that the instructions have been followed and that APL has been installed in the directory `[APL.RELn]`, where *n* is the release number of your APL⋆PLUS System.

1-1 Hardware and Software Requirements

Required Hardware Configuration:

- 2 megabytes main memory
- 2000 blocks free on the installation disk

Operating System Requirements:

- VMS or MicroVMS Version 4.4 or later

Recommended Hardware Configuration:

- Additional memory is required for efficient use of large workspaces or for multiple users.

Terminals Supported:

- Digital VT100
- Digital VT200 series
- HDS AVT
- HDS Concept 108
- HDS 200 series
- HP 2641
- IBM PC running APL⋆PLUS PC System
- Macintosh running APL⋆PLUS Mac System
- Generic APL video or hardcopy terminals

Note: Terminals are supported through the use of the `termcap` database (see Appendix D). You can add new terminals to the database.

1-2 *Beginning and Ending an APL Session*

Once installed, the APL⋆PLUS System can be invoked by the following DCL command (the `$` used here is the prompt displayed by VMS):

```
$ apl
```

A list of supported terminals should be displayed:

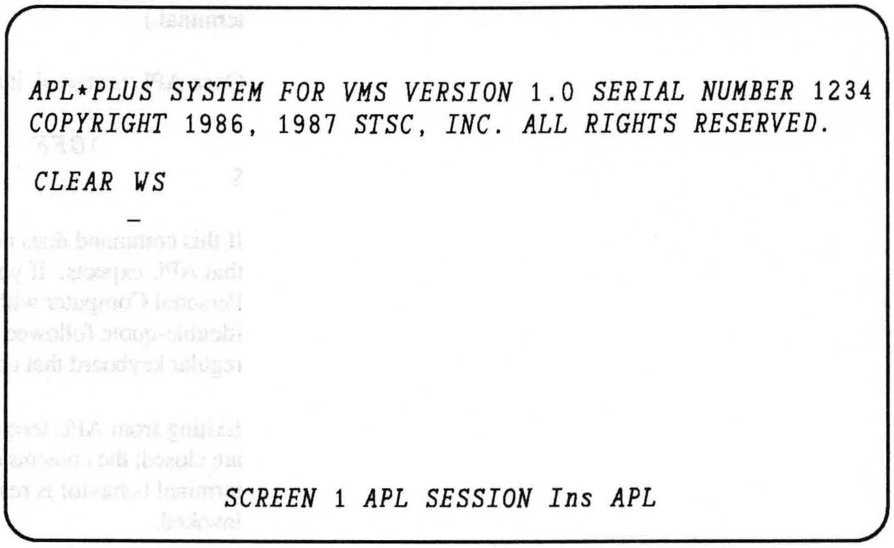
```
pc           IBM PC running APL*PLUS PC
c108         HDS Concept 108
avt          HDS AVT
c200         HDS 200
hp           HP 2641
vt100        DEC VT100
vt200        DEC VT220 and VT240
g            generic APL video terminal
h            generic APL hardcopy terminal

Terminal name:  _
```

Enter the abbreviation that identifies your terminal. If your terminal does not appear on this list, select `g` or `h`. APL will use a generic terminal definition which assumes a line-oriented terminal that behaves like a hardcopy terminal. Full-screen editing is not available for the generic terminal.

APL is a large program and usually takes several seconds to load from disk. When the system is loaded, a welcoming banner displays, and the interactive session begins. The message *CLEAR WS* appears, and the cursor indents six spaces on the next line. Depending upon the kind of terminal you are using, the screen may be cleared and the terminal switched into the APL character set.

APL should then display a screen similar to this:



```
APL*PLUS SYSTEM FOR VMS VERSION 1.0 SERIAL NUMBER 1234
COPYRIGHT 1986, 1987 STSC, INC. ALL RIGHTS RESERVED.

CLEAR WS

      _







                SCREEN 1 APL SESSION Ins APL
```

The file named APL is a DCL command procedure that performs several actions to set up the APL session. It will be installed as [APL.REL*n*]APL (where *n* is the release number) on one of the system disks. If the System Administrator at your site has followed the installation procedure recommended by STSC, the name APL will be defined as a synonym for this command in the file SYSLOGIN.COM. The installation procedure also recommends that the synonym APL_DISK be defined for the disk on which APL is installed.

Once you become more familiar with the APL*PLUS System, you will probably want to customize [APL.REL*n*]APL to your preferences (see Section 1-3).

Now test that APL is configured correctly for your terminal by typing in a simple expression such as:

```
      2+3
5
```

If APL prints the answer properly, the configuration is correct. (If the result displayed is 0.666667 instead of 5, you typed in the symbol for APL division ÷, which is on the key where + appears on a non-APL terminal.)

Once APL is started, it remains active until you type the command:

```
        )OFF
$
```

If this command does not work, you probably are not using the keys that APL expects. If your terminal is not an APL terminal or a Personal Computer with STSC's APL characters installed, try `"off` (double-quote followed by unshifted off). These are the keys on a regular keyboard that correspond to *)OFF* on an APL keyboard.

Exiting from APL terminates the APL process for VMS. Open files are closed; the contents of the active workspace are discarded, and the terminal behavior is restored to what it was when APL was first invoked.

1-3 The APL Command Procedure

The command procedures provided with the APL⋆PLUS System for VAX/VMS in directory `[APL.RELn]` are for configuring the system to specific APL terminals. These can be used instead of the general `apl` procedure, which prompts for the kind of terminal you are using.

Files of the form `nnnAPL.COM`, where `nnn` is a terminal type, are command procedures for running the APL⋆PLUS System. Each command file turns off the DCL Ctrl-y and Ctrl-t functions and starts APL with a specific initialization file.

The distributed command files and their associated initialization files are:

Terminal Type:	**Command File**	**Init File**
general or unknown:	`APL.COM`	`GENERIC.INIT`
HDS AVT terminal:	`AVTAPL.COM`	`AVT.INIT`
HDS Concept 108:	`C108APL.COM`	`C108.INIT`
HDS 200 series:	`HDSAPL.COM`	`HDS200.INIT`
HP 2641 terminal:	`HPAPL.COM`	`HP2641.INIT`
IBM PC running the APL⋆PLUS PC System in terminal mode:	`PCAPL.COM`	`PC.INIT`
DEC VT100 terminal:	`VT100APL.COM`	`VT100.INIT`
DEC VT200 series:	`VT200APL.COM`	`VT200.INIT`
generic APL video:	`GENERICAPL.COM`	`GENERIC.INIT`
generic APL hardcopy:	`HARDCOPYAPL.COM`	`HARDCOPY.INIT`

If the System Administrator at your site has followed the installation procedure recommended by STSC, the following synonyms will be defined and may be used to execute the corresponding `.COM` command files:

```
APL         PCAPL
AVTAPL      HPAPL
C108APL     VT100APL
HDSAPL      VT200APL
```

If the System Administrator has not already established global synonyms for these commands for all users, users can edit their own `LOGIN.COM` file to add a line such as:

```
$ hdsapl   :== "@apl_disk:[apl.reln]hdsapl"
```

This will allow the user to invoke APL on an HDS 200 by typing:

```
$ hdsapl
```

1-4 Terminals for APL

The APL⋆PLUS System for VAX/VMS uses a file of the same structure as the UNIX termcap database to describe the full-screen behavior of various terminals. A file named `atermcap` is provided on the distribution tape.

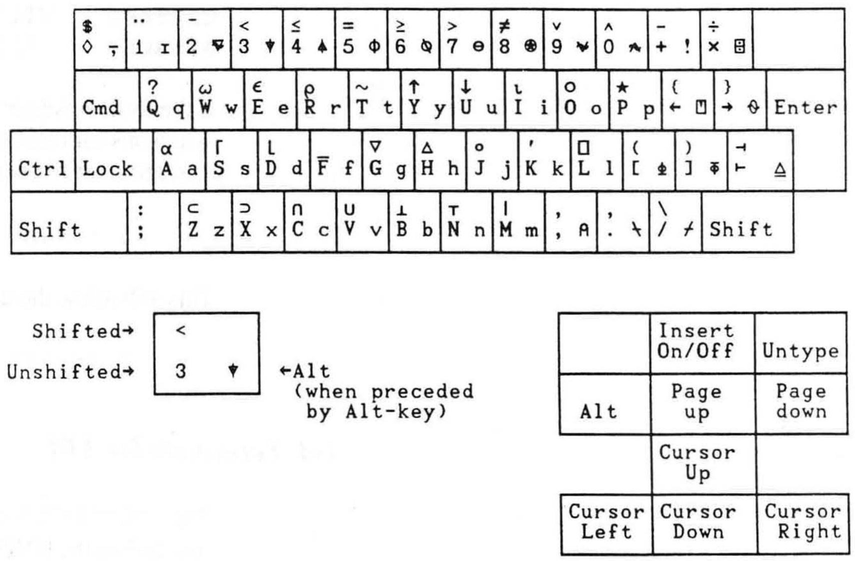
The `atermcap` database allows effective use of a wide variety of terminals with the system. Appendix D explains how to support other terminals not included in `atermcap`.

This section will also discuss the special support provided for the VT220 and VT240 terminals and the IBM PC when using the APL⋆PLUS PC System in terminal mode.

Using the VT220 and VT240 Terminals

The DCL procedure `VT200APL.COM` on the distribution tape takes advantage of the capability of fonts to be downloaded on the VT220 and VT240 terminals and makes them effective as APL terminals. The configuration file `VT200.INIT` loads useful values into the terminal's function keys and configures APL to recognize its editing keys. The APL keyboard for the VT220 is as follows:

VT 220 Keyboard



The function keys on a VT 220 are defined in `VT200.INIT` as follows:

Key	Unshifted	Shifted
F6	---	Insert blank line below (Cmd I)
F7	Overstrike	Delete line (Cmd D D)
F8	Delete char	Split line (Cmd .)
F9	Clear EOL	Join lines (Cmd ,)
F10	O-U-T	Insert saved lines below (Cmd P)
F11	Undo	Insert saved lines above (Cmd *)
F12	Refresh	Save line (Cmd Y Y)
F13	Scroll down	String search (Cmd /)
F14	Scroll up	Repeat search (Cmd / RETURN)
F17	Cmd Z (editor end)	
F18	Cmd S (to session)	
F19	Cmd E (editor begin)	
F20	Cmd Q (editor quit)	

See Section 5-1 for more information on function keys.

Using the APL⋆PLUS PC System as a Terminal

A PC running the APL⋆PLUS PC System in terminal mode can be a very effective terminal for the APL⋆PLUS System for VAX/VMS. The following steps are necessary in order to make the best use of the PC as a full-screen terminal.

On the PC (version 5.0 or later required):

- Execute the function *TERMINIT*, listed below.
- Turn off the status line, using Scroll Lock, to prevent conflict with the status line produced by APL⋆PLUS for VAX/VMS.
- Switch into terminal mode (Alt-F8).

On the VAX, run the DCL program `PCAPL.COM`, which runs APL with the terminal type of `pc`. Alternatively, run `APL.COM` and specify `pc` in response to the `terminal:` prompt.

After you complete the previous steps, the most common keystrokes will have the same effect on the APL⋆PLUS System for VAX/VMS as they do on the PC. The keystrokes affected are:

Backspace	Acts as the Untype Key
Ctrl-C	Signals an interrupt to halt APL execution
Home	Moves cursor to left end of the line
Ctrl-Home	Moves cursor to the top left of the screen
End	Moves cursor to the right end of the line
Ctrl-End	Moves cursor to the bottom right of screen
↑	Moves cursor up one line
Ctrl-↑	Moves cursor up four lines
↓	Moves cursor down one line
Ctrl ↓	Moves cursor down four lines
←	Moves cursor left
Ctrl-←	Moves cursor left eight spaces
→	Moves cursor right
Ctrl-→	Moves cursor right eight spaces
PgUp	Scrolls up one line
Ctrl-PgUp	Scrolls up one page
PgDn	Scrolls down
Ctrl-PgDn	Scrolls down one page
Alt →	Clears to end of line
Del	Deletes character and close up
Ins	Forms overstrike with next keystroke

The PC's Alt keys (such as Alt-4 for ⍋), will produce the expected composite character only if the APL★PLUS System for VAX/VMS is in overstrike mode. The PC's terminal mode transmits '∆',⎕TCBS,'|' when Alt-4 is pressed, and the overstrike will not be formed if the VAX is in insert or replace mode. You can achieve the effect of the Alt-keys even if the system is not in overstrike mode, by first pressing ESC and then the key that you would ordinarily use with Alt on the PC.

Execute the following APL function on the PC to customize terminal mode appropriately:

```
      ∇ TERMINIT;S;⎕SEG;TAB;⎕IO
[1]  ⍝ Initializes APL★PLUS PC for use as a VMS terminal
[2]    ⎕SEG←''
[3]    S←1 ⎕POKE 115 ⍝                 Full duplex
[4]    S←127 0 ⎕POKE 171 172 ⍝         Suitable untype sequence
[5]
[6]  ⍝ Other pokes for special editing keys
[7]    ⎕IO←0 ◊ TAB←⎕AV[9]
```

```
[8]    S←6↑138,⎕AV⍳TAB,'h'   ⍝      Home ←→ TAB H
[9]    S←S,6↑111,⎕AV⍳TAB,'k',TAB,'h' ⍝Ctrl-Home←→TAB K TAB H
[10]   S←S,6↑136,⎕AV⍳TAB,'1y'   ⍝ PgUp ←→ TAB 1 Y
[11]   S←S,6↑112,⎕AV⍳TAB,'1c' ⍝  Ctrl-PgUp ←→ TAB 1 C
[12]   S←S,6↑132,⎕AV⍳TAB,'l'   ⍝  END ←→ TAB L
[13]   S←S,6↑114,⎕AV⍳TAB,'j',TAB,'l' ⍝Ctrl-END←→TAB J TAB L
[14]   S←S,6↑165,⎕AV⍳TAB,'1u'   ⍝ PgDn ←→ TAB 1 U
[15]   S←S,6↑115,⎕AV⍳TAB,'1v' ⍝  Ctrl-PgDn ←→ TAB 1 V
[16]   S←S,6↑177 16  ⍝              INS ←→ Ctrl-P (overstrike)
[17]   S←S,6↑176 4  ⍝DEL ←→ Ctrl-D (delete char)
[18]   ⎕SEG←'' ⎕SEG←256⊥⎕PEEK 130 129
[19]   0 0 ρS ⎕POKE 1102+⍳ρS
     ∇
```

See "Using Terminal Mode" in the APL⋆PLUS PC User's Guide (Release 5.0 or later) for details on how the terminal mode behavior can be customized.

1-5 Using the Keyboard

The APL⋆PLUS System for VAX/VMS has been designed to provide a user-friendly, full-screen interface, no matter which of dozens of different terminals you may be using. Getting the most out of so many different terminals means that APL must be very flexible in how it treats the terminal. This means, however, that you have to ensure that APL is set up properly to work with your terminal.

Terminal features and behavior can vary considerably. For example, if your terminal has a cursor-up key labeled with an upward-pointing arrow, you would probably like to be able to press this key in APL to move the cursor up one line. If your terminal is an IBM PC running the APL⋆PLUS PC System, pressing the cursor-up key transmits a Ctrl-K character to VMS. On the other hand, if your terminal is a DEC VT100 , the cursor-up key transmits three characters: Escape, [, and A. Some terminals have no cursor key at all.

In order to deal with these differences, the APL⋆PLUS System uses the concept of logical keystrokes. A logical keystroke is defined in terms of the effect that a key on the terminal is expected to produce. For example, "cursor-up" is a logical keystroke — it is a command that you type at the terminal that instructs APL to move the cursor up one line. The exact key that you press to achieve this command will vary from one terminal to another, but the effect is the same.

Of course, this can only work if APL knows what sequence of characters is intended to represent cursor-up, which is why it is so important that the terminal interface be set up properly. The following tables show APL logical keystrokes and their effects on some popular terminals.

Logical Keystrokes

Input and Editing Keys

Keystroke	Effect
Untype	Delete the most recently typed character (the character to the left of the cursor) and close up. Also called destructive backspace.
Delete	Delete the character at the cursor and close up the space.
Clear-EOL	Erase all characters from the cursor to the end of the line.
Undo	Undo all changes that have been made to the line and reprompt.
Alt-key	Interpret the subsequent keystroke as an "Alt" key, or a logical Program Function key (see Section 5-1).
Overstrike	Overstrike the character to the left of the cursor with the next character typed.
Enter	Enter the current line as input to APL.

Cursor Keys

Keystroke	Effect
Cursor-up	Move the cursor up one line.
Cursor-down	Move the cursor down one line.

Cursor-left	Move the cursor left one space without erasing what is there (non-destructive backspace).
Cursor-right	Move the cursor right one space without erasing what is there (non-destructive space).

Window Control Keys

Scroll-up	Scroll the active window up one line to reveal a line saved in off-screen memory.
Scroll-down	Scroll the active window down one line.
Page-up	Move the active window up one page (one screen-full of text).
Page-down	Move the active window down one page.

Input Mode Control:

APL-keyboard	Switch APL to interpret keystrokes as coming from a terminal with an APL keyboard.
Text-keyboard	Switch APL to treat the keyboard as a standard non-APL keyboard.
Insert-mode	Switch between insert mode, replace mode, and overstrike mode. The current mode is shown on the status line.

Other Commands:

Command	Interpret the next keystrokes as session manager/full-screen editor command. See Chapter 2 for details on editor commands.
Interrupt	Interrupt an executing APL program or the display of output (functions like the BREAK key in some other APL systems). Always a ^C under VMS.

O-U-T	Interrupt ⎕ input (like the O-U-T escape sequence used on older APL systems).
Refresh	Clear the screen and re-display its contents.

The following table shows the physical keys used to produce logical keystrokes when the proper configuration file has been used on these terminals:

- VT220: `VT200INIT`
- HDS Concept 108: `C108.INIT`
- Standard ASCII terminal with no `.INIT` file used
- APL*PLUS PC terminals: `PC.INIT`

Logical Keystrokes on Selected Terminals

Keystroke	**VT200-type Terminals**	**HDS C108**	**ASCII terminal with no config file**	**APL*PLUS PC Terminals**
Untype	Remove	Backspace	Ctrl-R	Untype
Delete	F8	Ctrl-D	Ctrl-D	Del
Clear-EOL	F9	Ctrl-E	Ctrl-E	Alt-right-cursor
Undo	F11	Ctrl-B	Ctrl-B	Ctrl-B
Alt-key	Select	Ctrl-A	Escape	Escape
Overstrike	F5	Ctrl-P	Ctrl-P	Ins
Enter	Enter	Return	Return	Return
Cursor-up	Cursor-up	Cursor-up	Ctrl-K	Cursor-up
Cursor-down	Cursor-down	Cursor-down	Ctrl-J	Cursor-down
Cursor-left	Cursor-left	Cursor-left	Ctrl-H	Cursor-left
Cursor-right	Cursor-right	Cursor-right	Ctrl-L	Cursor-right
Scroll-up	F14	Scroll-up	Ctrl-Y	Ctrl-Y or Page-up
Scroll-down	F13	Scroll-down	Ctrl-U	Ctrl-U or Page-down
Page-up	Previous-Screen	Page-up	Ctrl-C	Ctrl-C or Ctrl-Page-up
Page-down	Next-Screen	Page-down	Ctrl-V	Ctrl-V or Ctrl-Page-down
APL-keyboard	Ctrl-Y	Ctrl-N	Ctrl-N	Ctrl-N
Text-keyboard	Ctrl-O	Ctrl-O	Ctrl-O	Ctrl-O
Insert-mode	Insert-Here	Ctrl-T	Ctrl-T	Ctrl-T
Command	TAB	TAB	TAB	TAB
O-U-T	F10	Ctrl-Z	Ctrl-Z	Ctrl-Z
Refresh	F12	Ctrl-F	Ctrl-F	Ctrl-F

Interrupting APL

The interrupt character for APL is Ctrl-C. You can use this character to:

- halt execution whenever APL is running
- interrupt the display of a large volume of output
- break out of the delay from ⎕DL and ⎕*FHOLD*.

During the time that APL is expecting input, Ctrl-C is the default page-up character, and it does not interrupt anything. On full-screen terminals, it is easy to tell whether APL can be interrupted:

- If the status line is hidden, then APL is running, and Ctrl-C acts an interrupt.

- If the status line is visible, then APL is awaiting input, and Ctrl-C does not act as an interrupt.

You can also use Ctrl-Y to end the APL process and return you immediately to VMS. This event cannot be trapped by any error handling routines, and your active workspace and all record of your APL session will be lost. You can suppress this effect of Ctrl-Y by using the DCL command:

```
set nocontrol=y
```

in the DCL procedure before invoking APL. This prevents Ctrl-Y from interrupting APL, although the message *INTERRUPT* is still displayed on the terminal. All of the APL command procedures provided by STSC already contain this setting.

Editing Input

APL input is displayed on the terminal as you type it, but no action is taken until you press Return. You can make any sort of correction before entering the line as input, and the terminal displays the corrections. The Untype key is particularly useful for immediately erasing a mistyped character and typing the correct one in its place.

If you want to discard the line completely and start fresh, the Undo key will erase everything you have typed on the line so you can begin again. Part of the line can be erased by backspacing and using the Clear-EOL key.

Forming Overstrikes

The APL composite characters, also called overstrikes, can be formed in three distinct ways:

- The overstrike key will combine the character to the left of the cursor (the one just typed) with the next keystroke typed. To produce ⍋, type the three keystrokes ∆, Overstrike, and |.

- The Alt-key mechanism is often handier for typing composite characters. For example, Alt-4 will produce ⍋ (assuming that you have not reprogrammed the Alt-key contents).

- In overstrike mode, a character already on the screen can be overstruck with another by placing the cursor on the character and typing the second character. This does not work in "insert" or "replace" mode.

Input Errors

If you type a keystroke that has no meaning to APL, or if you attempt to form an overstrike character from two keystrokes that do not combine to form a composite character, the terminal will indicate an error either by flashing the screen or sounding the bell, depending upon the kind of terminal you are using.

Insert, Overstrike, and Replace Mode

For CRT terminals, the system is in "insert" mode by default. Text typed in the middle of a line is inserted at the cursor, pushing any characters beyond the cursor to the right. The insert-mode key will switch the keyboard between insert, overstrike, and replace mode. CRT terminals may also have a status line that indicates whether the terminal is in insert, replace, or overstrike mode.

In "replace" mode, characters typed at the terminal replace the character already present. Replace mode is convenient for reading user input under application control. In replace mode, typing a space will substitute a blank space for the character already present on the screen. The cursor-right key spaces past a character.

In "overstrike" mode, an APL composite character can be formed by typing one symbol, backspacing, and typing the second symbol. Overstrike mode can be useful when you want to form multiple overstrikes at once; for example, you can enter ⍟ ⌽ ⊖ ⍉ by typing ⋆ | − \, followed by four backspaces and four ∘ characters.

APL and Text Keyboards

If you are using a terminal that can switch between displaying APL characters and regular ASCII characters, then you can use the APL-keyboard and text-keyboard keys to switch keyboards. These keys do not actually modify what the keys on the terminal transmit, but they do control how APL interprets characters received from the terminal.

In the APL keyboard state, typing the unshifted A key yields an upper case A, and typing it while holding down the shift key yields alpha (α). In the text keyboard state, the unshifted key produces lowercase "a" and the shifted key the upper case A.

Some characters can be produced from either keyboard. For example, the asterisk (*) is the same character whether produced with a shift-P in the APL keyboard or shift-8 on the text keyboard, and lowercase "a" on the text keyboard is exactly the same as A overstruck with an underscore (<u>A</u>).

APL Alphabets

The APL⋆PLUS System's character set contains the uppercase letters *ABCDE . . . XYZ* and the lowercase letters *a b c d e . . . x y z*. The lowercase letters correspond to the underscored alphabet used in traditional APL systems. On terminals that are incapable of displaying both APL and ASCII characters at the same time, including most hard-copy terminals, the lowercase alphabet is displayed in its traditional form of *<u>ABCDE</u> . . . <u>XYZ</u>*.

1-6 The Session Manager

The Session Manager manages the input and output between the user and the APL interpreter. The Session Manager and the full-screen editor are actually the same program, except that the full-screen editor is used to edit APL functions and variables. The same keystrokes and commands are used by both, providing a consistent environment for the APL user. More details on the full-screen editor can be found in Chapter 2.

All APL input you type and all output produced by APL is recorded in a logical screen image stored in the computer. The logical image typically contains more lines than can appear on the screen at once. One of the most important functions of the Session Manager is to keep the image that appears on the terminal consistent with the image stored in memory.

With the Session Manager, you can use the entire CRT screen for APL input. You can move the cursor to any position on the screen, insert or delete characters, or scroll up and down to reveal output that has moved out of the visible window.

Moving the Cursor

The cursor movement keystrokes (cursor-up, cursor-left, and so on) are used to move the cursor on the terminal. The Session Manager keeps track of the codes transmitted when you press the cursor keys. It also transmits control codes to the terminal to keep the visible cursor position consistent with the cursor position known to APL. Many full-screen editor commands can be used to move greater distances on the screen or even to locate a specific text string in the session image.

Re-Using Previous Lines

When the APL⋆PLUS System is expecting input and you press Return, the line on which the cursor rests is passed to the APL interpreter as input. Thus, you can re-enter any line of input or output that is visible on the screen by moving the cursor up to that line and pressing Return. You can modify the current form of a line using editing keystrokes and then enter the modified line.

If APL is in immediate execution input mode, and you move the cursor up to use an earlier line to satisfy the input, the line is copied to the bottom of the screen when you hit Return. If you make changes to the line before pressing Return, the new form is copied to the bottom of the screen and the old line is restored to its original form, keeping an accurate record of the APL session. When not in immediate execution mode (that is, in the del editor or ⎕ input), APL does not copy re-used lines to the bottom of the screen.

Scrolling

To understand the scrolling capabilities of the Session Manager, think of the terminal screen as a window that displays only part of the input and output that has been generated in the course of the APL session. The Session Manager maintains in memory a record of recent output, storing more lines than can appear on the screen at once. Scrolling is the process of moving the window up and down through the stored screen images.

Scrolling occurs automatically when all lines on the screen have been used and a new line of output is produced. All lines on the screen move up one position, and the topmost line disappears. If you move the cursor to the top line of the screen using cursor-up and press cursor-up again, you will "push" the window up one line to reveal the line that had disappeared. Similarly, pressing cursor-down when the cursor is at the bottom of the screen will push the window down one line.

The scroll-up and scroll-down keystrokes are used to scroll the window up and down a line at a time without changing the cursor position. Page-up and page-down move up and down a screen-full or "page" at a time. Other editor commands can be used to move the window in additional useful ways. Chapter 2 discusses the full-screen editor and its commands in detail.

If you are at the top of the scrolling memory and try to move up farther, the terminal will beep or flash the screen. The same effect is produced by attempting to move down past the last line. Only a limited number of screens of output are saved for scrolling; you can specify how many with the session parameter `screens=` (see Section 1-7). By default, the system saves four screens-full of output. When the scrolling memory is full, new output is inserted at the bottom of the scrolling memory, and the top lines are discarded.

On-Line Session Manager Documentation

The system command `)HELP` displays a summary of the keystrokes and commands used by the Session Manager and full-screen editor. The information displayed is extracted from a VMS file whose name is specified by the APL session parameter `help=` (see Section 1-7). If no help file is specified, the default file is used:

```
[APL.RELn]HELP.HLP.
```

1-7 APL Session Parameters

Session parameters control various options for APL sessions. They also convey information to APL about the terminal you are using. Specify session parameters by typing them on the same line as the APL invocation, or by including them in your configuration file. Following are some examples of session parameters:

```
$ apl 400000
```

Specifies the initial size of the active workspace to be 400,000 bytes.

```
$ apl editmem=40000 screens=10
```

Specifies 40,000 bytes for session manager and full-screen editor memory, with 10 screens of output saved in the scrolling memory.

```
$ apl initfile=[apl.rel1]pc.init
```

Specifies that the file `APL_DISK:[APL.REL1]PC.INIT` will be used as a configuration file; this particular file (`PC.INIT`) is used when the terminal is a PC running the APL⋆PLUS PC System in terminal mode.

Many other session parameters also exist. Since there are so many different session parameters, we recommend that you use a configuration file to contain them. Configuration files are files that contain APL session parameters; their structure is described in Section 1-8. The default APL command file, which is typically installed as the command `APL`, selects a configuration file for you based on the type of terminal you specify.

The parameters that you can supply when starting APL are listed in the following table:

Parameter	Description
nnn	Workspace size in bytes (default=16,384)
`editmem=`	Session/editor memory (default=65,536)
`help=`	Name of help file
`initfile=`	Name of configuration file
`initialws=`	Initial workspace name
`library=`	Define library/directory mapping
`outputtrt=`	Name of output translate table file
`prompt=`	String used as input prompt
`screens=`	Number of screens saved in session
`status=`	Initial terminal state (default=0)
`termcap=`	Name of termcap database
`termdinit=`	String sent to terminal on exit from APL
`terminal=`	Name of terminal you are using
`terminit=`	String sent to terminal at entry to APL
`termtype=b`	Bit-paired terminal

Each of the session parameters is described in more detail in this section.

Workspace Size
nnn

The initial size of the APL workspace is specified by a session parameter that consists of a number with no qualifier. For example:

```
$ apl 200000
```

specifies a 200,000-byte workspace. If the size parameter is missing or incorrectly formed, or if not enough memory is available for the requested size, the default workspace size is used. The default size is the smallest possible workspace, 16,384 bytes.

The maximum workspace size is determined by the process size limit and the amount of process memory available to VMS at the time APL is invoked. If your request is for a larger workspace than VMS can provide, APL displays the message *INSUFFICIENT SPACE FOR WS* and uses the default size.

Once APL is running, you can increase or decrease the workspace size using *)CLEAR*:

```
      )CLEAR 150000
CLEAR WS
      )CLEAR 900000
INSUFFICIENT SPACE FOR WS
CLEAR WS
```

See Section 10-3 for information on how to increase the workspace size limit.

Full-Screen Editor Memory
`editmem=`

The amount of memory allocated for the APL session manager and full-screen editor can be specified with the `editmem=` *bytes*. For example:

```
      $ apl editmem=50000
```

allocates 50,000 bytes of memory for the editor. This memory stores the logical image of the APL session as well as the images of objects being edited in the full-screen editor. The message *NOT ENOUGH MEMORY AVAILABLE* from the full-screen editor indicates that the current allocation has all been used.

The default allocation is 65,536 bytes, and the `editmem=` *bytes* can specify any value from 16,384 to as large as VMS will allow.

Help Facility
`help=`

The session parameter `help=` specifies the name of the file that is read and displayed by the APL system command *)HELP*. The help facility allows you to display the contents of the file, one screen at a time, or to review a screen that has already been displayed. If the `help=` option is not used, the default help file is used:

```
[APL.RELn]HELP.HLP
```

)HELP provides only a very simple help facility and is not recommended for incorporation into applications. The primary use of the facility is to help you remember the session manager and full-screen editor commands. For this reason, you may want to write a specific help file for individual terminals and to include the `help=` parameter in the configuration file for that terminal.

The distributed file `HELP.HLP` is a good example of the structure and content of a help file. Note that the VMS line-feed character (*⎕TCLF*) is interpreted as the line delimiter in `HELP.HLP`.

Configuration File
`initfile=`

The initial file is a configuration file that contains session parameters. See Section 1-8 for details on using `initfile=` and creating a configuration file.

Initial Workspace
`initialws=`

The initial workspace parameter `initialws=` specifies the name of a workspace that APL will load automatically at the start of the session. If the initial workspace contains a latent expression (*⎕LX*), an application can be started automatically. The workspace name is specified as the name of the VMS file that contains the saved workspace but without the `.WS` extension. For example:

```
$ apl initialws=autostart
```

If no explicit path is provided, the workspace is assumed to be in the current default directory.

Library Number
`library=`

The session parameter `library=` equates a library number with a VMS directory. This feature can be used to increase compatibility with other APL systems that use library numbers. For example:

```
library=11[apl.rel1]
```

defines library 11 as corresponding to the directory `[APL.REL1]`. More information on numbered libraries is in Chapter 3.

Output Translation
`outputtrt=`

The session parameter `outputtrt=` specifies an optional output translation table that is used instead of the translate table built-in to the system. This table controls the character sequence transmitted to the terminal for each of the 256 possible APL character values. All normal APL output to the terminal is affected, including input that is displayed while you type.

The `outputtrt=` parameter is primarily used to make APL work with specialized terminals that do not conform to the APL/ASCII typewriter-paired standard. Thus, the file `VT220TAB` contains translations specific to STSC's downloaded character set for the VT220 terminal. The following tables are available:

`aplotab`	a translate table for APL/ASCII typewriter-paired terminals
`bitotab`	a translate table to be used with bit-paired terminals, such as the HP 2641A.
`rawotab`	a translate table that causes each element of ⎕*AV* to be output with no translation at all
`vt220tab`	a translate table for VT220 terminal with downloaded APL characters.

An output translation file should be a regular VMS sequential Stream_LF file containing exactly 256 lines. The lines should be delimited by the linefeed character (⎕*TCLF* in APL, not ⎕*TCNL*), as would be produced by the VMS editor. For example, the 48th character of ⎕*AV* (origin 1) is slash (/), so line 48 of the file should contain the character /.

A new translate table can be used to change the display form of characters in APL's ⎕*AV* or to define display forms for characters that are not presently defined. It does not allow you to define new overstrikes that are accepted for input, although it can be used to control how valid overstrikes are displayed. For example, it can be used to control whether characters from ⎕*AV*[97] to ⎕*AV*[122] are displayed as *A̲B̲C̲* . . . *Z̲* or *abc*...*z*.

For convenience, the same notation as is used in termcap files can be used to denote characters that are difficult to produce explicitly using a text editor:

\E	means the escape character (decimal 27)
^G	means the BEL character (decimal 7)
\010	means octal 10 (decimal 8), also known as backspace or Ctrl-H
\\	means a single backslash.

Each sequence of characters should actually cause only a single character to be displayed on the terminal. If the sequence causes the cursor to move anywhere other than one position to the right, the terminal display will become unsynchronized with the session manager.

Input Prompt String
prompt=

The session parameter `prompt=` is used to specify a string that serves as the APL input prompt. Any string can be used, but the most useful value is the ASCII BEL character, which causes the terminal to beep when input is expected (much like an APL time sharing system). If a personal computer is being used as a terminal, the BEL prompt can provide a useful termination character for a ⎕*ARBIN*-based data transfer protocol on the PC; for example:

```
$ apl prompt=^G
```

where `^G` is a notation recognized by APL for Ctrl-G.

"Customizing Logical Keystrokes" later in this section contains a description of this notation.)

Scrolling Memory Pages
`screens=`

The session parameter `screens=` specifies the number of screens of scrolling memory reserved for the image of the APL session. For example:

```
$ apl screens=10
```

specifies ten screens. This memory is allocated from the block of storage reserved for the editor, and it may be necessary to increase the editor memory (with the `editmem=` session parameter) in order to have enough space.

Terminal Initialization
`status=`

Several session parameters are used to initialize the terminal or specify terminal behavior. The `status=`*n* is used to set the initial state of insert mode, status line, and keyboard:

`status=0`	insert off, status line off, APL keyboard
`status=1`	insert mode on
`status=2`	status line on in APL Session Manager (Default)
`status=4`	text keyboard instead of APL keyboard.

Combinations are specified by the sum of the values for the intended state. For example:

```
$ apl status=5
```

specifies insert mode on, status line off, and text keyboard.

Initialization Strings
`terminit=` and **`termdinit=`**

The session parameter `terminit=` specifies a character sequence that is transmitted to the terminal upon entry into APL. The default sequence is the ASCII character SO (Ctrl-N) to switch the terminal into the APL character set. The session parameter `termdinit=` specifies a corresponding string that is transmitted upon exit from APL; its default is the ASCII character SI (Ctrl-O). The `termdinit=` and `terminit=` strings are also transmitted when ⎕*CMD* and)*CMD* are used to execute a VMS DCL command from APL.

Both strings can be set to empty, effectively nullifying the default. For example:

```
$ apl terminit= termdinit=
```

Specifying empty strings is recommended on terminals where the default strings produce unwanted effects.

Identifying the Terminal to APL
`terminal=`

The `terminal=` parameter identifies the terminal name to APL. The terminal name will then be used in conjunction with the termcap database to provide a common set of terminal facilities for a wide variety of terminal types. The terminals supported in the supplied terminal database are:

`pc`	IBM PC running APL⋆PLUS PC
`c108`	HDS Concept 108
`avt`	HDS AVT
`c200`	HDS 200
`hp`	HP 2641
`vt100`	DEC VT100
`vt200`	DEC VT220 and VT240
`g`	generic APL video terminal
`h`	generic APL hardcopy terminal

To specify a terminal, use the appropriate abbreviation, as in:

```
$ apl terminal=vt200
```

Custom Termcap
`termcap=`

The APL⋆PLUS System for VAX/VMS uses exactly the same structure for entries in the termcap database as are used in the UNIX environment. The facility is general enough to enable APL to provide effective full-screen control on virtually any CRT display terminal. It is possible to develop your own termcap entries for terminals not presently included in the atermcap file supplied by STSC. If you do, you can specify your own file instead of STSC's atermcap file by using the termcap= parameter, as in:

```
$ apl termcap=filename
```

Since most terminals used in the VMS environment are designed to emulate either a DEC VT100 or VT220, you should try identifying your terminal to APL as one of these before developing your own termcap entry and configuration file. See Appendix D for additional information on the structure of a termcap database.

Customizing Logical Keystrokes

In order to work with a large variety of terminals, the APL★PLUS System can recognize a sequence of one or more characters as one of the logical editing keystrokes. Many terminals have cursor keys and other special keys that transmit special character sequences.

Each of the logical keystrokes recognized by APL can be customized to your terminal's behavior by specifying a session parameter. For example, the cursor-up key on a DEC VT100 terminal transmits a three-character sequence: ESC, '[', and 'A'. The cursup= session parameter defines the cursor-up keystroke, thus :

```
$ apl cursup=\033[A
```

causes APL to recognize the VT100's cursor-up key. The phrase 033 represents the ESC key, which is encoded as octal 033.

The following table lists the logical keystrokes used in this APL★PLUS System, the keywords used to specify them on the DCL command line or in a configuration file, and the default characters used for the keystrokes.

Logical Keystroke	Parameter Keyword	Default
Enter	`return=`	RETURN
Delete character	`delete=`	Ctrl-D
Clear-EOL	`clreol=`	Ctrl-E
Command prefix	`command=`	TAB
Refresh	`refresh=`	Ctrl-F
Page-up	`pageup=`	Ctrl-C
Insert/Replace/Overstrike	`toggle=`	Ctrl-T
Undo (restore line)	`undo=`	Ctrl-B
Alt-key prefix	`altkey=`	ESC
Cursor-left	`cursleft=`	Ctrl-H
Overstrike	`overstrike=`	Ctrl-P
Untype	`untype=`	Ctrl-R
Cursor-right	`cursright=`	Ctrl-L
Scroll-down	`scrolldown=`	Ctrl-U
Cursor-up	`cursup=`	Ctrl-K
Page-down	`pagedown=`	Ctrl-V
Cursor-down	`cursdown=`	Ctrl-J
APL-keyboard	`aplkeyb=`	Ctrl-N
Text-keyboard	`textkeyb=`	Ctrl-O
Scroll-up	`scrollup=`	Ctrl-Y
O-U-T	`out=`	Ctrl-Z

The defaults apply if no configuration file is used. When APL is invoked using the DCL command procedure `[APL.REL`*n*`]APL.COM`, the correct configuration is selected based on the terminal name you supply. For many terminals, the cursor-movement keys are defined in the `atermcap` database; if present, APL will implicitly use these definitions, and the configuration file need not supply them.

The ASCII characters with decimal values in the range 0 through 31 are called "control characters" and must be represented by a special notation. The APL★PLUS System follows conventions for denoting control characters either as a backslash followed by three octal digits or with ^ followed by a letter. Thus, the character SO (decimal 14) is represented as \016 (14 decimal is 16 octal) or as ^N, since Ctrl-N on the keyboard produces an SO character.

The logical keystrokes supplied to APL must be selected so each is distinct. If the same character sequence is specified for two different functions or one key sequence is a substring of another, the message *WARNING - CONFLICTING EDITING STRINGS* appears when APL is first invoked. For many terminals, the sequences transmitted by the cursor keys are part of the information stored in the termcap database. If so, it is not necessary to explicitly describe the cursor key codes in the APL configuration file.

The most common source of conflict is the character ESC, which is APL's default PF-key prefix and which is also transmitted by many terminals when the cursor keys are pressed. Specifying a unique sequence for the PF-key keystroke will silence the warning message. Ctrl-A (decimal 1) is a common choice for the PF-key keystroke in this case.

1-8 Configuration Files

APL session parameters contain information provided to APL when it is first invoked. These parameters control the characteristics of the APL session. Session parameters are specified either in the VMS command that invokes APL or in configuration files.

The number of session parameters you may need to provide is often too many to type as part of the VMS command that invokes APL. The APL*PLUS System, therefore, provides a means for you to use a configuration file that contains an arbitrary number of additional session parameters.

You can think of a configuration file as an extension of the VMS command line that invokes APL. Use the `initfile=` session parameter to specify the name of the configuration file. For example:

```
$ apl initfile=apl_disk:[apl.reln]pc.init
```

invokes APL with the file named `[APL.RELn]PC.INIT` as the configuration file.

A configuration file is an ordinary VMS file of ASCII text, that would be produced with the VMS editor. Each line of the configuration file contains the definition for one session parameter, just as it would be specified in the DCL command line. It is also possible for an configuration file to contain comments; any line that begins with "#" is considered a comment.

For example, suppose the file `[USERID]SAMPLE.INIT` contains the following lines:

```
[line 1]  # This is an APL init file that runs APL
[line 2]  # with a 150,000-byte workspace:
[line 3]  150000
[line 4]  # and 10 screens of session memory:
[line 5]  screens=10
[line 6]  # and initial workspace [userid]autostart:
[line 7]  initialws=[userid]autostart
```

Then the following two VMS commands are exactly equivalent:

```
$ apl initfile=[userid]sample.init

$ apl 150000 screens=10 initialws=[userid]autostart
```

Specifying Configuration Files

The session parameter `initfile=` is used to specify a configuration file explicitly by name. You can specify more than one configuration file, and it is possible for a configuration file itself to contain the `initfile=` session parameter to specify another configuration file. One configuration file can be linked to another, up to a maximum of 15 files deep.

Whenever the `initfile=` parameter is encountered, the effect is as if the contents of the specified file replaced the `initfile=` parameter. If a session parameter is specified more than once, the last definition is the one used. Thus, the effect of:

```
        $ apl initfile=[userid]sample.init 50000
```

is to run APL with a 50,000-byte workspace, even though the configuration file specifies 150,000 bytes.

EDITING

EDITING

Chapter Two
Editing Functions and Variables

This chapter describes the two editors that are available in the APL⋆PLUS System:

- a full-screen editor that can handle functions, character variables, and the record of your APL session.
- the traditional APL function editor, known as the del (∇) editor, which is line-oriented and only applicable to user-defined functions.

Section 2-1 of this chapter describes the full-screen editor and its associated ring of multiple images undergoing editing at any given time. The descriptions include the concept of the edit ring and its usage: placing a copy of an object from the active workspace in the ring, moving between images (including your APL session and its immediate execution mode), moving within the image on the screen and off, changing the image, and redefining an object from an image in the ring.

Section 2-2 of this chapter describes the del editor and its associated function definition mode. The descriptions include entering and leaving function definition mode, and selecting, displaying, entering, editing, or deleting individual lines of the function.

2-1 The Full-Screen Editor

The full-screen editor is the part of the APL⋆PLUS System that manages the input and output between the user and the APL system. The full-screen editor includes a program called the Session Manager that controls what appears on the terminal screen. Because the Session Manager and full-screen editor are one unified program, you can edit ⎕ and ⍞ input and output, and ⎕*WPUT* output, as well as APL functions and data.

Using the Session Manager/editor, you can:

- create or modify functions or variables in the active workspace
- edit several objects at the same time
- edit the image of the APL session and save text into functions and variables
- use cursor movement, scrolling, and paging
- insert and delete characters
- copy, move, and delete blocks of text
- move blocks of text between functions, variables, and the session
- search for specific character sequences and optionally replace them with another string
- switch to text mode to allow entry of ASCII text (upper- and lower-case letters, characters such as #%&", and so on).

A Ring of Editing Images

The full-screen editor operates on visual images of APL functions or variables. The editor can work with many images at the same time, although only one image at a time can be displayed on the terminal. Each image is stored in a *logical screen* that has a name and a unique screen number. A logical screen exists outside of the workspace where actions like `)LOAD` and `)CLEAR` do not affect it.

The screen images stored in the editor are arranged in a ring, like a rotating desktop telephone card file where many cards are stored but only one is visible. You can move freely from one image to another without losing its contents. Commands are provided to create new screens, move between them, delete them, and define functions or variables from them.

Changing the image of a function or variable in the editor has no effect on the object in the workspace until the image is explicitly written back into the workspace.

The APL session is simply one of the images in the edit ring. Editor commands can be used on the APL session, providing some very useful capabilities. For example, you can develop APL statements interactively in the session, then copy the lines of code into the image of a function being edited. The string search feature can be used to find a line of text in the APL session. You can even write the session image into the workspace as a variable.

Special Behavior of the APL Session Screen

There are three main differences in behavior between working in the session screen and editing a function or variable:

- The response to the Return key is different. In the APL session screen, pressing the Return key causes APL to read the current line as input and act upon it. When editing an object such as an APL function, pressing Return simply moves the cursor to the beginning of the next line.

- The maximum number of lines preserved in the APL session screen is fixed when APL is invoked. The default maximum is four terminal pages, but you can use the startup parameter `screens=` to specify a larger maximum (see Section 1-7). Once all the lines are used, the topmost lines are discarded as new ones are appended. When editing a function or variable, however, data is never lost. New lines are added to the object as it grows, limited only by the amount of memory allocated to the editor at the start of the APL session (see Section 1-7 for information on the `editmem=` parameter).

- Line numbers can be displayed in edited objects but not in the APL session screen.

Status Line

The last line of the terminal screen is reserved for a status line which contains information about the active screen and keyboard input state. The status line can be switched on and off using Command T (described later in this section). Here are some examples of status lines:

```
SCREEN 1:APL SESSION Ovs APL
```
(The APL session screen.)

```
SCREEN 2:FN READ Ins APL
```
(An APL function named *READ*.)

```
SCREEN 3:CHAR MAT PRICES Ins Text
```
(A character matrix *PRICES*.)

The status line consists of three fields:

- the screen number, which uniquely identifies the logical screen.
- object identification, which identifies the object in the screen; the field will contain *APL SESSION* to identify the session screen; for other screens, it will contain the designation *FN* for an APL function, or *CHAR* followed by *VEC* or *MAT* to identify a character vector or matrix, followed by the object name
- editor state, which contains the indications *Ovs* for overstrike mode, *Rep* for replace mode, *Ins* for insert mode, or *Cmd* for command mode; *APL* indicates APL font keyboard and *Text* indicates ASCII text keyboard input.

Full-Screen Editor Commands

Full-screen editor commands can be divided into two classes:

- basic and frequently used commands such as "move the cursor up one line," "untype the last character," or "start insert mode." They are typed with single keystrokes. These keystroke commands include Untype, Delete, Undo, Overstrike, Clear-EOL, and keystrokes for cursor movement and window scrolling. They are listed in Section 1-5.

- powerful and general commands such as "replace one string with another," "delete five lines," or "delete all characters through the next '◊'." These commands are formed starting with a prefix key and generally require multiple keystrokes.

General Full-Screen Editor Commands

General commands all start with the Command keystroke (usually the TAB key), which tells the editor that the keystrokes that follow are part of a command and not APL input. Most of the commands allow an optional count or repetition factor and a modifier or arguments. Here are some examples of general commands:

`TAB 10` *A* (Move the cursor to absolute line number 10.)

`TAB 6` *D* `SPACE` (Delete 6 characters to the right.)

`TAB 4` *D* `RETURN` (Delete 4 lines.)

In these examples, the Command prefix is the `TAB` character, the default. However, you can use a different keystroke as the Command key. Section 1-7 explains how to specify the keystroke. The words `SPACE` and `RETURN` represent the terminal's space bar and Return keys. Spacing between the characters of the command provides clarity in the example; the spaces would not be typed in the actual command. Likewise, brackets, used to indicate fields that are optional, are not typed in to the actual command. The general form of an editor command is:

`TAB` *[count] name parameters*

The *count* is typically optional, and *parameters* are not needed for all commands.

If you press the Return key to end a command, the word `RETURN` appears in the description. Many commands do not require you to press `RETURN`; they are completed as soon as you have typed the entire command. The characters used in commands assume the APL keyboard; for example, TAB *E* means the TAB key followed by the unshifted *E* key.

Commands for Editing Functions and Variables

The following commands are used to control the contents of the edit ring. Commands are provided to create a new logical screen, to move from one screen to another or back to the APL session, and to define APL objects from their screen images.

Edit object TAB *E name [options]* RETURN

Create a logical screen containing an image of the named object and display the image on the screen for editing. The name of the screen is the same as that of the object.

Write object TAB *W [name] [options]* RETURN

Write a copy of the current logical screen into the active workspace. If a name is specified, the object is given that name; otherwise, the name from the function header or, for a variable, the name of the screen is used.

Exit editor TAB *Z [name] [options]* RETURN

Write a copy of the current screen into the workspace (as *name*), delete the screen, and return to the APL session. This command is the quick way to exit from the editor.

Quit editor TAB *Q [ALL]* RETURN

Delete the current logical screen and return to the previous one. If *ALL* is specified, all screens except the APL session screen are deleted.

Get object TAB *G name* RETURN

Get the object (*name*) from the workspace and insert its image at the current location in the screen.

Forward screen `TAB` *F [name]* `RETURN`

Move forward to the next logical screen or, if *name* is specified, to the screen with that name.

Backward screen `TAB` *B* `RETURN`

Move backward to the previous logical screen.

APL **S**ession screen `TAB` *S* `RETURN`

Move to the APL session screen.

Display **R**ing `TAB` *R*

Display the contents of the ring of logical screens. The table of contents remains visible until you press Return.

The *options* field in the above commands can contain one or more of the following options, as appropriate, to specify the type and rank of the object:

`-FN`	edit or write the object as an APL function
`-CHAR`	edit or write the object as character data
`-VEC`	edit or write the object as a vector
`-MAT`	edit or write the object as a matrix
`-LN0`	edit the object without line numbers
`-LN1`	edit the object with line numbers (default).

When you begin editing an object with `TAB` *E*, the default object type is that of the object, if it already exists. If the object does not exist, the default type is function (`-FN`). When you write the object back to the workspace (`TAB` *W*), it will be stored as the same object type as it was in the editor.

Note that changing the active screen with **edit** (`TAB` *E*), **forward** (`TAB` *F*), **backward** (`TAB` *B*), or **session** (`TAB` *S*) does not destroy the current logical screen. The current screen becomes inactive, and you can return to it by moving around the edit ring.

Commands for Moving Around on the Screen

In additon to the basic cursor movement keystrokes, the following commands can also be used to move the cursor around the screen and to scroll and page up and down:

Pseudonym	**Keystrokes**	**Result**
←←	`TAB` *H*	Move to the left margin.
→→	`TAB` *L*	Move to the right end of the line.
Bottom	`TAB` *J*	Move to the last line of the logical screen.
Top	`TAB` *K*	Move to the first line of the logical screen.
n **←**	`TAB` *n H*	Move *n* spaces to the left.
n **→**	`TAB` *n L*	Move *n* spaces to the right.
Down	`TAB` *n J*	Move *n* lines down.
Up	`TAB` *n K*	Move *n* lines up.
Scroll ↑	`TAB` *n Y*	Scroll the window up *n* lines.
Scroll ↓	`TAB` *n U*	Scroll the window down *n* lines.
Page ↑	`TAB` *n C*	Page the window up *n* terminal screens. If *n* is omitted, page up one screen.
Page ↓	`TAB` *n V*	Page the window down *n* terminal screens. If *n* is ommitted, page down one screen.
Locate →	`TAB` *X c*	Move right to the character *c*.
Locate → ***n***	`TAB` *n Xc*	Move right to the *n*th occurrence of character *c*.
Locate ←	`TAB` *X c*	Move left to the character *c*.

Locate ← *n*	`TAB` *n X c*	Move left to the *n*th occurrence of character *c*.
Locate ◊	`TAB ◊`	Move right to the next diamond.
Locate ◊ *n*	`TAB` *n* `◊`	Move right to the *n*th diamond.
Jump	`TAB` *O c*	Move to the line marked with the letter *c* (see "Marking a Line," below).
Jump *n*	`TAB` *n A*	Move to absolute line number *n*.

Commands for Deleting Characters

You can delete multiple characters with the **delete** command. Each form of the command takes an optional number which defaults to 1 if omitted.

Delete →	`TAB` *n D L*	Delete *n* characters to the right.
	or `TAB` *n D* `SPACE`	(Alternate form of above command.)
Delete ←	`TAB` *n D* `H`	Delete *n* characters to the left.
Delete → *n*	`TAB` *n D X c*	Delete right to the *n*th occurrence of character *c*.
Delete ← *n*	`TAB` *n D X c*	Delete left to the *n*th occurrence of character *c*.
Delete *n* ◊	`TAB` *n D* `◊`	Delete right to the *n*th occurrence of ◊.

Commands for Marking a Line

A line can be marked with any one-letter name for reference in other commands. Once marked, the line can be referenced by the *O* modifier followed by the character. For example, `TAB` *O A* moves the cursor to the line marked with letter *A*.

Mark	`TAB` *M c*	Mark the current line with character name *c*.

Commands for Deleting and Saving Lines

One or more lines can be deleted from the logical screen with a single command. The deleted lines are saved into a buffer and can be inserted from the buffer into a different place in the logical screen or even into a different logical screen. Lines can also be saved into the buffer without deleting them. Only one group of lines is saved in the buffer at once, so deleting or saving a block of lines causes the previous buffer contents to be lost.

Delete ↓	`TAB` *n D D* or `TAB` *n D J*	Delete *n* lines downward, including the line on which the cursor appears; default number is 1 to delete current line only.
Delete ↑	`TAB` *n D K*	Delete *n* lines upward.
Delete Block	`TAB` *n D A*	Delete from current line upward (positive *n*) or downward (negative *n*) to line number *n*; if *n* is unspecified, all lines to the last line in the logical screen are deleted.
Delete Thru	`TAB` *D O c*	Delete through line marked with letter *c*.
Copy ↓	`TAB` *n* ← ← or `TAB` *n* ← *J*	Save *n* lines downward into the buffer, including the current line. Note that ← represents the APL assignment arrow, not the cursor-left key.
Copy ↑	`TAB` *n* ← *K*	Save *n* lines upward into the buffer.
Copy Block	`TAB` *n* ← *A*	Save from the current line to absolute line number *n*; if *n* is unspecified, all lines to the end of the logical screen are saved.
Copy Mark	`TAB` ← *O c*	Save from the current line to the line marked with letter *c* into the buffer.

Commands for Inserting New and Saved Lines

New lines can be inserted with the **insert line** command. The lines saved in the save buffer (from **delete** or **copy**) can be inserted with the **put** command.

Insert Line ↓	`TAB` *I*	Insert a blank line below the current line.
Insert Line ↑	`TAB ι`	Insert a blank line above the current line.
Put ↓	`TAB` *P*	Put the contents of the save buffer below the current line.
Put ↑	`TAB *`	Put the contents of the save buffer above the current line.

Note that moving from one logical screen to another with **edit** (`TAB` *E*), **forward** (`TAB` *F*), **backward** (`TAB` *B*), or to **session** (`TAB` *S*) does not alter the contents of the save buffer. **Copy** and **put** can be used to move lines between functions or from the session to a function.

Commands for Splitting and Joining Lines

Join	`TAB ,`	Join the next line to the current line to form one line.
Split	`TAB .`	Split the current line at the cursor position to make two lines.

Commands for Searching and Replacing

The **search** and **replace** commands are used to locate a sequence of characters in a screen and optionally replace it with a different string. String searches wrap around to the start of the logical screen if the end of the screen is reached.

Search	`TAB` / *string* `RETURN`	Search the logical screen for the next occurrence of *string*.
Repeat Search	`TAB / RETURN`	Repeat the last string search command.
Replace *n*	`TAB` *n* \ \ *string2* `RETURN`	Change *n* occurrences of the search target from the previous string search to the given *string2*. Default *n* is 1.
Replace All	`TAB $` \ \ *string2* `RETURN`	Change all occurrences of the search target from the previous string search to the given *string2*.
Repeat Replace	`TAB SPACE`	Repeat the previous string replace command.
Replace → *n*	`TAB` *n* \ *L string2* `RETURN`	Delete *n* characters to the right and replace them with *string2*.
Replace ← *n*	`TAB` *n* \ *H string2* `RETURN`	Delete *n* characters to the left and replace them with *string2*.
Replace Mark →	`TAB` *n* \ *X c string2* `RETURN`	Delete from the cursor to the *n*th occurrence of character *c* and replace with *string2*.
Replace Mark ←	`TAB` *n* \ *X c string2* `RETURN`	

		Delete from the cursor left to the *n*th occurrence of character *c* and replace with *string2*.
Replace ◊ *n*	`TAB` *n* `\ ◊` *string2* `RETURN`	Delete from the cursor to the *n*th occurrence of `'◊'` and replace with *string2*.

Miscellaneous Commands

Status	`TAB` *T*	Turn the status line on and off.
Help	`TAB ?`	Display the help screen and allow the user to browse through a list of editor commands.

Canceling a Command

Any general editing command can be canceled before it is completed by repeating the Command prefix; for example,

```
TAB 5 D TAB
```

This command, which was intended to delete 5 lines, does not take effect because it is terminated with Command.

Command Mode

There is a command mode in which all keystrokes are treated as editor commands. In this mode, the command prefix can be omitted in all commands. This mode is useful for editing an existing object, moving data around, making global changes, and so on. As an example, the commands to move to line 7 of a function and delete 3 lines normally would be:

```
TAB 7 A
TAB 3 DD
```

In command mode they would be:

```
7 A
3 DD
```

Command mode overrides insert mode and overstrike mode. Characters typed in during command mode do not become part of the text.

`TAB ×`	Switch in and out of command mode (the keystroke is the APL multiply key).

Entering the insert line command from command mode temporarily returns you to insert or overstrike mode to enter data on the new line. As soon as you leave the new line with any editor command, you are back in command mode.

Editor System Commands and System Functions

The system command `)EDIT` and system function `⎕EDIT` invoke the full-screen editor and have the same effect as the TAB *E* key sequence. The system command `)HELP` is the same as TAB `?`.

Using Hardcopy or Limited Terminals

The features of the editor are available on any CRT terminal with a minimum set of capabilities. The minimum functions required to support a terminal in full-screen mode are :

- clear to end of line
- clear screen (or alternatively, clear to end of page)
- cursor addressing (or alternatively, home-cursor).

Most CRT terminals have at least these facilities. If the terminal has more advanced features than these, they will be used, but they are not required. The terminal need not have any cursor control keys or function keys, but if it does they can be used. Information on how to control the terminal is extracted from the termcap database. For further information on termcap and APL's use of it, see Section 1-4 and Appendix D.

The APL⋆PLUS System will function with hardcopy terminals and terminals without the minimum set of features, but editing capabilities are limited. In these limited terminals the system works in line mode, and none of the general full-screen editor commands can be used. The system commands `)HELP` and `)EDIT` and the system function `⎕EDIT` are disabled, but some of the basic editor commands can be used. These provide the user with basic editing capabilities like those used on earlier APL system implementations. In line mode, the following editing keystrokes can be used:

Untype	Work as backspace followed by linefeed.
Delete	Work only on the last character of the line; the visual effect is a linefeed.
Clear-EOL	Erase all characters from the cursor to the end of the line. The visual effect is a linefeed.
Enter	Enter the current line as input to APL.
Cursor-down	Move the cursor down one line.
Cursor-left	Move the cursor left one space without erasing what is there (non-destructive backspace).
Cursor-right	Move the cursor right one space without erasing what is there (non-destructive space).
Undo	Discard everything typed on the line and re-prompt.

Editor Errors

An incorrectly formed editor command will produce a warning to the user by flashing the terminal screen or sending a bell character. This may occur when you try to delete more lines than a screen contains, when you try to form an illegal overstrike, or from a variety of other circumstances. Additionally, the editor may produce the following error messages:

CANNOT GET UNDEFINED OBJECT

The object requested does not exist in the workspace.

CANNOT QUIT APL SESSION

The **quit** command (TAB *Q*) is not valid from the APL session screen.

CANNOT REPLACE FN WITH VAR OR VAR WITH FN

The **write** command (TAB *W*) was issued to write an object as a variable when it already exists as a function in the workspace or vice versa. Use the **session** command (TAB *S*) to move back to the session to erase the object and the **forward** command (TAB *F*) to get back to the original screen to retry the **write** command.

DEFN ERROR

An error in the structure of the function prevents the function from being defined in the workspace.

DOMAIN ERROR

The object of an **edit** command (TAB *E*) is not an allowable data type. Numeric data, nested arrays, and heterogeneous data cannot be edited.

HELP FILE NOT FOUND

A **help** command (TAB *H*) was issued and the help file cannot be found. The default help file is `[APL.RELn]HELP.HLP`. Alternate help files can be specified with the `help=` session parameter. See Section 1-7 for details.

IMPROPER NAME

An **edit**, **get**, **forward**, or **write** command (TAB *E*, TAB *G*, TAB *F*, or TAB *W*) was issued with an improperly formed variable or function name. Names must be well-formed APL identifiers.

`INCORRECT OPTION TO QUIT COMMAND`

The **quit** command (`TAB` *Q*) was used with an option other than `ALL`. `ALL` is the only option allowed with the **quit** command.

`LENGTH ERROR`

The **edit** or **get** command (`TAB` *E* or `TAB` *G*) was issued for a matrix that has a second dimension greater than 1014 or for a vector that has more than 1014 characters between newline characters.

`LINE TOO LONG`

The **join** or **replace** command (`TAB ,` or `TAB \`) issued would have resulted in a line of more than 1014 characters in length.

`NAME MISSING`

An **edit** or **get** command (`TAB` *E* or `TAB` *G*) was issued without specifying the name of the variable or function to be edited.

`NO PREVIOUS CHANGE`

A **repeat replace** command (`TAB SPACE`) was issued when there is no last change to repeat.

`NOT ENOUGH MEMORY AVAILABLE`

The editor has run out of memory and cannot perform the requested operation. You can delete an unneeded logical screen from the edit ring by moving to it and using the **quit** command (`TAB` *Q*). If this problem occurs in the **delete** command (`TAB` *D*), it is because the editor is attempting to save the deleted lines so you can later retrieve with "Undo." Try deleting fewer lines at once. The memory allocated to the editor can be increased or decreased with the `editmem=` parameter at APL startup. See Section 1-7 for details.

NOT FOUND

The object of the **search** command (`TAB` */*) was not found.

OBJECT ALREADY EXISTS

The **write** command (`TAB` *W*) was used to rename an object, but an object with the new name already exists in the workspace. Use another name or return to the session with the **session** command (`TAB` *S*), erase the object, and return to the edit screen with the **forward** command (`TAB` *F*).

ONLY IN IMMEDIATE EXECUTION

During ⎕ or ⍞ input, an edit command was issued that would move to a new logical screen. This is allowed only during immediate execution mode input.

RANK ERROR

The **edit** or **get** command (`TAB` *E* or `TAB` *G*) was used to edit an array of more than two dimensions.

SI DAMAGE: WRITE IGNORED

The **write** command (`TAB` *W*) was issued for a function which has been altered in such a way as to affect the localization of names in the header or the line labels. A version of the same function is suspended. Use the **session** command (`TAB` *S*) to move back to the session to clear the suspension and the **forward** command (`TAB` *F*) to get back to the original screen to retry the **write** command.

UNKNOWN OPTION

An undefined option was specified in a command.

UNKNOWN WRITE OPTION

An undefined or invalid option was specified in a **write** command (`TAB` *W*).

UNNAMED OBJECT - SUPPLY NAME

A **write** command (`TAB` *W*) was issued for the APL session screen without supplying a name. A name must be specified for this case.

2-2 The Del Editor

This section describes the del editor, a traditional line-oriented function editor. Topics covered in this section include how to:

- enter and leave function definition mode
- understand the prompt
- select a line of the function (with `[ ]`)
- display the function or a line of the function for possible editing or replacement (with ⎕)
- delete selected lines (with ~)
- enter or edit a selected line
- exit from the editor, leaving the function unchanged.

Entering Function Definition Mode

To enter function definition mode, you must first be in immediate execution mode (that is, the system has typed a prompt of six spaces and is awaiting your input). Then:

- Type a ∇.

- Type a function name (*NAME* is used in the discussion below).

- Press Enter.

Places where variations are possible are indicated by dots:

```
∇  . . .  NAME  . . .
```

To create a new function called *NAME* (when there is no function or variable called *NAME* in the active workspace), enter:

```
∇ RESULT ← LEFT NAME RIGHT
```

This creates the new function, and establishes its syntax by using this line as the header line. Any of the following can be omitted, thereby establishing a different syntax for the function name:

```
RESULT ←

LEFT

LEFT AND RIGHT.
```

Thus, ∇ *NAME* can create a new function called *NAME* that takes no arguments and yields no explicit result.

To re-open an existing unlocked function called *NAME* that is already present in the workspace, enter:

```
∇ NAME
```

Regardless of the syntax of the function name, the system will prompt for a new line at the end of the function.

Leaving Function Definition Mode

To leave function definition mode, end any line with a ∇ and press Enter. Alternatively, using ⍫ will lock the function.

The effects of leaving function definition mode are:

- The lines of the function are reordered according to their line numbers. The header is treated as line 0. The other lines are renumbered with consecutive integers, if necessary.
- The current header, if different from the original header, determines the name and syntax for subsequent use of the function.
- The function is locked (can never be redisplayed or edited, but can be erased) if a ⍫ was used when entering or leaving definition mode.
- If the function being edited was suspended (noted in the state indicator with a `*`) and the renumbering has changed the value of labels, the new values will apply when execution is resumed. The error message *SI DAMAGE* indicates that the editing changes will not permit the function to be restarted. The damaged levels in the state indicator are indicated by `[¯1]` in `)SI`. Entering `)RESET` will clear the damaged state indicator, and the newly edited function can be run.
- The six-space prompt for immediate execution mode replaces the square-bracketed prompts of definition mode.

Discarding Editor Changes

If you change your mind and want to discard all the changes you have made in function definition mode, you can exit from the editor by typing:

```
      [→]
```

in response to the numbered prompt. The system displays the message *EDIT ABORTED*, and the original form of the function, if any, is restored.

Prompts in Definition Mode

A function definition is a numbered sequence of lines, each containing an APL statement. When displayed, the lines are shown in order, each preceded by the line number enclosed in square brackets (`[]`).

The prompt in function definition mode is also a function line number enclosed in square brackets; it is the number of the line where the system expects to store what you type. The complete prompt also includes the cursor in the seventh column of the same line, waiting for your input. Note that if the line number is long (such as `[1234.567]`), the cursor is displayed farther to the right.

Selecting Function Lines

To select the function line you want to enter, edit, or delete, specify the line number in square brackets and press Enter. You can use such a line choice to override the prompt just printed or to end the line you used to enter definition mode. The system responds by prompting for the information to be stored under that line number. For example:

```
      ∇OLD[5]

[5]
```

Displaying Function Lines

The symbol ⎕, used within square brackets, causes the lines already in the function definition to be redisplayed.

`[⎕]`	Displays the entire function as currently defined.
`[⎕n]`	Displays the function from line *n* through the last line of the function (or until you press Ctrl-*C*).
`[n⎕]`	Displays the line numbered *n* and prompts for a replacement. (You can override the prompt, avoiding replacement by typing a different line number within brackets.)
`[n⎕c]`	Displays the line numbered *n* for possible editing, leaving the cursor in column *c*. If a zero value is entered in *c*, the cursor is displayed at the end of the line.

Deleting Function Lines

To delete a line from the existing definition of the function, precede the line number with a ~ and enclose the entry in brackets. For example, `[~12]` (followed by Enter) in response to a definition mode prompt completely removes the line stored as line [12]. To delete several lines at once (even lines just inserted between others), type a ~, type the line numbers to be deleted (separated by a space), and enclose the entry in brackets; for example:

```
[~4 5 12 15 8 3.5 3.6]
```

The same technique can be applied in a line used to enter or leave definition mode (or used to both enter and leave definition mode in the same line). For example:

```
∇NAME[~2 9]∇
```

enters definition mode only long enough to delete the second and ninth lines after the header of the function name, then leaves definition mode and reprompts for immediate execution mode on the next line.

Entering or Editing Function Lines

Once the line has been selected (and possibly displayed), all of the single character editing actions are available for adding new material to the line or changing new or old material in the line. This is true regardless of how the line was selected, how it is displayed, and whether or not the line number is an integer.

Even the number of the line can be changed. In this case, both the line with the old number and the line with the new number will remain.

While in function definition mode, you can edit and re-enter any line displayed on the screen. Typically, the steps are :

- Move the cursor to the already displayed line .
- Edit the line .
- Press Enter.

Combining Function Lines

It is possible to combine most elements of definition mode into a single line. The following examples show several of the combined actions you can use:

```
     ∇RESULT←LEFT NAME RIGHT;LOCAL1;LOCAL2
[1]   LOCAL1←2×LEFT
[2]   LOCAL2← ρRIGHT ◊ RESULT ← LOCAL2 + LOCAL1
     ∇
```

In the above example, only the line number prompts [1] and [2] were typed by the system. The first line enters definition mode, creates a new function with two arguments, two local variables, and an explicit result. The second line enters a first line for the function. When the system prompts for the second line, the user enters a line and ends it by leaving definition mode.

```
     ∇NAME[1.5] RESULT ← RESULT + ρ RIGHT
∇
```

The example above enters definition mode, inserts a new line between lines [1] and [2], and leaves definition mode, causing renumbering of lines. When the function is redisplayed, the new line is line [2], and the numbers of the following lines are all one higher than they were before. (This is an important reason for using labels on lines to which you want to branch within the function. If the function is subsequently revised, line numbers may change, but the labels remain with the intended destination line.)

To redisplay the function without staying in definition mode, enter:

```
     ∇NAME[⎕]∇
```

If you enter the statement:

```
     ∇NAME[⎕]
```

the system displays the entire function, stays in definition mode, and then prompts for a new line to follow the last line of the function. If name is not the *NAME* of an existing function, the system reports a *DEFN ERROR* is reported. Entering the statement:

```
     ∇NAME[3⎕]∇
```

displays line [3] and returns the system to immediate execution mode.

```
∇NAME[⎕3]∇
```

skips the header and lines `[1]` and `[2]` but lists the remainder of the function and then leaves definition mode.

If you omit the final ∇ in the last example, the system stays in function definition mode and prompts for entry of the next function line (one plus the highest line number). You can either enter information for the new line or request redisplay of an existing line. For example:

```
[12]    [3⎕7]
```

redisplays line `[3]` with the cursor in column 7. You can then insert additional information at the beginning of line `[3]`, possibly separated from the existing statement with a ⋄.

Error Reports in Function Definition Mode

DEFN ERROR

One of the following has occurred:

- An incorrectly formed header was used.
- A line was entered that does not begin with a square-bracketed line.
- A ∇ not in quotes or in a comment occurs within a line.
- An attempt was made to delete the header, to define a function with the same name as an existing variable or label, to re-open definition of an existing function by specifying something other than the function name after the ∇, to open a locked function, or to open a pendent function (use ⎕*VR* to see it).

SI DAMAGE

You have closed definition after making changes to suspended or pendent functions that invalidate the current suspended state indicator. The state indicator must be cleared before execution can be restarted.

SI DAMAGE PENDING

You have made changes to a suspended or pendent function that invalidate the current suspended state indicator. You can recover at this point by canceling the changes made using `[→]`.

SI WARNING

You are opening definition on a suspended or pendent function. Changing the syntax or labels of a suspended function or making any changes to a pendent function will damage the suspended state indicator. However, other corrections to a suspended function are permitted.

WS FULL

There is not enough space in the active workspace for the function being defined.

FILES

FILES

Chapter Three
Using Files with APL

The APL⋆PLUS System for VAX/VMS, like other APL⋆PLUS Systems, gives APL the ability to store and access data in disk files outside of the APL workspace. As a result, you can use the flexible sharing capabilities of the APL⋆PLUS System's component files in conjuction with other types of files native to the VMS operating system environment. With these capabilities, your VMS operating system becomes an excellent environment for multi-user (shared data) APL applications.

The following general capabilities are available with the APL⋆PLUS System's disk files:

- the ability to enter, retain, and access data in disk files, where the data can remain after the APL session ends
- the ability to store more data than can all fit at a time in the APL workspace and to retrieve it in manageable portions
- the ability to perform file operations (like creating, erasing, appending to, and renaming a file) under program control
- the ability to use native VMS files from APL, providing an interface between APL and non-APL programs such as database managers, word processors, and spreadsheets.

How This Chapter Is Organized

Section 3-1 introduces files in the APL⋆PLUS System; it defines files, covers the fundamentals of file use, and describes the minimum set of file operations needed for working with files.

Section 3-2 explains additional file operations useful for managing personal (nonshared) files. Sections 3-2 and 3-3 together describe all the file operations needed for personal file management.

Section 3-3 introduces file sharing. File operations are discussed that allow both casual personal exchange of files and information and formal, regulated access to a common information base.

Section 3-4 discusses detailed control of file access. It is intended for individuals who will be implementing major applications that require the use of shared files.

Section 3-5 discusses the access matrix in more detail and explains how to apply it and the effects of making changes to it.

Section 3-6 explains how the APL*PLUS System relates the VMS directories to APL libraries.

Section 3-7 compares the APL component files and the native files available through the APL*PLUS System.

3-1 Fundamentals of File Use

The APL*PLUS System includes a built-in file system for the APL environment. This file system allows easy use of either floppy disk drives or hard disk drives to store data either in APL component files or other file types native to the environment. You can store, retrieve, and otherwise manipulate data with APL system functions.

Key Concepts

The material in this manual presumes that you have a clear understanding of several key concepts:

files: A file is a place for storing and organizing data in permanent disk storage. The APL*PLUS System uses two distinct kinds of files: (1) APL component files, which are used to hold APL arrays, and (2) native files, which are standard to the VMS operating system environment.

APL component files: An APL file is structured as a set of components, each of which contains one APL array. An APL file component is much like a variable, except that it has a number instead of a name. Any value that can be stored in a variable can also be stored in a file component. The file component contains complete information about the APL array, including its shape and datatype.

native files: These are VMS sequential Stream_LF files, of the form used by the VAX C run-time library. The APL⋆PLUS System treats native files as simple sequences of characters (bytes of data), leaving the programmer responsible for any structuring and housekeeping. Native files provide a means for APL applications to share data with non-APL applications.

directories: The VMS operating system organizes files using a hierarchy of directories. A directory is a special kind of native file that contains a list of file names. The names in the list can be files or other directories. Any file in the system can be located by specifying the directory in which it resides. Every user has a default directory where files and workspaces are stored and sought if no directory is explicitly named. Users can also create and access files in other directories by explicitly referring to the directory in a command.

libraries: Traditional APL systems organize files and workspaces into libraries. Libraries are much like directories except that they are identified by numbers instead of names, and they are not hierarchical (that is, a library cannot contain another library). The APL⋆PLUS System provides a mechanism for simulating traditional libraries by associating a library number with a VMS directory. This allows applications that were developed on other APL⋆PLUS Systems to run without modification on this APL⋆PLUS System.

users: To the APL⋆PLUS System, a user is one individual. Every user is recognized by a user name, which is used to log in to the system. Every user also has an account number or user identification code (UIC), which is used by the APL⋆PLUS System to identify individual users for file-sharing purposes. You can discover your user account number by entering 1↑⎕*AI*.

access: A user has access to an APL file when the APL⋆PLUS System permits that person to use it in some way. Typically, you have access to every file that you own, and can give access to other users by appropriately setting the access matrix.

ownership: Owning a file or workspace means that you are the person (user account number) who originally created it or most recently renamed it.

Comparison of APL Workspaces and Files

There are two ways to make permanent copies of APL data: in a saved workspace or in an APL file. APL files and saved workspaces are alike in that:

- Both reside on a disk.
- They can be created, used (and perhaps modified) over a period of time, and erased when they are no longer needed.
- Both have names and optional library numbers.
- Both can contain many objects, each of which can be any APL value. Stored data can be a scalar, vector, matrix, or higher-dimensional array, nested or simple, holding either characters or numbers. The only limit on the size of a component or variable is that it must fit in the available space in the active workspace and on the disk.
- They are owned by a particular user,generally the person who created them.

There are some important differences between a file and a saved workspace:

- A workspace can contain executable programs as well as data; a file can contain only data.
- More than one file can be active at the same time.
- A file can hold amounts of data that are larger than can fit in one workspace (which is limited by the amount of memory installed in your computer). The size of a file can range from a few bytes up to several megabytes, or more.

- The APL⋆PLUS System refers to a variable in a workspace by its name. The system refers to a component in an APL file by a component number that gives the component's position within the file. Component numbers are consecutive positive integers ranging from the number of the first component to the number of the last component.

File operations are used to bring the data contained in one component into the active workspace for processing and to save values generated in the active workspace as components of a file. File operations are performed through a collection of APL system functions. As system functions, file operations share many of the properties of APL primitive functions: they are always available for use in the workspace; they can be incorporated in user-defined functions; and many return explicit results that can be used in subsequent operations.

The names of the functions used with APL files all start with ⎕*F*, and each name indicates the kind of operation being performed. Examples of such functions are ⎕*FREAD* (for reading a component from a file) and ⎕*FAPPEND* (for appending to a file a value from the active workspace). The names of the functions for use with native files all start with ⎕*N*, and each name indicates the kind of operation being performed.

Improper use of file operations can lead to file errors. Such errors are indicated by error reports. Errors generated by file functions are just like errors generated by APL primitive functions in terms of their effect on the APL statement in which they occur. Each file operation is described in detail in Section 3-3.

For an APL file, the directory name or library number and file name together are termed the file identification. A file identification used as an argument to a file function is represented as a vector of characters enclosed in single quotes or as any other APL expression whose value is a character vector. The directory name is represented as a path: a sequence of directory names, separated by the character ".", which lead to the directory that refers to the file itself. If a library number is used instead, it is represented as a positive integer. The directory name or library number can be omitted if it is the same as the default directory being used.

A file name consists of from one to eleven uppercase letters (*A* - Z) and numeric characters (0 - 9), and must begin with a letter. If the library number is present, it is separated from the file name by one or more spaces. Leading and trailing blanks in the character vector are permitted.

The rules for file identifications differ somewhat depending upon whether directory mode or library mode is in effect. If any libraries are defined (using ⎕*LIBD* or a startup parameter), the system is said to be in library mode, in which file identifications follow the forms used in other APL⋆PLUS Systems. The following are valid file identifications in library mode:

'*SAMPLE*'	(Absence of library number implies default directory.)
'11 *TOOLS*'	(file named *TOOLS* in library 11)

If no numbered libraries are defined, the system is in directory mode and uses file identifications in the following form:

'*SAMPLE*'	(in the current working directory)
'[*JOE*.*USR*]*DATA*87'	(the file named *DATA*87 in the sub-directory named *USR*, which in turn is in the directory named *JOE*)

Most of the examples in this manual show the use of library mode because using it improves portability of code to other APL⋆PLUS Systems on other operating systems.

APL file identifications are patterned after workspace identifications but are logically distinct. A workspace and a file can have the same name. They are distinguished at the VMS operating system level by the file extension with which they are stored (`.VF` for files, `.WS` for workspaces), and they are distinguished in practice by the operation with which they are used.

Native, or VMS, file identifications use the same names both within APL and outside the APL environment. See your VMS operating system manual for the full set of conventions used by your system.

To be used, a file must first be paired with an integer file tie number. All operations on the file will refer to the file by its tie number rather than by its name. The particular value of the tie number can be any number in the acceptable range that is different from any other file tie numbers already in use. The pairing of a file and a file tie number is called a file tie, and a file is said to be tied if such an association has been made.

Descriptions and examples of file operations in this chapter use the two files described in the following section. Defining the functions used in this chapter will enable you to execute each example in turn.

Creating and Building Files

In the following example, the APL file named *PERSONS* has four components, each of which is a vector of characters. The APL file *SALES* also has four components, each of which is a vector of numbers.

File name:

Component number	*PERSONS*	*SALES*
1:	*'SMITH'*	5 6 3 1
2:	*'JONES'*	2 6 1
3:	*'KELLEY'*	4 6 2 9 1
4:	*'BECKER'*	20 6 4

A new APL file is brought into existence with the file function *⎕FCREATE*, and new components are placed in the file with the function *⎕FAPPEND*. The following program can be used to build the two files *PERSONS* and *SALES*:

```
     ∇ SAMPLE1;NAME;PMAT;T;VALUES
[1]    ⍝ CREATE THE FILES 'PERSONS' AND 'SALES':
[2]    'PERSONS' ⎕FCREATE 5
[3]    'SALES' ⎕FCREATE 20
[4]    PMAT← 4 6 ⍴'SMITH JONES KELLEYBECKER' ◊ I←1
[5]    ⍝ PROMPT FOR SALES FOR EACH PERSON:
[6]    ⍝ THEN STORE THE VALUES IN THE FILES:
[7]    ⍝
[8]    LOOP:→(I>1↑⍴PMAT)/END ◊ NAME←PMAT[I;]
[9]    'ENTER VALUES FOR ',NAME ◊ VALUES←⎕
```

```
[10] CN←VALUES ⎕FAPPEND 20
[11] 'VALUES STORED IN COMPONENT NUMBER ',⍕CN
[12] ⍝APPEND TO FILE 'PERSONS'
[13] T←((NAME≠' ')/NAME) ⎕FAPPEND 5
[14] I←I+1 ⋄ →LOOP
[15] ⍝
[16] END: ⎕FUNTIE 20 5
     ∇
```

The ⎕*FCREATE* operation establishes a new file and prepares it for further operations. The syntax (where *fileid* stands for file identification and *tieno* stands for tie number) is

'*fileid*' ⎕*FCREATE tieno*

The file name must be different from that of any existing file in that library, and the tie number must not be in use. When *SAMPLE* 1 is executed, line [2] creates the file *PERSONS* and ties it to 5. At this point, the file *PERSONS* exists in the library, but it is empty since it contains no components. Components are added to the file *PERSONS* by the function ⎕*FAPPEND* on line [11].

Similarly, ⎕*NCREATE* is used to create a native file. Native files follow somewhat different naming conventions and are used with negative tie numbers. See ⎕*NCREATE* in Chapter 3 of the *APL ⋆PLUS System Reference Manual* for more information.

The function ⎕*FAPPEND* puts a new component, containing an APL value from the active workspace, at the end of an APL file. The value in the workspace is not altered. The syntax is :

compno ← *value* ⎕*FAPPEND tieno*

The left argument, which provides the value for the new component, can be the name of a variable; for example:

name ⎕*FAPPEND* 5

The left argument can also be the result of any APL calculation. For example:

```
(5+2×4 6⍴⍳24) ⎕FAPPEND 5
```

The right argument is the tie number of the file to which data is being appended. *⎕FAPPEND* returns the new component number as its result. Examples of appending to files *SALES* and *PERSONS* appear in lines `[11]` and `[13]` of *SAMPLE*1.

Similarly, *⎕NAPPEND* appends the contents of an array, byte for byte, at the end of a native file.

Files created the way *PERSONS* and *SALES* were created have no maximum size. It is possible to specify a size limit for the file by specifying the number of bytes after the file identification in the left argument of *⎕FCREATE*. You will receive the error `FILE FULL` if you attempt to put more than that number of bytes of data into the file. The use of file size limits allows you to budget your disk storage and prevent "runaway" programs from filling the entire disk.

If line `[2]` of *SAMPLE*1 had been:

```
'PERSONS 10000' ⎕FCREATE 5
```

the size limit would have been 10,000 bytes. The default value 0 indicates no size limit.

Untying Files

The function *⎕FUNTIE* has the following syntax:

⎕FUNTIE tienos

It is used to break the pairing of an APL file and its tie number. The argument is a vector of zero or more file tie numbers, so *⎕FUNTIE* can untie several files at once (as in line `[16]` of *SAMPLE*1). *⎕FUNTIE* has no effect on the values stored in a file.

Similarly, *⎕NUNTIE* is used to break the pairing of a native file and its tie number.

Tying Files

An existing file that is not tied can be made available for processing, as shown in the next sample function. This function prints each of the components of the file named *PERSONS* along with the sum of the numbers in the corresponding components of *SALES*. The two file functions introduced in this example are ⎕*FTIE* and ⎕*FREAD*.

```
      ∇ SAMPLE2;I;X
[1]    ⍝ DISPLAYS TOTAL SALES FOR EACH PERSON
[2]     'PERSONS' ⎕FTIE 1
[3]     'SALES' ⎕FTIE 2
[4]     'NAME        SUM'
[5]     '----        ---' ◊ I←1
[6]    ⍝
[7]    LOOP:→(I>4)/END ◊ X←⎕FREAD 2,I
[8]     (10↑⎕FREAD 1,I),⍕+/X
[9]     I←I+1 ◊ →LOOP
[10]   ⍝
[11]   END: 'COMPLETE.'
      ∇
```

The function ⎕*FTIE* is used to establish an association between a file identification and a tie number. This procedure is referred to as tying a file. In *SAMPLE*2, *PERSONS* is tied to tie number 1 and *SALES* is tied to tie number 2. The syntax of ⎕*FTIE* is

'*fileid*' ⎕*FTIE* *tieno*

The function ⎕*FTIE* has the same effect as ⎕*FCREATE*, except that the file named in the left argument must already exist. Notice that in *SAMPLE*1 the file *PERSONS* was tied to the tie number 5, while in *SAMPLE*2 the file *PERSONS* was tied to 1. File tie numbers for use with ⎕*FTIE* can be any integer from 1 through 2147483647, and the number for a particular file can be different on different occasions.

Similarly, tie numbers for use with ⎕*NTIE* can be any integer from ¯1 through ¯2147483648.

Reading from a File

The function ⎕*FREAD* copies the value of a file component into the active workspace. The value can be placed in a variable (as in line [6] of *SAMPLE2*), used in an expression (as in line [7]), or displayed directly at the terminal. The syntax of ⎕*FREAD* (where *compno* stands for component number) is:

res ← ⎕*FREAD tieno compno*

The argument of ⎕*FREAD* is a two-element vector. The first element is the file's tie number, and the second element is the component number. ⎕*FREAD* returns the value of the specified file component as its explicit result.

Reading from a native file with ⎕*NREAD* is somewhat more complicated, since the file is not logically divided into components and does not internally track the datatype of what has been stored there. See ⎕*NREAD* in Chapter 3 of the *APL*PLUS System Reference Manual* for more information.

Duration of File Ties

The function *SAMPLE2* tied the files *PERSONS* and *SALES* but did not untie them, so those two files remain tied as files 1 and 2. A file remains tied until you untie it or end the APL session. Thereafter, you must re-establish the appropriate ties in order to use the same files.

The status of tied files is not changed by any system commands except)*OFF*. Therefore, loading into or clearing the active workspace leaves active any ties set previously during the terminal session. This makes it possible to load different workspaces to process files without having to re-establish the ties.

Listing Files in a File Library

The names of the files in a particular library or directory are obtained with the function ⎕*FLIB*. The syntax (where *lib* stands for library number or directory name) is :

result ← ⎕*FLIB lib*

result is a character matrix. Each row holds the identification of a file in the designated library. If, for example, the program *SAMPLE*1 had been created by a user in library [*JOE*.*USR*], then the expressions:

```
⎕LIBD '101 [JOE.USR]'
⎕FLIB 101
```

produce:

```
101 DATA1
101 SALES87
```

All the APL files in a library or directory appear in the result of ⎕*FLIB*, even those that the user account number requesting the display is not authorized to use.

⎕*FLIB* ⍳0 lists the APL files in the current default library (working directory). The system command)*FLIB* yields the same information as ⎕*FLIB*.

Similarly, ⎕*LIB* and)*LIB* return a list of all files (native, APL, and even workspaces) in a format consistent with native file names.

Erasing Files

When an APL file is no longer needed, it can be erased using the function ⎕*FERASE*:

fileid ⎕*FERASE tieno*

The file name is then deleted from the library. All of its components are destroyed, and the space they occupied is made available for use by other files. Since the file no longer exists, the file tie is also broken.

A file must be tied before it can be erased, and both the file name and the tie number must be given. Similarly, ⎕*NERASE* can be used to erase native files.

Copying Values from One APL File to Another

The following function is an example of changing the arrangement of filed information. The components of *PERSONS* and *SALES* are to be merged into a new file *RECORDS*. Each odd-numbered component will come from *SALES*. The two original files are erased after the merge is complete. Recall that *PERSONS* and *SALES* are still tied following execution of *SAMPLE2*.

```
     ∇ SAMPLE3;T
[1]    ⍝MERGE FILES TIED TO 1 AND 2 INTO 'RECORDS'
[2]    'RECORDS' ⎕FCREATE 3 ◊ I←1
[3]    ⍝
[4]    LOOP:→(I>4)/END
[5]    T←(⎕FREAD 1,I) ⎕FAPPEND 3 ⍝ FROM 'PERSONS'
[6]    T←(⎕FREAD 2,I) ⎕FAPPEND 3 ⍝ FROM 'SALES'
[7]    I←I+1 ◊ →LOOP
[8]    ⍝
[9]    END:⎕FUNTIE 3
[10]   'PERSONS' ⎕FERASE 1 ◊ 'SALES' ⎕FERASE 2
     ∇
```

Note that when the files have been merged, the new file *RECORDS* (tied to 3) is untied. It is not necessary to untie the files tied to 1 and 2 because erasing the files unties them automatically.

Sample Handling of a Native File

This example shows how a native file can be accessed and manipulated from within APL. Suppose that a VMS file NAMES contains a list of names to be sorted into alphabetic order. The file is structured like a typical VMS Stream_LF file: it is a linear sequence of characters, and the VMS newline character (⎕*TCLF*) separates the names. (Note that, in native VMS conventions, ⎕*TCLF* and not ⎕*TCNL* is used as the newline character.) The lines in the file are of varying lengths.

The system function ⎕*NTIE* ties ("opens") the native file:

```
          'NAMES' ⎕NTIE ¯5
```

and the function ⎕*NSIZE* reports the total number of bytes in the file :

```
      ⎕NSIZE ¯5
5129
```

Using the system function ⎕*NREAD*, it is possible to read a sequence of data from the file. For relatively small files such as this, the simplest method may be to read the entire file into the APL workspace at once. The following command reads the entire file and stores it in the variable *F*:

```
      F←⎕NREAD ¯5 82,(⎕NSIZE ¯5),0
```

The character vector can then be arranged into ascending order by converting it into a nested array (a vector of names) and arranging those names into the desired order. Once the names are rearranged, the list can be written back to file. In this case, since the length of the file does not change, you can replace the file with the new values. If *FS* is a character vector containing the sorted form of the file, then the following statement will write it to file:

```
      FS ⎕NREPLACE ¯1 0
```

The statement means "take the contents of *FS* and write them over the present contents of the file, starting at byte 0 (the first byte)."

Putting the ingredients together, the following function sorts a native file into alphabetic order by lines:

```
    ∇ SORTFILE FNAME;LINES;GRADEVEC
[1]   FNAME ⎕NTIE ¯1 ⍝ TIE THE FILE
[2]   LINES ← ⎕NREAD ¯1 82,(⎕NSIZE ¯1),0 ⍝ AND READ IT
[3]   LINES←⎕TCLF,LINES ⍝ LEADING DELIMITER ON FIRST LINE
[4]   LINES←1↓¨(LINES=⎕TCLF)⊂LINES ⍝CONVERT TO NESTED ARRAY
[5]   GRADEVEC←⎕AV⍋↑(⌈/⊃,/⍴¨LINES)↑¨LINES ⍝ GRADE
[6]   LINES←LINES[GRADEVEC] ⍝ REARRANGE INTO SORTED ORDER
[7]   LINES←¯1↓⊃,/LINES,¨⎕TCLF ⍝ CONVERT TO SIMPLE AGAIN
[8]   LINES ⎕NREPLACE ¯1 0 ⍝ WRITE BACK TO FILE
[9]   ⎕NUNTIE ¯1
    ∇
```

Comparisons with Corresponding Native File Operations

The correspondence between APL file operations and native file operations is not always exact. In particular, note that:

- Native files are always used with directory names. Library numbers do not apply to native files.

- Negative tie numbers are required for native files, rather than the positive tie numbers used with APL files.

- ⎕*NREAD* needs a more complex right argument since the file is not organized into components.

- ⎕*NAPPEND* places the exact contents of the array into the file, retaining no information about the array such as shape, datatype, or origin.

- There is no native file feature that is a close analogy for ⎕*FLIB* and)*FLIB*. The results of ⎕*LIB* and)*LIB* include all APL files and all APL workspaces too, since these are stored on disk as native files.

A more detailed comparison of native and APL files can be found in Section 3-7.

3-2 More Personal File Management Facilities

This section covers the remaining file functions used with both shared and nonshared files. The examples in this section assume prior creation and tying of files and are not intended for direct keyboard execution as were the examples in Section 3-1.

Inquiring about File Ties

Two system functions, ⎕*FNAMES* and ⎕*FNUMS*, report the names and numbers of the current APL file ties. ⎕*FNAMES* returns a character matrix in which each row is the library number and name of a currently tied APL file. ⎕*FNUMS* returns a vector of their current tie numbers, in an order corresponding to the names returned by ⎕*FNAMES*.

A convenient tabular display of this information is created by:

```
      ⎕FNAMES,'I10' ⎕FMT ⎕FNUMS
SAMPLE          1
PERSONS         2
```

The simplest way to untie all tied APL files is to execute the statement `⎕FUNTIE ⎕FNUMS`, since the vector returned by `⎕FNUMS` is exactly what `⎕FUNTIE` needs to untie all files.

Similarly, the functions `⎕NNAMES` and `⎕NNUMS` report on the status of native file ties. Given the negative tie numbers for native files, the corresponding formula for a tabular display is:

```
      ⎕NNAMES,'I11' ⎕FMT ⎕NNUMS
```

Replacing a File Component

A file component can be replaced by a new APL value using the function `⎕FREPLACE`. The syntax is:

value `⎕FREPLACE` *tieno compno*

For example, to replace the second component of the file tied to 1 with the character vector `'REPLACED'`, the following statement could be used:

```
      'REPLACED' ⎕FREPLACE 1 2
```

The new value can be any APL value that fits in the active workspace. If the new value is larger than the previous value, additional file storage is automatically provided.

A common use of `⎕FREPLACE` is to update a component by modifying its value. In this example, the value of the variable debit is catenated to the existing value of component 3 of the file tied to 7:

```
      ((⎕FREAD 7 3),DEBIT) ⎕FREPLACE 7 3
```

Note that the new value is larger than the value it is replacing. A replacement component need not occupy the same amount of space as the old value.

The corresponding native file operation, ⎕*NREPLACE*, cannot refer to components; in place of a component number you supply the first byte in the file to be replaced by the given value. That value will replace exactly the amount of the file needed to store it, regardless of what was there before. The programmer must evaluate positioning and space requirements. If more data is to be stored than the remaining space in the file can accommodate, ⎕*NREPLACE* will lengthen the file to make room for the rest of the data. For additional details, see ⎕*NREPLACE* in Chapter 3 of the *APL*PLUS System Reference Manual.*

Dropping Components from Either End of a File

The function ⎕*FDROP* removes components from either end of a file. The syntax is:

```
⎕FDROP tieno n
```

The argument to ⎕*FDROP* is a two-element vector. The first element is the tie number of the appropriate file. The second element is an integer specifying both the number of components to be dropped, and from which end of the file to drop them (shown by the sign of *n*). If *n* is positive, the specified number of components are removed from the front of the file; if *n* is negative, the components are removed from the end of the file; if *n* is zero, no components are removed. The component numbers of the remaining components are not changed. For example, if the file tied to 99 has ten components numbered 1 through 10, then after executing:

```
⎕FDROP 99 4
```

the file will have six components remaining, numbered `5`, `6`, `7`, `8`, `9`, and `10`. A further execution of:

```
⎕FDROP 99 ¯2
```

will leave components `5`, `6`, `7`, and `8`.

Components cannot be dropped from the interior of a file. Other techniques can be used to signify that an interior component holds no information and should be bypassed in later processing. Perhaps the simplest way is to replace the component with an empty vector (' ').

There is no native file operation that corresponds to ⎕*FDROP*, since native files are not organized into components.

Determining the Size of a File

The function ⎕*FSIZE*, with syntax :

```
     result ← ⎕FSIZE tieno
```

returns a four-element vector. The first two elements are the component limits of the file: the number of the first component, and a number that is 1 higher than the number of the last component. The component limits of a newly created file are 1 1.

A count of the number of components in the file is given by the following expression where *tn* is the tie number:

```
     |-/2↑⎕FSIZE tn
```

The third and fourth elements of the result of ⎕*FSIZE* are the amount of file storage currently occupied by the file, and the file storage limit, in bytes. If the fourth element is zero, the file has no imposed size limit other than available space on the disk.

The corresponding native file operation, ⎕*NSIZE*, returns a single number representing the number of bytes in use. Since native files have neither components nor automatic checking for maximum size, there can be no meaningful equivalents for the other numbers in ⎕*FSIZE*.

Suppose the result of ⎕*FSIZE* for the file tied to 10 is as follows:

```
      ⎕FSIZE 10
1 126 259584 265000
```

The result of ⎕*FSIZE* indicates that the components in the file are numbered 1 through 125, inclusive, and that the size limit for the file is 265,000 bytes, of which a total of 259,584 bytes are now occupied.

Following are examples of how to implement two common file organizations -- "first-in, first-out" (FIFO) and "last-in, first-out" (LIFO) -- using ⎕*FDROP* and ⎕*FSIZE*. The examples use a file tied to 50.

FIFO organization:

Data collection:	*data* ⎕*FAPPEND* 50 *SIZE* ← ⎕*FSIZE* 50
Subsequent processing:	*INFO* ← ⎕*FREAD* 50,*SIZE*[1] (process info) ⎕*FDROP* 50 1

LIFO organization:

Data collection:	*data* ⎕*FAPPEND* 50 *SIZE* ← ⎕*FSIZE* 50
Subsequent processing:	*INFO* ← ⎕*FREAD* 50,*SIZE*[2]-1 (process info) ⎕*FDROP* 50 ¯1

With each of these schemes, the data collection and processing phases can be executed when it is convenient to do so. You can collect data whenever information is available and process data when the file becomes large and unwieldy or at some fixed interval; once a day or once a week, for example.

After a component is processed, the storage it occupies is generally released by executing ⎕*FDROP*. In some cases, ⎕*FDUP* is needed to reclaim all dropped storage; see Chapter 3 of the *APL*PLUS System*

Reference Manual for more information. Consequently, data can be collected and processed almost indefinitely and never require more than a relatively small amount of storage space.

As an example of using ⎕*FDROP* with a file whose components are released in an arbitrary sequence, the following function produces an invoice from information in a file named *CDATA*. After the invoice is produced, the appropriate component of *CDATA* is dropped (if it is the first component) or replaced with an empty component (if it is not the first component). Each time it is used, the function checks to see if the last component of data is an empty vector; if it is, that component is dropped. This technique tends to minimize the file size.

```
     ∇ SAMPLE4;LIM;N
[1]   'CDATA' ⎕FTIE 5
[2]   'ENTER RECORD NUMBER' ◊ N←⎕
[3]    ⍝ TEST COMPONENT NUMBER:
[4]   LIM←2↑⎕FSIZE 5◊→((N<LIM[1])∨N≥LIM[2])/ERROR
[5]    ⍝ READ RECORD AND PRODUCE INVOICE:
[6]   RECORD←⎕FREAD 5,N ◊ →(0=⍴RECORD)/ERROR
[7]   PRINTINVOICE RECORD
[8]   '' ⎕FREPLACE 5,N
[9]    ⍝ DROP ANY EMPTY COMPONENTS
[10]  ⍝ FROM FRONT OF FILE
[11]  LIM←2↑⎕FSIZE 5
[12]  LOOP:→(LIM[1]=LIM[2])/END⍝TEST FOR EMPTY FILE
[13]   ⍝ TEST FOR EMPTY COMPONENT
[14]  →(0≠⍴,⎕FREAD 5,LIM[2]-1)/END
[15]   ⍝ DROP EMPTY COMPONENT
[16]  ⎕FDROP 5 ¯1 ◊ LIM[2]←LIM[2]-1 ◊ →LOOP
[17]   ⍝
[18]  ERROR:'THIS RECORD NUMBER NOT IN FILE'
[19]  END:⎕FUNTIE 5
     ∇
```

In line `[4]` of *SAMPLE4*, a branch is taken to error if *N* is not within the range of the lowest and highest component numbers. Otherwise, line `[6]` reads the component, branching to error if the component is empty. Line `[7]` calls the function *PRINTINVOICE* (not shown) which actually produces an invoice based on the record. Lines `[11]` through `[14]` drop empty components, if any, from the file. Note the checking for an empty file in line `[12]`.

The above discussion presents a limited representation of possible file organization techniques. There are more static forms of database file organization that subdivide the data and keep directories. Also, there is a widespread convention of reserving the first component for a description of the file.

Renaming a File

The function ⎕*FRENAME* is used to change an APL file name. The syntax is

'*fileid*' ⎕*FRENAME tieno*

⎕*FRENAME* can be used to change the name of a file in a library:

```
      'REPORTS' ⎕FTIE 100
      'OLDREPORTS' ⎕FRENAME 100
      ⎕FUNTIE 100
```

Note that ⎕*FRENAME* does not create a second copy of a file. After the execution of the previous example, *REPORTS* would have disappeared from the library.

In addition to changing the name, ⎕*FRENAME* also sets the ownership of the file to match the user account number under which the operation was performed.

The system function ⎕*NRENAME* is used for renaming native files and for moving a native file into a different directory.

Changing a File's Size Limit

The function ⎕*FRESIZE* can impose a size limit on an APL file beyond which the file is not permitted to grow. For example, if you did not want a limit when you first created the file but have changed your mind, use ⎕*FRESIZE*.

Sometimes a file needs to hold more data than its current size limit will allow. When a file does not have enough room for the value to be stored in it, a *FILE FULL* error occurs:

```
      'OLDREPORTS' ⎕FTIE 12
      REPORT ⎕FAPPEND 12
FILE FULL
      REPORT ⎕FAPPEND 12
      ^
```

The fourth element of the result of ⎕*FSIZE* shows that the size limit for the file *OLDREPORTS* is 100,000:

```
      ⎕FSIZE 12
1 34 99512 100000
```

The function ⎕*FRESIZE* can be used to increase the file size limit so that there will be enough room to perform the desired append operation:

```
      200000 ⎕FRESIZE 12
      ⎕FSIZE 12
1 34 99512 200000
      REPORT ⎕FAPPEND 12
34
```

The syntax of ⎕*FRESIZE* is:

newsize ⎕*FRESIZE* *tieno*

where *newsize* is the new file size limit. It is also possible to decrease the size limit of a file, provided you do not specify less storage than is already being used by the data in the file. In addition, you can always specify:

```
      0 ⎕FRESIZE 12
      ⎕FSIZE 12
1 34 99512 0
```

to remove the size limit restriction completely.

There is no system function for native files that corresponds to ⎕*FRESIZE*, since native files do not have size limits.

Copying a File

The function ⎕*FDUP* is used to make a duplicate copy of the contents of an APL file in another APL file. The copy need not be in the same library as the file being copied, and the name need not be the same (although it can be if the library number is different).

```
'5440 FIGHT' ⎕FTIE 48
'1844 OREGON' ⎕FDUP 48
⎕FUNTIE 48
```

The ownership of the original file is unchanged by ⎕*FDUP*, but the user account number that used ⎕*FDUP* is the owner of the copy.

There is no need to untie the new copy (`'1844 OREGON'`) since it was never tied. If you were to tie the new copy and compare its size (using ⎕*FSIZE*) to the size of the original file, you might find that the new file was smaller, since ⎕*FDUP* eliminates any wasted space while copying (see "Compacting a File", below).

There is no system function for duplicating native files, but you can use the VMS command "`copy`", either from VMS or with)*CMD*, or under program control with ⎕*CMD*. This method can also be used to copy any file, including APL files and saved workspaces, without compacting them.

To move a file from one library to another, use ⎕*FRENAME* for APL files and ⎕*NRENAME* for native files.

Compacting a File

The function ⎕*FDUP* can also be used to compact the contents of an APL file. First, make a new compacted copy using ⎕*FDUP*. If you want the compacted file to have the same name as befor, first use ⎕*FDUP* to create a new file with a different name. Then erase the original, rename the new compact copy to the original file name, and tie the compacted file to the same tie number that the original file was using.

```
'5440 FIGHT' ⎕FTIE 1844
'TEMP' ⎕FDUP 1844
'5440 FIGHT' ⎕FERASE 1844
'TEMP' ⎕FTIE 1844
```

```
'5440 FIGHT' ⎕FRENAME 1844
⎕FUNTIE 1844
```

The user account number that used ⎕FDUP to compact the file becomes the owner of the file.

Since storage space is limited, it may be necessary periodically to reclaim file space containing abandoned data. The data may have been abandoned by the use of ⎕FDROP or ⎕FREPLACE with a different amount of data. The APL⋆PLUS System does not automatically release for reuse all the space occupied by abandoned data within APL files. It is the user's responsibility to reclaim storage space as needed by using ⎕FDUP.

There is no system function for compacting native files, since the system does no tracking of their contents.

3-3 File Sharing — Concepts and Functions

The shared use of a file falls into one of two general categories. The first, discussed in Sections 3-1 and 3-2, is that in which only one person at a time may use a file (that is, have it tied). The second category of sharing, discussed in Sections 3-3 and 3-4, involves the concurrent use of a file by two or more people; rather like the use of several telephone receivers on one telephone number, which may require some protocol among speakers to prevent conflicts. Section 3-4, describes some mechanisms for the detailed control of concurrent file sharing.

The File Access Matrix

Associated with every APL file is an access matrix that records which user account numbers are authorized to use the file and which file operations each can perform.

Every access matrix is an integer-valued matrix with three columns and any number of rows:

- Column 1 is the user number to which access is granted.
- Column 2 indicates which operations a user is authorized to perform.
- Column 3 is a passnumber to the file.

The following table shows typical entries in an access matrix. The details of columns 2 and 3 are of interest only when exercising detailed control over access; they are discussed in Section 3-4.

Account Number	Encoding of Permitted Operations	Passnumber	Description
12345	¯1	0	Access granted to user 12345.
23456	9	5	Access granted to user 23456.
0	1	0	Access granted to all other users.

A user account number in column 1 can be that of any user, or it can be zero. Zero represents all user account numbers, except the file owner's, that do not appear elsewhere in column 1 of the access matrix.

When ⎕*FTIE* or ⎕*FSTIE* (a variant of ⎕*FTIE* described later in this chapter) is executed, the access matrix for the file being tied is searched for a row containing the user account number. If a row is found whose first element matches the user account number or failing that, a row is found whose first element is zero, then the authorized forms of access are taken from the second element of that row. Any attempt to use a file operation that is not authorized will result in a file access error.

The owner of a file has complete access to the file if column 1 of the access matrix does not contain his user account number. No one else has any access at all unless there is a match with his user account number or a zero in column 1. If a person with no access attempts to tie the file, the system will produce a file access error.

Functions like ⎕*FNAMES* and ⎕*FNUMS* are not restricted by access matrix settings and never produce a file access error because they are not performed on a particular file. Also, there are no limitations on ⎕*FSIZE* or ⎕*FSTIE*; these are permitted if any other file operation is authorized.

The functions ⎕*FRDAC* and ⎕*FSTAC* (described in Section 3-4) can be used to manipulate the access matrix.

File Component Information

Along with the value stored in a component by appending or replacing, four other pieces of information are carried with each component of a file. They are:

- the workspace storage needed to hold the component's value
- the user account number of the person who last replaced or appended the component
- the time that the component was last replaced or appended in a packed-timestamp form
- the component timestamp in a seven-element unpacked form (year, month, date, hour, minute, second, millisecond).

The function ⎕*FRDCI* reads this component information; its syntax is:

result ← ⎕*FRDCI tieno compno*

The result of ⎕*FRDCI* is a ten-element numeric vector holding the information described above. The packed timestamp in the third element of the result is given in microseconds since 00:00, 1 January 1900; under program control, it is convenient for comparing the timestamps of two components.

The function ⎕*FRDCI* is particularly useful in data collection or audit trail applications, where several persons may have access to append to a file. As each person appends, the component is automatically tagged with the time and the person's user account number. Later, when a user with ⎕*FREAD* and ⎕*FRDCI* authorization processes the file, the source of each component is clearly indicated.

Concurrent Sharing of Files

The previous discussion covered the first category of shared file use, in which only one person at a time is using a file. The use of ⎕*FTIE* to establish a file tie assures this kind of exclusive file use, since ⎕*FTIE* will produce a file tied error if any other user tries to tie the file. Thus, ⎕*FTIE* provides you with exclusive use of a file until you untie it.

The second category of file use covers situations in which several users may have the same file tied concurrently. Two file functions are needed for concurrent file use: ⎕*FSTIE* and ⎕*FHOLD*.

The function ⎕*FSTIE*, with syntax :

```
'fileid' ⎕FSTIE tieno
```

establishes a tie to a file, exactly as ⎕*FTIE* does. However, this "shared tie" also permits others to share-tie the same file. ⎕*FSTIE* can be used to share-tie a file if the file's access matrix allows the user any form of access whatever, provided no one has the file exclusively tied using ⎕*FTIE*.

When several persons are using a file concurrently, their file operations proceed asynchronously, and it is quite likely that one person's processing may be interleaved with another's. For example, suppose that users P and Q have the file tied to 77 and are trying to add 1 to the value of component 5; they use the following statement:

```
(1+⎕FREAD 77 5) ⎕FREPLACE 77 5
```

When both are finished, component 5 should be increased by 2. But suppose the parts of the statement are executed in this sequence:

```
P                        Q

                         ⎕FREAD 77 5
⎕FREAD 77 5
                         (add 1)
                         ⎕FREPLACE 77 5

(add 1)
⎕FREPLACE 77 5
```

The value of component 5 is increased by only 1, since P's program destroyed the value stored by Q.

What is needed here is an interlock to prevent P's program from executing any part of the foregoing APL statement while Q's program is executing it (and vice versa). The function ⎕*FHOLD* provides this interlock. Its syntax is :

    ⎕*FHOLD tienos*

The effect of executing ⎕*FHOLD* is to place an interlock on each file whose tie number is included in the right argument. The concept of this interlock is subtle; in effect, using ⎕*FHOLD* means "wait until no one else has these files held, then hold them for me."

⎕*FHOLD* executed by P, for example, has three effects:

- Any interlocks in effect from a previous ⎕*FHOLD* executed by P are released (even if they held the same file).
- P is placed in a queue behind every other user who has already executed ⎕*FHOLD* for any of the files specified by P. P's program is delayed until no one else is holding any of these files.
- Interlocks are set simultaneously on all the designated files, and P's program then resumes execution.

Many people misunderstand file holds when they are first learning about APL files. They assume that "holding a file" means they are preventing others from using the file while they have it held. The only effect of a file hold is to delay the execution of ⎕*FHOLD* in other users' programs. In other words, no two users can have the same file held at the same time. File holds do not block other file operations such as ⎕*FREAD* or ⎕*FREPLACE*.

⎕*FHOLD* does not prevent other users from using a file while you have it held. But ⎕*FHOLD* does provide a means for two or more users to cooperate and avoid conflict in file use. For example, suppose the program executed by P and Q in the previous example is changed to:

```
⎕FHOLD 77
(1+⎕FREAD 77 5) ⎕FREPLACE 77 5
⎕FHOLD ⍳0
```

Then the first user to execute `⎕FHOLD 77` is able to proceed without delay, while the other user's program is delayed until the first user's program executes `⎕FHOLD ⍳0`. The sequence would look like this:

```
P                          Q

                           ⎕FHOLD 77
⎕FHOLD 77
(delay)                    (proceed)
                           ⎕FREAD 77 5
                           (add 1)
                           ⎕FREPLACE 77 5
                           ⎕FHOLD ⍳0
(proceed)                  (proceed)
⎕FREAD 77 5
(add 1)
⎕FREPLACE 77 5
⎕FHOLD ⍳0
(proceed)
```

When Q executes `⎕FHOLD 77`, he establishes a hold on the file that prevents P from establishing a hold. In P's program, `⎕FHOLD` simply waits until its turn comes to hold the tie, which happens when Q executes `⎕FHOLD ⍳0`. P is then able to hold the file and execution proceeds. This makes it possible for cooperating users to avoid conflict.

However, if Q does not use `⎕FHOLD` properly, then there is nothing P can do alone to prevent conflict. Suppose P is using `⎕FHOLD`:

```
⎕FHOLD 77
(1+⎕FREAD 77 5) ⎕FREPLACE 77 5
⎕FHOLD ⍳0
```

And Q is **not** using `⎕FHOLD`:

```
(1+⎕FREAD 77 5) ⎕FREPLACE 77 5
```

The interaction could then be just as if P were not using ⎕*FHOLD* at all:

P	Q
⎕*FHOLD* 77	
(proceed)	⎕*FREAD* 77 5
⎕*FREAD* 77 5	(add 1)
(add 1)	⎕*FREPLACE* 77 5
⎕*FREPLACE* 77 5	
⎕*FHOLD* ⍳0	

Only when all parties cooperate can conflicts be avoided. For this reason, it is often wise to require the use of a specific function to access a file -- one that obeys the desired protocol. File passnumbers (discussed in Section 3-4) provide a means of enforcing the use of a given protocol.

Here are some situations that do or do not require file holds in order to work properly:

- Several users are concurrently appending to a file but make no other use of the file. File holds are not needed since ⎕*FAPPEND* will keep track of components added to the file by number. Although it is not possible to predict in what order the various values will appear in the file, any request for a file operation (in this case ⎕*FAPPEND*) always waits until a previous operation on the same file is complete. This statement is being used:

  mycomp ← value ⎕*FAPPEND tn*

  The result of ⎕*FAPPEND* lets each person know which component contains his value.

- Several users are reading and replacing components of a file, and it is known that no two users ever reference the same component. For example, P's program refers only to component 1, Q's program only to component 2, and so on. No hold is needed, since no conflict can occur on the concurrent use of a single component.

- An application involves the use of three files in which like-numbered components contain related data. One program is updating the files while the other programs are reading the files concurrently. To ensure that no program reading from the files will encounter a mixture of old and new data, the updating program has this appearance:

```
⎕FHOLD 21 22 2  (process and replace component n
                 of each of the three files)
⎕FHOLD ⍳0
```

and the file-reading programs that are operating concurrently have this appearance:

```
⎕FHOLD 21 22 23
A ← ⎕FREAD 21,N
B ← ⎕FREAD 22,N
C ← ⎕FREAD 23,N
⎕FHOLD ⍳0
```

This is a situation where file holds are needed in a program that is itself not modifying the contents of any files.

As these examples show, the need for file holds depends upon the interrelation of program and file structure. The design of any application involving concurrent use of files requires careful analysis for possible "races" between programs. You can resolve such conflicts with appropriate use of *⎕FHOLD*.

Duration of File Holds

All interlocks are released when the user who set them executes another *⎕FHOLD*, signs off, or enters immediate execution mode. The interlock on a single file can be released by untying it or retying it. Note that *⎕FHOLD* ⍳0 releases all interlocks.

The immediate execution case is particularly important to remember. If you type the following three statements as three different immediate execution inputs:

```
⎕FHOLD 1
PROCESS 1
⎕FHOLD ⍳0
```

the ⎕*FHOLD* has no effect at all. However, a compound statement like the following will work correctly, since there is no immediate execution input between ⎕*FHOLD* and process:

```
⎕FHOLD 1 ◊ PROCESS 1 ◊ ⎕FHOLD ⍳0
```

3-4 Detailed Control of File Sharing

This section covers some mechanisms for limiting file access to particular parts of a file or to specific sequences of file operations, primarily for designers of large-scale, shared-file applications. Most readers will find the facilities described in Section 3-3 adequate for their needs. Section 3-5 contains a more detailed discussion of the access matrix.

For simplicity in the examples that follow, the syntax of APL file operations does not include the optional passnumber argument.

Setting the Access Matrix

The value of a file's access matrix is set using the function ⎕*FSTAC*. The syntax is :

matrix ⎕*FSTAC* *tieno*

The left argument is an integer-valued, three-column matrix. A three-element integer vector is reshaped to become a one-row matrix. It replaces the previous value of the access matrix. Initially, when a file is created, its access matrix has no rows: its shape is `0 3`. The file owner has complete access since his user account number does not appear in column 1 and no one else has any access for the same reason.

Reading the Access Matrix

The access matrix is read into the active workspace using the function ⎕*FRDAC*. The syntax is:

result ← ⎕*FRDAC* *tieno*

The result is the access matrix for the file tied to the number in the right argument. The second element in a row of the access matrix is the sum of access code values for the particular file operations that are allowed to the user account number in the first position of the row. The access code values are :

Code	Operation
1	⎕*FREAD*
2	⎕*FTIE*
4	⎕*FERASE*
8	⎕*FAPPEND*
16	⎕*FREPLACE*
32	⎕*FDROP*
64	(Not used.)
128	⎕*FRENAME*
256	(Not used.)
512	⎕*FRDCI*
1024	⎕*FRESIZE*
2048	⎕*FHOLD*
4096	⎕*FRDAC*
8192	⎕*FSTAC*
16384	⎕*FDUP*
32768	(Not used.)
65536	(Not used, represents ⎕*FSTATUS* on other APL⋆PLUS Systems.)

The sum of all possible access codes can be used to authorize all possible file operations. A ¯1 can also be used to grant full access.

A value of 9 indicates authorization to execute the functions ⎕*FREAD* and ⎕*FAPPEND* on the file. Any non-zero value permits use of ⎕*FSIZE* and ⎕*FSTIE*; obviously, not all combinations of values make sense.

Using Passnumbers

The third element in a row of the access matrix, the passnumber, is usually zero. Nonzero passnumbers are used only to exercise detailed control over file access. If the third element contains a passnumber other than zero, the user whose user account number is on that row must provide a matching passnumber to operate on a file.

Omitting a passnumber from an argument (as in all uses of file functions in Sections 3-2 through 3-4) is equivalent to providing an explicit passnumber of zero. A mismatching passnumber causes a *FILE ACCESS ERROR*.

Passnumbers are intended for use within locked APL functions that the application programmer gives to users in place of the standard file functions. Suppose each component of the personnel file `2345 PERS` is a vector holding an employee's telephone extension and room number followed by salary information. The user with user account number `9876` is to be allowed to retrieve the telephone and room number but not the salary information.

For example, the programmer chooses a passnumber, `10349`, and defines these locked functions:

```
      PTIE
[1]   '2345 PERS' ⎕FSTIE 25 10349

      R←PREAD N
[1]   R←2↑⎕FREAD 25,N,10349
```

Next, he sets the access matrix for `2345 PERS` to authorize access by user `9876`. The row of the access matrix will be

```
          9876  1  10349
```

Finally, the programmer gives the locked functions to user `9876`, but does not tell him what the passnumber is. Now `9876` has permission to tie and read from the file, but only if he gives the passnumber with each `⎕FSTIE` or `⎕FREAD`. Since he does not know the passnumber, he can access file `2345 PERS` only through the functions *PTIE* and *PREAD*.

Using this general technique, it is possible to impose complex restrictions on file authorization; in fact, you can impose any sort of restriction that can be stated as an APL function. Since different passnumbers can be imposed on different user account numbers, it is easy to provide multiple levels of access authorization to confidential data. The previous example showed access restricted to specific fields within a component. Examples of other forms of control are :

- access only to even-numbered components:

```
[1]  ⎕ERROR (0≠2|N)/'EVEN NUMBERED CNS ONLY'
[2]  R←⎕FREAD 25,N,32049
```

- access only during the afternoon:

```
[1]  ⎕ERROR(~⎕TS[4]∊12 13 14 15 16 17)/'PM ONLY'
[2]  R←⎕FREAD 25,N,32049
```

- automatic logging of information requests:

```
[1]  A←N ⎕FAPPEND 99 2888 ⍝ LOG TO FILE 99
[2]  R←⎕FREAD 25,(N),32049
```

Later use of ⎕*FREAD* and ⎕*FRDCI* on the file tied to 99 will give the value of *n*, the timestamp, and the requestor's user account number.

- access only after validating data:

```
[1]  →(1≠ρρV)/ERR ◇ →((20≠ρV)∨0∨.>V)/ERR
[2]  R←V ⎕FAPPEND 45 14149
```

- access only after verifying user identity by questions and answers (to protect a momentarily unattended computer from passersby).

- access only to summary data (for instance, salary averages but no individual salaries).

3-5 The Access Matrix

All APL files have access matrices that regulate which users can use them and which operations each user can perform on them. The rules applicable to an access matrix are as follows:

- Every access matrix is an integer matrix with three columns and any number of rows. Each row in the access matrix represents the authorization granted for a single user or class of users. The first column of the matrix contains a user number; the second column contains an encoding of the corresponding authorized operations, and the third column contains a passnumber.

- The user numbers in column 1 can be those of any users. A zero value in column 1 refers to all users other than the owner or those explicitly specified elsewhere in column 1.

  The owner has full implicit access only if his user number does not explicitly appear in column 1 of the access matrix. If the owner does appear explicitly in the access matrix, then he is treated just like any other user.

- The value in column 2 explicitly indicates which operations the user is authorized to perform. Each operation subject to access control is associated with a value, called the access code, that is a power of 2. The sum of these access codes is the nominal value in column 2.

  Formally, the value in column 2 is the integer representation of a Boolean mask that has a bit for each controllable operation. If

  ```
  MASK ←→ (32⍴2)⊤VALUE
  ```

  then *MASK*[32-*N*] (origin 1) is the bit regulating the operation whose access code is 2⋆*N*. If the bit is 1, the user is authorized for the operation. Many bit positions are not associated with any operations; the value of these bits is immaterial. Thus, ¯1 grants authorization for every operation, since (32⍴2)⊤¯1 ←→ 32⍴1.

  This last property is often used to grant all but certain kinds of access to a file, workspace, or library. The technique is to subtract from ¯1 the access codes for the operations to be denied. For example, ¯5 grants all but ⎕*FERASE* access (¯1-4), and ¯7 grants all but ⎕*FERASE* and ⎕*FTIE*(¯1-(4+2)).

- The passnumber in column 3 can be any positive or negative integer value. Together with the user number in column 1, the passnumber determines which row of the access matrix is to be applied in verifying authorization for each operation. A single user or class of users can have multiple rows in the access matrix, with different privileges granted with different passnumbers.

Example of an Access Matrix

Suppose you want to set up a file to which all users have only ⎕*FREAD* access, and you want to be able to test the application that accesses the file and obtain the same permission as another user. You would then set up an access matrix like this:

0	1	0	⎕*FREAD*-only access to all but owner.
owner	1	0	⎕*FREAD*-only access to owner.
owner	¯1	1234	Full access with passnumber to owner so he can perform other file operations. Note that if the owner forgets the passnumber, he will be unable to do anything except a ⎕*FREAD* unless he uses)*FILEHELPER* on the file.

Importance of the File Passnumber

The passnumber used to tie a file remains associated with the file tie. After you have tied a file, you must supply the same passnumber in all subsequent operations on the file as you supplied when the file was tied. Use of another passnumber will always produce a *FILE ACCESS ERROR*, even if some row of the access matrix happens to grant the appropriate permission with that passnumber. To use the permission granted by a different row of the access matrix, you must re-tie the file using that row's passnumber.

Changing the Access Matrix When a File is Tied

In the APL⋆PLUS System, access to an APL file is determined only when the file is tied. Consequently, changing the access matrix of an APL file while it is tied has no immediate effect on access. The new value of the access matrix will be used when the other user tries to tie the file.

Similarly, if a user changes the access matrix of a file in a way that limits or otherwise changes his own access to the file, the change does not affect his access until the next time he re-ties the file. Therefore, a change to prevent accidental erasure by the owner of the file gives no protection until he unties or re-ties the file.

Overriding an Access Matrix

Under the rules of access control, it is possible to be locked out of one of your own files. Since the ability to set the access matrix is one of the operations governed by the access matrix, you may be unable to correct the problem with *⎕FSTAC* alone.

Facilities are available to assist the file owner who is locked out by the access matrix. The system command *)FILEHELPER* enables the owner to access the file and change its access matrix. The syntax is:

```
)FILEHELPER filename
```

where *filename* is a valid APL file identifier. See Chapter 2 of the *APL*PLUS System Reference Manual* for details on the use of *)FILEHELPER*.

)FILEHELPER grants the owner full explicit access with no passnumber. The old access matrix for the file is discarded. You can then reset the matrix to an appropriate value.

3-6 APL Libraries and VMS Directories

APL has traditionally grouped files and workspaces together in collections known as libraries, each identified by a library number. The VMS operating system organizes all of the files in the system by grouping them into directories and subdirectories. This chapter explains how the APL⋆PLUS System for VAX/VMS relates the VMS directories to APL libraries.

VMS Directories as Libraries

A VMS directory can be made to appear like a traditional APL library by giving it a library number. Library numbers are associated with directories when the APL session begins according to the contents of the configuration file specified when APL is invoked. A line of the

APL*n*.INIT causes the library number to be associated with the specific path. For example, the two-line file:

```
library=11[apl.tools]
library=9123[stuart]
```

associates library number 11 with directory [APL.TOOLS] and library number 9123 with [STUART]. A command of the form:

```
)LOAD 11 WSDOC
```

would then cause the workspace saved as [APL.TOOLS]WSDOC.WS to be loaded.

Once the APL session is started, library definitions can be added using the system function ⎕*LIBD*. For example:

```
        ⎕LIBD '244 [HARRIET]'
```

associates library number 244 with directory [harriet].

The Default Directory and Default Library

When APL is in library mode and a file identification omits the library number, a default value for the library number is the user's VMS account number (1↑⎕*AI*). The default directory assumed is the current working directory in the VMS environment, as is set with the DCL command SET DEFAULT. At session start, the working directory for the APL session is inherited from the working directory in effect when APL is invoked. During the APL session, the current working directory can be changed using the system function ⎕*CHDIR*.

```
        CHDIR '[TEST.SCRIPTS]'
[VICKI]
```

changes the current working directory to [TEST.SCRIPTS]. The explicit result contains the name of the former default directory. From this point on, the library number that matches (1↑⎕*AI*) refers to that directory.

You can use the system command)*LIBS* (see Chapter 2 of the *APL*PLUS System Reference Manual*) or the system function ⎕*LIBS* (see Chapter 3 of the *APL*PLUS System Reference Manual*) to display the library assignments in use for this session. You can distinguish whether the APL session is in path mode or library mode with ⎕*LIBS*, which returns a zero-row result in path mode.

3-7 Comparison of APL and Native Files

You should compare the APL component files and the native files available through the APL*PLUS System in some detail. The APL files provide more automatic housekeeping and control and greater convenience when making changes. The native files permit an easy interface with non-APL systems such as word processors, since these are the files that non-APL programs use.

The native files created by ⎕*NCREATE* are sequential Stream_LF files of the same type used by the VAX C compiler's run-time library.

Files Are a Sequence of Stored Data Items

Both APL files and native files can be viewed as a simple sequence of stored data items. They differ as follows:

- The APL file is a sequence of APL arrays. Each array, independent of the others, can be of any datatype, of any rank, of any shape, and of any size. One can be a table of decimal numbers, while the next can be a four-page memo. Regardless of the nature or size of the array, it is referred to by a single component number and can be retrieved by that number. When a component is retrieved, the array's internal organization (the number of bytes per element and the interpretation of the arrangement of the bits, known as datatype) and external organization (shape) are recognized and handled automatically.

- The native file, by is a sequence of bytes (one character's worth of data). How those bytes are organized or what they represent is determined entirely by the programmer. This represents maximum flexibility at the cost of maximum programming effort. The program performing the retrieval must deal with where to start reading the material, how far to read before reaching the other end, whether to convert to numeric form (and which numeric form), and whether to reshape the data and how.

Space Reservation and Checking

An APL file can be given a size limit. Each time you attempt to add or replace material in the file, the system automatically checks whether the operation would cause the file to exceed the limit. If the limit would be exceeded, the operation is not performed. A `FILE FULL` is signaled, and the programmer (or the program) must deal with that condition before continuing. This permits budgeting of disk space and recognition of "runaway" programs.

There is no such size checking in writing to a native file.

Ownership and Access Control

The APL⋆PLUS System tracks the user account number that created or last renamed the file (its owner). Through the use of the access matrix, the file owner and those individuals he specifies can extend or limit the types of file operations that can be performed by any given user account number. The use of non-zero passnumbers in the access matrix also makes it possible to limit the allowed operations. Usage can be restricted to locked APL functions or special kinds of ties.

Access to native VMS files is controlled by a less specific mechanism. Each native file has a set of "permissions" that determine how it can be used and by whom. The three permissions are:

- *read* permission -- the privilege to read any data that the file contains

- *write* permission -- the privilege to modify the file or erase it completely

- *execute* permission -- the right to attempt to load and execute the file as an executable binary program.

Rather than track distinct permissions for each user in an access matrix, the VMS operating system recognizes three classes of users:

- the owner
- the members of the owner's group (a concept that is not used in APL)
- everyone else.

The VMS operating system tracks each of the three classes of permission for each of the three classes of users. The system functions ⎕NRDAC and ⎕NSTAC can be used to read and modify native file permissions. See the detailed description of these features in Chapter 3 of the *APL *PLUS System Reference Manual*.

Replacement of Data

You can replace one component in an APL file with any other APL array using ⎕FREPLACE. You need not match the physical size in bytes, the datatype, or the shape of the array being replaced. In a file with 20 components, you could replace the integer in component 7 by a huge table of numbers without stopping to ask how much room on the file the integer occupied:

```
      ASTRO ⎕FREPLACE 5 7
```

Consequently, it is extremely easy to replace an object with an enlarged (updated) version of itself; for example, a customer list with one or more new customers added.

In a native file, you can only replace data byte-for-byte. You (or the program) must ask how large the amount of data to be put into the file is, and how that compares with the amount already there. Other questions you should ask include: Where does the data that is already there begin? What information currently in that space will be destroyed by a simple byte-for-byte replacement? Will space be wasted? Is more space needed than is used by the data to be replaced? If so, where else can it be put, or how can a larger version of this file (one that has more room for the replacement) be created?

Clearly, an updated list or otherwise enlarged object cannot simply replace the earlier version on the file. To preserve the same relative position would entail copying the rest of the file with the new data into a new file. To avoid copying of all the data, you must set up some sort of directory system to track the data. This can include currently unused space abandoned as data grew. If unused space is not tracked, it cannot be reclaimed.

If an APL file is cramped for space, reclaiming wasted space is simple. The system function ⎕*FDUP* uses the internal component-tracking data to compact the file down to only its useful data. Using ⎕*FDUP* requires that enough space be available on the disk to hold all the compacted data.

Since native files do not predictably include such tracking information, no compression of space wasted in native files can be provided by the system; it must be provided by the programmer.

Timestamp Information

For each component, an APL file stores not only the data, its shape and datatype, but also (retrievable separately with ⎕*FRDCI*) three other pieces of information not tracked by native files:

- how big it is (the number of bytes of workspace storage that would be needed to read the array)
- who stored it there (the user account number that put this array in the file)
- when it was stored (the date and time encoded into a single number called a timestamp).

The syntax of ⎕*FRDCI* is:

```
result ← ⎕FRDCI  tieno compno
```

Different Naming Conventions

The names of APL files are restricted to combinations of alphabetic and numeric characters, while the names of native files can contain other characters permitted by VMS naming conventions (see your VMS operating system manual) and are not restricted to names beginning with a letter.

FORMATTING

FORMATTING

Chapter 4
Formatting Data in the APL★PLUS System

This chapter describes the formatting capabilities of the APL★PLUS System function *⎕FMT* and identifies the formatting functions available in the workspace *FORMAT*.

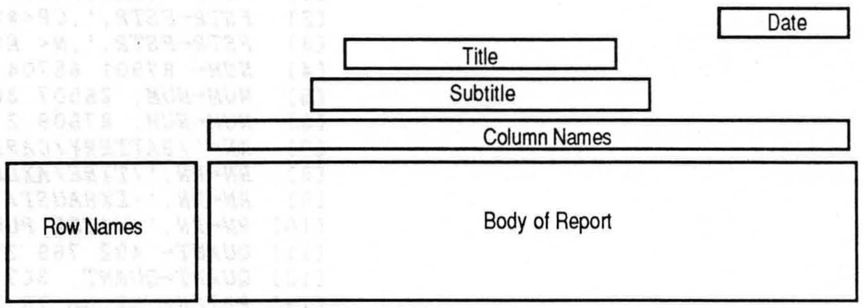
4-1 Designing A Report

The first step in generating a report is designing it. Consider several factors before you format the data for your report:

- the order of the columns
- the representation of the columns (integer, decimal, pattern, exponential)
- the width of the columns
- the width of the entire report.

You also must decide what kinds of decorations you want to use to clarify the meaning of the data. For example, you may want to include dollar signs, commas, negative signs, and so on. In some cases, a pattern may be the best way to represent to data; for example, a telephone number can be displayed with parentheses around the area code. The diagram below shows what the structure of a typical report might look like.

Design of a Typical Report



Once you have decided on the basic layout of your report, use the functions described in Section 4-14 to add and position titles and labels. The following report is an example of what a typical report might look like using the structure shown in Exhibit 4-1.

```
                    HARRIS GARAGE
           EASTERN DIVISION INVENTORY REPORT

PART NAME  NUMBER QUAN   PRICE      VALUE     REORDER <6MO.

BATTERY    879-01  492  $92.85  $45,682.20 12/13/87
CARBURETOR 657-04  769  $73.23  $56,313.87  5/06/87    R
FUEL TANK  876-03  371  $71.80  $26,637.80  6/24/87
WHEEL      234-06  287  $41.75  $11,982.25  4/12/87    R
BATTERY    876-07  381  $96.45  $36,747.45  9/03/87
TIRE       876-02   98  $60.90   $5,968.20  4/25/87    R
AXLE       265-07  205  $55.85  $11,449.25 11/13/87
TIRE       361-08  387  $66.95  $25,909.65  9/26/87
BRAKE      876-06  201  $32.00   $6,432.00  3/01/87    R
CARBURETOR 876-04  879 $157.80 $138,706.20  6/25/87
TIRE       234-01  298  $68.90  $20,532.20  2/11/87    R
EXHAUST    876-05  367 $354.00 $126,615.00  6/25/87
IGNITION   876-09  652  $22.50  $14,670.00  3/12/87    R
SPARK PLUG 273-03  391   $2.85   $1,114.35  8/05/87
RADIATOR   872-05  738  $63.80  $47,084.40  2/28/87    R
WATER PUMP 251-09  276  $53.78  $14,843.28  9/08/87
ALTERNATOR 729-07  493  $96.70  $47,673.10  7/14/87
RADIATOR   316-02  387  $69.30  $26,819.10 10/26/87
COIL       582-08  492  $25.50  $12,546.00  1/21/87    R
```

This report was produced with the following program, called *INVENTORY*. The program uses the system function ⎕*FMT* and several functions from the *FORMAT* workspace.

```
    ∇ INVENTORY;COLNAME;DATE;FSTR;NUM;PRICE;QUANT;RN;VALUE
[1]   FSTR←'13A1,T16,G<Z99-99>, I6, T29, P<$>F8.2, T38'
[2]   FSTR←FSTR,',CP<$>F12.2, T51, G<Z9/99/99>'
[3]   FSTR←FSTR,',N< R>Q<          >I13, X¯13, 5< >'
[4]   NUM← 87901 65704 87603 23406 87607 87602
[5]   NUM←NUM, 26507 36108 87606 87604 23401 87605
[6]   NUM←NUM, 87609 27305 87205 25109 72907 31602 58208
[7]   RN←'/BATTERY/CARBURETOR/FUEL TANK/WHEEL/BATTERY'
[8]   RN←RN,'/TIRE/AXLE/TIRE/BRAKE/CARBURETOR/TIRE'
[9]   RN←RN,'/EXHAUST/IGNITION WIRE/SPARK PLUG/RADIATOR'
[10]  RN←RN,'/WATER PUMP/ALTERNATOR/RADIATOR/COIL'
[11]  QUANT← 492 769 371 287 381 98 205 387 201 879 298
[12]  QUANT←QUANT, 367 652 391 738 276 493 387 492
[13]  PRICE← 92.85 73.23 71.8 41.75 96.45 60.9 55.85 66.95
```

```
[14] PRICE←PRICE, 32 157.8 68.9 345 22.5 2.85 63.8
[15] PRICE←PRICE, 53.78 96.7 69.3 25.5
[16] DATE← 121383 50683 62483 41283 90383 42583 111383
[17] DATE←DATE, 92683 30183 62583 21183 62583 31283
[18] DATE←DATE, 80583 22883 90883 71483 102683 12183
[19] VALUE←QUANT×PRICE
[20] COLNAME←'/PART NAME/NUMBER/QUAN/PRICE/VALUE'
[21] COLNAME←COLNAME,'/REORDER/<6 MO.'
[22] COLNAME←FSTR COLNAMES COLNAME
[23] RN←(⍳0) ROWNAMES RN
[24] FSTR RJUST '1/5/87'
[25] FSTR CENTER 'HARRIS GARAGE'
[26] FSTR CENTER 'EASTERN DIVISION INVENTORY REPORT'
[27] ''
[28] COLNAME
[29] ''
[30] FSTR ⎕FMT(RN;NUM;QUANT;PRICE;VALUE;DATE;×DATE-60587)
    ∇
```

The program defines a format string containing formatting instructions, assigns the data variables, uses the title functions and column and row name functions to set the titles and label the rows and columns, and finally calls *⎕FMT* to format the data into the report.

4-2 What Is ⎕FMT?

The system function *⎕FMT* is a simple, economical, and adaptable tool for detailed tabular formatting. *⎕FMT* allows you to:

- format several data arguments at a time
- vary the printing order of columns of data from the normal left-to-right order
- insert message text
- display tables of numbers in designated rows and columns
- decorate monetary values with dollar signs or other identifiers
- express the results of floating-point calculations as standard decimals or integers
- place flags to mark negative or zero values in results.

⎕*FMT* is a dyadic system function. It can be represented as either:

result ← '*formatstring*' ⎕*FMT* *data*

or

result ← '*formatstring*' ⎕*FMT* (*data1* ; *data2* ; *data3* ; . . . ; *datan*)

The left argument, *formatstring*, is a character vector that specifies how you want the data to look. It contains specific formatting instructions that control the editing and display of the right argument, the data you want to format. A data item in the right argument can be the name of a variable or an APL expression that produces a result.

The result of ⎕*FMT* is always a character matrix. It can be stored and used within a larger expression like any other APL expression. For example, the result of the expression:

```
'I4' ⎕FMT 987
```

is a one-row, four-column character matrix containing the value `' 987'`.

The number of rows in the matrix result is determined by the maximum number of matrix rows or vector elements in the data. The number of columns is determined by both the format string and the right argument.

The length of an edited line is limited only by the workspace storage required to hold the values. The value of ⎕*PW* does not control the length of the line, and the value of ⎕*PP* does not affect precision.

4-3 Right Argument – The Data List

The right argument to ⎕*FMT* is a data list containing APL variables, constants, or expressions that return numeric or character scalars, vectors, or matrices. A single data expression in the right argument needs no parentheses; for example:

'*formatstring*' ⎕*FMT* *data*

If the right argument contains two or more data expressions, you can separate them with semicolons, and enclose the entire set in parentheses:

```
'formatstring' ⎕FMT (data1;data2;data3;...;datan)
```

or you can use strand notation to specify the data as a nested array:

```
'formatstring' ⎕FMT data1 data2 data3...datan
```

You can replace any item in the data list with an expression that produces the desired value as a result (though you may need parentheses). The following examples show some permissible right arguments:

```
SCALAR ← 15
VECTOR ← 3.5 4E3 0.007 1
CHAR ← 'MONDAY'
MAT ← 3 4 ρι12

'formatstring' ⎕FMT SCALAR

'formatstring' ⎕FMT 2 2 ρVECTOR

'formatstring' ⎕FMT (SCALAR;VECTOR;CHAR;MAT)

'formatstring' ⎕FMT SCALAR VECTOR CHAR MAT

'formatstring' ⎕FMT (MAT;+/MAT)

'formatstring' ⎕FMT MAT (+/MAT)
```

The APL★PLUS System formats data expressions of different shapes as follows:

- A matrix has each column formatted separately.
- A vector is treated as a one-column matrix.
- A scalar is treated as a one-row, one-column matrix.

If you want to display a vector horizontally, reshape it as a one-row matrix.

Each column of data (which can be one scalar, one vector, or one column of a matrix) is formatted individually as specified by the left argument.

4-4 Left Argument – The Format String

The left argument to *⎕FMT* is a character vector containing one or more format phrases. There are two classes of format phrases:

- **Editing** format phrases edit data in the right argument; for example, the *I* format phrase displays numeric data in integer format.
- **Positioning** and **text** format phrases change the appearance of the result without editing any data; for example, the *T* format phrase specifies tab stops for column placement.

You can also use special **parameters** with many of the format phrases to specify the field width and precision of the image, and you can use **modifiers** to add special effects.

Separate individual format phrases with commas and enclose the entire string in single quotes (the phrase must be character-valued). You can use the format string directly as a character vector, or you can store it in a variable and use it later.

```
'I4' ⎕FMT V
```

or

```
F ← 'I4'
F ⎕FMT V
```

Text Delimiters

Place text in the left argument between pairs of delimiters. Any of the pairs of delimiters shown in the following list are valid. The closing delimiter must correspond to the opening one.

Delimiters	Examples
< >	<19__>
⊂ ⊃	⊂*CR*⊃
¨ ¨	¨=¨
⎕ ⎕	⎕-⎕
⍞ ⍞	⍞*YES*⍞
/ /	/*NO*/

You can use any of the delimiters as text characters within a format phrase; but one of them cannot be the closing delimiter. For example, to use the characters

```
5>3?
```

as text in a format phrase, you could enclose them in different delimiters:

```
⊂5>3?⊃
```

How the Format String is Processed

The format string is scanned phrase-by-phrase from left to right. Editing format phrases are matched with columns of data, and positioning and text phrases are processed as they are encountered without reference to the data.

You need not provide the same number of format phrases as columns of data in the right argument. The number of format phrases does not have to divide the number of columns of data evenly. If the format string contains an insufficient number of phrases to edit all of the data in the right argument, ⎕*FMT* cycles through the format string repeatedly until all of the data has been edited. If the format string contains more phrases than are necessary to edit all of the data, the trailing phrases are ignored.

Spaces in the format string have no effect, except between text delimiters.

4-5 *How to Construct a Format String*

A format string comprises several parts:

- Format phrases edit and position text and data.
- Parameters are used with format phrases to give ⎕*FMT* detailed instructions.
- Modifiers add special effects and decorations to edited data.

For example, in the format string

```
CI13
```

I is the format phrase telling ⎕*FMT* to format numeric data as integers, 13 is the parameter to the *I* format phrase specifying a field width of 13 columns, and *C* is a modifier specifying that commas should separate every three digits in the data.

To construct a format string, determine a report width and the width of individual fields within the report. Then select appropriate format phrases for the corresponding data, and code the required parameters, such as field width and the number of digits to be displayed. You can also include modifiers and decorations for special effects. Add positioning phrases to the format string to control the location of fields.

Simplify the coding of repeated format phrases or groups of format phrases by using repetition factors within parentheses. A repetition factor is a non-negative integer indicating the number of times to apply a phrase or a group of phrases. The default repetition factor is 1.

You may find it helpful to store format strings as variables, especially if you will be using them with several different right arguments.

4-6 The Editing Format Phrases

The *A*, *I*, *F*, *E*, and *G* format phrases place columns of data in the right argument into corresponding fields in the result.

A - Character Editing

The *A* format phrase is the only phrase that edits only character data. The *A* phrase takes the form:

Aw

where *w* is the field width. The field width specifies the number of columns in the result to be occupied by the edited value. (Remember that a vector is treated as a one-column matrix and is formatted as a single column, unless it is reshaped to a one-row matrix.)

```
      'A1' ⎕FMT 'SALT'
S
A
L
T
```

A character matrix in the data can contain several columns. In that case, you must specify a separate *A* format phrase for each column.

```
      CMATRIX ← 1 4 ρ'SALT'
      'A1,A1,A1,A1' ⎕FMT CMATRIX
SALT
```

Instead of typing the format phrase repeatedly, you can use a repetition factor to repeat the phrase. In the following example, a repetition factor of 4 is used.

```
      '4A1' ⎕FMT CMATRIX
SALT
```

When the same format phrase is used for the entire right argument, as in the above example, you can omit the repetition factor.

```
      'A1' ⎕FMT CMATRIX
SALT
```

If the field width is greater than 1, leading positions of the field are filled with blanks.

```
      WORDSQ ← 3 3 ρ'BOAURNSEQ'

      'A1, A2, A3' ⎕FMT WORDSQ
BO A
UR N
SE Q

      'A2, A2, A2' ⎕FMT WORDSQ
 B O A
 U R N
 S E Q
```

If you try to use an *A* format phrase to edit numeric data in the right argument, the result field is filled with stars (*).

```
      'A6' ⎕FMT 133
******
```

Stars also result if you try to use an edit phrase other than *A* to edit character data.

```
      'I4' ⎕FMT 'ABC'
****
****
****
```

I - Integer Editing

The *I* format phrase edits numeric data in integer format. The *I* phrase takes the form:

```
 Iw
```

where *w* is the field width. An *I* format phrase displays numeric data as integers. Be sure to include space in the field width for negative signs in the result that correspond to negative values in the data.

```
      TABLE ← ¯3 2 ¯1 ∘.* ¯2 ¯1 1 7

      TABLE
0.1111111111 ¯0.3333333333 ¯3 ¯2187
0.25          0.5           2   128
```

```
1              ¯1              ¯1    ¯1

      '3I4, I6' ⎕FMT TABLE
0  0  ¯3 ¯2187
0  1   2   128
1 ¯1  ¯1    ¯1

      'I8' ⎕FMT (3×4), 6E2 0.4 476.85912
      12
     600
       0
     477
```

format. The *F* phrase takes the form:

Fw.d

where *w* is the field width and *d* is the number of decimal positions. For example, the format phrase *F*9 . 2 creates a field of nine positions, in which every value is formatted to two decimal places. Allow two positions in the field width for a possible negative sign and a decimal point.

```
      'F9.2' ⎕FMT TABLE
     0.11    ¯0.33    ¯3.00 ¯2187.00
     0.25     0.50     2.00   128.00
     1.00    ¯1.00    ¯1.00    ¯1.00
```

For numbers between ¯1 and 1, a zero is placed to the left of the decimal point in the result. Dyadic format, in contrast, displays such numbers without a zero to the left of the decimal point in fixed-point editing.

E - Exponential Editing

The *E* format phrase edits numeric data in exponential (scientific notation) format. The *E* phrase takes the form:

Ew.s

where *w* is the field width and *s* is the exact number of significant figures in the nonexponent part of the result. For example, the format phrase *E*1 5 . 7 produces a field width of 15 with 7 significant digits. The number of significant digits specified must be between 1 and 16, inclusive. The width must be at least seven more than the number of significant digits to provide space for a negative sign, a decimal point, and a five-position exponent of the form *E ¯nnn*. Additional width leaves visible space between columns.

```
      'E10.2' ⎕FMT TABLE
 1.1E¯1    ¯3.3E¯1    ¯3.0E0     ¯2.2E3
 2.5E¯1     5.0E¯1     2.0E0      1.3E2
 1.0E0     ¯1.0E0     ¯1.0E0     ¯1.0E0

      'E12.4' ⎕FMT 2*50 ¯50
   1.126E15
   8.882E¯16
```

G - Pattern Editing

The *G* format phrase edits numeric data according to a pattern or picture format. With this phrase, you can arbitrarily mix text characters and data in a formatted column by creating a pictorial template for the data. Special characters within the pattern indicate where to display digits in the data; other characters are displayed as they appear in the pattern. The *G* phrase takes the form:

G<pattern>

where *pattern* consists of the special characters 9 and Z, called digit selectors, along with other text characters to insert in the field. Use any valid pair of text delimiters to enclose the pattern (see Section 4-3). For example, to format a date with a *G* pattern, enter:

```
      'G<99/99/99>' ⎕FMT 11887
01/18/87
```

The number of characters, including blanks, between the pattern delimiters determines the field width. The field is formatted with the exact number of characters and digits specified in the pattern.

Data in the right argument is rounded to the nearest integer. Each digit of the integer is transferred to the field to replace a digit selector (9 or Z) in the pattern. A 9 in the pattern transfers the corresponding digit from the integer into the result. A Z suppresses leading and trailing zeros, transferring the corresponding digit only if the digit is nonzero or if it is between two transferred digits, as described below.

Use the *Z* digit selector to omit leading or trailing zeros in the edited fields. If a *Z* corresponds to a zero in the integer, the zero is transferred only if digits on both sides are transferred; otherwise, the corresponding position in the field is unchanged. Since a `9` always transfers a digit to the result, a *Z* between two `9`s acts as a `9` digit selector. For example, *G*<*Z*9*Z*.99> is equivalent to *G*<*Z*99.99>. The following table compares the effects of using `9`s and *Z*s as digit selectors:

Format Phrase	**Data**	**Result**
`G<Z99.99>`	`2460 1200 0`	`24.60 12.00 00.00`
`G<Z99.9Z>`	`2460 1200 0`	`24.6  12.0  00.0`
`G<ZZ9.9Z>`	`2460 1200 0`	`24.6  12.0   0.0`
`G<ZZ9.ZZ>`	`2460 1200 0`	`24.6  12     0`
`G<ZZZ.ZZ>`	`2460 1200 0`	`24.6  12`

You must insert text characters or *Z* digit selectors in the pattern to display negative signs and decorations (decorations are explained in Section 4-10). Leading and trailing text characters always transfer directly to the result. Text between digit selectors transfers only if digits to the right and to the left of the pattern text are transferred. Compare the appearance of zeros and commas in the next two examples.

```
      NUM ← 298738472 389487.987 387 0.35
      ZEES ← 'G<$ ZZZ,ZZZ,ZZ9 AND NO CENTS>'
      NINES ← 'G<$ 999,999,999 AND NO CENTS>'

      ZEES ⎕FMT NUM
$ 298,738,472 AND NO CENTS
$     389,488 AND NO CENTS
$         387 AND NO CENTS
$           0 AND NO CENTS

      NINES ⎕FMT NUM
$ 298,738,472 AND NO CENTS
$ 000,389,488 AND NO CENTS
$ 000,000,387 AND NO CENTS
$ 000,000,000 AND NO CENTS
```

You can use *G* formatting to produce visually effective reports when numbers are displayed in traditional patterns.

```
      EMPLO ← 4 7 ρ'ABEL   GALOIS GAUSS  ZORN   '
      SSN←298374562 298750385 384716453 273069857
      TEL←4086729873 7187364782 8063948726 3138472637
      SALES ← 567 309 4958 312
      SAL ← 49800 50000 59500 41200

      FS ← '7A1,G< 999-99-9999 >,G< (999) 999-9999>
      FS ← FS,',I6,G<  ZZ,ZZ9>'

      FS ⎕FMT (EMPLO;SSN;TEL;SALES;SAL)
ABEL    298-37-4562  (408) 672-9873    567  49,800
GALOIS  298-75-0385  (718) 736-4782    309  50,000
GAUSS   384-71-6453  (806) 394-8726   4958  59,500
ZORN    273-06-9857  (313) 847-2637    312  41,200
```

Although data values edited by a *G* format phrase are rounded to integers, fractional values in the data can be displayed by multiplying the data by the appropriate power of ten. In the following example, the data is multiplied by 100.

```
      MONEY ← 1 4 ρ14.6 52 17 2.44

      'G<  $99.99>' ⎕FMT 100×MONEY
$14.60  $52.00  $17.00  $02.44
```

A better way to achieve the same effect is to prefix the *G* format phrase with the *K* scaling modifier (see "K - Scaling" in Section 4-10). In the following example, the data is scaled by `10*2`.

```
      'K2G<  $99.99>' ⎕FMT MONEY
$14.60  $52.00  $17.00  $02.44
```

4-7 The Positioning and Text Format Phrases

The positioning format phrases allow you to position fields without having to count individual positions. The positioning phrase specifies the column where the result of the next format phrase should begin:

- The *T* phrase specifies the starting column relative to the left margin.
- The *X* phrase specifies the starting column relative to the current position.

The *<text>* format phrase inserts text directly into the result field. These three phrases do not edit data; they simply position it in the result.

T - Absolute Tabbing

The *T* phrase specifies absolute tabbing (jumping to a particular column). It takes the form:

Tp or *T*

where *p* is the column position (counting from the left margin) at which to format the next edited value. For example, *T* 15 moves directly to position 15 regardless of the previous position. The first available position on a line is *T* 1.

If you use *T* without specifying the position, the next field is formatted to the right of the rightmost column of the result formatted thus far.

```
      '3A1, T10, I2' ⎕FMT (1 3 ⍴'TAB';10)
TAB      10
```

Using a *T* phrase before individual format phrases can make the format string easier to modify. The position of a field is clear and is independent of previous formatting instructions.

X - Relative Tabbing

The *X* phrase specifies relative tabbing, or positions to be skipped. It takes the form:

Xp

where *p* is the number of positions to be skipped from the present position before formatting the next field of the result.

- If *p* is positive, *p* is the number of positions to be skipped to the right.
- If *p* is negative, *p* is the number of positions to move left.
- If *p* is zero, the phrase is ignored.

Specify a negative value with a negative sign (X ¯12) or a minus sign (X-12).

```
      'I12,X¯12,2I4,X4,I4' ⎕FMT 1 4 ⍴1 2 3 4
2   3   1   4
```

<text> - Text Insertion

The <*text*> phrase inserts the text between the delimiters into a line. The phrase takes the form:

<*text*>

where *text* is any combination of characters and spaces. All the text between the delimiters, including blanks, is inserted directly into the edited line. You can use any valid pair of text delimiters to enclose the text (see Section 4-3).

```
      DPT ← 3 4 5
      REV ← 344.50 89.74 250.13

      FS←'<REVENUES FOR DEPT.>,I2,< ARE..$>,F6.2'
      FS ⎕FMT (DPT;REV)

REVENUES FOR DEPT. 3 ARE..$344.50
REVENUES FOR DEPT. 4 ARE..$ 89.74
REVENUES FOR DEPT. 5 ARE..$250.13
```

Using Positioning and Text Phrases

You can use the T and X format phrases to position fields without having to count individual spaces. In the format string :

```
I5, X3, 25A1, T51, 4F7.2
```

the first floating point field begins in position 51, and if all four F phrases are used, the result is a matrix with 78 ($50+4\times7$) columns.

Positioning phrases may re-order data in the result. This example uses parentheses and repetition factors to simplify the left argument (see Sections 4-8 and 4-9).

```
      FS ← '3(A1, <-   >), T1, 3(I3,X2)'
      FS ⎕FMT (3 3ρ'ABCDEFGHI';3 3ρι9)
A-1  B-2  C-3
D-4  E-5  F-6
G-7  H-8  I-9
```

You can use text phrases to override previously formatted fields. The decimal point is replaced by a colon in this example.

```
      FS ← 'F7.2, X¯3, <:>, T'
      FS ⎕FMT 1 4ρ10+0.15×¯1+ι4
10:00  10:15  10:30  10:45
```

You can use symbol substitution to achieve the same effect (see "S - Standard Symbol Substitution" in Section 4-10).

A backward-pointing relative or absolute tab may cause a previously formatted field to be overlaid by a new field. This new field need not match the width or alignment of any previously formatted field. In this case, nonblank characters in the new field replace the corresponding characters in the old field. Blank characters in the new field that occur as the result of explicit mention in <*text*> phrases, G phrases, or certain modifiers also replace the corresponding characters in the old field. However, blanks used as fill characters do not replace any characters in the old field.

In the following example, the background fill modifier R (see "R - Background Fill" in Section 4-10) alters the normal blank background fill character.

```
      IOTA ← 1 4 ρ1 2 3 4

      'I12, T1, 2I4, T, I4' ⎕FMT IOTA
   2   3   1   4

      'R<∘>I12, T1, 2I4, T, I4' ⎕FMT IOTA
∘∘∘2∘∘∘3∘∘∘1   4

      'R<∘>I12, T1, 2R< >I4, T, I4' ⎕FMT IOTA
   2   3∘∘∘1   4
```

In the last example, the blanks in the rightmost *R* modifier override previously formatted fields because they are text characters rather than fill characters.

4-8 Parameters

You can use special parameters within format phrases to give ⎕*FMT* detailed instructions. Use them to specify:

- the number of times to apply a phrase or a group of phrases
- the width of an edited column
- the number of digits to be displayed
- the position for a tab or skip.

Most parameters specify essential formatting considerations and are required with their corresponding format phrases; however, repetition factors are optional. Repetition factors improve the readability of a format string containing repeated format phrases and often clarify the structure of the format string at a glance. They are useful when a long format phrase is needed.

w - Field Width Parameter

The field width parameter *w* determines the number of positions in the result to be occupied by the edited value. It is required with the following phrases: *A* as in *Aw*, *E* as in *Ew.s*, *F* as in *Fw.d*, and *I* as in *Iw*.

```
      'F6.1, F12.1, F8.1, F18.1' ⎕FMT 2 4 ρ58.8
  58.8        58.8    58.8                58.8
  58.8        58.8    58.8                58.8

      ANAGRAMS ← 6 4 ρ'OPTSPOSTPOTSSPOTSTOPTOPS'

      'A1' ⎕FMT ANAGRAMS
OPTS
POST
POTS
SPOT
STOP
TOPS
```

If the field width is greater than the number of digits or characters, the field is padded with leading blanks.

```
      'A2' ⎕FMT ANAGRAMS
 O P T S
 P O S T
 P O T S
```

```
S P O T
S T O P
 T O P S
```

Be sure to leave space in the field width for:

- a negative sign in an *E*, *F*, or *I* phrase
- a decimal point in an *F* phrase, or in an *E* phrase if more than one significant digit is specified
- an exponent of the form *E* ¯*nnn* in an *E* phrase
- any spacing you might want between the columns of a matrix.

If the field width is insufficient, the field is filled with stars (see Section 4-12).

```
      TABLE ← ¯3 2 ¯1 ∘.* ¯2 ¯1 17
      'F5.1' ⎕FMT TABLE
0.1  ¯0.3*****
0.3   0.5*****
1.0  ¯1.0  ¯1.0
```

d - Decimal Position Parameter

The decimal position parameter *d* controls the number of digits that appear to the right of the decimal point. It is required with *F* phrases (*Fw.d*).

```
      FS ← 'F2.0, F7.3, F10.6, F13.9, F16.12'
      FS ⎕FMT 1 5⍴○1
3.  3.142  3.141593  3.141592654  3.141592653590
```

s - Significant Digits Parameter

The significant digits parameter *s* controls the number of significant digits in the nonexponent part of the result. The value of this parameter must be at least 1. This parameter is required with *E* phrases (*Ew.s*).

```
      'E10.2. E9.1, E17.9' ⎕FMT 1 3 ⍴¯57.98765
 ¯5.8E1    ¯6E1        ¯5.79876500E1
```

<pattern> - Pattern Text Parameter

The pattern text parameter *<pattern>* provides detailed control over the location of the individual digits of the value to be formatted. It is required with *G* phrases (*G<pattern>*). See "G - Pattern Editing" in Section 4-6 for detailed information on this parameter.

p - Position Parameter

The position parameter *p* controls the location at which to display the next field. It is required with *X* phrases (*Xp*) but is optional with *T* phrases (*Tp*). When used with an *X* phrase, *p* is the number of positions to be skipped from the present position. For example, because *X*1 says to skip the first position, it specifies the second. It can be a positive or negative integer, or zero. A zero causes the phrase to be ignored. Specify a negative value with either a negative sign (*X*¯12) or a minus sign (*X*-12).

When used with a *T* phrase, *p* is the number of positions from the left margin and is always a positive integer. For example, *T*1 indicates the first position.

```
      'I12, 4(X¯4, I1)' ⎕FMT 1 4 ⍴⍳4
   4  3  2  1

      FS ← 'T12, I1, T9, I1, T6, I1, T3, I1'
      FS ← ⎕FMT 1 4⍴⍳4
   4  3  2  1
```

r - Repetition Factor

The optional repetition factor *r* determines how many times to use a single format phrase or a group of format phrases enclosed in parentheses. You can use it with any format phrase. The repetition factor precedes all other elements in the format phrase. For example, a repetition factor of 2 to the left of an editing format phrase causes that phrase to edit two successive columns from the data list. For example, `F6.3, F6.3` is the same as `2F6.3`.

A nonzero repetition factor to the left of a *T* phrase has no effect. For example, `3T5` is the same as `T5` and `3T` is the same as `T`.

A nonzero repetition factor to the left of an *X* phrase repeats the specified skip. For example, `3X5` is the same as `X15` and `3X¯5` is the same as `X¯15`.

A nonzero repetition factor to the left of a <*text*> phrase repeats that phrase. For example, `2<PAGO >` is the same as `<PAGO PAGO >`.

A zero repetition factor to the left of any phrase causes that phrase to be ignored.

The default repetition factor is 1.

4-9 Grouping Symbols

Use parentheses in the left argument of `⎕FMT` to simplify the construction of repeated sets of format phrases and to limit further scanning of the format string when the data has been exhausted.

Enclose the group of format phrases in parentheses and place the repetition factor to the left of the left parenthesis.

```
12A1, 4(F10.2, I4)
I15, 4(X¯4, I1)
```

Without parentheses, you would have to specify the second format string in the preceding example as:

```
I15, X¯4, I1, X¯4, I1, X¯4, I1, X¯4, I1
```

A zero repetition factor to the left of a left parenthesis causes the phrases in the group to be ignored.

If parentheses are used and no repetition factor is specified, 1 is assumed.

After all columns of data have been used, *T*, *X*, and <*text*> format phrases in the format string continue to be used until one of the following occurs:

- an editing phrase (*A*, *E*, *F*, *G*, or *I*) with a nonzero repetition factor
- the end of the format string
- a right parenthesis with a repetition factor that has not been fully used
- a left parenthesis.

The following example demonstrates how to nest parentheses and how to use them to limit scanning of a format phrase when no more data is left to be edited.

```
         ADD ← 2 4 ⍴1 2 3 4 5 6 7 8
         ADD←(+/ADD),ADD
         'I2, < = >, 4(I1, (<+>))' ⎕FMT ADD
10 = 1+2+3+4
26 = 7+6+7+8
```

4-10 Modifiers

Modifiers add decorations and special effects to edited data. Place them between the repetition factor and the format phrase. You can use any number of modifiers in any order.

B - Blank

The *B* modifier leaves the field blank if the edited value is zero. Use it with *F*, *G*, and *I* format phrases.

```
         EX ← ¯65423.43 ¯10 ¯0.4 0 100

         'BF10.1' ⎕FMT EX
   ¯65423.4
      ¯10.0
       ¯0.4
                 (0 shows as blank.)
      100.0
      'BG<ZZZZZZ99>' ⎕FMT EX
 ¯65423
    ¯10
                        (0 . 4 shows as blank.)
                        (0 shows as blank.)
    100
```

C - Comma

The *C* modifier inserts commas between each group of three digits in the integer part of the edited value. Use it with *F* and *I* format phrases.

```
      DATA ← 2987309 3870.23 96874382 38E5
      'CI13' ⎕FMT DATA
  2,987,309
      3,870
 96,874,382
  3,800,000
```

Remember to provide extra positions in the field width for commas.

K - Scaling

The K modifier scales (multiplies) a number before displaying it. It takes the form:

Ki

where *i* is a positive or negative integer, or zero. A negative value can be specified by a negative sign (*K* ¯2) or a minus sign (*K* − 2). Each number to which the K modifier applies is multiplied by 10⋆*i* before it is formatted.

```
      'F8.2,K1F10.2,K¯2F8.2' ⎕FMT 1 3 ρ470.6
  470.60    4706.00     4.71
```

Use the *K* modifier with *E*, *F*, *G*, and *I* phrases. With an *F*, *G*, or *I* phrase, the *K* modifier controls how far the digits are shifted left or right of the decimal point in the result. With an *E* phrase, the *K* modifier adjusts the exponent in the result.

The following table shows the use of scaling with decimal, exponential, and integer editing.

Format Phrase	Number	Result
*F*6.2	24.60	24.60
*K*1*F*6.2	24.60	246.00
K¯2*F*6.2	24.60	0.25
*E*9.4	24.60	2.460*E*1
*K*1*E*9.4	24.60	2.460*E*2
K¯2*E*9.4	24.60	2.460*E*¯1
*I*3	24.60	25
*K*1*I*3	24.60	246
K¯2*I*3	24.60	0

Scaling is particularly useful when formatting numbers with the *G* phrase.

Format Phrase	Number	Result
G<9.*ZZ*>	24.60	0.25
*K*1*G*<*Z*9.*ZZ*>	24.60	2.46
*K*2*G*<*ZZ*999>	24.60	2460
*K*2*G*<*ZZ*.99>	24.60	24.60
*K*2*G*<*ZZ*.9*Z*>	24.60	24.6

L - Left Justify

The *L* modifier left-justifies the edited value in the result field. Use it with *F* and *I* format phrases.

```
      'LI9' ⎕FMT 2*⍳8
2
4
8
16
32
64
128
256
```

When numbers with fixed decimal points or negative signs are left-justified, the alignment may be unusual.

```
'LF10.2' ⎕FMT 34.5 266 0.300 ¯49.04
34.50
266.00
0.30
¯49.04
```

***M* - Negative Left Decoration**

The *M* modifier replaces negative signs with text you specify. It takes the form

M<text>

where *text* replaces the negative sign to the left of the result. Use this modifier with *F*, *I*, and *G* phrases. Be sure to provide space in the field width for the decoration text. *M<¯>* is the default negative left decoration.

In the following example, the *M* modifier replaces the APL negative sign (¯) by the minus sign (-).

```
      EX ← ¯65423.43 ¯10 ¯0.4 0 100

      'M<->F10.1' ⎕FMT EX
  -65423.4
     -10.0
      -0.4
       0.0
     100.0
```

This kind of replacement can be helpful when you use a different type font to print a report. For example, since the APL negative sign (¯) displays as an "at" sign (@) in most fonts, you could use *M<->* to display negative values with a minus sign (-) in another font. You can also use standard symbol substitution to solve similar problems (see "S - Standard Symbol Substitution" in this section).

N - Negative Right Decoration

The *N* modifier places text you specify to the immediate right of an edited negative value. It takes the form

N<text>

where *text* represents the text. Use this modifier with *F*, *I*, and *G* phrases. Be sure to provide space in the field width for the decoration text.

```
      'N<  MINUS>F20.2' ⎕FMT EX
     ¯65423.43  MINUS
        ¯10.00  MINUS
         ¯0.40  MINUS
                 0.00
               100.00
```

O - Format Zeros As Text

The *O* modifier places text you specify in fields with zero values. It takes the form

O<text>

where *text* represents the text. Use this modifier with *F*, *I*, and *G* phrases. Be sure to provide space in the field width for the decoration text.

The *O* modifier overrides any *P* or *Q* modifier that specifies text or decoration for zero values (see "*P* - Positive Left Decoration" and "*Q* - Positive Right Decoration" in this section). If the text is shorter than the field width, the text is right-justified in the result.

```
      'O<NONE> Q< DR> F9.2' ⎕FMT EX
¯65423.43
   ¯10.00
    ¯0.40
     NONE
100.00 DR
```

However, if the *L* modifier is specified (for *F* or *I* phrases only), the text is left-justified. If the text is the same length as the field width, no justification occurs.

```
           'O<NONE> LQ< DR> F9.2' ⎕FMT EX
¯65423.43
¯10.00
¯0.40
NONE
100.00 DR
```

P - Positive Left Decoration

The *P* modifier places text you specify to the immediate left of an edited positive or zero value. It takes the form

P <text>

where *text* represents the text. Use this modifier with *F*, *I*, and *G* phrases. Be sure to provide space in the field width for the decoration text.

```
      'P<+>F10.1' ⎕FMT EX
 ¯65423.4
     ¯10.0
      ¯0.4
      +0.0
    +100.0
```

Q - Positive Right Decoration

The *Q* modifier places text you specify to the immediate right of an edited positive or zero value. It takes the form:

Q <text>

where *text* represents the text. Use this modifier with *F*, *I*, and *G* phrases. Be sure to provide space in the field width for the decoration text.

```
      'Q< DR>I10' ⎕FMT EX
 ¯65423
    ¯10
      0 DR
      0 DR
    100 DR
```

When *M*, *N*, *O*, *P*, and *Q* modifiers are used with the *G* format phrase, the decoration text supersedes text characters in the pattern. The decoration tex appears adjacent to leading or trailing nonzero digits in the result. Therefore, the text phrases may not align.

```
      FMT ← 'M<>N< CR>O<NONE>Q< DR>G<Z,ZZZ--->'
      FMT ⎕FMT ¯123 120 242 0 ¯100 1000
  123 CR
  12 DR-
  242 DR
    NONE
    1 CR--
1 DR ---
```

R - Background Fill

The *R* modifier fills the result field with text you specify in all positions not filled with the edited value. It takes the form

R<text>

where *text* represents the text. Starting at the left side of the field, *text* is repeated as many times as necessary to fill the field. Use this modifier with *A*, *E*, *F*, *I*, and *G* phrases. Be sure to provide space in the field width for the decoration text.

```
      'R<∘ >I10' ⎕FMT EX
∘ ∘ ¯65423
∘ ∘ ∘ ∘¯10
∘ ∘ ∘ ∘ ∘0
∘ ∘ ∘ ∘ ∘0
∘ ∘ ∘ ∘100

      'R<BACKGROUND>I21' ⎕FMT EX
BACKGROUNDBACKG¯65423
BACKGROUNDBACKGROU¯10
BACKGROUNDBACKGROUND0
BACKGROUNDBACKGROUND0
BACKGROUNDBACKGROU100
```

When used with the *G* format phrase, the *R* modifier is displayed only in positions not occupied by text characters or decorations.

```
      RDEC ← 'R<*>G<$ ZZZ,ZZZ,ZZ9>'

      RDEC ⎕FMT 23987458 38794 287 0
$ *23,987,458
$ *****38,794
$ ********287
$ **********0
```

You can use the *R* modifier to replace leading blanks in the result of an *A* format phrase with nonblank characters.

```
      'R<->A2' ⎕FMT 3 3⍴'BOAURNSEQ'
-B-O-A
-U-R-N
-S-E-Q
```

However, the *R* modifier will not put the background text into a data position; rather, it will only fill in leading positions created by using a repetition factor.

```
      'R<*>A2' ⎕FMT 1 8 ⍴'JOHN DOE'
*J*O*H*N* *D*O*E
```

The blank between *JOHN* and *DOE* is not filled in with a background star.

S - Standard Symbol Substitution

The *S* modifier substitutes symbols of your choice for standard symbols used by ⎕*FMT*. Use the *S* modifier with *F*, *G*, and *I* format phrases. You can also use it with the *G* format phrase to free the digit selectors 9 and *Z* to serve as characters to be inserted in the result, and with an *F* or *I* format phrase to tailor other formatting effects to individual needs. *S* allows only a one-to-one substitution of symbols.

The *S* modifier takes the form:

S<symbolpairs>

where the first symbol in each pair must be one of the symbols in the following table, and the second symbol is the temporary replacement for the first.

This table contains a list of symbols that can be replaced using the *S* modifier. Applicable format phrases are shown in parentheses.

Symbol	Purpose (Applicable Format Phrases)
9	Digit selector (*G*)
Z	Digit selector, with leading and trailing zero suppression (*G*)
*	Field overflow fill character (*E*, *F*, *G*, *I*)
.	Decimal point (*F*)
,	*C* modifier insert character (*F*, *I*)
0	*Z* modifier fill character (*F*, *I*) or leading-zero fill character from a 9 digit selector (*G*)

The following example shows how you can use the *S* modifier to display decimal numbers as clock times.

```
      'S<.:>F7.2' ⎕FMT 1 4 ρ10+0.15×¯1+ι4
10:00  10:15  10:30  10:45
```

The substitution affects only the format phrase with the *S* modifier, as in the next example.

```
      'S<.V>F5.1, F9.3' ⎕FMT 1 2 ρ470.6 370.168
470V6 370.168
```

You can substitute for more than one symbol in a single format phrase. For example, symbol substitution can be used to follow European conventions, where periods rather than commas are used as number separators, and commas rather than periods are used as decimal points to mark the fractional part of a value.

```
      'S<.,,.>CF14.2' ⎕FMT 1234567.89
1.234.567,89
```

The only substitutions permitted for symbols in the left argument to ⎕*FMT* are those for the digit selectors 9 and *Z* in the *G* format phrase. The other substitutions that are permitted affect symbols that ⎕*FMT* places in the result. In the next example, the *S* modifier frees the digit selector 9 for use as a character in *text*.

```
      'S<9_>G<19__>' ⎕FMT 64 57 72 54
1964
1957
1972
1954
```

Use the *S* modifier to replace the stars that mark field overflow.

```
      A ← ○ 1E2 1
      'F5.2' ⎕FMT A
*****
 3.14
      'S<*$>F5.2' ⎕FMT A
$$$$$
 3.14
```

The following uses of standard symbol substitution produce a *FORMAT ERROR*:

- An odd number of symbols appears between the delimiters; for example, *S*< . >.
- The first symbol in a pair of symbols is not in the preceding table; for example, *S*<*V* . >.
- More than one substitution is made for the same symbol in the same format phrase; for example, *S*< . : . , >.
- The same symbol is substituted for the digit selectors 9 and *Z*; for example, *S*<9_*Z*_>.

Z - Zero Fill

The *Z* modifier fills unused leading positions in the result field with zeros instead of blanks. Use the *Z* modifier with *I* and *F* format phrases.

```
      'ZI10' ⎕FMT 3×1 2 3 4 5
0000000003
0000000006
0000000003
0000000012
0000000015
```

The *Z* modifier can also be used to display telephone numbers when exchanges and trunk numbers have been stored separately.

```
      EXCHANGES ← 355 298 385 448
      NUMBERS ← 56 7980 230 66

      'I3,<->,ZI4' ⎕FMT EXCHANGES NUMBERS
355-0056
298-7980
385-0230
448-0066
```

When the *Z* modifier is used with an editing phrase to format a negative value, the negative sign is left-justified in the result field.

```
      'ZI5' ⎕FMT ¯1 0 1
¯0001
00000
00001
```

4-11 Combinations of Modifiers

When the *B* modifier and *O*, *P*, or *Q* modifiers are used together, any data values that are zero appear as blanks, and the decoration text does not appear in the result field.

```
      'BO<NONE>I5' ⎕FMT ¯1 0 1
   ¯1

    1

      'BP<+ >I5' ⎕FMT ¯1 0 1
   ¯1

  + 1
```

When the *B* and *R* modifiers are used together, any data values that are zero do not suppress the *R* fill text in the result field.

```
      'BR< NONE>F5.2' ⎕FMT ¯1 0 1
¯1.00
 NONE
 1.00
```

When the *B* and *Z* modifiers are used together, any data values that are zero appear as blanks in the result field.

```
      'BZI5' ⎕FMT ¯1 0 1
¯0001

00001
```

When the *L* and *O* modifiers are used together, text specified by the *O* modifier is left-justified.

```
      'LO<NONE>I4' ⎕FMT ¯1 0 1
¯1
NONE
1
```

When the *L* and *Z* modifiers are used together, the *Z* modifier has no effect.

```
      'LZI5' ⎕FMT ¯1 0 1
¯1
0
1
```

When the *M* and *Z* modifiers are used together, each negative data value appears with its negative sign replaced by the *M* decoration text. The decoration text is left-justified in the result field.

```
      'M<- >ZI5' ⎕FMT ¯1 0 1
- 001
00000
00001
```

When the *O* modifier and the *P* or *Q* modifier are used together without the *B* modifier, any data values that are zero are formatted by the *O* text rather than the *P* or *Q* text.

```
      'O<      ∇>P<[>Q<]>LI5' ⎕FMT 0 1 2 3
     ∇
[1]
[2]
[3]
```

When the *P* and *Z* modifiers are used together, each zero or positive data value appears with the *P* decoration text and is left-justified in the result field.

```
      'P<+ >ZI5' ⎕FMT ¯1 0 1
¯0001
+ 000
+ 001
```

When the *R* modifier is used with one or more *M*, *N*, *O*, *P*, or *Q* modifiers, the decoration text overrides the background text for the portion of the field covered by the *M*, *N*, *O*, *P*, or *Q* modifier.

```
      FS←'R<*>M<->N<->O<NONE>P<+>Q<+>I5'
      FS ⎕FMT ¯1 0 1
**-1-
*NONE
**+1+
```

4-12 Useful Applications

Here are some examples of commercial applications that use ⎕*FMT* with combinations of modifiers. The numeric array *NUM* is used in most of the examples.

```
      NUM
1316026.715     755715.3407  4587093.116
  11586.9      2190044.457   ¯471009.5613
6789137.817          0       9347189.744
3836077.871   ¯5195105.458       234.5
```

Float Dollar Signs and Place Negative Values in Parentheses

```
      'C P<$> Q< > M<($> N<)> F19.2' ⎕FMT NUM
$1,316,026.72       $755,175.34     $4,587,093.12
   $11,586.90     $2,190,044.46      ($471,009.56)
$6,789,137.82             $0.00     $9,347,189.74
$3,836,077.87   ($5,195,105.46)           $234.50
```

Format Negative Values into Separate Columns and Omit Negative Signs

```
      'M<>N<                   > CI40' ⎕FMT  ,NUM
                                        1,316,027
                                          755,715
                                        4,587,093
                                           11,587
                                        2,190,044
                        471,010
                                        6,789,138
                                                0
                                        9,347,190
                                        3,836,078
                      5,195,105
                                              235
```

Display Decorations Only

```
      BOOLEAN ← 0 0 1 0 1
        'P<YES>R<NO  >BI4, X-1, < >' ⎕FMT BOOLEAN
NO
NO
YES
NO
YES
```

4-13 Stars or Unknown Digits in Result

Stars in Result

If stars appear in your result when you did not expect them, you probably used ⎕*FMT* incorrectly. Stars appear in the result when:

- A formatted value is larger than the field width (often because you forgot to account for the width of decorations in the field width).
- You tried to use an *A* format phrase to edit numeric data.
- You tried to use a format phrase other than *A* edit character data.

You can avoid these conditions by using a larger field width, by scaling the data, or by using the correct format phrase for the datatype. You can always substitute another symbol for the star by using the *S* modifier (see Section 4-10).

Unknown Digits in Result

The precision of the result of ⎕*FMT* is independent of the value of ⎕*PP*. ⎕*FMT* displays up to 17 digits. A format phrase requesting more significant digits than are available in the internal representation uses zeros for the missing digits.

```
      'F24.1' ⎕FMT ○1E20
 314159265358979330000.0
```

4-14 Workspace *FORMAT* Overview

The workspace *FORMAT* supplied with your APL⋆PLUS System contains several functions that will help you place titles and row and column names on a report. These functions fall into the following categories:

- Title functions allow you to position titles over your report.
- Label functions allow you to label rows or columns of your report.
- A special function allows you to build custom titling routines.

Use these functions in conjunction with ⎕*FMT* to help you generate your reports. Instead of loading the entire workspace, you can copy any of these functions into your workspace with the command

```
      )COPY [APL.RELn]FORMAT objects
```

where *objects* is a list of one or more desired functions.

Title Functions

Each of the following titling functions produces a character matrix as an explicit result. The matrix has the same width as the result of ⎕*FMT* used with the same format string, which permits catenation of titles onto formatted data to form larger character matrices.

CENTER	Center a title over a report.
LJUST	Left justifie a title over a report.
RJUST	Right justifie a title over a report.

Label Functions

The following functions set up labels for rows and columns of a report.

COLNAMES	Label columns of a report.
ROWNAMES	Label rows of a report.

Custom Titling Routines

The *CENTER*, *LJUST*, *RJUST*, and *COLNAMES* functions analyze a format string using the *RWTD* function. You can use *RWTD* directly to design you own titling routines to meet your needs.

Using the Functions in Formatting Programs

The following program *GO* uses functions in *FORMAT* in conjunction with ⎕*FMT*.

```
      ∇ GO DATA;FSTR;RN
[1]    FSTR←'8A1, X2, I5, F10.2, F15.2'
[2]    RN←8 ROWNAMES '/NUTS/BOLTS/SCREWS'
[3]    FSTR CENTER 'INVOICE'
[4]    FSTR COLNAMES '∘MATERIAL∘QTY∘COST∘EXTENSION'
[5]    FSTR ⎕FMT (RN;DATA)
      ∇

      INFO ← 3 3 ρ20 0.15 3 9 0.71 6.39 3 0.42 1.26

      GO INFO
                INVOICE
MATERIAL QTY        COST      EXTENSION
NUTS      20        0.15           3.00
BOLTS      9        0.71           6.39
SCREWS     3        0.42           1.26
```

To speed up execution of your reports, use the following two techniques:

- Separate report setup from report production.
- Avoid repeated analysis of the same format string by the *RWTD* function.

Separating Report Setup from Report Production

To save execution time when producing reports, use a separate "setup" program to prepare and store as variables portions of the report that do not change, such as titles and row and column labels. The report program runs faster since it is only printing the saved variables rather than computing them. Using this approach, the function *GO* (shown previously) is replaced by two functions *SETUP* and *REPORT*.

```
     ∇ SETUP
[1]  FSTR←'8A1,X2,I5,F10.2,F15.2'
[2]  RN←8 ROWNAMES 'NUTS/BOLTS/SCREWS'
[3]  TITLE←FSTR CENTER 'INVOICE'
[4]  COLN←FSTR COLNAMES '∘MATERIAL∘QTY∘COST∘EXTENSI
     ∇

     ∇ REPORT DATA
[1]  '' ◊ TITLE ◊ ''
[2]  COLN
[3]  FSTR ⎕FMT (RN;DATA)
     ∇
```

The *SETUP* function places four global variables in the workspace, to be used by the *REPORT* function. The *REPORT* function can be run as many times as needed once *SETUP* has been run. If the report specifications change, revise *SETUP*, run it once, and save the workspace for future use.

Avoiding Repeated Analysis by the RWTD Function

When a format string is given as the left argument to *LJUST*, *RJUST*, *CENTER*, or *COLNAMES*, these functions use *RWTD* to analyze the format before proceeding. However, if the left argument to these functions is numeric data, the functions assume it is the proper result of an earlier analysis by *RWTD* and bypass the use of *RWTD*.

Therefore, to save execution time use one of the following procedures:

- Call *RWTD* separately and save its result for use as the left argument to *LJUST*, *RJUST*, *CENTER*, or *COLNAMES*. Lines [3] and [4] of *SETUP* in the previous example would run faster as:

```
[3] RR←RWTD FSTR ◊ TITLE←RR CENTER 'INVOICE'
[4] COLN←RR COLNAMES '∘MATERIAL∘QYT∘COST∘EXTENSION'
```

- Use a numeric vector of field widths instead of *RWTD*. The *COLNAMES* program will right justify headings over fields with positive widths, and left justify headings over fields with negative widths. Lines [3] and [4] of the previous example would then be:

```
[3] RR←¯8 7 10 15 ◊ TITLE←RR CENTER 'INVOICE'
[4] COLN←RR COLNAMES'∘MATERIAL∘QYT∘COST∘EXTENSION'
```

SCREEN/KEYBOARD FACILITIES

Chapter Five
Keyboard and Screen Management

In addition to manipulating the screen from the terminal keyboard, the APL⋆PLUS System provides system functions to manipulate the screen under program control. Using these facilities changes the image of the APL session screen in memory and also causes APL to transmit the necessary control sequences to the terminal to produce the same effect.

5-1 Simple Input and Output Management Facilities

The building blocks of simple input and output facilities are the ⎕ (quad) for numeric data and ⍞ (quote-quad) for character data. The ⎕ can be used in an expression like ⎕←*A* to display the contents of *A* at the terminal. It can be used in the expression *A*←⎕ to wait for input from the terminal and store it in the variable *A*. The ⍞ works the same way for character input, except that on output (⍞←*A*) it does not display a carriage return at the end of the data.

A typical function might use ⎕ or ⍞ both to accept input data for processing and to display the result of the operation:

```
      ∇UPDATE
[1]  ⍞←'ENTER NAME: '
[2]  NAME←⍞
[3]  ⍞←'OK, ',NAME,' HOW OLD ARE YOU? '
[4]  AGE←⎕
 .
 .
 .
      ∇
```

The function above would begin execution as follows, with user input in bold:

```
      UPDATE
ENTER NAME:
SMITH
OK, SMITH, HOW OLD ARE YOU? 34
      .
      .
      .
```

In the *UPDATE* function above, *AGE* will include leading spaces to indicate the location of the entry on the line. Some older APL systems return the entire line (including prompt) rather than spaces. If you have programs that require this behavior, you can set ⎕*PR* to `''` to emulate the older style. See Chapter 3 of the *APL★PLUS System Reference Manual* for details on ⎕*PR*.

Other APL★PLUS systems use the expression:

```
      ⍞←'PROMPT' ◊ ⎕ARBOUT ⍳0 ◊ ANSWER←⍞
```

to eliminate anything on the line except the user's entry. The APL★PLUS System for VAX/VMS supports this idiom.

⎕*ARBOUT* ⍳0 is a special definition of ⎕*ARBOUT*. Normally ⎕*ARBOUT* can be used to send any ASCII character to the screen. It sends arbitrary character data to the screen, interpreting the numeric codes given in its argument as the corresponding ASCII character.

Terminal Control Characters

Several system constants are provided to simplify sending control characters to the terminal. They can be used to produce special actions on the terminal. The system constants take the form ⎕*TCnn* and produce the following control characters:

• Bell:	⎕*TCBEL*
• Backspace:	⎕*TCBS*
• Delete:	⎕*TCDEL*
• Escape:	⎕*TCESC*
• Form Feed:	⎕*TCFF*
• Linefeed:	⎕*TCLF*
• New Line:	⎕*TCNL*
• Null:	⎕*TCNUL*

Note how some of these are used in the example below. For details on the terminal control system constants, see Chapter 3 of the *APL*PLUS System Reference Manual*.

In conjunction with ⎕ and ⍞ on output, ⎕*CURSOR* can position the output anywhere on the screen; ⎕*TCFF* can be used to clear the screen altogether. The session-related variable ⎕*CURSOR* contains the current location of the cursor within the current window on the screen. Its value is a vector containing the row and column of the current cursor position relative to the upper left corner of the window in use (see ⎕*WINDOW*).

You can assign ⎕*CURSOR* a new value, which will immediately cause the cursor to move to a new location. The following function, a slightly enhanced update program, clears the screen and positions the question on the screen:

```
     ∇UPDATE2
[1]   ⎕TCFF
[2]   ⎕CURSOR ← 4 15
[3]   ⍞←'ENTER NAME: ' ◊ ⎕ARBOUT ⍳0
[4]   NAME←⍞
[5]   REPLY←'OK, ',NAME,'HOW OLD ARE YOU? '
[6]   ⎕CURSOR ← 6 10
[7]   ⍞←REPLY,⎕TCBEL ◊ ⎕ARBOUT ⍳0
[7]   AGE←⍞
     ∇
```

Screen Operations

If you have a large quantity of text to display on the screen or if you want to specify display attributes such as reverse video, ⎕*WPUT* is a useful tool. This system function replaces the characters or attributes (or both) in a specified screen window. ⎕*WPUT* has the syntax:

wspec ⎕*WPUT data*

where *wspec* is the optional specification of a window other than the current one, and *data* is the data to display in the window. The *wspec* argument takes the form of the session-related variable ⎕*WINDOW*. ⎕*WINDOW* is a vector whose first two elements are the absolute screen position of the upper-left corner of the window and whose last two elements are the shape of the window.

The system function that corresponds to ⎕*WPUT* on input is ⎕*WGET*. ⎕*WGET* reads the characters or attributes (or both) from a specified screen window. It has the syntax:

```
result ← wspec ⎕WGET rtype
```

where *wspec* is the optional specification of a window other than the current one and *rtype* is the type of result desired: an integer code from 1 to 4.

Display Attributes

Most CRT terminals provide a facility for display attributes, allowing the display of characters to be emphasized by reverse video, high intensity, or blinking. The APL*PLUS System allows you to manipulate the display attributes on the screen using ⎕*WGET* and ⎕*WPUT*. APL uses a logical attribute approach to standardize the handling of attributes on a wide variety of terminals.

The logical attribute is a number representing the combination of attributes that are used to display a character. Each possible attribute has a numeric value, and the combination of attributes is expressed as the sum of these numbers. This APL*PLUS System is capable of handling up to 8 distinct display attributes in 256 different combinations. Most monochrome terminals support 4 attributes. By convention, the attributes are identified to APL by the following attribute values:

0	default display form for the terminal
1	reverse video
2	alternate intensity (brighter or dimmer than usual)
4	blinking
8	underlined

For example, to make a 10-by-10 section of the screen display in reverse video with blinking characters, use:

```
0 0 10 10 ⎕WPUT 5
```

These attribute conventions work only if the termcap database being used contains the necessary definitions. See Appendix D for information on the structure of the termcap database.

Single Keystroke

To capture a single keystroke, use the system function ⎕*INKEY*. Where ⍞ captures all input until the user presses Return, ⎕*INKEY* takes only a single input character. For example:

```
⎕←'PRESS ANY KEY TO CONTINUE' ◊ KEYSTROKE←⎕INKEY
```

⎕*INKEY* allows you to write APL programs that react to each keystroke as the user types it.

Using the Programmable Function Keys

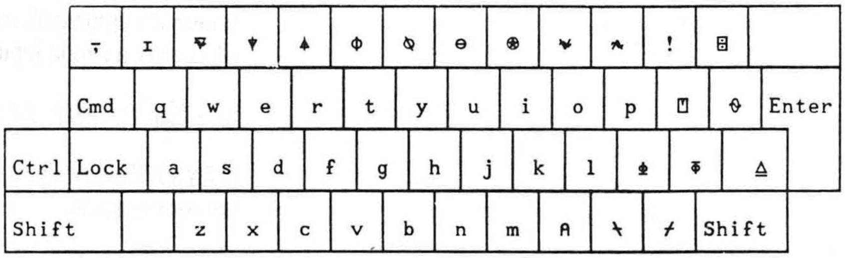
The APL⋆PLUS System includes a facility for defining a group of logical programmable function keys that provide a faster method to type pre-defined sequences of keystrokes as input. This facility is independent of any real function keys that you may have on your terminal keyboard, and you can use this facility even if there are no function keys on the terminal.

A function key is used by typing the PF-key keystroke (also known as the Alt-Key) followed by any other single character. A distinct function key is defined for each of the 128 ASCII characters. For example, suppose that the function key value for the "1" key is defined to be the six-character sequence (')*VARS*',⎕*TCNL*). Then, when you press PF-key followed by 1, a)*VARS* and a simulated Return are displayed as though you had entered them yourself from the keyboard.

Each function key has a default definition. The most commonly used APL overstrikes can all be produced using the default function keys. These keys have been chosen to correspond to the way the Alt-shift key is used to produce overstrikes on the APL⋆PLUS System and on IBM 3270-series terminals. Thus, PF-key followed by " , " produces ⍝, and PF-key *A* produces a. For this reason the PF-key keystroke is also known as Alt-key.

The diagram belows shows the default definitions that correspond to the Alt-keys on other keyboards. For the other ASCII characters, the default function key yields the character itself.

Default Alt-key Definitions


⍪ ⌶ ⍫ ⍒ ⍋ ⌽ ⍉ ⊖ ⍟ ⍱ ⍲ ! ⌹
Cmd q w e r t y u i o p ⍞ ⍣ Enter
Ctrl Lock a s d f g h j k l ⍎ ⍕ ⍙
Shift z x c v b n m ⍝ ⍀ ⌿ Shift

A single function key sequence can be defined to have as many as 64 characters, but the total for all 128 function keys cannot exceed 512 characters.

The system function ⎕*PFKEY* reports and optionally modifies the definitions of the function keys. Used monadically, it reports the current definition:

```
      ⎕PFKEY '1'
⌶
```

Used dyadically, it changes the definition:

```
      ( ')VARS',⎕TCNL  )  ⎕PFKEY '1'
```

The effect of the above statement is to make PF-key 1 expand into the command)*VARS*. Note that ⎕*PFKEY* makes the distinction between shifted and unshifted keys. This makes it possible to assign key definitions that, when the text keyboard option is in use, are the same as the "unified" keyboard used by the APL⋆PLUS PC System.

⎕*PFKEY* is described in detail in Chapter 3 of the *APL ⋆PLUS System Reference Manual.*

Character to Numeric Conversion

Frequently, data captured from screen input must be converted from character to numeric. The system function ⎕*FI* will convert character vectors that are images of numbers to their numeric form. Its syntax is:

```
     result ← ⎕FI data
```

where *data* is character data. The result is a numeric vector formed by taking successive groups of nonblank characters from the argument and converting them to numbers. Any characters that do not represent a well-formed number appear as zero in the result. For example:

```
      ⎕FI 'ANSWER: 666'
0 666
```

Other algorithms using the execute primitive (⍎) could be used for the same purpose but ⍎ will produce an error message for any incorrectly formed data, rather than returning a zero.

The system function ⎕*VI* can be used in conjunction with ⎕*FI* to provide a validity check on the input character vector. The syntax for ⎕*VI* is the same as for ⎕*FI* but the result is a Boolean vector with a 1 in each position that represents a well-formed number and a 0 for each group of characters that does not form a valid number. For example:

```
      ⎕VI 'ANSWER: 666'
0 1
```

Another idiom commonly used to convert only valid character input into numeric input is:

```
      ANSWER←⍞
      NUMANSWER←(⎕VI ANSWER)/⎕FI ANSWER
```

Here is another example of character conversion:

```
      C←' .25 -6.25 8,9,10'
      ⎕FI C
0.25 0 0
```

```
      ⎕VI C
1 0 0
```

As suggested by the last example, a minus sign (-) cannot be used in place of a negative sign (¯) to negate a number; and commas and other symbols do not serve to separate fields. An example of a technique for modifying such symbols to form a valid argument for ⎕*FI* is:

```
      EXT←' 0123456789E.,+-'
      INT←' 0123456789E.   ¯ '
      ⎕FI INT[EXTι' .25 -6.25 8,9,10']
0.25 ¯6.25 8 9 10
```

For more detailed information on the use of each of these system functions and constants, see the *APL★PLUS System Reference Manual.*

5-2 The *INPUT* Workspace

The APL★PLUS System includes the workspace *INPUT* that contains utility functions incorporating the system functions described above. The utility functions are easy to use, have widely accepted default behavior, and can handle special requirements as well. The functions include the following built-in features:

- same-line prompting
- help messages
- normalization of input
- checks and limits on input length
- error messages
- keyword detection (such as HELP or END)
- customized processing.

The input workspace contains fuctions that accept and manipulate input. Six input functions whose names begin with IP form the core of the workspace:

- *IPCHR* accepts character input.
- *IPNUM* accepts numeric input.
- *IPMATCH* accepts forced choice input (character or numeric).
- *IPMIX* accepts combined character, numeric, and forced choice input.
- *IPBULK* accepts bulk input.
- *IPYN* accepts variations of "yes" or "no" and calls *IPMIX*.

These functions can be used either as stand-alone functions or as functions that call the input primitive *IP*. All six functions prompt for and accept input, remove superfluous blanks, check input length, display more detailed prompts on empty input, and reprompt, if necessary, with error messages. If the optional global *ipesc* is defined, the functions detect keywords like *HELP*, *END*, or *QUIT* and return a distinguished result if a keyword is encountered. You can also compose customized input functions using these functions as building blocks:

- *DEB* deletes excess blanks (leading, trailing, and consecutive).
- *DLB* deletes leading blanks.
- *DTB* deletes trailing blanks.
- *SSDEB* deletes excess blanks from substrings of a segmented vector.
- *SSMATCH* matches data against a segmented vector of keywords.
- *SSGLB* converts a segmented vector into a set of global variables.
- *SSMAT* converts a segmented vector into a matrix.
- *SSVEC* converts a segmented vector into an aligned vector.

The input workspace also serves as a source for these commonly used input functions:

- *AKI* accepts keyboard input.
- *NIP* accepts numeric input.
- *AYN* accepts "yes" or "no" responses.

Broader capabilities, however, are provided by the functions *IPCHR*, *IPNUM*, and *IPYN*.

To most effectively use the input workspace, you need to answer some basic questions about your data input requirements.

What Kind of Input Are You Requesting?

Are you asking for numbers, such as salary information, or are you requesting character data, such as an employee's name? Do you want to restrict input to specific choices, as in:

```
CHOOSE ONE OF:  ADD, CHANGE, DELETE
```

Do you need to request input of mixed datatypes, such as name, age, and salary? Do you need to collect several lines of input at once?

If you want this kind of input	**Use this function**
numeric	*IPNUM*
character	*IPCHR*
specified choices (numeric or character)	*IPMATCH*
mixed (numeric or character)	*IPMIX*
bulk input	*IPBULK*

How Do You Set Up a Prompt?

The right argument to each input function is a vector of one or more prompts. The leading prompt should be brief since it is displayed each time input is requested. Subsequent prompts should be detailed enough to clarify the leading prompt. They are displayed when a user requests help by pressing the Return key when prompted for input.

If you need to request phone numbers, a possible line of code is:

```
A←3 IPNUM '|PHONE: |3 NUMBERS (E.G. 301 657 1872)'
```

An experienced user would generally need to see only the first prompt:

```
PHONE: 301 320 4089
```

But if the user needs help, he can press Return to request more details:

```
PHONE: (Return)
3 NUMBERS (E.G. 301 657 1872)
PHONE: 301 320 4089
```

How Do You Restrict Input?

The left argument contains control information.

IPNUM and *IPCHR*

Do you want to restrict the input length? Can input be of any length? Must your input fit in a range of lengths, such as 1 to 30 characters? Or do you need exactly two numbers?

Length restrictions?	**Your left argument should be**
No	⍳0
Yes, exact lengths	one or more non-negative integers, such as 2 4 for two or four characters or numbers
Yes, a range of lengths	one or two non-positive integers such as ¯3 for one, two or three characters or numbers, or ¯2 ¯4 for two, three, or four characters or numbers

If you want to request exactly three numbers, you can use this statement:

```
      3 IPNUM '|PHONE: |ENTER 3 NUMBERS'
```

If you need to store employee names with a maximum length of 40 characters, you can use the following input statement:

```
¯40 IPCHR '|NAME: |LASTNAME, FIRST MI'
```

to accept responses containing 1 to 40 characters.

IPMATCH

What kind of choices do you require? Are your choices numeric (for example, a valid zone number of 1, 2, or 3) or character? If character, are you accepting exact matches only (*ADD* exactly matches *ADD*), or are you taking partial matches as well (*A*, *AD*, and *ADD* match *ADD*)? Are you taking only "yes" or "no" responses?

If your choices are	**Your left argument should be**
numeric	a numeric vector
character, partial matches	a segmented string of choices (*LA←'|SINGLE|MARRIED'*)
character, exact matches	a segmented string reshaped into a one-row matrix (*LA←1 15⍴'|SINGLE|MARRIED'*)
"Yes" or "no"	*LA←'|YES|NO'*

You can also use the unlocked cover function *IPYN* if you want only "yes" or "no" responses. *IPYN* has a built-in left argument of *|YES|NO* and returns a Boolean result.

The result of *IPMATCH* is a one-element integer vector indicating the number of the matched segment:

```
    R←'|SINGLE|MARRIED' IPMATCH 'MARITAL STATUS: '
MARITAL STATUS: M
    R ◊ ⍴⍴R
2
1
```

IPMIX

What combination of data do you require? Once you have determined what types of data you want, you need to decide what length restrictions (as for *IPCHR* and *IPNUM*) and specific input choices (as for *IPMATCH*) you need. Each field of input requires a control specification. For example, if you are requesting a name and age, you can specify two control specifications in the left argument:

```
      ¯402 13 IPMIX '/NAME/AGE:   '
```

The integer ¯402 specifies character data (type 2) of length 1 to 40; the integer 13 specifies integer data (type 3) of length exactly 1.

Do You Want to Detect Keywords Like HELP or END?

If so, you must create a global variable called *ipesc*. It is a segmented string of keywords, for example:

```
      ipesc←'|HELP|END'
```

and it is defined in the same way as the left argument to *IPMATCH* with regard to exact and partial matches. A match to *ipesc* is returned as a scalar integer to distinguish it from other data which is always returned as a vector. Note that on some hardware, lowercase APL characters (*ipesc*) may appear as underscored APL characters (*IPESC*).

Do You Have Special Processing Needs?

The input functions perform quote-quad (⍞) input and output, including the display of error messages. If you have special terminal requirements, you can write a non-default *IP* function to handle them. For example, you can have *IP* clear the screen before a user enters input.

COMMUNICATIONS

Chapter Six
Communications

Computers often need to exchange information with other computers or peripheral devices. The primary feature of the APL★PLUS System enabling such communication is the system function ⎕*ARBIN*. ⎕*ARBIN* can be used to create print files (see Chapter 8) or to gather information from the keyboard (see Chapter 5).

This chapter will survey the use of ⎕*ARBIN* and then review the application built with ⎕*ARBIN*, *SERHOST* (Section 6-3), to move APL workspaces and files from one APL System to another.

The other way to communicate between APL Systems is to move an APL workspace or file to a native file with the utility functions in the *SLT* or *TRANSFER* workspaces. A file transfer program such as Kermit can move the file from one machine to the other.

The *SLT* workspace is discussed in Section 6-5. Ways to move files from machine to machine are discussed in Section 6-6.

6-1 How to Communicate with Remote Devices

The system function ⎕*ARBIN* provides a facility for detailed control of input and output between APL on the VAX and the user's terminal. If the user's terminal is actually another computer, such as a PC running the APL★PLUS System, ⎕*ARBIN* can be used as the basis for transferring data between the two computers under program control. The *SERHOST* workspace is an example of this kind of data transfer.

The syntax for ⎕*ARBIN* is:

result ← *paramlist* ⎕*ARBIN* *prompt*

where *result* contains any received data, *paramlist* is a list of parameters that describes where and how to communicate, and *prompt* is the data to be transmitted.

The left argument is a vector (or singleton) of integers describing how the communication is to be carried out. If the left argument is a singleton, it specifies the outport and awaits no input, but immediately resumes local processing. If input is expected, or if you are specifying values other than default values, the left argument contains the following elements in this order:

[1] outport — destination of data

[2] inport — source from which data is to be received

[3] translation — translation table to be used

[4] protocol — transmission protocol to be used

[5] wait — number of seconds to wait for data

[6] charlimit — maximum number of characters of data

[7] terminators — list of termination codes.

The elements are used starting from the left; if fewer than six elements are given, the remaining parameters assume the default values. For detailed information on syntax, default values, and behavior of ⎕*ARBIN*, see Chapter 3 of the *APL*PLUS System Reference Manual*.

The first two elements of the parameter list tell ⎕*ARBIN* where to send data and from where to receive it. These are described as the **outport**, or the destination of data you are sending, and the **inport**, or the source of data sent to you. The other parameters are used to control how these ports are handled.

6-2 Transferring Data from Other APL Systems

The APL*PLUS System contains several utility workspaces and programs to help exchange workspaces and files with APL Systems on other computers:

- *SERHOST* -- a workspace for transferring APL component files between the APL*PLUS System for VAX/VMS and the

APL ⋆ PLUS Systems for the PC or Macintosh. *SERHOST* can be used to move workspaces as well as files since workspaces can be stored in APL component files and later recreated into workspaces.

- *TRANSFER* -- is in a workspace containing a number of useful utilities that can be used in conjunction with Kermit or another file transfer technique to transfer APL workspaces and APL data.

- *SLT* - Source Level Transfer - is a workspace with functions that will transfer workspaces from one APL system to another using the Workspace Interchange Standard.

The kind of data being transferred and the degree of compatibility between the source and destination systems determines which tools are most appropriate to use. There are several different kinds of compatibility:

- APL workspace and component file compatibility - - A workspace saved on the APL ⋆ PLUS PC or UNIX system cannot be loaded by the APL ⋆ PLUS System for VAX/VMS. This is true even between the VMS and ULTRIX versions of APL for the VAX. The internal structures of saved workspaces and component files are different for each APL ⋆ PLUS System implementation.

- Character set compatibility - - The order of characters in ⎕*AV* for the APL ⋆ PLUS System for VAX/VMS is exactly the same as the APL⋆PLUS PC System, but differs from the APL ⋆ PLUS UNX and APL ⋆ PLUS Mainframe systems. APL character data must be translated when moved to systems with different ⎕*AV* order. The *SERHOST* workspace handles ⎕*AV* differences automatically. Explicit translation is needed, however, if you use a native file-based transfer between the APL ⋆ PLUS System for VAX/VMS and APL⋆PLUS System for UNIX. The *TRANSFER* workspace contains functions to perform this translation.

- APL source level compatibility - - APL programs that look the same produce the same effect. The APL ⋆ PLUS PC, Mac, UNIX, VM, MVS, and VAX/VMS systems are source level compatible at the APL standard level and the APL ⋆ PLUS standard level with only a few exceptions. Nested arrays are not available on the Macintosh or PC, for example. The APL⋆PLUS Systems for VAX/VMS and for UNIX are completely compatible at the source level.

The APL standard level consists of the APL primitives, system functions, and system variables that are defined by the ISO/ANSI standard for the APL Programming Language. The APL⋆PLUS standard level is a superset of the base APL standard, containing the STSC enhancements to APL whose behavior is compatible across all versions of the APL⋆PLUS System. Any system feature that is not identified as "system dependent" or "experimental" can be assumed to be an APL⋆PLUS standard.

It is be practical to transfer APL programs and data in source form between the various APL⋆PLUS systems. Conversion is likely to be required only where "system dependent" or "experimental" features are used.

The APL⋆PLUS Systems for PC, UNIX, and VAX/VMS Systems are partially compatible in certain system-dependent device control features such as ⎕*ARBIN*, ⎕*WGET*, ⎕*WPUT*, and ⎕*INKEY*, which are as similar in the three environments as is technically practical. In many cases, it is possible to move code that uses these features between the systems with very little conversion.

6-3 Using the *SERHOST* and *SERXFER* Workspaces

The easiest way to move APL functions and variables between either the PC or Mac systems and the VAX/VMS System is with the *SERHOST* workspace. For additional information on *SERXFER* see the APL⋆PLUS PC *Programmer's Manual*, or the APL⋆PLUS Mac System documentation.

The PC or Macintosh must use the terminal mode of the APL⋆PLUS System to sign on to the VAX in order to use the *SERHOST* and *SERXFER* workspaces.

Summary of Personal Computer to VMS Transfer Using SERXFER

1. If transferring workspaces, create a transfer file on the personal computer:

```
)LOAD myws
)COPY SERXFER DTF DTFALL
'myws' ⎕FCREATE 1
DTFALL 1
```

2. Switch the personal computer into terminal mode and sign on to the VAX. Invoke the APL⋆PLUS System for VMS, then:

```
)LOAD SERHOST
'myws' ⎕FCREATE 1
```

3. Switch the personal computer back into local mode, then:

```
1 SENDSFILE 1
```

4. When the transfer is complete, switch the personal computer into terminal mode. If the file is a transfer file, reconstitute the workspace as follows:

```
)CLEAR 300000   ⍝ As large as necessary
)COPY SERHOST LFF LFF 1
```

. . . (objects are defined from transfer form)

```
)ERASE LFF DTF DTFALL
)SAVE myws
```

Summary of VMS to Personal Computer Transfer Using SERXFER

1. If necessary, create the transfer file on VAX:

```
)LOAD myws
)COPY SERHOST DTF DTFALL
'myws' ⎕FCREATE 1
DTFALL 1
```

2. On the personal computer:

```
)LOAD SERXFER
'myws' ⎕FCREATE 1
```

3. On the VAX:

```
)LOAD SERHOST
```
   (Make sure the source file is tied to 1)

4. On the personal computer:

```
1 GETSFILE 1
```

5. When the transfer is complete, a transfer file can be reconstituted into a workspace on the personal computer:

```
)CLEAR
)COPY SERXFER LLF
LLF 1    ⍝ Restore the workspace from transfer form.
```

   . . . Objects are defined from transfer form

```
)ERASE LFF DTF DTFALL
)SAVE myws
```

For more information on using some of the functions in the *SERHOST* workspace, see Chapter 4 of the *APL*PLUS System Reference Manual.*

6-4 Using the *TRANSFER* Workspace

An alternative transfer method that requires more programming effort is a native file transfer, which serves the same function as the component transfer files used by the *SERXFER* protocol. The transfer file is a native file containing character vectors delimited by a specific character (the default is ⎕TCBEL). Each character vector is the representation of one APL object in the same "transfer form" as is used by the *SERXFER* workspace.

The *TRANSFER* workspace contains the functions *DTFN* and *DTFNALL*, which build a native transfer file, and *LFFN*, which reconstitutes a workspace from the native file transfer form.

The syntax of these functions is the same as *DTF*, *DTFALL*, and *LFF* in the *SERXFER* and *SERHOST* workspace. A more detailed description is provided in chapter 4 of the Reference Manual.

The native transfer file is the recommended technique for exchanging workspaces with machines running the APL⋆PLUS System for UNIX. It can also be used for transfers with the APL⋆PLUS PC System.

Summary of the TRANSFER Functions

names DTFN tieno

DTFN takes a matrix of *names* as the left argument and writes the corresponding objects in source code form to the native file tied to *tieno*.

DTFNALL tieno

DTFNALL takes a native file *tieno* as the right argument and writes the entire contents of the active workspace to the corresponding native file.

LFFN tieno

LFFN takes a native file *tieno* as the right argument and recreates the active workspace from its contents.

(tieno1 , size) UNBLOCKS tieno2

UNBLOCKS takes a native file *tieno1* and file size as its left argument and unblocks the file to the file corresponding to *tieno2*. The result is a sequential STREAM_LF file. Since Kermit builds files in 512-byte blocks for transmission, the desired size of the file must be specified so that *UNBLOCKS* can remove the padding on the last block. Get the *size* by performing a ⎕*NSIZE* on the original file on the other system.

The translate table *∆TRTUNXTOVMS* is also provided in the *UTILITY* workspace. This variable is used to translate characters from their representation in APL⋆PLUS System for UNIX to the APL⋆PLUS System for VAX/VMS. The character set used in the APL⋆PLUS System for VAX/VMS is the same as that used in the APL⋆PLUS PC System. Therefore, translation is not needed to move workspaces from the DOS environment to VMS. The translate table may be applied by removing the comment symbol (⍝) from line [13] of function *LFFN*.

Summary of the Native File Transfer: UNIX to VMS

1. Build the transfer file on the APL⋆PLUS System for UNIX:

```
)LOAD myws
)COPY TRANSFER DTFN DTFNALL
'MYWS.T' ⎕NCREATE ¯1
DTFNALL ¯1
SIZE←⎕NSIZE ¯1  ⍝ Note size of the file for step 3 below
⎕NUNTIE ¯1
```

2. In UNIX, use Kermit in image mode or any other file transfer program that will transfer binary data to move the file to the VAX.

3. Run the APL⋆PLUS System on the VAX, then:

```
)CLEAR 300000   (as large as necessary)
)COPY TRANSFERFNS LFFN ∆TRTUNXTOVMS UNBLOCKS
'MYWS.T' ⎕NTIE ¯1
'MYWS.S' ⎕NCREATE ¯2
(¯1,size) UNBLOCKS ¯2 ⍝Get size from step 1, above
```

Modify the function *LFFN* to translate between the two ⎕AV orders by deleting a single ⍝ as indicated in the function itself.

```
LFFN ¯2
)ERASE LFFN DTFN DTFNALL UNBLOCKS
)ERASE ∆TRTUNXTOVMS UNBLOCKS
⎕NUNTIE ¯1 ¯2
)SAVE myws
```

6-5 Using a Source Level Transfer

The functions in the *SLT* workspace conform to the Workspace Interchange convention (as described in the ISO/ANSI APL standard) for converting APL functions, variables, and environmental information into character variables called canonical representation vectors, or CRVs. The Standard also prescribes a technique for mapping a sequence of CRVs into a bit stream for transfer to another installation on a physical medium. The technique permits transfers between any two APL systems, regardless of character sets or the internal representations of characters.

The following overview summarizes the procedures for transferring workspaces from the APL⋆PLUS System and for installing transfers from other systems onto the APL⋆PLUS System. For details on the functions in workspace *SLT*, see *DESCRIBE* in the workspace and the function descriptions in Chapter 4 of the *APL ⋆PLUS System Reference Manual.*

Transfer FROM the APL ⋆PLUS System

1. Condition the workspace. The workspace should be loaded using *)XLOAD* if appropriate. All functions should be unlocked and all global variables should have their correct values.

2. Copy *SLT* into the active workspace:

```
      )COPY [APL.RELn]SLT
SAVED 17:15:54 08/13/87
```

3. Send the workspace to file:

```
      DUMPWS 'MYFILE.SLT'
FILE SIZE: 3218
⎕PP
⎕IO
⎕CT
⎕RL
⎕LX
⎕ELX
⎕ALX
NAME1
NAME2
NAME3
```

```
FILE SIZE: 8943
```

4. Repeat the above steps for each workspace to be transferred.

5. Output files can then be written to tape or transferred to another system by a file transfer utility such as Kermit (see Section 8-5).

Transfer TO the APL*PLUS System

1. A Source Level Transfer file must be read into the VMS System using a file transfer utility such as Kermit.

2. Use *WSLIB* to produce a catalog of the transfer file:

```
      )LOAD SLT
SAVED 17:15:54 08/13/87

      'MYFILE.SLT' ⎕NTIE ¯1
      WSLIB ¯1
OFFSET: 961         WSID: WS TRANSFERWS
OFFSET: 14014       WSID: FILE TRANSFERFILE
OFFSET: 50868       END OF FILE.
```

3. Bring in a workspace from the file:

```
      'WICTEST' LOADWS ¯1 961
OFFSET: 961       WSID: WS TRANSFERWS
⎕PP
⎕IO
⎕CT
⎕RL
⎕LX
⎕ELX
⎕ALX
MAINFUNCTION
SUBFUNCTION
VARIABLE
OFFSET: 14014   WSID: FILE TRANSFERFILE
SAVING WICTEST
WICTEST SAVED 17:59:31 08/13/87
```

Note that the filed workspace is automatically created and saved. The second element of the right argument of *LOADWS* is the offset for the workspace to be loaded.

4. Repeat the above step for each workspace on the transfer file.

6-6 Communicating with Other Machines

There are two facilities for transferring non-APL files directly from other machines to the VAX: the VMS command SET HOST, and Kermit. Kermit is an implementation of the popular KERMIT file transfer protocol, which can send and receive native files between incompatible hardware and different operating systems.

The SET HOST Command

The VMS command `SET HOST` provides a means of connecting your terminal to another system. It allows you to communicate through a DECNET node or a hard-wire from your serial port.

To use a DECNET node, the syntax for `SET HOST` is:

```
$ SET HOST node
```

where *node* is the number of a DECNET node. To communicate with another machine through a serial port, the syntax is:

```
$ SET HOST/DTE TXAn
```

where `TXA`*n* is the name of a serial port on your machine. For details and a list of options to go with the SET HOST command, use the VMS Help facility (`HELP SET HOST`). To terminate the connection, hit Ctrl-\ (Control key and ASCII backslash).

The `SET HOST` command can be used to transfer data between other machines and the VAX by using the `/LOG` option. The `/LOG` option keeps a record of your session in the VMS file `SETHOST.LOG`. Once you are connected to the other machine, any data that you can display on your screen can be found in the `SETHOST.LOG` file.

Since the VMS data buffers can overflow when you are communicating at 9600 CPS, data can be lost using this transfer method. We recommend setting your terminal speed to 1200 CPS or less when using the /LOG option for data transfer.

For moving large data files from other systems, the Kermit utility, described in the next section, is a better tool.

Using Kermit

The file `[APL.RELn]KERMIT.EXE` distributed with the APL⋆PLUS System for VAX/VMS is a stand-alone program that uses the popular Kermit file transfer protocol defined at Columbia University. The Kermit program is distributed free of charge by STSC with the permission of Columbia University; it is not considered to be part of the supported APL⋆PLUS System. Columbia's policy on distribution of Kermit is reproduced in Appendix D. Users who require full Kermit documentation or the code for Kermit on other machines should contact Columbia University.

We have found Kermit an effective technique for transferring files between different UNIX systems, and between UNIX and non-UNIX systems such as VMS. We suggest that users who find Kermit useful make a contribution to Columbia University to help fund this valuable service to the computing community.

Kermit starts in "command mode," but can operate in either "connect mode," in which it serves as a terminal emulator for logging onto a remote system, or in the "send" or "receive" mode. When transferring files that contain other than ASCII text (for example, a source-level workspace transfer files) the file type must be set to binary.

Before Kermit can be used, the `KERMIT.EXE` file must be installed with `LOG_IO` privilege. A typical input sequence that will start Kermit is:

```
$ RUN KERMIT
C-kermit> SET LINE TXAn
C-kermit> SET SPEED 9600
C-kermit> CONNECT
$ [Return]
```

Kermit is sensitive to parity. If Kermit does not respond, check your parity setting for your terminal. For an HDS terminal for example, parity must be set to "space." When using a PC as a terminal, parity must be off, with eight-bit data and one stop bit.

For more information on the use of Kermit, use the Kermit help facility. The help facility is accessed by entering a `?` from the Kermit prompt.

INTERFACES

Chapter Seven *Using APL with Non-APL Programs*

This chapter discusses the variety of ways that you can use the APL⋆PLUS System to interact with non-APL programs, such as operating systems and other languages.

There are three basic techniques for communicating with other programs from APL. From simplest to most complex and interactive, they are: Native Files, ⎕*XP*, and ⎕*NA*. A section in this chapter is devoted to each.

The Native File facility is the loosest coupling with non-APL programs. You can read and write data to a native file with a set of built-in functions. Then you can leave APL, either with)*OFF*, or temporarily with)*CMD* or ⎕*CMD* and run the non-APL program using the data in the native file. Finally, you can read the result or any new data back into the APL workspace using ⎕*NREAD*. The system function ⎕*CMD* allows you to perform these steps under program control.

To run another non-APL process simultaneously with APL, the APL⋆PLUS System provides the external process interface, ⎕*XPn*. Each system function (⎕*XP*1, ⎕*XP*2, ⎕*XP*3, ⎕*XP*4, and ⎕*XP*5) is the complete interface to one external process, permitting as many as five external processes to be running at once.

The system function ⎕*NA* provides the facility to incorporate non-APL code as part of the APL workspace. This allows APL functions to be built from programs written in other languages.

7-1 Using Native Files

Native VMS files can be accessed from APL with the native file system functions. These system functions parallel the suite of component file system functions, such as ⎕*FCREATE* and ⎕*FREAD*, but have the letter *N* as a prefix (⎕*NCREATE* and ⎕*NREAD*) instead of the letter *F*.

For a brief tutorial on the fundamentals of native file use, see "Sample Handling of Native Files" at the end of Section 3-1.

Issuing DCL Commands

The system command `)CMD` permits the APL user to execute DCL commands from the APL environment, and then return to the active APL workspace. If no command is given, `)CMD` displays the reminder `type log to return to apl` and then accepts a sequence of DCL commands on subsequent lines until given the instruction `logout` in response to the VMS prompt. You are then returned to APL. The syntax is:

```
)CMD
```

or

)*CMD command*

where *command* is the DCL command to be executed.

The system function ⎕*CMD* provides a similar capability and can be used under program control. The effect of ⎕*CMD* is to temporarily exit from APL while preserving the contents of the active workspace and to execute the command given it, after which it returns to APL and excutes any code that follows. For details on the use of ⎕*CMD* and)*CMD*, see the *APL*PLUS System Reference Manual.*

The native file system and ⎕*CMD* can be used together to integrate an APL program with a non-APL program. The APL program could first prepare some data and store it in a file in the format required by the non-APL program. ⎕*CMD* could then be used to request the operating system to start executing the non-APL program and use the file previously created by APL as the input. The output generated by the non-APL program could be stored in a file and later retrieved by APL after the ⎕*CMD* task completes execution.

7-2 Interfacing APL to Non-APL Programs

The APL*PLUS System provides two distinct facilities for calling non-APL programs from the APL environment:

External routines mapped into the APL process's address space can be associated with APL names using ⎕*NA* and called as though they were defined functions. The facility is similar to that provided by the *FASTFNS* utilities in the APL⋆PLUS Mainframe System and by ⎕*NA* in IBM's APL2 Program Product.

External processes are complete modules that APL runs as subprocesses, exchanging data through VMS mailboxes. The facility is very similar to the external process interface used in the APL⋆PLUS System for UNIX.

The two facilities provide quite different capabilities. The appropriate facility to use depends upon the nature of the program being interfaced to APL. As a rule, external routines are used to make discrete utility subroutines available to APL programs. These may be useful runtime library routines or fast implementations of algorithms that are hard to express efficiently in APL. External processes are more suited to interfacing APL to other major subsystems, such as database managers or large graphics packages.

Comparisons of External Routines and External Processes

- External Routines are mapped into the APL process and share the same address space. They can, therefore, operate on objects in the APL workspace. In contrast, external processes execute in a separate address space, receiving copies of APL variables passed through a mailbox interface.

- The external routine interface is simpler to use than external processes, particularly if the routine already exists. Once ⎕*NA* has been used to describe the routine to APL, APL will automatically pass parameters of the correct type to the routine whenever it is called. The external process interface requires that the programmer understand the internal form of variables in the APL workspace and write code specifically to process APL variables.

- External routines must be linked into sharable modules and the full names of the resulting `.EXE` files must be defined as logical names to VMS. External processes need not be sharable and are invoked by specifying the fully qualified name of the `.EXE` file.

- External routines are associated with a monadic APL function in the workspace; each routine requires a distinct APL function name and uses a small amount of workspace. External processes are interfaced through a system function and have no effect at all on the APL workspace; the effect is more like defining a new system function.

- External routines are treated like user-defined functions by workspace operations such as `)SAVE`, `)LOAD`, `)COPY` and `)CLEAR`. External processes are not associated with the workspace and are not affected by workspace operations.

- External routines are unable to create new APL arrays, though it is easy to create arrays from APL whose values can be filled in by external routines. An external process always creates a new array each time it is invoked.

- Since external routines are part of the APL process, they cannot execute concurrently with APL. It is possible, however, to design an external process so that it can execute concurrently with APL. For example, APL can start a time-consuming computation, such as a large database transaction running in an external process, without waiting for the transaction to complete.

- External routines are restricted in the kinds of operating system services they may use, while no such restrictions apply to external processes. For example, if you wish to process RMS files from APL in ways that are not provided with built-in system functions, it is wiser to write an external process for this purpose. An external process can open and close files with no side effects on the files that have been opened by APL. Also, external processes can request memory allocation from VMS, and use variables to preserve information between calls. External routines generally use only their own automatic variables and the data areas passed to them as formal parameters.

7-3 Using External Processes

An external process is a program that runs concurrently with APL, but as a separate VMS process or task. The external process is created by the APL process and runs until explicitly terminated by APL or until the APL process itself terminates.

The program running as an external process is an independently compiled, executable module that can be executed under VMS. The external process creates a unique mailbox, QMB*xxxx*, which it uses to communicate with APL. In the mailbox name QMB*xxxx*, *xxxx* represents the external (child) process ID number and can be observed by issuing the VMS command `show logical/job`. Unlike the typical VMS program, however, the input and output are not ASCII text, but instead are APL variables, in the internal data representation used in the APL⋆PLUS System workspace.

APL communicates with an external process through a VMS mailbox. When ⎕*XPn* is called, it writes first its left and then its right argument into the external process's mailbox, and then it begins to read the mailbox. The external process is constructed so that it receives two arrays, computes a result array, and then writes that result array to the mailbox. APL reads the mailbox to obtain the array that becomes the explicit result of ⎕*XPn*.

Using External Processes

Up to five external processes can be running at once. The system functions ⎕*XP*1, ⎕*XP*2, ⎕*XP*3, ⎕*XP*4, and ⎕*XP*5 refer to these external processes. The monadic form of all the ⎕*XPn* system functions controls the interface. The dyadic form passes data to the external process and returns computed results to the workspace.

An external process is created by monadic ⎕*XPn*. If the right argument is a character vector, it is assumed to be the name of the program to be run.

One of the sample external processes distributed with the APL⋆PLUS System is named VTOM. To initate the process:

```
      ⎕XP1  '[APL.RELn]VTOM.EXE'
8192
```

The explicit result (8192) is the VMS ID for the new process. If the external process cannot be created, a two-element integer vector is returned. The first element is 0, and the second element is the VMS System Service Condition Value.

Once the process is running, dyadic ⎕*XPn* can be used to pass APL variables to it and to return the computed result. The VTOM process is designed to convert a blank-delimited character vector into a matrix.

```
          VECTOR←' ONE TWO THREE FOUR FIVE'
          MATRIX←'' ⎕XP1 VECTOR
          ⍴MATRIX
5 5
          MATRIX
ONE
TWO
THREE
FOUR
FIVE
```

Note that in this example the external process actually uses only its right argument. An empty vector is used as a dummy left argument since ⎕*XPn* must be called dyadically. The program VTOM will read but ignore the left argument.

You can query the name of the program running as an external process by calling ⎕*XPn* monadically with an empty argument:

```
          ⎕XP1 ''
[APL.REL1]VTOM.EXE
```

If the result is empty, no external process is presently running for the particular ⎕*XPn* system function:

```
          ⍴⎕XP1 ''
12                         (⎕XP1 is already being used.)

          ⍴⎕XP2 ''
0                          (⎕XP2 is currently available.)
```

To terminate the external process before the end of the APL session, freeing ⎕*XP*1 for re-use, use monadic ⎕*XP*1 with a numeric argument to signal the external process. In the case of VTOM, the untrappable "kill" signal (9) will work:

```
⎕XP1 9
0
```

The 0 indicates that the external process is no longer running and ⎕*XP*1 can be re-used. In the current release, the only acceptable numeric argument to monadic ⎕*XPn* is the scalar 9, which terminates the external process. Any other number produces a *DOMAIN ERROR*.

Debugging an External Process

While an external process is being developed, it is useful to use `fprintf` to display messages that trace the status of the program. The sample external process source file `VTOM.C` contains examples of this debugging technique. Output produced by `fprintf` is not a part of the logical screen image maintained by the APL Session Manager, so it disappears when the Refresh key is pressed.

If ⎕*XPn* indicates normal creation of a process, yet the external process does not function at all, the problem may be that system quotas have been exceeded preventing the system from starting the process. You can check to see if the process is running with the following command:

```
)CMD SHOW SYSTEM
```

The operating system should display the XP*nxxxx* process. If it does not, check your quotas with the VMS command:

```
)CMD SHOW PROCESS/QUOTAS
```

The open file, subprocess, paging file, and buffered I/O byte count quotas are affected by each external process. Additional information on the appropriate values for each of these quotas is provided in the *APL*PLUS System Installation Manual.*

Stopping an Orphan External Process

The VMS command `show system` displays a list of process IDs and names. The name of an external process is XP*nxxxx* where n represents ⎕*XPn* and the *xxxx* represents the process ID number of the parent process (APL). If APL has terminated abnormally, leaving an orphan external process still running, the external process is stopped by issuing the VMS command `stop/id=`*yyy*, where *yyy* is the process ID of the XP*nxxxx* process itself.

Writing External Processes

The programs that run as external processes must be specifically written for this purpose and conform to the input and output conventions of `⎕XP1`. These programs can be written in any suitable language, provided the resulting program is a VMS `.EXE` file. Typically, the external process is used as an interface to a library of subroutines supplied with a software package such as a database manager. The external process is a relatively simple program that forms a front end to the package, interpreting APL arrays as control information or data, and dispatching the appropriate subroutines.

Programs written to run as external processes need to obey the following conventions:

- All input read from the mailbox QMB*xxxx* will be provided by APL in the internal form of an APL variable. The structure of APL variables is explained in the file `VARMAC.H` distributed with the APL⋆PLUS System; this file also contains macros to be used for constructing and manipulating arrays.

- All output written to the mailbox must be in the same format as an APL variable.

Following is an outline of the structure of a typical external process. Sample external processes are provided with the APL⋆PLUS System in both executable and C source forms. Users who want to write their own external processes are encouraged to begin by studying the sample source files. In many cases, it is practical to modify one of the sample programs to perform the task desired.

Structure of an External Process

The general structure of a typical external process is described below. For additional details and coding techniques, examine the following files distributed with this APL⋆PLUS System:

• `VTOM.C`	C language source code for a vector-to-matrix external process called `[APL.RELn]VTOM.EXE`
• `FILE.C`	C language native file interface for VMS
• `VARMAC.H`	C language header file describing the structure of APL arrays as used in the external process interface and containing macros for accessing the fields within the array
• `ERRMACRO.H`	C language header file containing the error codes used to signal specific errors from ⎕*XP* in the APL environment.

Main Program - Initialization

- Set traps for any signals that may be sent by the application. If no traps are set, any signal will terminate the external process.

- Initialize other variables or open files as the application warrants. The mailbox is created for inter-process communication here.

Main Program - Main Loop

- Read the ⎕*XPn* left argument (an APL array) from the mailbox by reading up to 512 bytes into the local buffer. The first 4-byte field is a 32-bit integer indicating the size of the array in bytes, including the 4-byte field. If the array is larger than 512 bytes, read the rest of the variable into a suitably large buffer. If the read terminates with fewer bytes than expected, go to exit.

- Read the ⎕*XPn* right argument from the mailbox by the same technique used for the left argument. You must read the entirety of both arguments out of the mailbox, even if one is a "dummy" argument, to dyadic ⎕*XPn*.

- Interpret the content of the APL arrays by using the appropriate macros to access the datatype, rank, shape vector, and data values of the array.

- Perform whatever processing is appropriate.

- Construct the APL array that is to become the explicit result of ⎕*XP*1 by writing the appropriate values into an internal buffer variable. You must fill in the following fields:

  leading length (4-byte integer)
  descriptor (type) byte
  rank byte
  number of elements (4-byte integer)
  shape vector (4-byte integers)
  data value; padded as necessary to fill out to a 4-byte boundary
  trailing length (4-byte integer)

  The routine `initbuf()` in `VTOM.C` fills in all but the shape vector integer(s) and the data values. The 6 bytes between the leading count field and the descriptor field can have any value, but are best set to zero.

- Write the result array to the mailbox. APL will wait until the entire array has appeared in the mailbox before allowing ⎕*XP*1 to return it as an explicit result.

- Go to main loop.

Exit

Close the mailbox and terminate.

7-4 Using External Routines

After an external routine has been defined in the workspace with ⎕*NA*, it can be used as though it were a locked APL function. External routines are always monadic; they are called with a right argument whose shape is equal to the correct number of parameters for the external routine. Typically, the argument to an external routine is a nested array, in the form of a vector of arrays, in which each item corresponds to a parameter to the external function.

The explicit result of the external routine is a vector of those items specified as output variables by ⎕*NA*. If the non-APL routine generates an explicit result, such as a return code, and this result is described with ⎕*NA*, the result becomes the first item of the result returned to APL. Additional items in the result consist of any parameters designated as output variables, in the order in which they appear in the parameter list.

For example, suppose an external routine named `ISUM` exists, whose effect is to add up a vector of integers and return a scalar containing that sum. In C, the function could be expressed as follows:

```
int sum ( input, length )
int *input; /*address of first number in vector */
int length;        /*length of the vector */
{
int i;
int sum = 0;
        for   ( i=0;  i<length; i++ )   {
                sum += input[i];
        }
        return(sum);
}
```

Also, suppose that this routine has been compiled and linked, and that the resulting `.EXE` file has been defined as a logical name `SUMFNS` (see "Writing Your Own External Routines" for details). To bind this function to APL, ⎕*NA* would be used as follows:

```
      3 0 ⎕NA 'SUMFNS:∆SUM ISUM (*I4, I4) I4'
1
```

This tells APL to create an external function in the workspace named ∆*SUM*, which is bound to a routine named `ISUM` that can be found in the shared module with logical name `SUMFNS`. The explicit result of 1 indicates that the arguments to ⎕*NA* were well formed, but it does not necessarily indicate that the external routine can be called with these arguments. Since the specifications for parameters are not preserved in compiled code, it is extremely important to provide the correct information to ⎕*NA*. Unpredictable results, including an APL system crash, can be caused by incorrect use of external routines.

Once the external function has been bound to an APL name, it is for all practical purposes a locked function in the workspace. For example:

```
      )FNS
∆SUM
      ⎕NC '∆SUM'
3
      )SAVE SAMPLE
...
      )ERASE ∆SUM
      )COPY SAMPLE ∆SUM
SAVED...
```

As defined to APL in the above example, the function ∆SUM is called with a 2-item nested vector. The first item is the vector containing the number to be summed, and the second is a scalar containing the length of the vector. For example:

```
      ∆SUM (⍳5) (5)
15
```

It is often convenient to define a cover function for an external routine, for example:

```
    ∇ Z←SUM R
[1]   Z←∆SUM (R) (⍴,R)
    ∇
```

How APL Calls External Routines

After an external routine has been defined to APL with ⎕NA, it can be invoked by calling it as a monadic function and supplying the proper right argument. When an external routine is called, APL performs the following steps:

- Check the right argument to make sure it is a scalar or vector whose length matches the number of parameters that were specified with ⎕NA.
- Build a standard VAX architecture argument list containing data extracted from the items of the right argument.
- Locate the module and routine, mapping them into the APL process's address space if they are not there already.
- Call the routine, passing the argument list.

- When the routine returns, build an APL variable to contain the explicit result and return it to APL.

How Arguments are Passed to External Routines

When an external routine is called, APL constructs a standard VAX argument list from the right argument supplied to the external routine. The argument list consists of 4-byte parameters that contain either the data value itself (which is called "pass by value") or an address of the data ("pass by reference"). The maximum number of parameters that the external routine may have is 255.

If an argument is passed by value, APL requires the corresponding item of the external routine's argument to be a scalar, whose value is placed in the 4-byte field. Double-precision floating point items can be passed by value, taking up two successive 4-byte values. This is the same way that the VAX C compiler passes double-precision floats by value; most other VAX languages pass only single precision (4-byte) floats by value.

If the argument is passed by reference, APL passes the address of the first data value of the corresponding item of the argument. For example, if a routine has two arguments, both passed by reference (`*I4,*C1`), it would require a 2-item nested array as its argument:

```
ARG ← ( 99 100 101 102 ) ('ABCDEFG')
```

The external routine will be passed an argument list that contains the address in the workspace of the value `99` in `1⊃ARG`, and the address of the `'A'` in `2⊃ARG`.

Arguments of type `G0` (general APL objects), are always passed by reference. APL passes the address of the APL array itself rather than the address of its data, which begins a few bytes into the array. The external routine must be written with a knowledge of APL array structure. See the preceding discussion of External Processes for hints on APL array structure.

Because of the efficient form that the APL*PLUS System uses to create and represent nested arrays, the execution cost of forming a nested array argument to an external routine is negligible. There is very little overhead in building the argument in instances like the following:

```
T ← PROCESS (ARG1)(ARG2)(ARG3)(ARG4)
```

Automatic Conversion of Numeric Datatypes

APL uses three different datatypes to represent numeric values (boolean, integer, and floating point). If a numeric array is passed as the argument to an external function, APL will automatically create a copy of the array in which the values have been converted to the datatype that the external routine expects. For example, the function ∆*SUM* described above could be called with integer values in any datatype:

```
T ← ∆SUM (1 0 1 1)(4) ⍝ BOOLEAN VALUES
T ← ∆SUM (1 + 0 -1 0 0)(4) ⍝ INTEGER
T ← ∆SUM (1E-12 + 1 0 1 1)(4)  ⍝ FLOAT
```

Floating point values will be converted to integer only if they are within APL's "integer tolerance" of a whole number. Values that are not close to a whole integer produce an error:

```
      T ← ∆SUM  (1.5 2 3)(3)
DOMAIN ERROR
      T ← ∆SUM  (1.5 2 3)(3)
        ∧
```

Explicit Results for External Routines

The explicit result that an external routine returns to APL is formed from the result of the routine (if described in the ⎕*NA* description) followed by any parameters marked as output arrays with '←' in the ⎕*NA* description. If any output arrays are passed by reference, they will be returned as enclosed items of the result. If the routine is described to ⎕*NA* as not returning a result, by omitting a result specification, any result produced when the routine is actually called will be discarded. If no result and no output variables are specified, the external routine returns an empty numeric matrix (0 0 ⍴0).

Output arrays provide the mechanism by which an external routine can return an entire array to APL rather than a single scalar. For example, consider an external routine *REVERSE* that reverses a character vector. It expects three parameters -- the address of input vector, the address of the output vector, and the length:

```
3 0 ⎕NA 'SMPLFNS:REVERSE(*C1,*C1←,I4)'
```

When *REVERSE* is called from APL, its second parameter must be the array that it will modify to contain the reversed character vector. This array should be of the same shape as the first parameter, which is the input.

Since APL always makes a copy of an output array before passing it to the external routine, the simplest way to call this routine is to pass the same array as both arguments:

```
      TXT ← 'THE INPUT UNMODIFIED'
      TXTR ← ⊃REVERSE (TXT)(TXT)(⍴TXT)
      TXTR
DEIFIDOMNU TUPNI EHT
      TXT
THE INPUT UNMODIFIED
```

If the array returns more than one item in its result, strand assignment is convenient for breaking up the nested array result. Suppose *REVERSE* also returns a return code to be returned to APL:

```
      )ERASE REVERSE
      3 0 ⎕NA 'SMPLFNS:REVERSE(*C1,*C1←,I4)I4'
```

REVERSE will now return a 2-item vector:

```
      RC TXTR ← REVERSE (TXT)(TXT)(⍴TXT)
```

It is important that all arrays that might be modified by the external routine are marked with '←' by ⎕*NA*. Consider the effect of not marking the output array in the above example:

```
      )ERASE REVERSE
      3 0 ⎕NA 'SMPLFNS:REVERSE(*C1,*C1,I4)'
      TXTR ← ⊃REVERSE (TXT)(TXT)(⍴TXT)
      TXTR
DEIFIDOMNU TUPNI EHT
      TXT
DEIFIDOMNU TUPNI EHT
```

APL passed the same address as both the input and output, with the consequence that the external routine modified the input variable as well. Since no copy was made, the variables *TXT* and *TXTR* refer to the same array in the workspace.

Caution: Unless care is taken to ensure that only one variable name points to an array, it is unwise to modify an existing array in the workspace. For example,

```
A←B←C←⍳1000
```

creates a single copy of the array `⍳1000` and causes three variable names to refer to it. If the contents of *A* are modified by an external routine, by passing *A* by reference and not defining it as an output variable, *B* and *C* will be modified as a side-effect of modifying *A*. An example of when it is safe to modify a named array can be found in "Case Study: Vector to Matrix Conversion," in this section.

Errors Produced by Calling External Routines

The following errors may be produced at the time that an external routine is called:

RANK ERROR	The argument is not a scalar or vector.
LENGTH ERROR	The shape of the argument does not match the number of parameters described to ⎕*NA*.
DOMAIN ERROR	The datatype of an item of the argument does not match the type specified with ⎕*NA* and the data can not be converted to the required type. *DOMAIN ERROR* is also produced when APL is unable to map the external routine into its address space. This can occur for a number of reasons, the most common of which are:

- The logical name of the module is undefined.
- The `.EXE` file associated with the logical name does not exist.
- The `.EXE` file contains no entry point with the specified name.
- The module is not linked as sharable.

Future versions of the APL⋆PLUS System will permit passing arguments by VMS descriptor, a common practice in the VAX environment. In the current release, you have to write your own intermediate routines that accept the currently supported datatypes and set up calls to routines that expect parameters passed by descriptor.

Writing Your Own External Routines

External routines can be written in any suitable language, including FORTRAN, PASCAL, C, BLISS, as well as assembler. The only restrictions are:

- The routine must accept parameters passed by standard VAX argument lists.
- The routine must use parameters of types that are accepted by ⎕*NA*.

The procedure for writing, compiling, and linking a module that contains routines to be called from APL is summarized by the following example:

1. Edit the source file. Suppose an editor is used to create a "C" language source file SUMFNS.C with these contents:

```
int isum(a,n) /* sum an integer vector */
int *a;
int n;
{
int z = 0;
       while   ( --n >=0 ) z += *a++;
       return   (z);
}

double fsum(a,n) /* sum a float vector */
double *a;
int n;
{
double z = 0;
        while ( --n  >= 0 ) z += *a++;
        return  (z);
}
```

2. Compile or assemble the source file. In this instance, the DCL command to compile a C program is:

```
$ cc sumfns
```

3. Link the object file as a sharable module, specifying both routines as universal (externally visible) entry points:

```
$ link/share  sumfns, sys$input:/opt
                                    (User types as many linker
universal=isum                       options as needed and types
universal=fsum                       Ctrl-Z to indicate end.)
```

4. Define a logical name for the module:

```
$ define sumfns $dua0:[jgw.text]sumfns.exe
```

5. Run APL, and use ⎕*NA* to bind external functions to the routines:

```
      3 0 ⎕NA 'SUMFNS:ISUM(*I4,I4)I4'
1
      3 0 ⎕NA 'SUMFNS:FSUM(*D8,I4)D8'
1
```

6. Use the external routines:

```
      ISUM (1 2 3)(3)
6
      FSUM (1.5 10.4)(2)
11.9
```

Calling Runtime Library Routines

Routines in VMS runtime libraries can also be called from APL as external routines. For example, the VAX C runtime library routine `times()` returns information on how much CPU time the APL process has consumed. Its C syntax is:

```
int times (t)
int t[4];       /* where values will be placed */
```

The VAX C Runtime Library is assumed to be equated with the logical name VAXCRTL, so an external routine can be defined thus:

```
      3 0 ⎕NA 'VAXCRTL:CPU∆TIME TIMES (*I4←)'
1
```

The monadic function *CPU∆TIME* now can be called, passing a 4-element integer vector by reference. The function `times()` will write its result as this address:

```
      ρT ← ⊃CPU∆TIME ⊂ι4
4
      T
234 0 0 0
```

Case Study: Vector to Matrix Conversion

This example shows how to deal with the restriction that external routines cannot create new APL arrays but can only modify the values of arrays that have been passed to them. A program that converts a delimited vector to a matrix must examine its input before the shape of the result can be calculated; and the shape must be known before the output array can be created.

The problem can be solved by splitting the calculation into two external routines. The first examines the input, calculates the dimensions of the result matrix, and returns them to APL. An APL statement then uses reshape to create a matrix with the correct shape and datatype. The matrix is passed to the second routine, which fills in its values.

In C, the two routines could be defined with the syntax below. Function `vmshape()` calculates the shape needed for the result matrix and `vmfill()` fills in the values.

```
vmshape(input,len,shape)
char *input;     /* delimited character vector */
int len;            /* length of input vector */
int shape [2]; /* where #rows and #cols are placed
{
int nrows;  /* number of rows needed in result */
int ncols;  /* number of cols needed for
                        longest row */
/* ... scan input, calculate nrows and ncols ...*/
                             shape[0] = nrows;
                             shape[1] = cols;
}

vmfill (input,output,shape)
char *input;    /* delimited character vector */
char *output;   /* character matrix */
int shape[2]    /* shape of output */
{
  /* ... scan input, copy each delimited string
                          to a row of output ...*/
}
```

After these routines have been compiled and linked, they can be defined as external functions:

```
      3 0 ⎕NA   'VTOM:VMSHAPE(*C1,I4,*I4←)'
1
      3 0 ⎕NA   'VTOM:VMFILL(*C1,*C1←,*I4)'
1
```

The two external routines can be incorporated into a stand-alone APL cover function:

```
     ∇ MATRIX←VTOM VECTOR;SHAPE
[1]  ⍝ Converts a delimited character vector
[2]  ⍝    into a matrix.
[3]  SHAPE ← VMSHAPE (VECTOR)(⍴VECTOR)(2 2)
[4]  MATRIX ← VMFILL (VECTOR)(SHAPE⍴' ')(SHAPE)
     ∇
```

Note the use of the third parameter to *VMSHAPE*. A two-element integer array with arbitrary values (2 2) is created and passed to *VMSHAPE* to be filled in. Since the third parameter is designated as an output variable and included in the result, APL will create a second 2-element vector which is passed to *VMSHAPE*. A similar process occurs with the second element of *VMFILL*. In both cases, an APL expression is used to create a new variable which is duplicated to become the explicit result. Two arrays are created in each case but only one is needed.

Efficiency can be improved in this case by passing arrays that the external routine is expected to modify and not marking them as output variables:

```
      )ERASE VMSHAPE VMFILL
      3 0 ⎕NA 'VTOM:VMSHAPE(*C1,I4,*I4)'
1
      3 0 ⎕NA 'VTOM:VMFILL(*C1,*C1,*I4)'
1

    ∇ MATRIX←VTOM VECTOR;SHAPE
[1]   ⍝ Converts a delimited character
[2]   ⍝    vector into a matrix.
[3]    SHAPE←2 2
[4]    VMSHAPE (VECTOR)(⍴VECTOR)(SHAPE)
[5]    MATRIX ← SHAPE⍴ ' '
[6]    VMFILL (VECTOR)(MATRIX)(SHAPE)
    ∇
```

In this case, it is safe to allow the external routine to modify the contents of the named variable, since it is certain that the array is referenced by only one name. The usage is considerably more efficient.

PRINTING

Chapter Eight
Printing

Because of the wide variety of printers used in the VMS environment, both with the VAX and MicroVAX, the APL⋆PLUS System does not yet have any standardized printing utilities. As support for various printers becomes available, information will be added to this chapter.

How Do I Get APL Data to My Printer?

Use the existing printing facilities of your printer and VMS. The APL⋆PLUS System's Native File capability allows you to write your APL data into native files (see Section 7-1 of this manual). Then, submit a print job for the native file, just as you would for any other VMS file.

How Do I Get APL Characters to Print?

There are hardcopy terminals such as the LA120 that have APL character sets and can print files with APL characters.

Alternatively, you can print APL characters by using a PC (running the APL⋆PLUS PC System in terminal mode) as a terminal to the VAX. A slave printer attached to the PC can be made to print any characters that appear on the screen.

WORKSPACES

Chapter Nine *Supplied Workspaces*

The workspaces supplied with this APL⋆PLUS System can be grouped into several categories:

- The tutorial workspaces are *APLCOURSE* and *LESSONS*. They will help you learn and practice APL in conjunction with the manual *APL Is Easy!*

- The demonstration workspace, *DEMOAPL*, illustrates features of this APL system.

- The file transfer workspaces are *SERHOST*, *TRANSFER*, and *SLT*. *SERHOST* is a data transfer application used with *SERXFER* on personal computers.

- The utility workspaces are *DATES*, *FORMAT*, *INPUT*, and *UTILITY*. They contain helpful functions that you can use to format dates and data, ease conversion from other APL systems, and collect and process input. You'll find them useful when building your own programs and applications.

- The supplemental workspaces are *COMPLEX* and *EIGENVAL*. The *COMPLEX* workspace provides APL cover functions to perform complex number arithmetic. The *EIGENVAL* workspace provides functions to compute eigenvalues and eigenvectors.

Where relevant, overviews of certain workspaces are given in other chapters of this manual as follows:

- *INPUT*---Chapter 5
- *SERHOST*, *SLT*, and *TRANSFER*---Chapter 6
- *FORMAT*---Chapter 4

This chapter briefly documents the other supplied workspaces. Each section consists of an overview and a list of the functions grouped by subject. Detailed function descriptions can be found in Chapter 4 of the *APL⋆PLUS System Reference Manual.*

Each workspace contains a *DESCRIBE* function or variable containing online documentation for the workspace. When you first load one of the workspaces, enter *DESCRIBE* to augment the documentation in this manual.

9-1 The *APLCOURSE* Workspace

The *APLCOURSE* workspace is an interactive tutorial that creates drill and practice exercises using primitive monadic and dyadic functions with scalar or vector integer and floating-point data. The workspace corresponds to the *AIDRILL* workspace discussed in the APL text *APL: An Interactive Approach* by Leonard Gilman and Allen J. Rose (Wiley, 1984). All the functions are unlocked.

The functions prompt for selection of the APL functions on which you want to be tested. You can select several function types for a session. You must solve the problem and respond with a correct answer. You get three chances to solve the problem correctly. After three incorrect answers, the correct answer is displayed. You will be drilled on the use of the function until you solve the problem. If you want, you can receive a record of your performance.

The functions *EASYDRILL* and *TEACH* create the practice drills; the function *INSTR* displays instructions on using these functions. *EASYDRILL* and *TEACH* are quite similar in action. You first select the function and datatype. If vectors are used, you can also choose reduction. The *TEACH* function differs from *EASYDRILL* in its ability to use vectors of length zero and to create drills with scalar monadic, scalar dyadic, mixed dyadic, and mixed monadic functions. The variables *SM* and *SD* contain the scalar functions, and *MM* and *MD* contain the dyadic functions.

The variables *PLEASE*, *CHANGE*, *STOP*, and *STOPSHORT* halt execution of the current exercise. If you have difficulty solving the problem, *PLEASE* returns the solution and creates another problem using the same function type. *CHANGE* gives the correct solution and switches to problems of another function type. *STOP* ends the session and displays the record of your performance. The variable *SCORE* stores the results of your session. *STOPSHORT* terminates the session without displaying your score.

9-2 The *COMPLEX* Workspace

The *COMPLEX* workspace contains functions that perform complex number computations and structural manipulations that invert complex matrices. The workspace also includes a function that derives the complex roots of an analytic function of a single complex variable.

The summary function provides a synopsis of all of the functions in the workspace. You can display documentation for each individual utility function by entering:

EXPLAIN functionname

The following conventions are used throughout the workspace:

- The notation *aJb* is used to reference the complex number *a*+*b*i in the documentation and function descriptions. Purely imaginary numbers are referenced as 0*Jb*. Purely real numbers, *aJ*0, are abbreviated as *a*.

- An array of complex numbers is represented by a numeric array whose rank is one higher and whose first dimension is 2. The first coordinate represents the real part of the complex number and the second coordinate represents the imaginary part. For example, the complex scalar 3*J*4 (3+4i) is represented by the vector `3 4`. The complex vector `1J2 3J4 5 6J7 0J8` is represented by `2 5⍴1 3 5 6 0 2 4 0 7 8`. Likewise, an *m*-by-*n* complex matrix is represented by a 2-by-*m*-by-*n* numeric array.

- Functions that operate on arbitrary complex arrays are prefixed *CX*; functions that operate only on complex scalars (two-element numeric vectors) are prefixed *Z*.

- The term "alternating string" refers to a numeric vector with an even number of elements where the odd elements are the real parts of several complex numbers and the even elements are the imaginary parts of the complex numbers.

The *CX* function filters an APL array to ensure it is a valid representation of a complex array. It also takes an alternating string and organizes it as a representation of a complex vector. The *CXRESHAPE* function can be used to create higher dimensional complex arrays. Once you have established representations of complex arrays, you can apply a variety of computational functions to them.

For example, the function *CXTIMES* multiplies two complex arrays. To multiply the complex vector `1J2  3  4J5` by the vector `1J¯1  2J¯2  3J¯3`, you can execute:

```
        1 2 3 0 4 5 CXTIMES 1 ¯1 2 ¯2 3 ¯3
```

When both operands are complex scalars, use *ZTIMES*. For example, the statement:

```
        6 7 ZTIMES 8 9
```

multiplies `6J7` by `8J9`.

Other functions are available to compute:

- sums, differences, and quotients
- square roots, powers, logarithms, and exponentials
- trigonometric and hyperbolic functions and their inverses
- conjugates, magnitudes, and amplitudes
- inverses and transposes of complex matrices.

Use the *SUMMARY* function in the *COMPLEX* workspace to display both the name and syntax of each of these functions.

The *CXROOTS* function finds the roots of a function of a single complex variable. This function must be named *CXfn* and must be monadic and return an explicit result. Both the argument and the result must be a complex scalar (two-element numeric vector). After you define *CXfn* , execute:

roots ← *CXROOTS guesses*

where the argument is an alternating string or complex vector of guesses. The number of guesses supplied limits the number of roots that will be returned. The *CXstate* global variable controls the behavior of the *CXROOTS* function -- the maximum number of iterations, the convergence tolerance, the display of intermediated results, and so forth. Execute *CXSTATE* for a review and interpretation of the elements of *CXstate*.

9-3 The *DATES* Workspace

The *DATES* workspace contains functions for converting dates (vector to scalar or scalar to vector), formatting dates, verifying dates, and performing miscellaneous calculations and manipulations of dates based on the Gregorian calendar. Each function handles any number of dates in its argument, except for *DATESPELL* and *DSPELL*, which handle only a single date at a time. All functions that accept the vector form of dates require that the argument be in ⎕*TS* timestamp form. No error-checking is performed. Invalid arguments may produce errors or incorrect results.

Conversion

The *DATEREP*, *HOURREP*, *MINREP*, *SECREP*, *TIMEREP*, and *FTIMEREP* conversion functions change dates from a compact base form (for example, as the number of days or minutes that have elapsed since, or prior to, 0:00:00, 1 January 1900) to a vector form similar to the ⎕*TS* timestamp. Conversely, the *DATATBASE*, *HOURBASE*, *MINBASE*, *SECBASE*, *TIMEBASE*, and *FTIMEBASE* conversion functions change dates from a vector form to a scalar form representing the elapsed number of time units from 0:00:00, 1 January 1900.

Formatting

The *TIMEFMT*, *FTIMEFMT*, and *DSPELL* functions format dates in fixed formats. *DATESPELL* spells out and formats dates according to various options, and allows an hour offset to time zones other than Eastern. *TIMEFMT*, *DSPELL*, and *DATESPELL* allow formatting of varying time precisions.

Verification

The *DATECHECK* function verifies a date to ensure that it is a valid date.

Calculation

A variety of functions in this workspace perform calculations with dates. *DAYSDIFF* and *WKDAYSDIFF* calculate the number of days or weekdays between dates. *DAYOFWK* determines the numeric value (1-7) of the day of the week for a given date; *DAYOFYR* determines the numeric value (1-365) of the day of the year. *LEAPYR* determines whether a given year is a leap year. *DATEOFFSET* calculates dates by offsetting to past or future dates. *YMDTOMDY* converts dates packed as [*YMD*] to dates packed as [*MDY*], and *MDYTOYMD* performs the inverse operation.

9-4 The *DEMOAPL* Workspace

The *DEMOAPL* workspace contains functions that demonstrate the diverse capabilities of your APL*PLUS System. The *DESCRIBE* variable explains the purpose of each function. Use of this workspace is discussed in Chapter 1 of *APL Is Easy!* The *LIST* function displays an entire function using ⎕*VR* when a function name is supplied as the right argument. All functions are unlocked and commented.

Formatting Output

REPORT, *CALENDAR*, and *CALEN* demonstrate use of the report generation facility (see Chapter 7). A report typical of a business application is generated by *REPORT*. The report title is centered over the table by the *CENTER* function. *DATA*, *PARTS*, *QUAN*, *TITLE*, and *TOT* are created as global variables by *REPORT*. *CALENDAR* creates a calendar for any specified month of a given year after October, 1752. *CALEN* uses the *CALENDAR* function twelve times to display a calendar for an entire year.

Mathematical Operations

COMB, *PERMX*, and *PRIMES* use primitive APL system functions to perform complex mathematical options. *COMB* uses a recursive algorithm to create all combinations of a given number from a fixed set of numbers. *PERMX* generates a table of all permutations of numbers from ⎕*IO* to a specified number. *PRIMES* returns all prime numbers from 1 to a specified number.

Character Manipulation

The *PIGLATIN* function translates a character string to pig-latin. Extra blanks are removed.

9-5 The *EIGENVAL* Workspace

The *EIGENVAL* workspace contains a variety of functions to find eigenvalues and, optionally, the eigenvectors of real matrices. Both real and complex eigenvalues are derived. Different routines embodying different techniques can be used depending upon whether or not the real matrix is symmetric.

The summary function provides a synopsis of all of the functions in the workspace. You can display documentation for each individual utility function by entering

EXPLAIN functionname

9-6 The *LESSONS* Workspace

The *LESSONS* workspace contains all of the functions discussed in the tutorial *APL Is Easy! APL Is Easy!* explains APL concepts with concise, detailed examples. You should be able to complete the tutorial in 10 to 15 hours; using *LESSONS* will help you learn.

As you work through the tutorial, you can verify the accuracy of your answers by examining the solutions in *LESSONS*, or you can copy the desired programs into your workspace to save time. Although all the functions are in the workspace, you must create the variables. The solutions to the exercises are also in the appendix in the tutorial. The *DEMOAPL* workspace contains the functions mentioned in Chapter 1 of *APL Is Easy!*

9-7 The UTILITY Workspace

The *UTILITY* workspace contains a group of functions to display nested arrays in pictorial form. The primary function is *DISPLAY*. It will return a character matrix containing a pictorial representation of the array provided as its right argument. For example:

```
      DISPLAY ⍳¨⍳3

.→------------------.
| .→. .→--. .→----. |
| |1| |1 2| |1 2 3| |
| '~' '~--' '~----' |
'∊------------------'
```

DISPLAY is used extensively in Chapter 1 of the *APL*PLUS System Reference Manual.*

PERFORMANCE TIPS

Chapter Ten

Performance Tips

To ensure that your APL⋆PLUS System is running efficiently:

- Have enough memory for the APL interpreter and editor and the user's workspace to be resident concurrently.

- Install APL as a shared code segment if you have multiple user's working simultaneously.

These two considerations are discussed in Sections 10-1 and 10-2, respectively.

10-1 Use of Memory

The size of the APL interpreter itself is approximately 280,000 bytes. The APL workspace size can be increased or decreased using)*CLEAR*; the largest size obtainable is a function of the way VMS is configured at your site.

How Much Do I Need?

The amount of memory APL uses consists of two parts: a fixed amount representing the minimum storage necessary to run APL, and a variable amount that depends on how much storage you need to use for the workspace and full-screen editor.

The fixed amount is approximately 280,000 bytes, the size of the APL interpreter.

The variable part of the APL process memory is the dynamic memory that APL allocates to use for the workspace and the Session Manager/full-screen editor. The amount of memory used can be controlled by APL session parameters. Once APL is running, you can change the size of the active workspace with the system command)*CLEAR*. For example:

```
      )CLEAR 500000
CLEAR WS
```

This sets the active workspace size to 500,000 bytes. If the new size is larger than the former size, APL requests additional memory from VMS.

Every machine has an effective maximum workspace size; requesting more workspace than is available produces an error message:

```
      )CLEAR 5000000
INSUFFICIENT SPACE FOR WS
```

Section 10-3 contains suggestions on how to proceed if you cannot obtain a workspace as large as you need.

Avoiding Excessive Paging

The size of APL processes can be a major factor in the effective performance of the VMS system. The total amount of memory allocated to active processes can be larger than the amount of main memory in the computer. In this situation, VMS pages inactive processes to disk, copying them into memory to run them, then copying them back to disk.

Often, the paging of tasks to disk has no perceptible effect on APL performance. However, when too much process space is allocated at once, the machine may begin to "thrash," copying pages between memory and disk. The effect can be very frustrating to the APL user--the system will appear to freeeze for seconds at a time--and the disk drive will be constantly active.

If you have a problem with thrashing while APL is running, it may help to use smaller workspaces or reduce the number of concurrent APL users. But the problem is most likely to be resolved by consulting the hardware vendor for advice on whether VMS can be "tuned" to better handle large processes. Often the problem can be resolved simply by adding more memory, but the hardware vendor can best determine how to address your particular situation.

VMS has excellent virtual memory facilities, but as with any such system, performance can suffer if the facility is abused. As a rule, the larger the APL workspace you use, the more your execution peformance will be diminished by paging to disk. It is generally wise to keep workspaces small enough so that they fit completely in real memory.

10-2 Shared Code Segment

The exectuable file `[APL.RELn]APLX` is set up to use a "shared code segment" so that only one copy of the executable program is resident in memory at the same time, no matter how many APL processes are running at once. This greatly reduces the amount of main memory consumed by the APL interpreter, and also makes APL load faster if another user is running it already. Each user has his own private storage for data and workspace, and there is no danger of one user's APL usage conflicting with another.

The shared code segment technique works only when a central copy of the executable interpreter file is used by each user. Making private copies of the executable file defeats the benefits of the shared code segment.

If multiple users are likely to be running APL at once, the APL interpreter should be installed as a shared image program. Have only one copy of the interpreter resident in memory even if many users are running APL simultaneously. The procedure for installing APL in this fashion is included in the Installation Guide.

10-3 Avoiding WS FULL

There are several possible reasons why you mat not be able to obtain as large a workspace as you want. Allocating the requested amount of memory might cause your APL process to exceed the system-wide process size limit (the maximum size of any one process). Or, the total amount of space available for all VMS processes combined may be used up.

The process size limit can often be increased by adding memory to the computer and then re-installing or re-configuring VMS. Refer to your VAX system administrator's manual or consult with the hardware vendor to see if this is possible on your machine.

The total process space can also be increased by reconfiguring VMS to provide a larger "swap area", the region of the disk that is reserved for storing inactive processes.

10-4 Monitor Facility

An important step in improving the performance of your APL applications is identifying where the execution bottlenecks. The system function ⎕*MF* monitors an APL function and returns the accumulated CPU times for each line of the funtion. When applied within an application, it identifies areas of code which may benefit from optimization efforts.

See Chapter 3 of the *APL ⋆PLUS Reference Manual* for details on how to use ⎕*MF*.

10-5 Partial Compilation of APL Functions

This APL⋆PLUS System uses a technique of partially compiling a line of APL code when it is executed for the first time. The line is translated into an internal form of pseudo-code, which is then executed by the interpreter. This pseudo-code differs from the internal form used by most other APL systems in that it does not require syntactic analysis when it is executed. STSC research has shown that a substanial portion of the work done by a traditional APL system is spent in performing syntax analysis.

When a function is first defined in the system, the input representation of the function (called the source form) is stored in the workspace exactly as typed. The source form is not examined until the first time the function is executed, when syntactic analysis is performed and pseudo-code is generated. The pseudo-code is then stored in the function, along with the source code, and is executed the next time the function is executed. The pseudo-code needs to be regenerated only if the syntactic context on which the function depends has changed. In practice this happens only with a change in valence of a user-defined object used in the APL statement.

Implications of this partial compilation include:

- A function runs faster after the first time it has been executed, since it needs syntactic analysis only once. The function also grows to occupy more workspace area as the pseudo-code is generated.
- When a function is displayed with the Del editor, ⎕*VR*, or ⎕*CR*, or is edited with the full-screen editor, the display form is the same as entered by the user.
- System functions such as ⎕*VR* and ⎕*DEF*, which maipulate the source form of function, are relatively fast.
- Copying a function into the workspace, or redefining it with ⎕*DEF*, ⎕*VR*, or the editor causes the pseudo-code to be discarded. Pseudo-code is preserved in a saved workspace.
- Derived functions that use non-scalar or user-defined functions are particularly well suited for representation in pseudo-code.

10-6 Multiple References

The APL⋆PLUS System makes copies of APL variables in the workspace only when absolutely necessary. In particular, using a variable as the argument to a defined function does not cause an extra copy to be made. Also, assignment to one variable of the unmodified value of another variable does not cause an extra copy to be made. For example:

```
A←B←C←D←⍳100
```

creates only one copy of the value (⍳100). The value is discarded only when the last reference to it is deleted, which occurs when a different value has been assigned to each of the variables or when all of them are erased.

Multiple references are particularly significant for nested arrays. For example:

```
A←1000⍴⊂⍳1000
```

creates only one copy of the value (⍳ 1000). The variable *A* consists of multiple references to the value, not multiple copies of it. This can yield a substantial space savings in many situations.

10-7 Dynamic Internal Structures

The data structures used by the APL workspace are all dynamic and their sizes can be easily changed. The execution stack (state indicator) extends automatically when necessary and releases the extra space when it is no longer needed. Thus, the error `STACK FULL` cannot occur, although `WS FULL` is possible if there is no room to extend the stack. Similarly, the symbol table is dynamic, and its size can be increased or decreased at any time using `)SYMBOLS`, even when names are in use in the workspace.

Since the symbol table is automatically extended by the system whenever necessary, we recommend beginning your session with only a small symbol table, preferably the default of 512.

GLOSSARY

Glossary

abort

Interrupt and abandon execution of a program or a session that is in progress.

absolute tab

A cursor position that is specified relative to the left margin.

absolute value

The magnitude of a number, regardless of any negative sign that might be present. The APL monadic function | returns the absolute value of the number or numbers in its argument.

access matrix

A matrix associated with each APL component file. The matrix consists of three columns containing the user account numbers of those who have been given access permission of some type, the sums of the access codes granted to each user, and the access passnumbers (if any) required of each user.

account number

An integer (ranging from 0 to 32,767) that identifies a user account. The user provides this number to the system at the beginning of each APL session. The user account number is the value of ⎕*AI*[1]. User account numbers appear in the first column of file access matrices.

accounting information

Information consisting of the user account number, the CPU time consumed by the APL session, and the elapsed time since the start of the APL session. This information is returned by the system function ⎕*AI*.

active workspace

The workspace that is currently in the computer's CPU memory. The last workspace loaded by the user. If the user has just begun an APL session, the active workspace is either *CLEAR WS* or a workspace specified as a start-up parameter.

address

An identification that designates a particular location in the computer's memory.

algorithm

A set of well-defined rules for solving a problem in a finite number of steps.

Alt key

The shift key for which the red key label has been provided.

ambivalent function

A function that can take either one argument (monadic) or two (dyadic).

APL file

A collection of sequentially numbered APL arrays (called components)e stored on disk. APL files are created and manipulated by system functions whose names begin with ⎕*F*.

application program

A program that enables the computer to perform a specific job. It is usually written in a high-level language such as APL, and often includes parts written in other languages.

argument

The data on which a function acts.

array

An arrangement of elements into zero, one, or more coordinates (dimensions).

array shape

See **shape of an array**.

ASCII

The acronym for the American Standard Code for Information Interchange, used as the standard code for information interchange among data processing systems, communication systems, and associated equipment. It uses a coded character set consisting of seven-bit coded characters with an eighth bit available for parity checking.

assembly language

A low-level symbolic programming language, closely resembling machine-code language, that allows a computer user to write a program using mnemonics instead of numeric instructions.

atomic vector

The vector that contains all of the possible character values in an APL system. These character values can be referenced using the system function ⎕*AV*.

attention

An interrupt issued by pressing Break while an APL statement or function is being executed.

attributes

The visible traits associated with displayed characters (colors, highlighting, inverse video, blinking, and so on).

autorepeat

A feature that allows you to hold a key, producing the same result as pressing the key multiple times.

backspace character

A non-graphic character that causes the cursor to move one position to the left. Available in APL⋆PLUS Systems as ⎕*TCBS*.

backup

An extra copy of stored data that can be used in case of accidental damage to or inadvertent erasure of the original (caused by either human error or machine failure).

baud rate	The speed of transmitting data to peripherals or other computers. See also **BPS**.
bell character	A non-graphic character that produces a bell sound on many ASCII devices. Available in APL⋆PLUS Systems as ⎕*TCBEL*.
binary	A number representation system using the base (radix) two. Numbers are written using only the digits 0 and 1, where each additional place to the left represents an increasing power of 2, just as in normal decimal numbers each place represents an increasing power of 10.
bit	Shortened form of binary digit. The smallest unit of data on computers. A single character in a binary number (that is, 0 or 1).
bit-pairing	The mapping of the APL characters onto the keys of a bit-pairing terminal. Compare to **typewriter-pairing**.
Boolean	Arrays in which all values are either 0 or 1.
boot	See **bootstrap**.
bootstrap	A procedure for loading a program (usually the operating system) into a computer as a result of preliminary instructions. This procedure is initiated by a switch or by a keyboard input command. Also called "IPL."
BPS	Abbreviation for "bits per second". A measure of data transmission speed showing the number of bits of information that pass a given point in one second. Bytes per second in ASCII transmissions is usually approximately BPS divided by 10.

branch potential

The indication of whether an APL statement has terminated with a successful branch. One of three potentials associated with an APL statement after it has executed. Contrast with **display potential** and **value potential**.

branch statement

A statement in a defined function which changes the order of execution of statements in that function. See **conditional branch** and **unconditional branch**.

branch target value

The line number to which a branch is to be made.

Break

A signal sent from the keyboard to indicate that current processing should be interrupted, or that the line being entered should be abandoned. It is signaled in this APL⋆PLUS System with the Interrupt key.

Break signal

The result of pressing Break.

buffer

A fixed-size internal storage area dedicated by the system to a particular purpose, such as holding data received from a port until it is used.

byte

A sequence of adjacent bits used as a unit of data. One byte usually contains 8 bits.

calling environment

The environment from which a function is called upon to execute and to which the computer returns after its execution.

canonical representation

The representation of a given function formed by converting the function into a character matrix in which each row is a line of that function. The explicit result of ⎕*CR*.

caret

In an error display, a pointer where the problem was encountered. Represented by the caret symbol (^).

catenate

To join two arrays to form a single larger array.

character

Any symbol, letter, digit, or punctuation mark used to control or represent data.

character constant

A series of characters enclosed between single quotes.

character data

Data, consisting of symbols (visible or not) used by the keyboard, screen, communications, or printing devices (also known as literal data). Each symbol occupies one byte of storage and is found among the 256 unique elements of the atomic vector (⎕*AV*). One of two datatypes in APL. Contrast with **numeric data**.

character image

A representation of a number using the characters "0123456789*E*. ¯" along with its spacing on the page.

character input mode

The input mode invoked when ⍞ executes. The system provides no distinguishing prompt for the user entry. The result is a character vector of the user entry (possibly preceded by a program-issued prompt) with no execution or command processing of the user entry. Character input mode is normally ended by pressing Enter, but can be aborted by using the O-U-T keystroke (typically Ctrl-Z). Contrast with **evaluated input mode**.

clear workspace

An active workspace created by the)*CLEAR* command. It has no workspace name and contains no user-defined functions or variables. All workspace-related system variables have their default values.

clock

A device capable of generating signals at periodic intervals.

command

A word related to a particular computer operation that can be used alone or with additional information to cause that operation to be performed. Most commonly used for the operating system or for APL system commands.

compaction

Reclaiming disk storage space abandoned in a component file by replacing file components with components of a different size.

comparison tolerance

The limit within which two APL values being compared are judged to be close enough to each other to be considered equal. The amount is controlled by the user through ⎕*CT*. Also referred to as "fuzz."

component

See **file component**.

component file

See **APL file**.

component information

The numbers returned by ⎕*FRDCI*, representing bytes needed to store the component in the workspace, the user account number that placed this data in the file, and the timestamp of when it was placed in the file.

component number

The number used to refer to a file component, including all uses of file system functions that refer to individual components (⎕*FREAD*, ⎕*FREPLACE*, ⎕*FRDCI*). This number is assigned to the component when it is first appended by ⎕*FAPPEND*. The number is one greater than that of the component just before it. The first component appended to a new file is numbered 1.

compound statement

Two or more statements written on a single line and separated from one another by diamond symbols (◊). Compound statements are executed in order, from left to right.

computer program

A set of instructions (called "statements" in APL) expressed in a form suitable for execution by a computer. Also called a "program."

conditional branch

A branch statement whose effect is dependent on some specified condition. The argument to a conditional branch may evaluate to a numeric array whose first element is used as the destination or to an empty array, resulting in no branching being done. Contrast with **unconditional branch**.

conformability

The required rank and shape relationship of the array arguments of a function.

constant

An array whose content is specified at the point of call (where it is to be used), rather than referring to another source for the data (such as a variable). See **character constant**, **numeric constant**, **variable**.

control character

A character produced by pressing a key while holding down the Ctrl-key.

control key

The key labeled "Ctrl."

conversion type

Specifications for interpreting and storing the data bits read from a native file.

coordinate

An axis along which data is arranged in an array, allowing indexing (subscripting) selection. To select a single element of data, a single index must be specified for each coordinate. The lengths of all coordinates of an array are given in the array's shape vector.

CPU

Acronym for Central Processing Unit. The part of a computer that controls the interpretation and execution of instructions.

crash

Also called a system crash, is a failure of either the hardware or the software. A crash leaves the system in an unusable state. The system must be restarted either by re-booting or by powering off and then on again.

CRT

Acronym for cathode-ray tube. A television-like device used for displaying text and possibly graphic images.

Ctrl

A shift key (also called control) used to enter control codes from the terminal keyboard. For example, Ctrl-S means that the user presses the S key while holding down the Ctrl key.

current directory

The directory that the operating system is currently using as the default directory.

cursor

Position where the next input or output character will appear on the screen. Indicated during input by a blinking underline on the screen.

data

A collection of numbers or characters. Datatypes are classified as numeric or character. See **character data** and **numeric data**.

database

Data items stored in order to meet the information processing and retrieval needs of an organization. The term implies an integrated file or files used by many processing applications as opposed to an individual data file for a particular application.

datatype

See **character data** and **numeric data.**

decimal

The number representation system using the base (radix) 10. Compare to **binary** and **hexadecimal notation.** Also, a synonym for decimal point.

decoration

An optional modifier used as part of the left argument to the APL⋆PLUS System function ⎕*FMT*. Displays the specified text with numeric data.

default

A value used by a system or language translator when no other value has been specified by the user or the user's program.

default disk drive

The disk drive that DOS uses when no disk drive has been specified. The letter representing this disk drive followed by a separator character is used as a prompt in DOS.

default library

The library used when no library number has been provided.

default value

A predetermined value used when no explicit choice is made or when no explicit action is taken to set the value.

default width

When the width specification of the dyadic format primitive is `0`, or when the width is unspecified, the default width is used to format the data. The default width is equal to one more than the widest representation of any value in any of the columns of the right argument whose formatting is controlled by that specification.

defined function

An APL program defined by the user while in definition mode or by using ⎕*DEF*, ⎕*DEFL*, ⎕*FX*, or the full-screen editor. A defined function consists of a header line (that formally specifies the function name, any arguments, and any result) and lines of APL statements. One of three function types in APL. Also called **user-defined function**. Contrast with **primitive function** and **system function**. See also **function**.

defined name

An APL identifier currently being used as the name of a function, variable, or label in the workspace. A name that does not produce a *VALUE ERROR* when referenced.

definition mode

See **function definition mode**.

defn error (definition error)

An error occurring when the system is unable to alter a defined function in the way requested.

delimiters

With regard to ⎕*FMT*, pairs of symbols used to open and close *text* and *pattern* phrases.

diamond symbol

The symbol ◊ used to separate the individual statements in a compound statement.

digit

A graphic character that represents an integer, one of the characters 0 through 9. Also called **numeric character**.

digit selectors

Special characters used in the g format phrase to indicate where digits in the data are to be displayed.

dimension

The number of elements along a single coordinate of an array. The maximum size of the number and arrangement of elements in an array. Also called **coordinate**. See also **shape of an array**.

directory

System table space used to control data storage and retrieval in a file. A table of identifiers and references to the corresponding items of data. Also, DOS's table of information on a group of files, possibly the entire contents of a disk.

directory mode

A mode of the system where files and workspaces in different directories are identified by prefixing the name with the operating system's directory or path designation. Contrast to **library mode.**

disk

A non-volatile, rotating, data storage device whose circular surface is coated with magnetic material.

disk drive

A hardware device that rotates a disk, permitting electromagnetic reading and writing of data.

diskette

Synonym for a disk (mini-disk). See **disk**.

display

A visual presentation of data.

display potential

The indication of whether the value of an APL statement is to be displayed. If the statement has no value, display potential is undefined. One of the three potentials associated with an APL statement after it has executed. Compare **branch potential** and **value potential**.

domain

The set of arguments (or of pairs of arguments) for which a function is defined.

domain error

An error occurring when a function is executed with one or more arguments not in its domain.

dyadic function

A function that requires both a left and a right argument.

edit

To add to, change, or delete from an existing program or set of data.

editing format phrases

Format phrases that convert data in the right argument of ⎕*FMT* into characters in the result.

elements

The individual items in an array. In an array of character data, each character is an element. In an array of numeric data, each number is an element.

empty array

An array having one or more dimensions of length zero. The number of elements in an empty array is zero.

empty vector

An empty array of one dimension (⍳0 or '').

Enter

The key that you press to indicate that you have finished typing a line of input and that the line should now be processed. Called "Return" on some computer manufacturers' keyboards.

error

A statement or command that cannot be executed, usually due to incorrect construction, unacceptable values, or hardware constraints or failures.

error handling

The use (via ⎕*ELX*) of preplanned responses to errors encountered in statement execution.

error message

See **error report**.

error report

A message produced by the system when it encounters a statement or system command it cannot execute.

escape character

A non-printing character, typically with transmission code 27, accepted by some communicating devices as a signal that subsequent characters are commands to alter option settings in the device.

evaluate

To calculate. See also **execute**.

evaluated input mode

The input mode when ⎕ executes. The system prompt in this mode is ⎕: . User entry is evaluated as an APL statement whose last explicit result is the value used as input. Evaluated input mode is normally ended by typing Enter. Evaluated input mode can be aborted by typing Exit or by entering a branch arrow (→) without an argument. Contrast with **character input mode.**

execute

In computer programming, to interpret a computer instruction and carry out the operations specified by the instruction. Also called **run**. Also, an APL function, denoted by the symbol ⍎, that takes a character vector or scalar as its argument and evaluates (executes) the APL statement represented in the argument. See also **evaluate**.

execution mode

Synonym for **immediate execution mode**.

explicit coordinate

A coordinate to a primitive function that is specified by enclosing it in brackets. For instance, in +\[2]*A*, the explicit coordinate is 2. Contrast with **implicit coordinate**.

explicit result

A value produced as a result of expression evaluation available for use in subsequent expressions or for screen display.

exponential editing

Type of formatting that displays all numbers as multiples of a designated power of ten. You specify the number of significant digits to be displayed.

expression

Identifiers, constants, and/or primitive functions and operators in syntactic combination. See also **latent expression.**

extension

See **file extension.**

field

A set of consecutive columns in the result controlled by an editing or *text* format phrase (with ⎕*FMT*), or a format pair (with dyadic format).

field width

The number of consecutive columns in the result controlled by one format phrase (with ⎕*FMT*) or one format pair (with dyadic format).

file

A linear collection of related data records arranged on disk in a prescribed sequence. The user can communicate information to, and access information from a file using the APL⋆PLUS PC System's file system functions. In an APL file, a linear collection of arrays (called components). In a native file, a linear collection of characters (called bytes). See also **native file**.

file component

The fundamental storage reference point in an APL⋆PLUS System component file. Any APL array can be stored in a file component.

file errors

Errors that occur during the execution of an instruction that requires a reference to a file on disk. May be concerned with a particular file or with a disk or disk drive.

file extension

The period and zero to three alphanumeric characters following the name of a DOS file (also known as file type).

file identification

The combination of library number and file name (or in the case of native files, a library letter, colon, file name, period, and optional extension) that identifies a file.

file name

The name that identifies a file. It consists of one to eight alphabetic and numeric characters, beginning with an alphabetic character.

file operations

All system functions that query or alter the files on the disks. For example, bringing components into the active workspace for processing or saving values generated in the active workspace as components of a file.

file passnumber

A number appearing in column 3 of a file access matrix that is used to control access to the file. Also called **passnumber**.

file reservation

The space reserved on a disk for future growth of a particular APL file. Also called **file size limit**.

file size limit

The size beyond which an APL file is not allowed to grow. Also called **file reservation.**

file tie

The arbitrarily established and temporary link that pairs a number to a particular file so that the number can be used to represent the file in the numeric arguments to file system functions.

file tie number

A unique number used for referencing a file while it is in active use.

file tie quota

The maximum number of files that can be tied concurrently.

file type

See **file extension.**

fixed point editing

Type of formatting that displays all numbers with the decimal point in a fixed position. If necessary, values are padded with zeros or rounded to comply with the number of decimal positions specified.

floating-point

A number able to have a fractional part. Also, the internal storage format used for such numbers.

floppy disk

A storage medium that consists of a flexible disk (diskette) of oxide-coated mylar stored in a paper or plastic envelope. The entire envelope is inserted in the disk drive. Also called **diskette**. See also **disk.**

format

The primitive function that forms numeric values into a character image. In the monadic form of format, the system chooses the width and precision of the image. In the dyadic form, the left argument controls the width and precision of the image. Represented by the thorn symbol (⍕). Also, to prepare a disk for accepting data.

format pair

A pair of numbers in the left argument of dyadic thorn that control the format field width and display precision when formatting one or more columns of data in the right argument.

format phrase

A sequence of characters in the left argument to *⎕FMT* that specify how a column of data is to be edited or positioned for display purposes. The specifications control the spacing, representation, precision, and decoration of the displayed data.

format string

A character vector containing one or more format phrases separated by commas that is used as a left argument to *⎕FMT*.

formatting

The process of arranging data for display purposes. Formatting specifications control the spacing, representation, precision, and decoration of the displayed data. Also, preparing a disk to accept data.

fractional part of a number

The part of a number to the right of the decimal point.

function

A named procedure (program or subroutine) that specifies how a job is to be done. A function is classified as primitive (for example, +), system (for example, ⎕*FREAD*), or defined (for example, *ROWNAMES*). See **primitive function, defined function**, and **system function.**

function definition

The rule or algorithm by which a user-defined function is to work. A user-defined function is entered, edited, and displayed in function definition mode or in the full-screen editor. The system functions ⎕*DEF*, ⎕*DEFL*, and ⎕*FX* can also define functions. See **function definition mode.**

function definition mode

Entering, modifying, or displaying a **defined function** using the on-screen editing facilities. Compare **immediate execution mode.**

function header

The initial line of a defined function that names the function, models and determines its syntax, and names any identifiers that are local to the function. Referred to as line [0] when editing or using ⎕*CRL* or ⎕*DEFL* .

function line number

See **line number.**

function type

Defined functions can either return or not return explicit results. They can also be niladic, monadic, or dyadic. See also **dyadic function**, **monadic function**, and **niladic function.**

fuzz

See **comparison tolerance**.

garbage

Unwanted data remaining from a previous operation that has not been erased from memory. Also, any useless or inaccurate data.

garbage collection

The rearrangement of the contents of memory that eliminates unwanted data (garbage) to reclaim space for new data.

global definition

The definition of a function or variable outside of the current (local) state of execution.

global environment

The collective term for the global definitions of all functions, variables, and system variables in the active workspace.

global variable

With respect to a level of function execution, any variable not local to that function.

hardware

The physical units making up a computer system (that is, the apparatus as opposed to the software or programs).

header

See **function header**.

hexadecimal notation

A notation of numbers using the base (radix) sixteen. The ten decimal digits 0 to 9 and the letters A through F are used to represent the digits 0 through 15, respectively, as single characters. One "byte" can be encoded using two hexadecimal symbols. Abbreviated and also called "hex."

identifier

Name of a function or variable.

idle character

See **null character**.

IEEE

Acronym for the Institute of Electrical and Electronics Engineers.

immediate execution mode

The method of operation in which any entered statement or system command is immediately executed. Immediate execution mode is the initial operating mode after a user has begun an APL*PLUS System session. Also referred to as **execution mode**. Compare with **function definition mode**.

implicit coordinate

A coordinate to a primitive function that is not specified explicitly. For instance, in `+\A`, the implicit coordinate is the last coordinate, while in `÷⌶α`, the implicit coordinate is the first coordinate. Compare with **explicit coordinate**.

implicit output

A value produced by an APL expression which is not re-used as an argument or assigned to a variable before the statement ends, and consequently is displayed.

inclusive

Including the numbers at each end of the range as well as the numbers between them.

index

A non-negative integer used to select an element along a coordinate of an array. Also called **subscript**.

index origin

The value that represents the first index position along a coordinate. The first number used in counting from the beginning. Can be either 0 or 1. Represented by `⎕IO`.

inport

A source from which input is received. For example, a serial port, keyboard, and screen.

input	Data to be processed and instructions to control processing entered into the internal storage of a computer system.
input buffer	See **type-ahead buffer**.
input/output	A general term for the equipment used to communicate with a computer and the data involved in such communication. Abbreviated and also called **I/O**.
input source	Where the input for an APL session is coming from -- usually the keyboard, but possibly an APL component file selected by ⎕*IN* to act as a surrogate for keyboard input, with each component containing one line of input (as built by ⎕*CAP*, for example).
integer	A number whose fractional part is zero. In this APL⋆PLUS System, an integer can range from ¯2147483648 to 2147483648 and be stored in four bytes. Outside that range, integer values are stored in eight bytes.
integer editing	Type of formatting that displays all numbers as integers. Each number is rounded (if necessary) and displayed without a decimal point.
integral part of a number	The part of a decimal number to the left of the decimal point.
integrated circuit	See **chip**.
integrity	Protecting data from errors in normal operation, such as accidental erasure or entering too many digits for a date.
internal precision	The number of significant digits provided by the internal representation of a number.

interpreter

A computer program that interprets and executes each source language statement before interpreting and executing the next statement. Often contrasted with a **compiler**, a program that translates an entire source language program into a machine language program without executing it.

interrupt

A signal, condition, or event that causes normal processing operations to be suspended temporarily.

I/O

See **input/output**.

keytop

The upper surface of a key on the keyboard and, by extension, the symbol or name printed there.

label

A name used to identify a line in an APL function. A label, followed by a colon, immediately precedes the rest of the statement. Labels are used in branching expressions as the values of the lines on which they appear.

lamp symbol

The symbol ⍝ used to separate a comment from a statement or a label. A comment preceded by a ⍝ can occupy a separate function line.

latent expression

A system variable (⎕LX) whose contents are executed immediately when a workspace is loaded. Represented by ⎕LX.

left argument

The value that appears to the left of a dyadic function.

length

The number of positions along a coordinate. See **shape of an array**.

library

A set of stored files and/or workspaces. In this APL⋆PLUS System, equivalent to an operating system directory.

library mode

A mode of the system where files and workspaces in different directories are identified by prefixing the name with a library number. The library numbers are assigned to a directory or path names with ⎕*LIBD* or a startup parameter. Contrast to **directory mode.**

library number

The number used to reference a library, particularly in the identification of an APL file or workspace, or in an inquiry regarding the contents of a given library.

line number

In a function definition, the number associated with a line.

linefeed character

A non-graphic character that, when displayed, causes the cursor to advance one line, but does not cause any horizontal motion. Represented by ⎕*TCLF*.

literal data

See **character data**.

local definition

The definition of a function at a particular state of function execution. Compare **global definition** and **most local definition**. If an object's name is not included in the header of any function in the state indicator, its local and global definitions are the same.

local environment

The collective name for all of the most local definitions of functions, variables, and system variables. An APL statement in a defined function can only use or modify the local definitions of objects. Compare **global environment**.

local value

The value of a variable at a particular state of function execution.

local variable

A variable that has a value only during execution of a defined function and that is explicitly localized in the function header.

location

An element of computer memory referred to by its address.

locked function

A defined function that cannot be edited or displayed. See also **function.**

loop

A closed sequence of statements performed repeatedly, usually until some test condition is met.

low-level language

A programming language that is machine-dependent, being translated by an assembler into instructions and data formats for a specific machine.

low-order digit

The rightmost (least significant) digit that appears in the representation of a number to a given precision.

machine language

The set of processing instructions native to the computer hardware, and therefore the only instructions the hardware can execute. Commands or instructions in higher level languages are translated into sometimes lengthy sequences of machine language by system software (such as the operating system command processor) and various computer language processors, such as this APL⋆PLUS System.

matrix

A rectangular arrangement of elements (numbers or characters). Each element requires two subscripts to identify it -- the first identifies the row, the second identifies the column. A two-coordinate or two- dimensional array, an array of rank two.

memory	Any device or medium capable of accepting and retaining data, so that data can be retrieved and used when needed. Often used more narrowly to refer to the computer's internal memory.
microsecond	One millionth of a second.
millisecond	One thousandth of a second.
minus sign	See **subtraction sign** and **negative sign.**
mnemonic	A name or symbol chosen to assist the human memory. For example, "fn" for "function."
modem	A device that permits the transmission of digital signals over analog transmission lines. Acronym for modulator-demodulator since it modulates and demodulates signals transmitted over communication facilities.
modifiers	The elements of format phrases that specify decorations and special effects with edited data.
monadic function	A function that takes only one data argument (appearing on its right).
most global definition	The definition of a variable or function at the state where there are no suspended functions (with a clear state indicator). Contrast with **local definition** and **most local definition**.
most local definition	The definition of a variable or function at the current state of function execution.

native file

Any operating system file considered as a stream of bytes and accessed with file functions beginning with ⎕*N* . Contrast with **APL file.**

negative sign

The sign used to indicate that a number is negative (for example, ¯8). This sign does not indicate an operation.

newline character

A non-graphic character that, when displayed, causes the cursor to move to the first column in the next line. Represented by ⎕*TCNL*.

niladic function

A function that takes no arguments.

non-graphic characters

Characters in the APL⋆PLUS System that, when displayed, do not have a visual representation.

non-negative

Numbers zero or higher.

null character

A non-graphic character that causes no movement of the cursor, but takes the same amount of time as displaying a character. Represented by ⎕*TCNUL*.

number

A quantity. Usable with arithmetic functions such as addition. The more limited set of numbers that can be stored in a computer can be expressed as numeric constants. Numbers are left unaltered by the APL identity function (for example, +5), while characters are rejected.

numeric constant

A number used directly from the keyboard or within a program, but not stored in a variable. It is formed of digits with the possible additional use of decimal point, negative sign, or even the letter *E*. A number is distinguished from the same character sequence within single quotes.

numeric data

A collection of numbers. One of the two datatypes in APL. See also **character data**, **elements**, **number**, and **numeric constant**.

operating system

An organized collection of software that controls the execution of computer programs and that can provide scheduling, debugging, input/output control, accounting, compilation, storage allocation, data management, and other related services. The operating system manages computer resources and provides services to programs. See also **system**.

operator

A primitive function operating on arguments that are themselves primitive functions to create another function. For example, the \ in + \ is an operator that combines with the primitive scalar dyadic function called addition to produce a primitive monadic function called addition-scan.

origin

See **index origin**.

origin-dependent

Affected by the value of the index origin.

outport

A destination to which output is directed. For example, a serial port, printer, or screen.

overstrike

A composite symbol formed by typing two symbols in the same character position. Also, by extension, symbols that can be formed that way even when they have been made available on a single key.

owner of a file

The account number by which the file was created or most recently renamed.

parallel port

A communications channel through which all the data bits of a byte pass concurrently (instead of consecutively). Typically used for impact printers. Compare to **serial port**.

parameter

A value specified for use by a program, software package, or operating system command, often from a limited set of acceptable values.

passnumber

An integer that can be used to extend the identification of a user. Used with files to exercise detailed control over file access as specified in the access matrix.

passthrough localization

Use of global values for localized system variables that have not yet been assigned local values.

pendent function

A function that is halted because of a suspension in another function that is called by this function. Pendent functions cannot be edited with function definition mode nor changed (under that name) with the full-screen editor.

peripherals

Devices that can be attached to a computer or terminal (for example, printers, disk drives, plotters, or microfiche viewers).

port

A communications channel between a computer and an external device such as a keyboard, a printer, or a communications line.

positioning format phrases

Format phrases (*X* and *T*) that change the appearance of the result of ⎕*FMT* by moving the cursor without reference to data in the right argument.

precision

The number of significant digits used to represent a number.

primitive function

A function built into the APL language and represented by a single non-letter symbol in APL symbol form.

print precision

The system variable ⎕*PP* that controls the number of significant digits (precision) of numeric output.

print width

The system variable ⎕*PW* that control the maximum number of character positions or print columns available to the system for displaying output.

program

(To develop) a set of sequenced instructions that cause a computer to perform particular operations, a plan to achieve a problem solution, or a routine. See also **application program**, **function**, and **operating system**.

programmed function key

A key that can be programmed to contain or perform varying functions. In the APL⋆PLUS System, the keys that can be programmed by ⎕*PFKEY* into multiple character sequences.

programming language

A language designed to express instructions in a form suitable for execution by a computer. Examples are APL, BASIC, COBOL, FORTRAN, and Assembly language. Also called "computer programming language."

prompt

A cue given to the user by a computer program asking the user to enter information.

protocol

A set of rules and conventions governing the communications between two or more devices. For example, XON/XOFF and RTS/CTS.

public comment

A comment at the end of a function line, beginning with ⍝∇, and retrievable even from a locked function by ⎕*CRLPC*.

quad functions	See **system functions.**
quad input	See **evaluated input mode.**
quad variables	See **system variables.**
query	A question directed to the user by a computer program, or vice-versa, to obtain specific information.
quote quad input	See **character input mode.**
radix	The base number in a number system. For example, the radix in the decimal system is 10.
RAM	An acronym for random access memory. Often, specifically a memory chip used with computers that can be read from and written on. Contrast with **ROM**. See also **memory**.
random link	The link (value of $\square RL$) used by the system's pseudo-random number generator.
rank	The number of dimensions (coordinates) of an array.
read	To obtain or interpret data from a storage device, data medium, or other source.
reboot	To reload the operating system and reinitialize all of the computer's internal memory, as by system reset. A final effort to regain control of the computer without powering off and back on. Loses all data in computer memory. See also **bootstrap**.

recursive function

A function that calls itself during execution.

register

An internal computer component capable of storing a specified amount of data (for example, one word). Registers hold the results of intermediate calculations.

relational function

A function used to compare two values that returns a result of either 1 or 0 to indicate true or false. See also **function**.

relative tab

A cursor position specified relative to the present position.

reset

To return components of a computer system to a specified starting state. See also **reboot**.

result field

See **field**.

return

To resume execution in, and pass a value back to, the calling environment (for example, the results of a function).

right argument

The value appearing to the right of a monadic or dyadic function name.

ROM

Acronym for read-only memory. Typically, a memory chip used with computers from which data can be read but to which no data can be written. Contrast with **RAM**. See also **memory**.

root directory

The main directory on a floppy disk or hard disk.

row (of a matrix)	A horizontal line of elements in a matrix or array of elements. Also, the first dimension of a two-dimensional array or, more generally, the next-to-last dimension of any array.
RS-232	The most common type of connection into a computer for a serial device, such as a terminal, modem, or plotter. Its characteristics are determined by a standard from the Electronic Industries Association.
run	To execute a particular computer program. Also, an execution of a program by a computer on a given set of data. Also called **execution**. See also **execute**.
saved workspace	A workspace stored in a library (on a disk).
scalar	An array of rank zero that must contain a single data element: a single number, or a single character. See also **scalar constant**.
scalar constant	A value coded explicitly, rather than being assigned to a variable. If numeric, a single number. If character, a single character enclosed in quotes.
scale factor	For the system function ⎕*FMT*, the amount by which a number is multiplied, expressed as a power of 10.
scan operator	Primitive facility that combines with any primitive scalar dyadic function to form a new monadic function. The new function forms successive elements of its result by using reduction to apply the scalar dyadic function to successive take (↑) operations of the right argument.

scope	With respect to an active state indicator, the collection of contiguous environments (function calls) in which a particular definition of an identifier persists.
scroll	The movement of text on a CRT (screen).
security	Limiting access to data based on established criteria (payroll information, for example).
seed value	See **random link.**
serial port	A communications channel through which each data bit passes consecutively. Typically used with terminals. Compare to **parallel port.**
session	A period of time during which a particular software system is in continuous use.
session-related system variables	A system variable whose value is not changed by loading or clearing a workspace. The values remain through an entire session unless explicitly changed. Contrast with **workspace-related system variables.**
shadowed	An APL object at a particular state of function execution that is inaccessible from the current state because an object with the same name is localized at the current state or at an intervening state. See **scope.**
shape of an array	A numeric vector whose elements are the dimensions of the array.
shared file	See **file.**

shift keys

Keys that alter the effect of pressing a character key.

significant digits

Those digits in a number which can be trusted not to have been distorted or rendered inaccurate by measurement, calculation, or the storage format.

singleton

An array containing a single number or character regardless of rank. Any elements in its shape vector are 1.

sink file

A file into which material destined for display on the screen is placed, either in addition to or in place of the screen.

software

A set of functions, procedures, and routines associated with the operation of a computer system. See also **hardware** and **program**.

spool

Acronym for simultaneous peripheral operations on-line. To write or read data to or from peripheral devices concurrently with execution of another program. For example, to spool data ready for printing so that it is printed while the computer is freed for other uses.

state indicator

A table in the active workspace that tracks the execution progress of defined functions. If function execution is halted, the state indicator shows the halted defined functions in order (the most recently active suspended function first) and the line in each function where execution stopped.

statement

A syntactically well-formed expression of APL primitive functions and operators, data constants, variables, and user-defined functions.

statement label

See **label**.

status line

See **system status line**.

steward

The individual or user account number that can add, delete, or modify the access of other user account numbers to a particular software application.

stopping

The use of the system function ⎕*STOP* to cause function execution to halt before executing specified function lines.

store

To enter data into or retain data in a device from which it can be retrieved at a later time.

strong interrupt

Two or more Breaks entered in quick succession during processing. A strong interrupt is recognized during the execution of an APL statement, and may halt execution before the entire statement has completed. Contrast with **weak interrupt**.

subprogram

Synonym for **subroutine**.

subroutine

A subsidiary routine called by the main program. Also called **subprogram**.

subscript

The value used to indicate the specific position(s) along a given coordinate of an array. Also called **index**.

subtraction sign

The sign used to indicate that the subtraction function is to be performed (for example, 5 − 3), represented by the hyphen.

suspended function

A function whose execution has been halted at some point because of an error, a Break signal, or a stop set by ⎕*STOP*. Suspended functions are marked in the state indicator by a ⋆. See **state indicator**.

suspended workspace

A workspace whose state indicator is not empty.

symbol table

The area of a workspace that stores the names of APL objects that have been referred to in the workspace.

syntax

In APL, the rules by which functions and their arguments are put together to form valid statements.

syntax error

An error occurring from an improperly formulated expression or an improperly called function. This may be caused, for example, by unmatched parentheses, by two variables juxtaposed with no function between them, or by a function used without correct arguments as specified in the function header.

system

The physical equipment and software used as a unit to process data. A system includes the central processing unit, its operating system, and the peripheral devices and programs under its control.

system command

An instruction used to direct the APL⋆PLUS System in performing certain housekeeping tasks. An instruction that allows the user to monitor and control the contents of workspaces, files, and libraries. Every system command begins with a "^". System commands cannot be used as part of an APL statement or defined function.

system constant

A constant value available from the system (like a system variable), but one that is not user-definable.

system functions

A special class of functions that can always be used from the active workspace. They occupy no storage in the workspace and do not appear in the)*FNS* list. System functions have names beginning with the quad symbol (⎕). Some system functions perform operations similar to system commands.

system status line

A line at the bottom of the screen that displays the current settings of such items as workspace identification, keyboard states, and printer activity.

system variables

A special class of variables, always in the workspace, that are used to monitor or control the workspace or session environment. They are distinguished by having names that begin with the quad symbol (⎕). See **workspace-related system variables** and **session-related system variables**.

text format phrases

Format phrases that insert characters into the result of ⎕*FMT* without referencing data in the right argument.

text insertion

Type of formatting used to insert specified text directly into the result field.

thorn symbol

The symbol ⍕ used to represent the format primitive function.

tie

To assign a unique integer to a file to allow for referencing the file when it is in active use. Also, the match of file to tie number so created.

tie number

See **file tie number**.

tied file

A file currently paired with a number (its tie number), and consequently available for processing. Currently tied files can be listed by ⎕*FNAMES* ⋄ ⎕*NNAMES*. See also **tie number**.

timestamp

A record of the date and time that some event occurred (for example, appending a file component).

tracing

The use of the system function ⎕*TRACE* to display the result of statements on specified lines.

translation

A character-by-character substitution to adjust for proper printing or for transmission to a device incapable of dealing with the original form.

translation table

An array of characters indexed by subscripts into a character set in order to map characters from one character set into a different character set. Such a table might be used to map ASCII character codes to corresponding internal APL characters, for example.

type-ahead buffer

An area in memory where typed keystrokes are collected until the system is ready to process them.

typewriter-pairing

A mapping of the APL characters onto the keys of a typewriter-pairing terminal. Compare to **bit-pairing**.

unbalanced quotes

An odd number of quotes (not counting any quotes in the optional comment) in an APL statement.

unconditional branch

A branch statement whose effect is always the same. Contrast with **conditional branch**.

user account number

The unique integer used by an APL⋆PLUS System to identify a user. The user account number is the value of 1↑⎕*AV*.

user-defined function

See **defined function**.

value

Data that can be assigned to a variable, specific data (either a constant, the contents of a variable or system variable), or the explicit result returned by a function.

value potential

The indication of whether an APL statement returned a value. One of the three potentials associated with an APL statement after it has executed. Compare **branch potential** and **display potential**.

variable

A named collection of data or a named array.

vector

A linearly arranged array, an array with one coordinate (dimension), an array of rank one, or a data structure that permits the location of any item by the use of a single index or subscript. See also **atomic vector** and **empty vector**.

video attributes

See **attributes.**

visual representation

The form of display of a defined function that includes leading and trailing del symbols (∇) and line numbers. The result of ⎕*VR*.

waiting function

A function called in the currently executing APL statement whose left argument is being evaluated prior to beginning the function.

weak interrupt

A single Break entered at the keyboard. A weak interrupt is recognized only at the completion of a line of a function. Contrast with **strong interrupt**.

window

A rectangular portion of the CRT screen, specified by four integers. The first two integers represent the location on the screen of the upper-left corner of the window. The second two integers represent the shape of the window (rows and columns).

window specification

A precise description of a particular window, given as an integer vector whose four elements represent the starting row and column (the position of the upper-left corner) and the size of the window.

workspace

The APL execution environment in which computation takes place and in which names have meaning. A workspace contains the variables, defined functions, and control information for an APL session. A repository for a collection of functions and data. Workspaces can be stored in libraries for later use. See also **active workspace** and **saved workspace**.

workspace identification

The library number and workspace name that identify a workspace.

workspace name

A sequence of one to eight characters, all of which must be capital letters or digits and the first of which must be a letter.

workspace parameter

See **system variables.**

workspace-related system variables

System variables associated only with a workspace. Their value is not preserved when another workspace is cleared or loaded, or when an APL session ends. See also **session-related system variables.**

wrap

To continue uninterrupted on a second line of the screen a line begun above.

wrap marker

The screen attribute used in the leftmost column of the current window to indicate that the line on which the marker occurs is part of a wrapped line begun above.

wrapped line

A line on a CRT screen that is longer than the width of the screen. The display of a wrapped line occupies more than one physical line on the screen or continues on the line or lines below its beginning.

APPENDIXES

Appendix A

System Characteristics and Limits

The characteristics and limits for this APL★PLUS System are listed below.

The character set is described in Appendix B of this Manual.

The following internal data representations are used:

• Boolean	one bit per element (range 0-1)
• Integer	32-bit 2's complement integers (range -2147483648 to 2147483637)
• Floating point	64-bit VAX-D format double precision (range: negative-number-limit to positive-number-limit)
• Character	8-bit characters
• Nested	four-bytes per element at outermost level (4-byte pointer)
• Heterogeneous	10 bytes per element (one byte datatype, one byte reserved, eight bytes to hold largest possible value).

In the following table, when a limit is given as "none," that there is no constant value that the system enforces as a limit. Available workspace area is the only limitation that applies in these cases.

Positive-number-limit:	1.0141183460469229*E*38
Negative-number-limit:	¯1.0141183460469229*E*38
Positive-counting-number-limit:	72057594037927936 (¯1+2*56)
Negative-counting-number-limit:	¯72057594037927936 (-¯1+2*56)
(Coordinate-length-limit) Index-limit:	2147483647
Length-limit:	2147483647
Rank-limit:	127
Identifier-length-limit:	100
Quote-quad-output-limit:	none
Comparison-tolerance-limit:	1*E*¯13
Integer-tolerance:	2*¯32
Print-precision-limit:	18
Full-print-precision:	18
Exponent-width-limit:	3
Indent-prompt:	6ρ' '
Quad-prompt:	'⎕:',⎕*TCNL*,6ρ' '
Function-definition-prompt:	'1234567890.[]'
Line-limit:	32767
Elements-per-array-limit:	2147483647
Printing-width-limit:	255
Symbols-limit:	32767
Input-line-length-limit:	1014
Function-line-length-limit:	none
Execute-argument-length-limit:	2147483647

Appendix B
System Character Set

The following table describes the characters that make up the Atomic Vector (⎕*AV*) for this APL⋆PLUS System. The table includes:

- decimal index in ⎕*AV* (origin 0)
- display form of the character
- the overstrike sequence (if any) that can be used to create the character
- the character's name
- the terminal required.

Your terminals may not be able to display all of the characters described here. Generally, this complete character set is available only on bit-mapped devices in which a custom font has been developed or on personal computers used as terminals in which the hardware has been modified to support APL (with a ROM or downloadable character set).

Characters that can require a special terminal (other than an ASCII terminal) are flagged with the following codes:

APL	An APL terminal is required to display this character.
extended	A terminal with an extended character set and a custom font that includes this special character is required.

Index	Char	Overstrike	Name	Terminal
0.			⎕*TCNUL*	
1.	☺	O U	smiling face	Extended
2.	≢	≠ _	inequivalent	Extended
3.	⍷	∊ _	epsilon-underscore	Extended
4.	◊		diamond	APL
5.	¨		dieresis	APL
6.	←		left arrow	APL
7.			⎕*TCBEL*	
8.			⎕*TCBS*	
9.			horizontal tab	
10.			⎕*TCLF*	
11.	⊂		implication	APL
12.			⎕*TCFF*	
13.			⎕*TCNL*	
14.	⊃		reverse implication	APL
15.	⍟	⋆ ○	logarithm	APL
16.	⌻	⎕ ∘	quad-jot	Extended
17.	⍂	⎕ \	sandwich	Extended
18.	⍸	⍳ _	iota-underscore	Extended
19.	⍫	∇ ~	del-tilde	APL
20.	ù		lower-case u grave	Extended
21.	⌶	⊥ ⊤	I-beam	APL
22.	⍬	0 ~	zilde	Extended
23.	⍵		omega	APL
24.	↑		take	APL
25.	↓		drop	APL
26.	→		right arrow	APL
27.			⎕*TCESC*	
28.	⊣		right tack	APL
29.	⊢		left tack	APL
30.	⍋	∆ |	upgrade	APL
31.	⍒	∇ |	downgrade	APL
32.			space	
33.	!	' .	shriek	
34.	"		double quote	
35.	#		number sign	
36.	$		dollar sign	
37.	%		percent sign	
38.	&		ampersand	

Index	Char	Overstrike	Name	Terminal
39.	'		single quote	
40.	(		left parenthesis	
41.	)		right parenthesis	
42.	*		power	
43.	+		plus sign	
44.	,		comma	
45.	-		minus sign	
46.	.		period	
47.	/		slash	
48.	0		zero	
49.	1		one	
50.	2		two	
51.	3		three	
52.	4		four	
53.	5		five	
54.	6		six	
55.	7		seven	
56.	8		eight	
57.	9		nine	
58.	:		colon	
59.	;		semi-colon	
60.	<		less than	
61.	=		equal	
62.	>		greater than	
63.	?		query	
64.	@		at-sign	
65.	*A*		upper-case A	
66.	*B*		upper-case B	
67.	*C*		upper-case C	
68.	*D*		upper-case D	
69.	*E*		upper-case E	
70.	*F*		upper-case F	
71.	*G*		upper-case G	
72.	*H*		upper-case H	
73.	*I*		upper-case I	
74.	*J*		upper-case J	
75.	*K*		upper-case K	
76.	*L*		upper-case L	
77.	*M*		upper-case M	
78.	*N*		upper-case N	

Index	Char	Overstrike	Name	Terminal
79.	O		upper-case O	
80.	P		upper-case P	
81.	Q		upper-case Q	
82.	R		upper-case R	
83.	S		upper-case S	
84.	T		upper-case T	
85.	U		upper-case U	
86.	V		upper-case V	
87.	W		upper-case W	
88.	X		upper-case X	
89.	Y		upper-case Y	
90.	Z		upper-case Z	
91.	[		left bracket	
92.	\		backslash	
93.	]		right bracket	
94.	^		logical and	
95.	_		underscore	
96.	'		left single quote	
97.	a	A _	lower-case a	
98.	b	B _	lower-case b	
99.	c	C _	lower-case c	
100.	d	D _	lower-case d	
101.	e	E _	lower-case e	
102.	f	F _	lower-case f	
103.	g	G _	lower-case g	
104.	h	H _	lower-case h	
105.	i	I _	lower-case i	
106.	j	J _	lower-case j	
107.	k	K _	lower-case k	
108.	l	L _	lower-case l	
109.	m	M _	lower-case m	
110.	n	N _	lower-case n	
111.	o	O _	lower-case o	
112.	p	P _	lower-case p	
113.	q	Q _	lower-case q	
114.	r	R _	lower-case r	
115.	s	S _	lower-case s	
116.	t	T _	lower-case t	
117.	u	U _	lower-case u	
118.	v	V _	lower-case v	

Index	Char	Overstrike	Name	Terminal
119.	*w*	*W* _	lower-case w	
120.	*x*	*X* _	lower-case x	
121.	*y*	*Y* _	lower-case y	
122.	*z*	*Z* _	lower-case z	
123.	{		left brace	
124.	¦		split stile	
125.	}		right brace	
126.	~		tilde	
127.			⎕*TCDEL*	
128.	Ç		upper-case C cedilla	Extended
129.	ü		lower-case u umlaut	Extended
130.	é		lower-case e acute	Extended
131.	â		lower-case a circumflex	Extended
132.	ä		lower-case a umlaut	Extended
133.	à		lower-case a grave	Extended
134.	≠		not equal	APL
135.	ç		lower-case C cedilla	Extended
136.	ê		lower-case e circumflex	Extended
137.	ë		lower-case e umlaut	Extended
138.	è		lower-case e grave	Extended
139.	ï		lower-case i umlaut	Extended
140.	î		lower-case i circumflex	Extended
141.	⌈		ceiling	APL
142.	Ä		upper-case A umlaut	Extended
143.	⌊		floor	APL
144.	É		upper-case E grave	Extended
145.	∆		delta	APL
146.	×		times	APL
147.	ô		lower-case o circumflex	Extended
148.	ö		lower-case o umlaut	Extended
149.	⎕		quad	APL
150.	û		lower-case u circumflex	Extended
151.	⍞	⎕ '	quote-quad	APL
152.	⌹	⎕ ÷	domino	APL
153.	Ö		upper-case O umlaut	Extended
154.	Ü		upper-case U umlaut	Extended
155.	¢	⊂ |	cent sign	Extended
156.	£	{ ⊥	British pound	Extended

Index	Char	Overstrike	Name	Terminal
157.	¥	Y -	Japanese yen	Extended
158.	⍪	- ,	catbar	Extended
159.	⍨	~ ¨	frown	Extended
160.	á	a /	lower-case a acute	Extended
161.	í	i /	lower-case i acute	Extended
162.	ó	o /	lower-case o acute	Extended
163.	ù	u /	lower-case u acute	Extended
164.	ñ		lower-case n tilde	Extended
165.	Ñ		upper-case N tilde	Extended
166.	⍝	∩ ∘	comment lamp	APL
167.	⍀	\ -	backslash-bar	APL
168.	¿		inverted query	Extended
169.	⎕	[ ]	squad	Extended
170.	⍡	⊤ ¨	snout	Extended
171.	⍢	∇ ¨	frog	Extended
172.	⍣	⋆ ¨	sourpuss	Extended
173.	¡		inverted shriek	Extended
174.	«		left guillemet	Extended
175.	»		right guillemet	Extended
176.	░		.25 shading character	Extended
177.	▒		.50 shading character	Extended
178.	▓		.75 shading character	Extended
179.	│		line-drawing character	Extended
180.	┤		line-drawing character	Extended
181.	╡		line-drawing character	Extended
182.	╢		line-drawing character	Extended
183.	╖		line-drawing character	Extended
184.	╕		line-drawing character	Extended
185.	╣		line-drawing character	Extended
186.	║		line-drawing character	Extended
187.	╗		line-drawing character	Extended
188.	╝		line-drawing character	Extended
189.	╜		line-drawing character	Extended
190.	╛		line-drawing character	Extended
191.	┐		line-drawing character	Extended
192.	└		line-drawing character	Extended
193.	┴		line-drawing character	Extended

Index	Char	Overstrike	Name	Terminal
194.	┬		line-drawing character	Extended
195.	├		line-drawing character	Extended
196.	─		line-drawing character	Extended
197.	┼		line-drawing character	Extended
198.	╞		line-drawing character	Extended
199.	╟		line-drawing character	Extended
200.	╚		line-drawing character	Extended
201.	╔		line-drawing character	Extended
202.	╩		line-drawing character	Extended
203.	╦		line-drawing character	Extended
204.	╠		line-drawing character	Extended
205.	═		line-drawing character	Extended
206.	╬		line-drawing character	Extended
207.	╧		line-drawing character	Extended
208.	╨		line-drawing character	Extended
209.	╤		line-drawing character	Extended
210.	╥		line-drawing character	Extended
211.	╙		line-drawing character	Extended
212.	╘		line-drawing character	Extended
213.	╒		line-drawing character	Extended
214.	╓		line-drawing character	Extended
215.	╫		line-drawing character	Extended
216.	╪		line-drawing character	Extended
217.	┘		line-drawing character	Extended
218.	┌		line-drawing character	Extended
219.	█		solid character	Extended
220.	▄		solid lower-half character	Extended
221.	▌		solid left-half character	Extended
222.	▐		solid right-half character	Extended
223.	▀		solid upper-half character	Extended
224.	α		alpha	APL
225.	ß		German double-S	Extended
226.	ι		iota	APL
227.	⍤	∘ ¨	hoot	Extended
228.	⍥	○ ¨	holler	Extended
229.	⍱	∨ ~	nor	APL
230.	⊥		base, decode	APL

Index	Char	Overstrike	Name	Terminal
231.	⊤		encode	APL
232.	⌽	○ |	rotate	APL
233.	⊖	○ −	rotate-bar	APL
234.	⍲	∧ ~	nand	APL
235.	⌿	/ −	slash-bar	APL
236.	∇		del	APL
237.	⍉	○ \	transpose	APL
238.	∊		epsilon	APL
239.	∩		intersection	APL
240.	≡	= _	equivalent	APL
241.	⍙	∆ _	delta-underscore	APL
242.	≥		greater than or equal	APL
243.	≤		less than or equal	APL
244.	⍕	⊤ ∘	thorn	APL
245.	⍎	⊥ ∘	hydrant	APL
246.	÷		divide	APL
247.	„	" ,	German open-quote	Extended
248.	∘		jot	APL
249.	○		circle	APL
250.	∨		or	APL
251.	⍴		rho	APL
252.	∪		union	APL
253.	¯		high minus	APL
254.	|		stile	APL
255.			undefined	

Appendix C
Error Messages

This chapter contains the error messages displayed by the system along with the probable cause of the error. Detailed explanation of the cause of an error may also be found in the description of the function or system command that created the error.

Error	**Explanation**
AXIS ERROR	An attempt has been made to select a non-existent or invalid axis (coordinate) of an array for use with a function or operator.
DISK ERROR	A VMS system file operation used internally by APL has produced an unexpected error. In most cases, the error message will include the VMS return code.
DOMAIN ERROR	The argument supplied is not the right type or has a value outside the acceptable range.
FILE ACCESS ERROR	You are not permitted to access this file, or the supplied passnumber is incorrect.
FILE ARGUMENT ERROR	The file name is incorrect. This often arises from using a directory name while the system is in library mode or a library number when in directory mode.
FILE DAMAGED	Internal inconsistencies have been found in an APL component file. This can be caused by writing to a component file with native file operations.

Error	Explanation
FILE DATA ERROR	The file component does not contain a well-formed APL array. The component was probably written by a non-APL program.
FILE FULL	The space required for this operation would cause the file to exceed its size limit.
FILE INDEX ERROR	The component number lies outside the range of valid component numbers for this file.
FILE NAME ERROR	A file with the same identification (*name*) already exists.
FILE NAME TABLE FULL	There is insufficient space available in the internal table used to contain file names.
FILE NOT FOUND	The file was not found in the library or directory specified (may be the default working directory if no library number or directory name is supplied).
FILE TIE ERROR	No file is tied to the tie number supplied.
FILE TIE QUOTA EXCEEDED	There is a maximum number of files that can be tied at one time. This operation would have exceeded this limit.
FILE TIED	Another user already has the file exclusively tied.

Error	Explanation
FORMAT ERROR	The format phrase provided to ⎕*FMT* is ill-formed. Up to 16 characters of the incorrect phrase is displayed preceding the *FORMAT ERROR* phase.
HOST ACCESS ERROR	The operating system's file permissions on this native file do not permit the file operation (see ⎕*NRDAC*).
INDEX ERROR	The index is not in the range of valid indices for this variable (check ⎕*IO*).
INCOMPATIBLE WS	The specified workspace was saved by a newer version of APL than the one you are using.
INCORRECT COMMAND	The system command is ill-formed. Some possible causes include: • A misspelled command • A library number has been used when the system is in directory mode • Miscellaneous and extraneous material following the command • The argument supplied is invalid
INSUFFICIENT MEMORY	There is not enough system memory available to complete the operation.
INSUFFICIENT PROCESS SPACE	Workspace size required for this operation would cause the APL process to exceed the operating system's configuration limit.
LENGTH ERROR	The argument does not have the correct shape. For example, it supplies five numbers when it should supply only four.

Error	Explanation
LIBRARY NOT FOUND	The specified APL library number is not defined in ⎕*LIBS*, or the directory referenced does not exist.
LIMIT ERROR	The number is out of the acceptable range of numbers or some other system limit has been reached.
NO SPACE FOR ⎕*DM*	There is not enough space in the workspace to record the diagnostic message.
NONCE ERROR	The supplied argument follows the design principles of the system, but for various implementation considerations is not valid in this version of the system.
NOT COPIED:	The indicated objects were not copied from the saved workspace because the active workspace is full, the symbol table is full, or the saved workspace is damaged.
NOT COPIED, WS DAMAGED	The copy operation could not be completed because the saved workspace is damaged.
NOT ERASED:	The requested objects were not erased from the workspace because they were pendent or suspended.
NOT FOUND:	The indicated objects were not found in the saved workspace.
NOT IN DEFN OR QUAD	This system command is not permitted while the system is in ⎕-input or function definition mode.

Error	**Explanation**
RANK ERROR	The argument supplied is of incorrect rank. For example, a matrix was supplied when the system was expecting a vector or a scalar.
SYNTAX ERROR	The statement supplied is not a valid APL statement.
SYSTEM ERROR	An error has occured in the APL interpreter code, possibly caused by damage to the workspace's internal data structures. The APL session is terminated after a system end.
VALUE ERROR	The object identifier (name) does not have a value in this workspace.
WS ARGUMENT ERROR	The workspace identifier is ill-formed or is too long to process.
WS DAMAGED	The source workspace is not in the correct form for a saved workspace, or a disk error has occurred.
WS FULL	This operation requires more workspace than currently available. *)CLEAR* can be used to enlarge the workspace.
WS NAME ERROR	A workspace already exists with the name supplied.
WS NOT FOUND	The workspace does not exist in the specified directory or you do not have permission to access it.
WS TOO LARGE	The saved workspace is too large to be used at this time.

Appendix D *APL and Termcap*

This APL⋆PLUS System for the VAX/VMS environment emulates the UNIX operating system's "termcap" (terminal capability) facility as the basis for providing full-screen support on a wide variety of terminals. A termcap file is a database that describes the control sequences appropriate to a large number of different screen terminals.

A small termcap database that describes popular APL terminals (the `atermcap` file) is included with the APL⋆PLUS System. This appendix provides the material you need to add other terminals to the termcap database by augumenting the `atermcap` file. Experience with the UNIX termcap facility is most useful in modifying the `atermcap` file. In fact, adding a UNIX termcap entry into `atermcap` is the quickest way to provide some terminal support for a new terminal since the termcap databases distributed with UNIX systems often contain entries for a large number of terminals and many terminal manufacturers will provide a recommended termcap entry for a terminal upon request.

Customization of the UNIX termcap entry will likely still be needed to fully utilize all the features of APL with a new terminal. If you are unable to locate a termcap entry that works with your terminal, contact our Help Line since we may be able to supply one or help you construct one for you terminal. The telephone number for our Help Line can be found at the end of the Introduction.

Identifying the Terminal and Termcap to APL

In order to provide full-screen support for your terminal, APL needs two essential pieces of information:

- the type of terminal you are using
- the name of the termcap database to be used.

The terminal type is determined from the value of the `terminal=` `startup` parameter.

```
$apl terminal=vt100
```

The termcap file is identified by the `termcap=` startup parameter.

Custom Termcap

The file `atermcap` that is distributed with the APL⋆PLUS System is a termcap file that describes most popular APL terminals. The following table lists the termcap entries used by the APL⋆PLUS System.

Termcap Entries Used by the APL⋆PLUS System

Name	**Use**
`ti`	Cursor movement initialization string
`te`	Cursor movement de-initialization string
`sr`	Scroll reverse
`up`	Move cursor up
`do`	Move cursor down
`le`	Move cursor left
`nd`	Non-destructive space (cursor right)
`cm`	Cursor movement
`im`	Insert mode
`ei`	End insert mode
`ic`	Insert character
`al`	Add line
`dl`	Delete line
`dc`	Delete Character
`ho`	Move cursor to home position
`ce`	Clear to end of line
`cd`	Clear to end of display
`cl`	Clear screen
`vb`	Visible bell
`so`	Begin standout mode*
`se`	End standout mode*
`kh`	Sequence transmitted by Home key**
`kr`	Sequence transmitted by cursor-right key**
`kl`	Sequence transmitted by cursor-left key**

`ku`	Sequence transmitted by cursor-up key**
`kd`	Sequence transmitted by cursor-down key**
`AT`	Attribute type (STSC enhancement)*
`AP`	Attribute prefix (STSC enhancement)*
`A0 - A8`	Attribute codes (STSC enhancement)*
`SC`	Spaces clear flag (STSC enhancement)
`SO`	Spaces overstrike (STSC enhancement)
`SR`	Spaces replace (STSC enhancement)

* Standout mode is used if the STSC enhanced termcap notation for attributes is not present in the file. See "Termcap Notation for Display Attributes," following, for a discussion of the STSC enhancements.

** APL will extract the logical keystrokes for cursor movement from the termcap file if these entries are present, in which case it is not necessary to explicitly define them in the configuration file.

Termcap Notation for Display Attributes

Regular UNIX termcap files only contain the information necessary to support one logical attribute, known as "standout mode". When APL is used with a regular termcap file, standout mode is used to produce display attribute 1, but attributes 2, 4, and 8 have no effect. Depending upon the terminal, attribute 1 may be reverse video or something different.

To provide a more complete set of attributes, the APL★PLUS System supports extensions to termcap that describe how attributes are produced on a terminal. APL handles two distinct ways in which terminals control combinations of display attributes called "Type 0" and "Type 1":

- On "Type 0" terminals, attributes are set individually, with a separate escape sequence used to set each attribute. APL sets the attributes on these terminals by first transmitting the "clear all attributes" sequence and then transmitting a separate escape sequence to turn on each desired attribute. Terminals that follow the ANSI standard, such as the DEC VT100 and VT200, are common examples of Type 0 terminals.

- On "Type 1" terminals, the display attributes are set by an escape sequence followed by an "attribute byte" that expresses the desired combination of attributes to be turned on. The APL*PLUS PC System in terminal mode is an example of a Type 1 terminal.

The following tables descibe how termcap entries are constructed for both types of terminals. The STSC termcap file `atermcap` contains examples of each type of attribute encoding.

Attribute Notation for Type 0 Terminals

Name	**Use**
`AT`	attribute type: `AT=0` means that this is a Type 0 terminal, with each attribute turned on individually
`A0`	default attribute (0): transmitting this sequence to the terminal will reset the terminal to its default display attribute
`A1`	string to enable attribute 1 (reverse video)
`A2`	string to enable attribute 2 (alternate intensity)
`A3`	string to enable attribute 4 (blinking)
`A4`	string to enable attribute 8 (underline)
`A5`	string to enable attribute 16
`A6`	string to enable attribute 32
`A7`	string to enable attribute 64
`A8`	string to enable attribute 128

Attribute Notation for Type 1 Terminals

Name	**Use**
`AT`	attribute type: `AT=1` means that this is a Type 1 terminal with all attributes set by one attribute byte
`AP`	attribute prefix: the invariant part of the escape sequence that sets the attribute byte; APL transmits this sequence to the terminal followed by a single byte computed to denote all of the attributes

`A0`	default attribute (0): a single character that represents the default attribute; the attribute characters for the other attributes are added to the decimal value of this character to compute the attribute byte; typically, `A0` is either `SPACE` or "@" (decimal 32 or 64)
`A1`	character added to A0 for attribute 1 (reverse video)
`A2`	character for attribute 2 (alternate intensity)
`A3`	character for attribute 4 (blinking)
`A4`	character for attribute 8 (underline)
`A5`	character for attribute 16
`A6`	character for attribute 32
`A7`	character for attribute 64
`A8`	character for attribute 128

For example, on a Type 1 terminal where

`A0`="@"	(decimal value 64)
`A1`=Ctrl D	(decimal value 4)
`A2`=Ctrl H	(decimal value 8)

APL would set attribute 3 by transmitting the AP string followed by the ASCII character "L" (whose decimal value 76 is 64+4+8).

Other STSC Termcap Enhancements

In addition to the enhanced notation for display attributes, STSC has also defined three new termcap control strings. These strings control the means by which ⎕*WPUT* writes data to the terminal screen when only a portion of the screen is being updated. On many APL terminals, it is necessary to first blank out the characters already present on the screen before the new data is written, since the terminals will overstrike the existing data instead of replacing it.

The new strings are

`SC`	Flag to indicate that the terminal always overwrites data already present on the screen, even in APL mode. This is true of non-overstriking terminals such as the VT200. If the termcap entry contains `:SC:`, ⎕*WPUT* will not blank out an area before overwriting it.

`SO=...`	If `SC` is absent, this is the string used to put the terminal into a state where spaces overstrike rather than replace existing characters. Default is `SO=^N`.
`SR=...`	If `SC` is absent, this is the string used to put the terminal into a state where spaces replace existing character, enabling ⎕*WPUT* to reliably blank out an area on the screen. Default is `SR=^O`.

Appendix E *Policy On Commercial Use And Distribution of Kermit*

*STSC provides Kermit with this APL *PLUS System in accordance with the following policy statement written by Frank da Cruz, Columbia University Center for Computing Activities, June 1984:*

The KERMIT file transfer protocol has always been open, available, and free to all. The protocol was developed at the Columbia University Center for Computing Activities, as were the first several KERMIT programs. Columbia has shared these programs freely with the worldwide computing community since 1981, and as a result many individuals and institutions have contributed their own improvements or new implementations in the same spirit. In this manner, the number of different systems supporting KERMIT implementations has grown from three to about sixty in less than three years. If Columbia had elected to keep the protocol secret, to restrict access to souce code, or to license the software, the protocol would never have spread to cover so many systems, nor would the programs be in use at so many sites, nor would the quality of many of the implementations be so high.

Although KERMIT is free and available to anyone who requests it, it is not in the "public domain". The protocol, the manuals, the Columbia implementations, and many of the contributed implementations bear copyright notices dated 1981 or later, and include a legend like

> Permission is granted to any individual or institution to copy or use this document and the programs described in it, except for explicitly commercial purposes.

This copyright notice is to protect KERMIT, Columbia University, and the various contributors form having their work usurped by others and sold as a product. In addition, the covering letter which we include with a KERMIT tape states that KERMIT can be passed along to others; "we ask only that profit not be your goal, credit be given where it is due, and that new material be sent back to us so that we can maintain a definitive and comprehensive set of KERMIT implementations."

Within this framework, it is acceptable to charge a reproduction fee when supplying KERMIT to others. The reproduction fee may be designed to recover costs of media, packaging, printing, shipping, order processing, or any computer use required for reproduction. The fee should not reflect any program or documentation development effort, and it should be independent of how many implementations of KERMIT appear on the medium or where they came from. It should not be viewed as a license fee. For instance, when Columbia ships a KERMIT tape, there is a $100.00 reproduction fee which includes a 2400' reel of magnetic tape, two printed manuals, various flyers, a box, and postage; there is an additional $100.00 order processing charge if an invoice must be sent. The tape includes all known versions of KERMIT, including sources and documentation.

Commercial institutions may make unlimited internal use of KERMIT. However, a quesiton raised with increasing frequency is whether a company may incorporate KERMIT into its products. A hardware vendor may wish to include KERMIT protocol into its comunications package, or to distribute it along with some other product. A timesharing vendor or dialup database may wish to provide KERMIT for downloading. All these uses of KERMIT are permissible, with the following provisos:

- A KERMIT program may not be sold as a product in and of itself. In addition to violating the prevailing spirit of sharing and cooperation, commercial sale of a product called "KERMIT" would violate the trademark which is held on that name by Henson Associates, Inc., creators of The Muppet Show.

- Existing KERMIT programs and documentation may be included with hardware or other software as part of a standard package, provided the price of the hardware or software product is not raised significantly beyond costs of reproduction of the KERMIT component.

- KERMIT protocol may be included in a multi-protocol communication package as one of the communication options, or as a communication feature of some other kind of software package, in order to enhance the attractiveness of the package. KERMIT protocol file transfer and management should not be the primary purpose of the package. The price of the package should not be raised significantly because KERMIT was included, and the vendor's literature should make a statement to this effect.

- Credit for development of the KERMIT protocol should be given to the Columbia University Center for Computing Activities, and customers should be advised that KERMIT is available for many systems for only a nominal fee from Columbia and from various user group organizaitons, such as DECUS and SHARE.

Columbia University holds the copyright on the KERMIT protocol, and may grant permission to any person or institution to develop a KERMIT program for any particular system. A commercial institution that intends to distribute KERMIT under the conditions listed above should be aware that other implementations of KERMIT for the same system may appear in the standard KERMIT distribution at any time. Columbia University encourages all developers of KERMIT software and documentation to contribute their work back to Columbia for further distribution.

Finally, Columbia University does not warrant in any way the KERMIT software nor the accuracy of any related documentation, and neither the authors of any KERMIT programs or documentation nor Columbia University acknowledge any liability resulting from program or documentation errors.

These are general guidelines, not a legal document to be searched for loopholes. To date, KERMIT has been freely shared by all who have taken the time to do work on it, and no formal legalities have proven necessary. The guidelines are designed to allow commercial enterprises to participate in the promulgation of KERMIT without seriously violating the KERMIT user community's trust that KERMIT will continue to spread and improve at no significant cost to themselves. The guidelines are subject to change at any time, should more formal detail prove necessary.

Commercial organizations wishing to provide by KERMIT to their customers should write a letter stating their plans and their agreement to comply with the guidelines listed above. The letter should be addressed to:

KERMIT Distribution
Columbia University Center for Computing Activities
612 West 115th Street
New York, NY 10025

INDEX

Index

U = *APL★PLUS System User's Manual*
R = *APL★PLUS System Reference Manual*