APL2 Programming:

System Services Reference

Version 2 Release 2

SH21-1054-01

APL2 Programming:

System Services Reference

Version 2 Release 2

SH21-1054-01

Note!

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

Second Edition (March 1994)

This edition replaces and makes obsolete the previous edition, SH21-1054-0. The technical changes for this edition are summarized under “Summary of Changes,” and are indicated by a vertical bar to the left of a change.

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

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

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

APL Products and Services
IBM Silicon Valley Laboratory, H36/F40
Reader Comments
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

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

Contents

Notices . . . xii
Programming Interface Information . . . xii
Trademarks . . . xiii

About This Book . . . xiv
Who Should Use This Book . . . xiv
APL2 Publications . . . xiv
Conventions Used in This Library . . . xv

Summary of Changes . . . xvii
Product . . . xvii

Part 1. Interactive and Batch Processing . . . 1

Chapter 1. Introduction to APL2 System Services . . . 3
APL2 System Structure . . . 3
APL2 Support Features . . . 5
Invocation Options . . . 5
APL2 Session Manager . . . 5
APL2 Libraries, Workspaces, and Data Files . . . 6
Batch Processing of APL2 . . . 6
Shared Variables and Auxiliary Processors . . . 6
Workspaces Distributed with APL2 . . . 6
External Names . . . 7

Chapter 2. APL2 Invocation and Termination . . . 8
Starting APL2—The APL2 Command under CMS and TSO . . . 8
Continuing with the Invocation Options . . . 8
Description of Invocation Options . . . 11
Session Termination under CMS . . . 32
Session Termination under TSO . . . 33

Chapter 3. The APL2 Session Manager . . . 36
Features of the APL2 Session Manager . . . 37
The Session Manager Screen . . . 37
Entering Multiple Lines of Input . . . 41
The Session Manager Log . . . 41
Reusing Previous Lines in the Session Log . . . 41
Controlling the Size of and Saving the Session Log . . . 42
APL2 Session Manager Commands . . . 43
Session Manager Command Summary . . . 43
COLUMN . . . 44
COPY . . . 46
DISPLAY . . . 50
FIND . . . 53
HELP . . . 54
LINE . . . 54
LOG . . . 55
PAGE . . . 56

PFK . . . 57
PROFILE . . . 58
SUPPRESS . . . 61
Session Manager Messages . . . 62

Chapter 4. APL2 Libraries: Workspaces and Data Files . . . 63
APL2 Libraries, Workspaces, and Data Files under CMS . . . 63
Updating LIBTAB APL2 to Create New APL2 Libraries . . . 64
Accessing Libraries, Workspaces, and Data Files . . . 65
Workspace Names . . . 65
Data File Names . . . 66
Transfer Files under CMS . . . 66
Library Passwords . . . 67
APL2 Libraries, Workspaces, and Data Files Under TSO . . . 67
Virtual Storage Access Method (VSAM) Library System . . . 68
Sequential Access Method (SAM) Library System . . . 70
Transfer Files under TSO . . . 73
Security and Integrity of APL2 Data . . . 73

Chapter 5. Named Editors . . . 74
Restrictions Using Named System Editors . . . 74
Using Named System Editors under CMS . . . 75
Using Named System Editors under TSO . . . 75

Chapter 6. Batch Processing . . . 78
Batch Jobs under CMS . . . 78
CMS Batch Facility Input . . . 78
CMS Batch Facility Output . . . 79
Batch Jobs under TSO . . . 80
TSO Batch Input . . . 80
TSO Batch Output . . . 80

Chapter 7. Controlling APL2 Invocation . . . 82
Providing Input to APL2 . . . 82
Directing APL2's Output . . . 83
Controlling APL2's Use of the Screen . . . 84
DBCS and APLIN/APLPRINT Files . . . 84
DBCS in Other IBM Products . . . 84
DBCS in APL2 . . . 85
Reading DBCS from APLIN . . . 85
Writing DBCS to APLPRINT . . . 85

Chapter 8. Using APL2 across Systems . . . 86
Cooperative Processing . . . 86
Processor Network Identification . . . 86
Processor Profile Structure . . . 87
Using the Port Server . . . 88
Sending a Share Offer . . . 88
Receiving a Share Offer . . . 88
Processor Profile Syntax . . . 89
Processor Profile Examples . . . 91
Transferring Workspaces . . . 94
Workspace Transfer between APL2 Systems . . . 94
Migration of TryAPL2 Workspaces . . . 95

Migration of VS APL Workspaces . . . 95
Transferring AP 211 Files . . . 95

Part 2. Auxiliary Processors . . . 97

Chapter 9. Summary of Auxiliary Processors Distributed with APL2 . . . 102
Using Auxiliary Processors . . . 103
Using the Share-Offer Utilities . . . 104
Suggestions for Use of Auxiliary Processors . . . 105

Chapter 10. AP 100—Host System Command Processor Under CMS . . . 107
Associated Workspace . . . 107
Shared Variable Overview . . . 107
Initial Value . . . 108
Communication Procedure . . . 108
Querying the Operating System . . . 109
CMS Communication and IMPCP and IMPEX Settings . . . 109
Cautions . . . 110
Return Codes . . . 110

Chapter 11. AP 100—Host System Command Processor Under TSO . . . 112
Associated Workspace . . . 112
Shared Variable Overview . . . 113
Communication Procedure . . . 113
Querying the Operating System . . . 114
Return Codes . . . 114
AP 100 Built-In Commands . . . 116

Chapter 12. AP 101—Alternate Input (Stack) Processor . . . 127
Associated Workspaces . . . 127
Shared Variable Overview . . . 128
Data Formats . . . 128
Initial Values . . . 128
Communication Procedure . . . 129
AP 101 Commands . . . 129
Using AP 101 within a Defined Function . . . 130
Using AP 101 with the TSO Fence Option . . . 131
Cautions . . . 132
Return Codes . . . 133

Chapter 13. AP 102—Main Storage Access Processor . . . 134
Shared Variable Overview . . . 134
Commands . . . 135
Communication Procedure . . . 135
Formatting the Result from AP 102 . . . 136
Cautions . . . 136
Return Codes . . . 137

Chapter 14. AP 110—CMS File Processor . . . 138
Associated Workspace . . . 138
Shared Variable Overview . . . 138
Initial Values . . . 139
Communication Procedure . . . 140

Record Variable . . . 141
Control Variable . . . 141
Cautions . . . 143
Return Codes . . . 144

Chapter 15. AP 111—QSAM File Processor . . . 146
Associated Workspaces . . . 146
Shared Variable Overview . . . 146
Initial Values . . . 147
Communication Procedure . . . 148
Cautions . . . 150
Return Codes . . . 151

Chapter 16. AP 119—Socket Interface Processor . . . 154
Shared Variable Overview . . . 154
The APL2 Socket Application Program Interface . . . 154
IUCV Paths and Sockets . . . 155
AP 119 and TCP/IP Commands Summary . . . 155
Definition of TCP/IP Terms . . . 156
Blocking . . . 158
Using AP 119—The TCPIP Commands . . . 158
ACCEPT . . . 158
BIND . . . 159
CLOSE . . . 159
CONNECT . . . 159
FCNTL . . . 160
GETHOSTID . . . 160
GETHOSTNAME . . . 161
GETPEERNAME . . . 161
GETSOCKNAME . . . 162
GETSOCKOPT . . . 162
LISTEN . . . 163
READ . . . 163
RECV . . . 164
RECVFROM . . . 164
SELECT . . . 165
SEND . . . 166
SENDTO . . . 166
SETSOCKOPT . . . 167
SHUTDOWN . . . 167
SOCKET . . . 168
WRITE . . . 168
Using AP 119—The AP Commands . . . 168
The APL2 Port Server . . . 169
PSLIST—Send LIST Command to the Port Server . . . 169
PSCLEAR—Send CLEAR Command to the Port Server . . . 170
PSSHUTD—Send SHUTDOWN Command to the Port Server . . . 170
UNREGSTR—Send an UNREGISTER Command to the Port Server . . . 170
Listening Ports . . . 170
GETLPORT—Get Listening Port . . . 171
SETLPORT—Set Listening Port . . . 171
Starting AP 119 . . . 171
Sample AP 119 Session Using the APL2 Socket API . . . 172
Return codes . . . 175

Chapter 17. AP 120—APL2 Session Manager Command Processor 179
Shared Variable Overview 179
Data Formats 180
Communication Procedure 180
Return Codes 181

Chapter 18. AP 121—APL2 Data File Processor 182
Associated Workspaces 182
Shared Variable Overview 182
Access Control Considerations 183
APL2 Data Files 183
File Identification 183
APL2 Data File Organization 184
Communication Procedure 185
Commands 185
Opening a File 187
Checking for End of File 190
APL2 Data File Maintenance 190
Library Query 190
Space Requirements for Storing APL2 Variables 191
Size Limitations 191
Cautions 191
Return Codes 192

Chapter 19. AP 123—VSAM File Processor 195
Associated Workspaces 195
Shared Variable Overview 195
VSAM Files—General Information 195
File Identification 196
File Formats and Keys 196
Commands 197
Communication Procedure 198
Opening a VSAM File 199
Processing a VSAM File 200
Obtaining the Key of the Last I/O Operation 201
Positioning the Record Pointer 202
Specifying Character Conversion 203
Closing a VSAM File 203
Cautions 203
Return Codes 204

Chapter 20. AP 124—Text Display Auxiliary Processor 207
Shared Variable Overview 207
Understanding Screen Management 208
Communications Procedure 209
Return Codes 221

Chapter 21. AP 126—GDDM Processor 222
Associated Workspaces 222
Licensed Program Requirements 223
Shared Variable Overview 223
Data Formats 224
Communication Procedure 226
GDDM Calls 228

AP 126 Commands . . . 228
Obtaining Copies through AP 126 . . . 237
GDDM FSOPEN Request or DSOPEN, DSUSE Sequence . . . 237
Alternating Paths . . . 237
Implications of Multiple Data Paths . . . 237
Page Sharing with the APL2 Session Manager . . . 238
Handling Attentions . . . 239
APL2/370 and GDDM EBCDIC Code Page Differences . . . 240
GDDM Error Diagnosis . . . 241
Return and Reason Codes . . . 241

Chapter 22. AP 127—SQL Processor . . . 244
Shared Variable Overview . . . 245
Communication Procedure . . . 245
AP 127 Commands . . . 246
Return Codes . . . 247

Chapter 23. AP 210—BDAM File Processor (TSO Only) . . . 248
Associated Workspace . . . 248
Shared Variable Overview . . . 248
BDAM File Requirements . . . 249
Communication Procedure . . . 249
Initial Values . . . 249
Formatting a Direct File Using AP 210 . . . 250
BDAM File Processing Procedure . . . 251
Cautions . . . 253
Return Codes . . . 253
Data Management Error Codes . . . 254

Chapter 24. AP 211—The APL2 Object File Processor . . . 255
Shared Variable Overview . . . 255
Commands Accepted by AP 211 . . . 255
Return Codes . . . 259

Chapter 25. APL2 Shared Variable Interpreter Interface . . . 261
Shared Variable Interpreter Interface Protocols . . . 261
Shared Variable Overview . . . 262
Interpreter Input Data . . . 262
Interpreter Output Data . . . 263

Part 3. Associated Processors . . . 265

Chapter 26. External Names and Associated Processors . . . 268
Applications of External Names . . . 268
Managing External Names from APL . . . 270
Creating and Destroying an Association . . . 270
Invoking an External Name . . . 271
Querying an Associated Name . . . 271
Avoiding Name Conflicts . . . 272
Environmental Considerations . . . 273

Chapter 27. Processor 10—Communication with REXX . . . 274
Overview . . . 274

Detailed Description . . . 275
Using REXX Functions . . . 275
Accessing REXX Variables and Constants . . . 279
Built-in Functions . . . 280
Unexpected Errors and Other Considerations . . . 286
Failure when Associating a Name . . . 286
APL Errors . . . 287
Non-APL Error Messages . . . 287
REXX Return Code 20040 . . . 287
"Missing" Argument Strings . . . 288
Truncated Data Returned under TSO/E . . . 289
Other Considerations . . . 289

Chapter 28. Processor 11—Calling Compiled Programs . . . 291
□*NA* Syntax for Non-APL Programs . . . 292
Processor 11 Overview . . . 292
Introduction . . . 293
Processor 11 Glossary . . . 293
Usage Overview . . . 295
Routine Descriptions . . . 295
Routine Lists . . . 298
BUILDRL and Interface Management Routines . . . 301
BUILDRL Interface Management and Self-Describing Routines . . . 302
Environments . . . 303
Interface Details . . . 307
Routine Description Tags . . . 307
Argument Patterns . . . 310
Result Patterns . . . 314
Explicit Results, Function Valence, and Operator Valence . . . 317
NAMES Files . . . 318
Processor 11 Non-APL Routine Description Tag Rules . . . 319
System Usage Guidelines . . . 321
Linkage Conventions . . . 321
Unexpected Errors . . . 322
Processor 11 Routine Search Order Guidelines . . . 325
External Function Names . . . 325
CMS Search Order Guidelines . . . 325
TSO Search Order Guidelines . . . 326
Link-Editing External Routines . . . 327
Link-Edit Tools . . . 328
Link-Editing External Routines on CMS . . . 329
Link-Editing External Routines on TSO . . . 330
Installation of External Routines . . . 331
Extended Addressing Considerations . . . 331
Preloading and Sharing External Routines . . . 331
Execution Time Libraries . . . 332
Other Processor 11 Considerations . . . 332
Using Self-Describing Routines from Non-APL Programs . . . 332
Using Modules with Routine Lists from Non-APL Programs. . . . 332
FORTRAN Considerations . . . 332

Chapter 29. Processor 11—Access to Namespaces . . . 334
Overview . . . 334
Detailed Description . . . 336

Creating Namespaces . . . 338
Workspace Names . . . 340
Accessing Objects in Namespaces . . . 340
NAMES Files . . . 342
Using Namespaces . . . 343
Namescopes . . . 344
Combining Several Namespaces in a Member . . . 347
CMS Namespace Routine List Example . . . 348
TSO Namespace Routine List Example . . . 349
Link-Editing Namespaces . . . 349
Unexpected Errors and Other Considerations . . . 349

Chapter 30. Processor 12—Files as Arrays . . . 352
□*NA* Syntax for Processor 12 . . . 352
Supported Primitive Operations . . . 354
APL Files as External Variables . . . 355
Record-oriented Files as External Variables . . . 356
Format Descriptors for External Variables . . . 358
Processor 12 Errors . . . 359

Appendixes . . . 363

Appendix A. Implementation Limits . . . 365

Appendix B. Deviations from APL2 Programming: Language Reference . . . 366
System Functions and Variables . . . 366
Full-Screen Editor—Editor 2 . . . 366
System Commands . . . 367

Appendix C. National Languages Supported by APL2 . . . 368

Appendix D. Auxiliary Processor Conversion Options . . . 370
APL . . . 370
370 or BCD . . . 371
BIT . . . 371
BYTE . . . 371
CDR . . . 372
COD1 . . . 372
DBCS[(nnn)] . . . 372
Reading DBCS Data . . . 373
Writing DBCS Data . . . 374
EBCD or 192 . . . 374
TN . . . 374
VAR . . . 374

Appendix E. Conversion of Atomic Vector Characters . . . 376

Appendix F. APL2 Files and Data Sets . . . 381
CMS Files . . . 381
CMS Filedef (DD) Names . . . 382
TSO DD Names . . . 382
TSO Data Set Names . . . 383

Appendix G. Sample Non-APL Programs to be Called through Processor 11 385
C/370 Examples 386
Updating Arguments with C/370 386
PL/I Examples 388
Updating Arguments with PL/I 388
VS FORTRAN Examples 390
Updating Arguments with VS FORTRAN 390
Link-Editing Examples 392
Link-Editing on TSO using a CLIST 392
Link-Editing on TSO using JCL 392
Link-Editing on CMS 393
Generating a MODULE on CMS 393

Appendix H. Summary of Terminal Information for APL2 Tasks 394
IBM 2741 Communication Terminal 394
IBM 3270 Information Display System 396

Appendix I. Printer Fonts Supplied with APL2 399

Bibliography 401
APL2 Publications 401
Other Books You Might Need 401

Index 404

Notices

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

IBM may have patents or pending patent applications covering subject material in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785, United States.

Programming Interface Information

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

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

However, this reference also documents Product-sensitive Programming Interface and Associated Guidance Information provided by APL2.

Product-sensitive programming interfaces allow the customer installation to perform tasks such as diagnosing, modifying, monitoring, repairing, tailoring, or tuning of APL2. Use of such interfaces creates dependencies on the detailed design or implementation of the IBM software product. Product-sensitive programming interfaces should be used only for these specialized purposes. Because of their dependencies on detailed design and implementation, it is to be expected that programs written to such interfaces may need to be changed in order to run with new product releases or versions, or as a result of service.

Product-sensitive Programming Interface and Associated Guidance Information is identified where it occurs, either by an introductory statement to a chapter or section or by the following marking:

Product-sensitive programming interface

Product-sensitive Programming Interface and Associated Guidance Information...

End of Product-sensitive programming interface

Trademarks

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

ACF/VTAM	MVS/XA
AIX	OS/2
AIX/6000	RACF
APL2	RISC System/6000
APL2/6000	Selectric
BookMaster	SQL/DS
DATABASE 2	System/370
DB2	System/390
GDDM	VM/ESA
IBM	VM/XA
MVS/DPA	VTAM
MVS/ESA	3090

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

Sun	Sun Microsystems, Inc.
Solaris	Sun Microsystems, Inc.
UNIX	AT&T Corporation

About This Book

This manual describes the services and facilities that, together with the language processor, make up the IBM* APL2* Licensed Program. These services support efficient APL2 program development.

Most of the services described in this manual are available under both CMS and TSO. Differences in availability and in use do exist, however, and they are noted throughout the manual. The standard order of presenting the differences is first to provide the information specific to CMS and then to discuss the comparable information for TSO.

The APL2 Licensed Program has several options or parameters that can be specified when it is installed. These options enable the product to be tailored to the services available in your computing system and allow consistency with the naming conventions established at your site. Your site, therefore, may not have chosen all the services described in this manual. Furthermore, your site may have overridden some of the default installation options used in this book. Where your installation of APL2 can differ from the default installation, the manual provides appropriate remarks. To find out whether your APL2 environment differs from the default environment described in this manual, check with your system administrator.

Who Should Use This Book

This manual assumes you are familiar with both APL2 and the host system (CMS or TSO). It neither describes the language nor provides details on the operating system. The manuals listed at the end of this preface are sources for this information.

"Part 2. Auxiliary Processors," assumes you are an experienced VS APL or APL2 application programmer or has had a VS APL or APL2 course that includes the use of auxiliary processors.

"Part 3. Associated Processors," assumes you are an experienced application programmer in either APL2 or another language. Some of the material covered in this part requires an understanding of the operating system level interfaces used by other languages.

APL2 Publications

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

Figure 1 (Page 1 of 2). APL2 Publications

Information	Book	Publication Number
General product	*APL2 Fact Sheet*	GH21-1090

Figure 1 (Page 2 of 2). APL2 Publications

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

For the titles and order numbers of other related publications, see the "Bibliography" on page 401.

Conventions Used in This Library

This section discusses the conventions used in this library.

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

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

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

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

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

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

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

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

Throughout this book, the following product names apply:

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

Summary of Changes

Product

APL2/370, Version 2 Release 2

Date of Publication: March 1994

Form of Publication: Revision, SH21-1054-01

Document Changes

- Added the NLT invocation option
- Updated all version numbers
- Changed default for the QUIET invocation option to OFF
- Added session manager GDDM* information
- Updated cooperative processing and migration information
- Updated AP 101 commands
- Updated AP 127 operations syntax
- Added RENAME command for AP 211
- Added new return codes for AP 211
- Added implementation limits table
- Added deviations appendix
- Added listing of national language support

Part 1. Interactive and Batch Processing

Chapter 1. Introduction to APL2 System Services . . . 3
APL2 System Structure . . . 3
APL2 Support Features . . . 5
Invocation Options . . . 5
APL2 Session Manager . . . 5
APL2 Libraries, Workspaces, and Data Files . . . 6
Batch Processing of APL2 . . . 6
Shared Variables and Auxiliary Processors . . . 6
Workspaces Distributed with APL2 . . . 6
External Names . . . 7

Chapter 2. APL2 Invocation and Termination . . . 8
Starting APL2—The APL2 Command under CMS and TSO . . . 8
Continuing with the Invocation Options . . . 8
Description of Invocation Options . . . 11
Session Termination under CMS . . . 32
Session Interruption under CMS . . . 33
Session Termination under TSO . . . 33
Session Interruption under TSO . . . 34

Chapter 3. The APL2 Session Manager . . . 36
Features of the APL2 Session Manager . . . 37
The Session Manager Screen . . . 37
Entering Multiple Lines of Input . . . 41
The Session Manager Log . . . 41
Reusing Previous Lines in the Session Log . . . 41
Scrolling through the Session Log . . . 42
Controlling the Size of and Saving the Session Log . . . 42
APL2 Session Manager Commands . . . 43
Session Manager Command Summary . . . 43
COLUMN . . . 44
COPY . . . 46
DISPLAY . . . 50
FIND . . . 53
HELP . . . 54
LINE . . . 54
LOG . . . 55
PAGE . . . 56
PFK . . . 57
PROFILE . . . 58
SUPPRESS . . . 61
Session Manager Messages . . . 62

Chapter 4. APL2 Libraries: Workspaces and Data Files . . . 63
APL2 Libraries, Workspaces, and Data Files under CMS . . . 63
Updating LIBTAB APL2 to Create New APL2 Libraries . . . 64
Accessing Libraries, Workspaces, and Data Files . . . 65
Workspace Names . . . 65
Data File Names . . . 66
Transfer Files under CMS . . . 66

Library Passwords . . . 67
APL2 Libraries, Workspaces, and Data Files Under TSO . . . 67
Virtual Storage Access Method (VSAM) Library System . . . 68
Creating APL2 VSAM Libraries . . . 68
Accessing VSAM Libraries . . . 69
Sequential Access Method (SAM) Library System . . . 70
Workspace Names . . . 71
CONTINUE workspace . . . 72
Transfer Files under TSO . . . 73
Security and Integrity of APL2 Data . . . 73

Chapter 5. Named Editors . . . 74
Restrictions Using Named System Editors . . . 74
Using Named System Editors under CMS . . . 75
Using Named System Editors under TSO . . . 75

Chapter 6. Batch Processing . . . 78
Batch Jobs under CMS . . . 78
CMS Batch Facility Input . . . 78
CMS Batch Facility Output . . . 79
Other APL2 Considerations . . . 79
Batch Jobs under TSO . . . 80
TSO Batch Input . . . 80
TSO Batch Output . . . 80

Chapter 7. Controlling APL2 Invocation . . . 82
Providing Input to APL2 . . . 82
Directing APL2's Output . . . 83
Controlling APL2's Use of the Screen . . . 84
DBCS and APLIN/APLPRINT Files . . . 84
DBCS in Other IBM Products . . . 84
DBCS in APL2 . . . 85
Reading DBCS from APLIN . . . 85
Writing DBCS to APLPRINT . . . 85

Chapter 8. Using APL2 across Systems . . . 86
Cooperative Processing . . . 86
Processor Network Identification . . . 86
Processor Profile Structure . . . 87
Using the Port Server . . . 88
Sending a Share Offer . . . 88
Receiving a Share Offer . . . 88
Processor Profile Syntax . . . 89
Identification Entries . . . 89
Authorization Entries . . . 90
Communicating with Version 2 Release 1 . . . 90
Processor Profile Examples . . . 91
Transferring Workspaces . . . 94
Workspace Transfer between APL2 Systems . . . 94
Migration of TryAPL2 Workspaces . . . 95
Migration of VS APL Workspaces . . . 95
Transferring AP 211 Files . . . 95

Chapter 1. Introduction to APL2 System Services

APL2 consists of a language processor, or interpreter, and a collection of services or features external to the language. These services are designed to increase the efficiency of your work with APL2 and to expand the capabilities of the APL2 implementation of the language.

The APL2 language processor is described in *APL2 Programming: Language Reference*. The features described in this publication are:

- The options available when you invoke APL2
- The APL2 session manager
- APL2 libraries and workspaces
- Batch processing of APL2
- Editing APL2 functions and variables
- Shared variables and auxiliary processors
- Associated processors

APL2 System Structure

APL2 consists of several system components. Optionally, the services of several other IBM licensed programs, such as the Graphical Data Display Manager (GDDM) and SQL/Data System (SQL/DS*) or DATABASE 2* (DB2*), can be used with APL2. Figure 2 on page 4 illustrates the basic components of an APL2 system. The numbers in the figure correspond to the numbered explanations.

1. **Your terminal**. Connects you to the host operating system when you log on.
2. **Your host operating system**, either CMS or TSO. When you invoke APL2, you are connected to the facilities of APL2.
3. **APL2 session manager**. An optional feature that requires the use of the Graphical Data Display Manager (GDDM), the session manager provides full-screen interface to APL2 during your APL2 session.
4. **APL2 executor** (different for CMS and TSO because it interfaces with the host system). Receives your APL2 requests and passes them to the APL2 interpreter.
5. **APL2 interpreter** (same for CMS and TSO because it does only APL2 processing and is independent of the host system). Processes your requests and interfaces with your active workspace.
6. **Active workspace**. This is where you define APL2 variables and their defined functions and operators, process APL2 expressions, functions, and operators, and communicate with other processors.
7. **APL2 libraries**. Stores saved workspaces, data files, and the optional session manager log.

8. **Shared Variable Processor (SVP)**. Manages the shared variable communication between your workspace and an auxiliary processor or other user.
9. **Auxiliary processors**. For access to facilities and data outside the active workspace, auxiliary processors receive your requests from the SVP and communicate with the host system.
10. **Associated processors.** Allows processing of routines written in other programming languages.

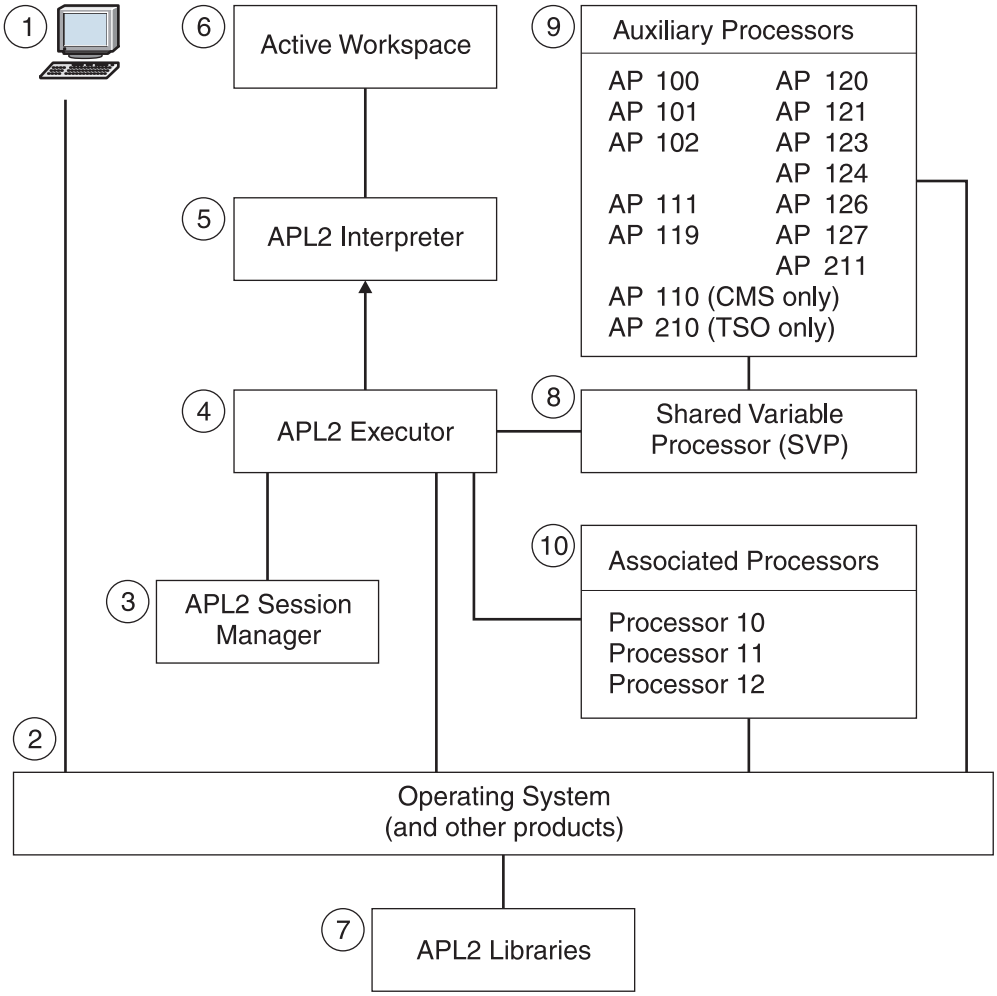


Figure 2. APL2 System Structure

APL2 Support Features

This section discusses some of the support features supplied with APL2.

Invocation Options

You invoke APL2 from the host system by command. The name of the command is APL2. Since your installation may have designated another name for the command, check with your system administrator for APL2 invocation requirements on your computing system.

The command to invoke APL2 allows you to specify several options that affect your APL2 session. These options include:

- The characteristics of the terminal you are using
- The size of your workspace and the size of storage areas used with auxiliary processors
- Debugging options to be in effect

Chapter 2, "APL2 Invocation and Termination," describes each option and the procedures for invoking APL2, terminating your APL2 session, and resuming your session after a break in service.

APL2 Session Manager

The APL2 session manager is an optional facility of APL2. Its use requires the IBM Graphical Data Display Manager (GDDM) Version 2 Release 3 or later, and GDDM-compatible display stations, such as the IBM 3270 family of terminals.

The session manager has many features that can increase your productivity during an APL2 session, including:

- A log file containing a history of the input/output transactions between you and APL2
- The ability to scroll backward and forward through the log file
- The facility to reuse and reprocess previous lines in the log without having to reenter the lines
- An easy way to print, or copy to a file, lines from the session
- Definition and use of function keys to automatically enter predefined commands or data
- The ability to enter and display DBCS characters on terminals that support them, such as the IBM 5550 running Japanese PC/G Version 6 software

Chapter 3, "The APL2 Session Manager," describes the features, commands, and display screen of the APL2 session manager. Session manager commands can be issued during immediate execution mode, or they can be issued from within a defined function using AP 120, the Session Manager Command Processor. See Chapter 17, "AP 120—APL2 Session Manager Command Processor" on page 179.

APL2 Libraries, Workspaces, and Data Files

Saved APL2 workspaces, APL2 data files, and the APL2 session manager log are all stored in special files called APL2 libraries. Structure and naming requirements for these libraries and files differ, depending on whether your host system is CMS or TSO.

Also, differences exist under TSO, depending on whether you have chosen a Virtual Storage Access Method (VSAM) library system or a Sequential Access Method (SAM) library system for storing workspaces. Chapter 4, "APL2 Libraries: Workspaces and Data Files" on page 63, describes the characteristics and requirements for the various files used with APL2. Appendix F, "APL2 Files and Data Sets" on page 381, lists the default files and data sets distributed with APL2.

Batch Processing of APL2

Although APL2 is designed to be used interactively at your terminal, there may be occasions when APL2 processing is more efficiently processed in batch mode. Such situations include the processing of a function that produces a printed report. Chapter 6, "Batch Processing" on page 78, describes how APL2 can be processed in batch mode. It specifies the requirements for submitting the job and the requirements for the input stream containing the job. Chapter 7, "Controlling APL2 Invocation" on page 82, describes APL invocation by display terminal-oriented application development programs. With this type of invocation, APL runs without a terminal, much the same as it does under TSO batch in the TSO environment or in a disconnected virtual machine in the CMS environment.

Shared Variables and Auxiliary Processors

Shared variables are used to communicate with other APL2 users or to interface with auxiliary processors. Auxiliary processors allow you to communicate with other services of the host system such as those that:

- Process files
- Issue commands to the host system (CMS or TSO)
- Execute user programs not written in APL2
- Provide information about the environment in which APL2 is running

Control information and data are passed between your workspace and an auxiliary processor by shared variables. Each auxiliary processor provided with APL2 is described later in this manual.

Your installation may not provide all the processors documented in this manual. Also, your installation may have additional processors, written by your systems staff, that are not described in this reference. To find out what is available on your computing system, see your system administrator.

Workspaces Distributed with APL2

APL2 is distributed with several APL2 workspaces that contain defined functions you can use to facilitate your APL2 coding. These functions use APL2 primitive functions, operators, and other objects to provide useful programming tools. Many of the defined functions are *cover functions* for use with auxiliary processors. Cover functions provide the necessary communication protocol so that you do not have to code the detailed protocol yourself.

The workspaces supplied with APL2 are contained in public libraries 1 and 2. The WSINFO workspace contains a list and a description of the supplied workspaces. Each individual workspace contains a *DESCRIBE* variable or function that describes the functions in the workspace in more detail. Each workspace also contains a *HOW* variable or function that explains how to use the defined functions. For additional information about the workspaces, see *APL2/370 Programming: Using the Supplied Routines*.

External Names

External names have a variety of uses in APL2 applications. They provide the ability to organize APL2 applications more effectively and allow access to non-APL facilities using normal APL2 syntax. By providing additional ways for you to process information using APL2, external names improve productivity. You can use external names for some of the following reasons:

- To access files quickly and easily
- To isolate APL programs to prevent name conflicts
- To use existing non-APL programs
- To improve performance
- To allow synchronous access to system information
- To maintain shared code

Chapter 2. APL2 Invocation and Termination

APL2 is invoked from the host system environment as a command. The default command name is APL2, but your installation may have changed the name of the command. Whatever its name, you can do any one of the following:

- Enter it manually after logging on to the host system.
- Process a CMS EXEC or TSO CLIST that invokes the APL2 command.
- Have the command automatically invoked when you enter CMS or log on to TSO. This is normally done through a CMS PROFILE EXEC or a TSO CLIST.CLIST(DEFAULT) member.

The)*OFF* or)*CONTINUE* system command terminates your APL2 session and allows either you or the invoking EXEC or CLIST to continue with your CMS or TSO session.

For more description of the underlying system during invocation and termination, see the appropriate user's guide for your system.

For information about requirements for APL2 library access when you initiate APL2, see Chapter 4, “APL2 Libraries: Workspaces and Data Files” on page 63.

Appendix F, “APL2 Files and Data Sets,” contains TSO CLIST and AP2EXIT EXEC samples.

Starting APL2—The APL2 Command under CMS and TSO

You can invoke APL2 under either CMS or TSO with the following command:

APL2 *option1 option2 ... option*

where:

option Is one of the keyword parameters used to tailor the APL2 environment for a particular APL2 session. See “Continuing with the Invocation Options” for the invocation options list.

Continuing with the Invocation Options

Invocation options entered by keyword parameters to the APL2 command are available to tailor the APL2 environment for a particular APL2 session. The invocation options are summarized alphabetically by keyword in Figure 3 on page 9 and described in detail starting on page 11. To see what the option keywords are, enter:

```
HELP APL2
```

Figure 3. Summary of APL2 Invocation Options

Keyword	Description
AISIZE(*size*)	**TSO only**. Specifies the number of bytes of data that can be stacked by AP 101, the alternate input (stack) processor.
APNAMES(*name*[(*string*)]...)	Specifies the auxiliary processors to be used in addition to those automatically loaded.
CASE(*n*)	Identifies the cases of the alphabets to be used when APL2 accepts, returns, or displays the names of the APL objects.
CODE(*nnnnn*)	Same as TERMCODE option.
DATEFORM(ISO|ES|EU)	Specifies the format for time and date stamps.
DBCS(TRY|ON|OFF|*nnn*)	Indicates whether mixed APL2/DBCS is supported for I/O during the APL2 session.
DEBUG(*nnn*)	Alters normal error recovery actions of APL2 for debugging purposes.
DSOPEN(*device-token*)	Specifies a device-token to be passed to GDDM (if it is available) on a DSOPEN call.
EXCLUDE(*name...*)	Specifies the auxiliary processors you do not want loaded into your session.
FREESIZE(*size*)	Specifies a minimum limit on the amount of virtual storage not used for the active workspace or shared variables.
HILIGHT(*setting*)	Specifies whether input, output, or both are to be highlighted on the screen.
ID(*nnnnnnn*)	Specifies an identifier number to be associated with the current APL2 session.
INPUT('*string*'...)	Specifies one or more input lines to be given to APL2 upon invocation.
LOADLIB(*dsname ...*)	**TSO Only**. Specifies one or more private load libraries from which APL2 modules, auxiliary processors, or AP 100 commands can be loaded.
NLT(*language*)	Specifies the national language to be in effect when APL2 is started. This can later be changed by assigning a new value to $\square NLT$.
PROFILE(*name*)	Identifies the name of the APL2 session manager profile to be loaded on invocation.
QUIET[(ON|OFF)]	Prevents APL2 from displaying output until it prompts for input.
RUN(['*locator*'] *function*)	Specifies name of a niladic function in a namespace as a part of entry to APL2.
SHRSIZE(*size*)	Specifies the amount of virtual storage to be reserved for the shared variable processor.
SMAPL(TRY|ON|OFF|*nnnn*)	Specifies whether the APL2 session manager is to be used during your APL2 session or whether the interpreter should share a variable with processor *nnnn*.
SVMAX(*nnnnn*)	Specifies the maximum number of shared variables that you can concurrently share.
SYSDEBUG(*nnn*)	Specifies special debugging settings for use by system programmers.
TERMCODE(*nnnnn*) *or* CODE(*nnnnn*)	Identifies the type of terminal you are using.
TRACE(*nnn*)	Provides special debugging aids for use by system programmers.
WSSIZE(*size*)	Specifies the amount of virtual storage in your CMS virtual machine or TSO region to be reserved for your active workspace.
XA(*nn*)	Identifies whether working storage in an XA or ESA system should be allocated above or below the 16-megabyte address boundary.

Each keyword option is usually followed by a value in parentheses. Default values are supplied for the options, and these are generally satisfactory for beginning APL2 users. You may, however, want to override some of the options to customize the APL2 session to meet your needs. To override one or more options, enter the APL2 command followed by the keyword and value of the option(s) you want to override. For example, you enter the following command to invoke APL2 when the APL2 session manager is to be suppressed:

```
APL2 SMAPL(OFF)
```

You can specify keyword options in any order after the APL2 command, and you can abbreviate the keywords. The abbreviation must be at least the first two characters of the keyword.

If the same keyword appears more than once in the APL2 command, the value of the last one entered is used. For example, the following APL2 command initiates an APL2 session without the session manager:

```
APL2 SMAPL(ON) ID(1007) SMAPL(OFF)
```

You can specify the DEBUG, SYSDEBUG, and TRACE options as positive or negative integers. The system treats them as binary flags and turns the corresponding flags on if the integer is positive or turns them off if the integer is negative. For example:

`DEBUG(7)` A value of 7 is treated as binary 1 1 1 and specifies setting `DEBUG (4 2 1)` on.

`DEBUG(-5)` A value of `-5` is treated as binary -(1 0 1) and specifies setting `DEBUG(4 1)` off.

Successive specifications successively turn the corresponding flags on and off. For example:

`DEBUG(6) DEBUG(-3)` First, the 4 and 2 DEBUG flags are set; next, the 2 and 1 DEBUG flags are turned off. The result is a setting of `DEBUG(4)`.

The defaults supplied with APL2 are given with the detailed descriptions of the options beginning on page 11. Check with your system administrator to find out which of these options may have been changed during the installation of APL2 at your site.

Description of Invocation Options

This section lists the invocation options in alphabetic order by keyword. The headings show the syntax of the options; the descriptions explain how to use each option and what values to specify.

Four options allow you to specify an integer *size* value. These options are:

`AISIZE(`*size*`)`

`FREESIZE(`*size*`)`

`SHRSIZE(`*size*`)`

`WSSIZE(`*size*`)`

Unless otherwise restricted by the maximum value allowed for the option, *size* can be specified as bytes, kilobytes, or megabytes. To specify kilobytes or megabytes, follow *size* with a K or M, respectively. To specify a size in bytes, enter a number only. For example, all the following are valid values for the FREESIZE option:

```
FREESIZE(2048000)
FREESIZE(2000K)
FREESIZE(2M)
```

You can also express *size* as a percentage. For example, to reserve 50% of the CMS virtual machine size or TSO logon size for your workspace, you can specify:

```
WSSIZE(50%)
```

Your installation of APL2 can provide defaults for each of the invocation options. Your installation can also prevent you from changing some of these options by providing system overrides for them. Check with your system administrator to find out if any options have system overrides.

AISIZE(*size*) (TSO Only)

Specifies the maximum number of bytes of data that can be stacked by AP 101, the alternate input (stack) processor.

size
Integer, expressed in bytes or kilobytes. For example:

```
AISIZE(8192)
     or
AISIZE(8K)
```

Note: "M" and "%" are permitted, but are not normally useful.

Minimum size: 0
Maximum size: Limited by region size
IBM-supplied default: 512 bytes

Note: The number of lines that can be stacked through AP 101 depends on the length of each line and the stack size specified by this option. If you attempt to place more lines than can fit on the stack, the entire stack is cleared, as indicated by a nonzero return code on the stack request.

APNAMES(*name*[*(string)*]...)

Identifies auxiliary processors not automatically available but that you plan to use during your APL2 session. Processors named with this option can be those distributed with APL2 or those written by your system programming staff.

name
Name of the load module for the auxiliary processor you want available. More than one name can be specified. For example:

```
APNAMES(USERAP1 USERAP2)
```

Your installation normally provides resident auxiliary processors that are automatically available to your session. If you specify a name in this option that is the same as the name of a resident auxiliary processor, the resident version is ignored and a load module with the specified name is searched for and loaded. If you specify an auxiliary processor that uses the same number as the resident auxiliary processor, the specified auxiliary processor should have the same name as the resident, or you must use the exclude option to exclude the resident. For example:

```
APN(A0121) EXC(AP2X121)
```

Figure 4 on page 19 contains the names of the auxiliary processor load modules supplied with APL2.

Under CMS: The following are searched to locate the specified module name:

- Nucleus extensions. One way to load a module as a CMS nucleus extension is the CMS NUCXLOAD command.
- Members of a CMS LOADLIB defined by the CMS FILEDEF command as AP2LOAD. For information on creating auxiliary processor LOADLIB files, see *APL2/370 Installation and Customization under CMS*.

Under TSO: The following are searched to locate the specified module name:

- Task libraries (including those specified in the LOADLIB invocation option—see the description of the LOADLIB option on page 23)
- Link pack area
- Link libraries

('*string*')
Optional character string of parameters to be passed to the auxiliary processor. For example:

```
APNAMES(USERAP1 USERAP2('EBCD') USERAP3('L'))
```

Most auxiliary processors distributed with APL2 ignore any character strings specified as parameters. The *string* parameter may be used with user-written auxiliary processors and with AP 119 and AP 127.

AP 119 Access: The following options can be specified with the APNAMES parameter:

SERVPORT(*nnn*)
Normally, the local port server is listening on port number 31415. If the port server is using a different port number, this parameter must be used to allow AP 119 to communicate with the server. Allowable values are 256 to 65535.

LISTEN(*nnn*)
AP 119 does not normally open a listening connection until a variable is offered to a remote processor. If you want AP 119 to open a listening connection on startup, use this parameter. AP 119 attempts to use the number specified as its listening port. If this number is unavailable, the listening connection is not started. Allowable values are 256 to 65535, or 0 to let TCP/IP assign an arbitrary number.

TCPID('*ccc*')
AP 119 expects the name of the local TCP/IP machine (under CMS) or started task name (under TSO) to be TCPIP. If the name is different, this parameter must be used to allow AP 119 to successfully communicate with TCP/IP.

This is an example of using the startup parameters:

```
APL2 APNAMES(AP2X119(SERVPORT(1234) LISTEN(2345) TCPID('TCPTEST')))
```

If no options are specified using APNAMES, the default is:

```
SERVPORT(31415) TCPID('TCPIP')
```

AP 127 Access: You can use APNAMES to override some default values for AP 127:

Under TSO:

```
APNAMES(AP2X127(RRPLAN(name) CSPLAN(name) SSID(sys) ISOL(level)))
```

name
Identifies a DB2 application plan. The default values are **APL2V22R** (for RRPLAN) and **APL2V22C** (for CSPLAN).

sys
Identifies the DB2 subsystem. The default is **DSN**.

level
Specifies the starting isolation level. Must be RR or CS. The default is **RR**.

Under CMS:

```
APNAMES(AP2X127(ISOL(level)))
```

level
Character string that specifies the starting isolation level. Must be RR or CS. The default is **RR**.

You need to specify AP 127 on APL2 invocation only if you want to override one or more of these defaults. For any keyword not specified, the defaults from the installation options module are used.

CASE(*n*)

Identifies the cases of the alphabets to be used when APL2 returns or displays the names of APL objects.

n One of the following:

0 Both lowercase and underbarred characters are valid and synonymous when evaluating names. Primitives that return names as results (⎕*NL*, ⎕*CR*, ⎕*FX*, ⎕*SVQ*, ⎕*TF*), and system commands and messages that produce names, produce them using the underbarred rather than the lowercase alphabet.

1 Both lowercase and underbarred characters are valid and synonymous when evaluating names. Primitives that return names as results, and system commands and messages that produce names, produce them using the lowercase rather than the underbarred alphabet.

2 In general, underbarred characters are not valid in names and are not accepted or produced in system functions, commands, or messages. Underbarred letters in the arguments to)*COPY*,)*PCOPY*,)*MCOPY*, and)*IN* are accepted as lowercase letters to aid migration.

Note: The response to)*ERASE*, given underbarred letters, is the underbarred letters.

The default shipped by IBM is CASE(1). Your installation can change the default or specify an overriding value.

The CASE(*n*) invocation parameter does **not** apply to all work done by you during the APL2 session. Instead, it is interpreted as an implicit parameter of a)*CLEAR* command. CASE is, in fact, a workspace attribute in all workspaces. (CASE(0) was the default for workspaces created by APL2 Version 1.) The workspace attribute is saved with the workspace and is not changed when the workspace is loaded.

Objects may be copied into the active workspace using)*COPY*,)*PCOPY*,)*MCOPY* or)*IN* without affecting the workspace attribute. The names of copied objects, as well as names referred to by copied functions or defined operators, are converted appropriately.

Note: Literal strings and comments within functions, and the content of indirect copy lists, are **not** converted.

No matter what CASE setting or workspace attribute is in effect, shared variable names containing underbarred characters are never transferred from the APL2 interpreter to the SVP The underbarred characters are instead translated to lowercase. Names returned to the interpreter from the SVP are handled and displayed in accordance with the workspace attribute.

CODE(*nnnnn*)

Used by APL2 to identify the type of terminal you are using. CODE is a synonym for TERMCODE. See the description of the TERMCODE invocation option on page 30.

DATEFORM(ISO|US|EU)

Specifies the format for date and time stamps displayed during operations such as)*LOAD* and)*SAVE*.

ISO
International Standards Organization convention, which follows the format *yyyy-mm-dd hh.mm.ss*.

ISO is the IBM-supplied default.

US
United States preferred convention, which follows the format *mm/dd/yyyy hh.mm.ss*.

EU
European convention, which follows the format *dd.mm.yyyy hh.mm.ss*.

For example, to display the European format, you specify:

```
DATEFORM(EU)
```

The date and time are formatted as:

```
27.03.1992 11.30.52 (GMT+1)
```

Note: The APL2 date and time display is always followed by the user's time zone offset from Greenwich Mean Time (GMT).

DBCS(TRY|ON|OFF|*nnn*)

Used to tell APL2 what double-byte character set (DBCS) data to display and input during the APL2 session. It also sets the default DBCS number for the DBCS translation option of various auxiliary processors and external functions.

Many languages have more characters than can fit in the APL2 atomic vector, ⎕*AV*. Japanese, Korean, and Chinese are examples. IBM has defined DBCS to represent these languages. These characters can be displayed on displays that support the DBCS, such as the IBM 5550 multistation. Each DBCS character occupies two columns on the screen. DBCS characters and characters from ⎕*AV* can be mixed on the screen.

Note: APL2 supports the display of DBCS only through the session manager, AP 126, and)*EDITOR* 2. GDDM is required. For testing purposes, GDDM can provide limited, emulated DBCS support on displays that do not actually support DBCS.

APL2 also supports DBCS in APLIN and APLPRINT files.

When DBCS data is being processed, the DBCS invocation option can control the identification of the DBCS used. In particular, a user with a display that does not support real DBCS or with a display supporting one DBCS can simulate operation on a display with a different DBCS.

If the DBCS invocation option is not used, APL2 determines whether or not mixed APL2/DBCS data is to be displayed based on the presence of DBCS support as reported by GDDM. If GDDM reports that the display has a DBCS, or that GDDM emulated DBCS support is available, then APL2 allows mixed DBCS data. For displays that have real DBCS support, the DBCS ID defaults to the CPGID of the DBCS display. For GDDM emulated DBCS support, the DBCS ID defaults to 0.

TRY
: If GDDM indicates that the display has a DBCS, then activate DBCS support and use the CPGID as the DBCS identifier.

  If GDDM indicates that emulated DBCS support is being used, then treat as DBCS(0).

  If GDDM is not available, or it indicates that no DBCS support is available, then treat as DBCS(OFF).

  Note: DBCS(TRY) is the default in the installation options module as distributed by IBM, but your installation can choose a different default.

ON
: Behaves like DBCS(TRY), except this ends the APL2 session if DBCS is not available rather than treating as DBCS(OFF).

OFF
: - Substitute `'ω'` for any DBCS characters on terminal output.
  - Do not accept DBCS on terminal input.
  - Use 0 as the DBCS identifier when auxiliary processors and function routines convert DBCS data to APL data.
  - Permit any DBCS identifier when auxiliary processors and function routines convert APL data to DBCS data.

nnn
: Use the decimal value *nnn* as the DBCS identifier and activate DBCS display support. The value of *nnn* must be between 0 and 32,767 inclusive.

  Only APL data with the specified DBCS identifier can be converted to DBCS data by auxiliary processors, function routines, or terminal output processors.

  Note: DBCS(0) is not equivalent to DBCS(OFF), even for auxiliary processors or function routines.

You can change this invocation option during the APL2 session by either the *OPTION* external function or *)CHECK SYSTEM DBCS*. This restarts APL2 display support. For the session manager, this means that any in-storage log file is discarded.

DEBUG(*nnn*)

Alters normal error recovery actions taken by APL2. It can assist in debugging errors.

Note: Your installation can supply one or more debug settings that you cannot override.

nnn

One or more numbers (positive or negative) of the debug settings you want to change. A positive number turns on one or more settings, whereas a negative number turns off one or more settings. Several settings can be turned on or turned off with a single number equal to the sum of the numbers for the individual settings. For example, to specify debug settings 2 and 4, specify either:

```
DEBUG (2 4)
```
or
```
DEBUG (6)
```

To turn off setting 4, specify:

```
DEBUG (-4)
```

IBM-supplied default value: 0

Your installation can provide a default DEBUG value other than 0.

The debug settings and their meanings are:

1—MSG Displays secondary error messages without prompting.

Note: Use of this setting produces messages for exceptional conditions that are not necessarily error conditions. For example, end of file may or may not be an error, depending on the situation. An error message is displayed immediately, before an attempt to determine if the condition is acceptable.

2—ECHO Causes all input to APL2 from the AP 101 input stack to be "echoed" (displayed) at your terminal. Normally, input from the AP 101 stack is not displayed at the terminal as it is executed.

4—XDUMP Default setting. Provides more complete dumps. Typical dumps are about 50 pages long. Under this setting, they can be 200 to 500 pages or more, depending on the workspace size. For a further explanation of APL2 storage dumps, see *APL2/370 Diagnosis Guide*.

8—ESTIMATE TSO Only. Provides estimates of the progress of long-running library operations, such as)*LOAD*,)*COPY*, or)*SAVE*.

32—MSGID Provides the message identifier number, along with the text of any message produced by APL2. For example, the following message:

```
 DOMAIN ERROR
```
displays with DEBUG(32) as:
```
 AP2IAPL254 DOMAIN ERROR
```

64—NOLX Suppresses processing of ⎕*LX*, the latent expression, when workspaces are loaded with the)*LOAD* command.

128—NOQUEMSG Discards secondary messages rather than queuing them. While 1—MSG is turned on, secondary messages are immediately displayed rather than queued, so the setting of this flag is irrelevant.

DSOPEN(*device-token***)**

Specifies a value to be passed to the Graphical Data Display Manager (GDDM) as the device-token parameter on a GDDM DSOPEN call. This option permits more precise declaration of the terminal type when GDDM cannot automatically determine the correct terminal characteristics.

You must specify a device token with DSOPEN if you want to access GDDM through AP 126 while running APL2 in TSO batch or in a CMS disconnected virtual machine. Otherwise AP 126 issues GDDM calls during AP 126 initialization that tell GDDM to use device characteristics of the current terminal type. Since no terminal exists in TSO batch or in a CMS disconnected virtual machine, GDDM may hang or otherwise fail to initialize.

You also may need to specify a device token with DSOPEN if you are using a 24-line display screen under TSO. In this case, GDDM cannot determine whether the terminal type is a 3277 or a 3278, and simply assumes some terminal type that may not be correct. For more information about terminal types, see the description of the TERMCODE option below.

This option is meaningful only when the APL2 session manager, AP 126, or)*EDITOR 2* is used.

device-token

Selected device tokens supplied by GDDM include:

ADMK772A 3277 model 2 with APL

ADMK782A 3278 model 2 with APL

For more values, see the description of the device tokens in *GDDM Base Application Programming Reference*.

An example of using the DSOPEN option is:

```
DSOPEN(ADMK782A)
```

EXCLUDE(*name...***)**

Specifies the module names of auxiliary processors normally available but that you do not want loaded at the start of your APL2 session. For example, to exclude AP 127, the SQL processor, from being loaded when you invoke APL2 in the TSO environment, include the EXCLUDE option in the APL2 command as follows:

```
EXCLUDE(AP2X127)
```

Figure 4 contains the load module names of auxiliary processors supplied with APL2. Check with your system administrator to find out the names of any additional processors that may be automatically available at your installation.

Figure 4. Module Names of Auxiliary Processors Supplied with APL2

Auxiliary Processor	CMS Module Name	TSO Module Name
AP 100	AP2V100	AP2T100
AP 101	AP2V101	AP2T101
AP 102	AP2V102	AP2T102
AP 110	AP2V110	*not available*
AP 111	AP2V111	AP2T111
AP 119	AP2X119	AP2X119
AP 120	AP2X120	AP2X120
AP 121	AP2X121	AP2X121
AP 123	AP2V123	AP2T123
AP 124	AP2X124	AP2X124
AP 126	AP2X126	AP2X126
AP 127	AP2X127	AP2X127
AP 210	*not available*	AP2T210
AP 211	AP2X211	AP2X211

Note: Excluding auxiliary processors can reduce storage requirements for APL2, particularly on TSO. However, excluding certain IBM auxiliary processors can cause problems because of internal APL2 dependencies on these processors. The following auxiliary processors can be safely excluded, if users do not need them: AP 119, AP 120, AP 121, AP 123, AP 124, AP 127, AP 210, and AP 211.

FREESIZE(*size*)

Specifies the amount of virtual storage that must remain unused after space is allocated for the active workspace, shared variables, and other areas allocated during invocation. These areas include modules and work areas used by other products such as VSAM and GDDM. FREESIZE may be needed during the session for the APL2 interpreter module, auxiliary processor modules, access method buffers, dynamically loaded modules, storage control blocks, and program products such as GDDM.

size

An integer expressed in bytes, kilobytes, or megabytes. For example:

```
FREESIZE(65536)
      or
FREESIZE(64K)
```

A percentage of your virtual machine size. For example:

```
FREESIZE(10%)
```

Your installation provides a default value for FREESIZE. If the minimum amount of FREESIZE is not available, APL2 invocation fails.

Because of the number of variable factors associated with APL2 storage allocations, you may find it better to omit the FREESIZE option and use the WSSIZE and SHRSIZE options to reduce the amount of space allocated to the active workspace and shared variables. If you cannot reasonably reduce WSSIZE and SHRSIZE and you have virtual storage problems, increase your CMS virtual machine size or your TSO region size.

HILIGHT(*setting*)

Specifies whether input, output, or both are to be highlighted on the screen.

Under CMS: This option applies only when the APL2 session manager is used.

Under TSO: This option applies, regardless of whether the APL2 session manager is being used.

setting
One of the following:

INPUT Highlight only input lines. **(CMS default.)**

OUTPUT Highlight only output lines. **(TSO default.)**

ON Highlight all lines.

OFF Highlight no lines.

For example, to have no lines highlighted during your APL2 session, specify:

```
HILIGHT(OFF)
```

CMS Comments: If the session manager is not used, the setting of the CP TERMINAL HILIGHT command determines whether or not user input lines are highlighted.

ID(*nnnnnnn*)

Specifies a numeric identifier to be associated with your current APL2 session. The number becomes the first item in the system variable ⎕*AI*. (For a description, see *APL2 Programming: Language Reference*.) This number identifies your:

- Default library for saved workspaces (including any *CONTINUE* workspace)
- Default library for files created through AP 121, the APL2 data file processor
- Library for APL2 session manager log files
- Library for)*COPY* work files (TSO only)

The number you enter in the parameter to the ID option is also used to identify you as a possible share partner for shared variable communication between users on the same system. The value of the ID parameter does not affect a user's ability to share variables across systems. TCP/IP profile files are used to define potential cross-system partners. (See “Processor Profile Structure” on page 87.)

nnnnnnn
Should be at least 1000 to avoid conflict with auxiliary processor identifiers (usually 100 through 999). The number cannot be greater than 9999999.

For example, to identify yourself as user 1234 when you invoke APL2, specify:

```
ID(1234)
```

If two or more APL2 users or auxiliary processors have the same ID number, only one of these users can communicate with the shared variable processor (SVP).

Your installation can set or change your ID and override any value you provide. It can also restrict the use of user-to-user shared variables on the same system. Check with your system administrator for APL2 ID requirements at your site.

Unless your installation overrides your specification, if you do not specify an ID (or if you specify ID(0)) when you invoke APL2, you are not permitted to share variables with other users except using cross-systems shared variables.

INPUT(**'** *string***'** ....)

Specifies one or more input lines used when APL2 is invoked. See Chapter 7, "Controlling APL2 Invocation" on page 82 for the relationship between the different methods for providing input to APL2.

' *string***' . . .**
Each character string represents a line of input to APL2. Strings are separated by one or more blanks or commas.

Each string is enclosed in a pair of quotation marks. If quotation marks are a part of an expression, a *pair* of single quotation marks must be entered for each *single* quotation mark in the expression.

The example below illustrates the use of the INPUT option to load an APL2 workspace named *PAYROLL*, and then execute a function named *START* with an argument **'***ABC***'**.

```
INPUT(')LOAD PAYROLL'  ' START ''ABC''')
```

If you do not specify the INPUT option when invoking APL2, the APL2 session begins with the *CLEAR WS* message or the loading of a *CONTINUE* workspace. If you have a damaged *CONTINUE* workspace and want to suppress the automatic loading of it, you can specify the INPUT invocation option with no data:

```
INPUT('   ').
```

TSO Note: If you specify the INPUT option and a *CONTINUE* workspace exists, you receive a message warning that a *CONTINUE* workspace exists, but it is not loaded. If the warning appears, you may want to load the contents of your *CONTINUE* workspace and save it into a workspace of a different name. Otherwise, forced termination of your session may cause your active workspace to be saved in the *CONTINUE* workspace, whose previous contents are lost.

If you invoke APL2 from a CLIST and pass a quoted string as a parameter, the CLIST processor strips off the single quotation marks around the strings. The sample CLIST below shows the recommended method of invoking APL2 with a symbolic INPUT parameter:

```
APL2.CLIST
PROC 0 IN('')      /* specify INPUT as a parameter */
CONTROL NOLIST NOPROMPT NOMSG
APL2 IN('&IN')   /* note the enclosing quotation marks */
```

You can invoke the CLIST by typing:

```
APL2 IN(')LOAD PAYROLL')
```

When APL2 is invoked from a CLIST, all lowercase characters and numbers are folded to uppercase. Some APL2 characters cannot be passed to APL2 if the INPUT option is used within a CLIST. You can avoid this restriction by invoking APL2 without using a CLIST.

For more information, see *OS/VS2 TSO Terminal User's Guide*.

The AISIZE parameter must provide enough space to hold all INPUT strings.

LOADLIB(*dsname...***)** **(TSO Only)**

Specifies one or more MVS private load libraries from which auxiliary processors, Processor 11 programs, or secondary APL2 modules (including the interpreter) can be loaded. The LOADLIB option is an alternative to the ALLOCATE command issued for load libraries needed during your APL2 session. You can use this option to:

- Specify the load library that contains an auxiliary processor specified in the APNAMES invocation option
- Replace, for your current APL2 session, the APL2 interpreter module, AP2INTRP (to test an APAR fix, for example)
- Identify a task library for modules that are loaded by AP 100 in response to the **APL ATTACH** built-in command
- Identify a library of modules loaded by Processor 11.

When you specify a load library with this option, APL2 makes the listed data sets part of the MVS TASKLIB search for the duration of the APL2 session.

Library names are separated by one or more blanks. For example, to specify that user library *userid*.MY.LOADLIB and system library PROD.PAY.LOADLIB are to be searched for modules that are used during your APL2 session, specify the following:

```
LOADLIB(MY.LOADLIB 'PROD.PAY.LOADLIB')
```

A maximum of 111 data set names can be specified. If ddname LOADLIB was previously allocated, it is replaced. The ddnames LOADLIB0 through LOADLIB9 are used temporarily as needed, and if more than 11 data sets are listed, ddnames LOADLI00 through LOADLI99 are also used temporarily. These names must not be in use when APL2 is invoked with the LOADLIB option.

Note: If an auxiliary processor is in none of the load libraries specified in the LOADLIB option, the MVS system libraries are searched. You can duplicate the effect of the LOADLIB option by using the ALLOCATE command. Prior to invoking APL2, allocate your libraries with the ddname LOADLIB. If you use the LOADLIB option in addition to allocation with the ddname LOADLIB, the LOADLIB invocation option overrides the allocation.

NLT(*language*)

Allows users to override the installation default for the national language in their invocation EXEC or CLIST, and allows the language to be set before APL2 is running. For example:

```
APL2 NLT(ESPANOL)
```

For a list of the national languages supported by APL2, see Appendix C, “National Languages Supported by APL2” on page 368.

PROFILE(*name*) **(SMAPL Only)**

Specifies the name of the file containing the profile to be used by the APL2 session manager. During its initialization, the session manager executes a PROFILE LOAD command using the name that you specify. (See the description of the PROFILE LOAD session manager command under “PROFILE” on page 58.)

name

CMS: File name of the profile to be loaded. The file type of the named profile must be VSAPLPR.

TSO: Second-level qualifier of the profile to be loaded. The first-level qualifier is the TSO profile-prefix and the third-level qualifier is VSAPLPR.

If you are using the session manager without specifying the PROFILE option, the profile named DEFAULT is loaded. (If more than one DEFAULT profile exists, the host system search order determines which one is loaded.)

If you are not using the session manager, the PROFILE option is ignored.

If you specify PROFILE( )—no profile name—the session manager does not execute a PROFILE LOAD command.

For further information on session manager profiles, see Chapter 3, “The APL2 Session Manager,” especially “PROFILE” on page 58 .

QUIET[**(ON|OFF)**]

ON

Prevents APL2 from displaying any output until APL2 prompts for input.

OFF

Permits APL2 to display all output from the beginning of the session. It is the default.

For instance, if you use the INPUT option and the QUIET option in an APL2 command stack using AP 101, the alternate input (stack) processor, you can load a workspace and start an application without displaying the APL2 initiation messages that would normally be displayed. If that application uses AP 101 to stack an *)OFF* system command, APL2 termination messages are suppressed.

Note: The QUIET option applies only to output from the APL2 language processor. All other output, such as that from AP 126, is not suppressed.

If you are using the session manager, the QUIET option does not suppress the initial display of the session manager screen if the session manager profile includes the DISPLAY(ON) command. To suppress initial display of the session manager screen, specify the PROFILE( ) option or load a session manager profile that does not include the DISPLAY(ON) command.

Use the external function OPTION to change the QUIET setting. See *APL2/370 Programming: Using the Supplied Routines* for more information.

RUN([*'locator'*] *function*)

This option is designed to simplify invocation of an external function as a part of entry to APL2.

locator
Optional information to be used by Processor 11 to locate the external function. If provided, this must be a character string that can be used as the first item of the left argument to ⎕*NA* for processor 11. When this information is provided, the external function must either reside in a namespace or be self-describing. The possible formats are:

'*member***'**
For CMS, the name of a nucleus extension, MODULE file, or TEXT file containing the external function. For TSO, the name of a load module that is available through the normal load library search order (including any LOADLIB ddname or invocation option.)

'*library.member***'**
For CMS, *library* is the filename of a LOADLIB file, and *member* is the name of a member in the library containing the function. For TSO, *library* is a ddname to which a load library must be allocated before invoking APL2, and *member* is the name of a member in the library containing the function.

If *locator* is omitted, the function must be described in a NAMES file entry for the *function* using the defaults NAMES files for the APL2 session.

function
A 1 to 8 character APL name containing only alphanumeric characters (no ⍙, _, ¯, or underbarred characters.) Note that lowercase letters are converted to uppercase while processing the invocation RUN option.

For more information, see Chapter 28, “Processor 11—Calling Compiled Programs” on page 291 and Chapter 29, “Processor 11—Access to Namespaces” on page 334. The RUN option produces a pair of statements that are executed before anything provided by the INPUT option, the stack, or the APLIN file. The first statement is a ⎕*NA* for the requested function, and the second is a simple niladic invocation of it.

RUN provides only minimum control of application invocation. For more complex situations, use the INPUT option or provide an APLIN file.

SHRSIZE(*size*)

Specifies the amount of virtual storage to be reserved for the shared variable processor. The maximum size of data in a variable shared with a local auxiliary processor is typically limited to 10 to 15 thousand bytes less than this value, though it can be limited further by available free storage, available workspace storage, or auxiliary processor limits.

size

An integer expressed as bytes, kilobytes, or megabytes, as in the following example:

```
SHRSIZE(256000)
      or
SHRSIZE(250K)
```

A percentage of the CMS virtual machine size or TSO logon size. For example:

```
SHRSIZE(5%)
```

Minimum size: 16K bytes
Maximum size: 16 megabytes
IBM-supplied default: 40K bytes

These sizes can be changed by your installation.

Note: Because of rounding, the actual SHRSIZE size assigned to your APL2 session can be slightly different from what you specify. Issue the *)QUOTA* system command to display the actual share size used. Your installation can allocate additional space for shared storage to support user-to-user shared variables. This space is not reported in the *)QUOTA* command. The *)QUOTA* command is described in *APL2 Programming: Language Reference*.

Typically, SHRSIZE values should be smaller than WSSIZE. If SHRSIZE is too small, the interpreter may signal a *SYSTEM LIMIT*+ error during the APL2 session, with a *)MORE* message indicating *Interface capacity*.

In a non-XA environment, SHRSIZE and WSSIZE specify areas that are allocated below the 16M line; in an XA environment, they specify areas that can be allocated above the line, unless invocation XA(24) is specified.

SMAPL(TRY|ON|OFF|*nnnn*)

Indicates whether you want to use the APL2 session manager for your APL2 session. The session manager requires GDDM and an IBM 3270 series display station.

TRY
Invokes the APL2 session manager if it is available.

TRY is the default setting.

ON
Invokes the APL2 session manager.

If the session manager is not available and you specify SMAPL(ON), you receive a message, and your APL2 session is not initialized.

OFF
Does not invoke the APL2 session manager.

If you specify SMAPL(OFF), APL2 uses the standard input/output protocol of the host system—CMS or TSO. If you do not use the APL2 session manager, but GDDM is available, you can still use the Editor 2 and AP 126, the GDDM Processor.

nnnn
If the SMAPL value is numeric, it indicates that the Shared Variable Interpreter Interface is to be used. The APL2 interpreter shares a variable with processor *nnnn*. All subsequent input and output to the interpreter is performed through the shared variable.

The Shared Variable Interpreter Interface is designed to allow the interpreter to be controlled by another user ID, using an APL2 session manager or an APL2 function. The other ID can be on the same VM or MVS system, or can be on a different system, provided the two systems are connected through TCP/IP. Once a variable is shared with the interpreter, the interpreter sends messages, arrays, and requests for input through the shared variable.

For further details about the shared variable interpreter's protocols, consult Chapter 25, “APL2 Shared Variable Interpreter Interface” on page 261. For information about how to interactively communicate with the interpreter from another CMS or TSO session, consult the discussion of the *RAPL2* function in *APL2/370 Programming: Using the Supplied Routines*.

SVMAX(*nnnnn*)

Specifies the maximum number of shared variables you can share concurrently.

nnnnn
Must be a positive integer.

Minimum value: 4
IBM-supplied default: 88
Maximum value: 32767

The default value may have been changed during installation.

To display the current value of this option after you invoke APL2, issue the)*QUOTA* system command.)*QUOTA* is described in *APL2 Programming: Language Reference*.

Note: The maximum number of shared variables is also limited by the size of shared storage. Each shared variable requires at least 128 bytes of space in shared storage.

SYSDEBUG(*nnn*)

Establishes special debug settings for your APL2 session. These settings provide information beyond that provided by the DEBUG option. They are intended for use by system programmers in diagnosing system or internal APL2 problems or problems with writing an auxiliary processor. They are not for use during normal APL2 operation; their use can significantly degrade APL2 performance.

For a description of the SYSDEBUG option, see *APL2/370 Diagnosis Guide*.

TERMCODE(*nnnnn*)
or
CODE(*nnnnn*)

Used by APL2 to identify the type of terminal you are using.

Under TSO, if you do not specify this option, you may be prompted to enter a *shift-6* character if APL2 cannot determine your terminal type dynamically. This *shift-6* character allows APL2 to identify your terminal.

If you are using the session manager or AP 126 and GDDM is unable to correctly identify your terminal type, use the DSOPEN invocation option rather than TERMCODE to specify the correct terminal type.

nnnnn
Code identifying the terminal.

The value for the TERMCODE option must be one of the IBM-supplied device codes shown in Figure 5 on page 30.

TERMCODE(-1) can be used to request controlled invocation.

The invocation parameter TERMCODE(-1) indicates that APL2:

- Is being invoked in a “controlled invocation” environment
- Is to use the file allocated (FILEDEFed under CMS) to ddname APLIN to satisfy its input requirements.
- Is to put all its output to the file allocated to the ddname APLPRINT.

The use of TERMCODE(-1) is described in detail in Chapter 7, “Controlling APL2 Invocation” on page 82.

Note: All values other than -1 are TSO only.

On CMS, the APLIN and APLPRINT file requirements are as follows:

The APLIN file can be defined on any input device supported by the FILEDEF command.

The APLPRINT file can be defined on any output device supported by the FILEDEF command. The default is variable length records with a data length limit of 121, including ASA print control characters. If the record format was specified as fixed, then the default record length is 80.

On TSO, the APLIN and APLPRINT file requirements are as follows:

The APLIN file can have the following characteristics:

- The record format (RECFM) can be U, F, FB, V, or VB.
- For fixed-length record files (RECFM=F or FB), the record length (LRECL) can be up to 255.
- For variable-length record files, (RECFM=V or VB), the record length (LRECL) can be up to 255, including the record descriptor word (RDW).
- For undefined record format files (RECFM=U), the record length (LRECL) or block size (BLKSIZE), whichever is specified, can be up to 255.

The APLPRINT file can have the following characteristics:

- The record format (RECFM) can be F, FB, V, or VB. The default RECFM is VB.
- The record format can also indicate either ANSI carriage control (A) or machine carriage control (M).
- The record length (LRECL) can be up to 255, including the record descriptor word (RDW) for variable-length record files (RECFM=V or VB). The default LRECL is 125. The default block size (BLKSIZE) is LRECL for fixed-length record files (RECFM=F or FB) or LRECL plus 4, which is the length of the block descriptor word (BDW) for variable-length record files (RECFM=V or VB).

Figure 5 (Page 1 of 2). Device Codes for the TERMCODE Option

Code	Description
-1	Controlled invocation. APL2 avoids using the terminal if controlled invocation is used. Instead, APL2 redirects input and output requests to files, much as if it were running in TSO batch or on a disconnected virtual machine under CMS. Controlled invocation is intended for use by applications that are themselves using the terminal and that do not want APL2 to interfere with their terminal input and output. See Chapter 7, “Controlling APL2 Invocation” on page 82 for more information. **Note:** TSO translates certain characters when entered using a CLIST. If you want to specify this termcode using a TSO CLIST, use the TSO minus character ('-') instead of the APL overbar ('¯').
0	Unknown terminal type. APL2 attempts to determine the terminal code from the available system information. If insufficient information is available, the APL2 characters can be displayed or translated incorrectly.
1	Pass through translation. The APL2 system passes characters to TSO without translation.
2	In MVS batch execution, this code uses the Technical Notation (TN) translation for output directed to the data set identified by ddname APLPRINT. Such translation is useful when output is directed to impact printers that have the TN print chain (train), instead of the APL print character set.
27411	IBM 2741 terminal with correspondence code using type element 987 for the APL feature.
27412	IBM 2741 terminal with BCD or EBCD code using type element 988 for the APL feature.
3101	IBM 3101 terminal.
3178	IBM 3178 terminal without the APL feature.
3179	IBM 3179 terminal without the APL feature.
31791	IBM 31791 terminal with the APL feature.
3180	IBM 3180 terminal without the APL feature.
31801	IBM 3180 terminal with the APL feature.
3232	IBM 3232 terminal.
3270	IBM 3270 terminal with no lowercase or APL feature. Lowercase input and output are folded to uppercase.
32702	IBM 3270 family terminal with the APL feature, including the characters ⍥, ⍁, ⍷, ≡, ⁖, ⍸, and ⍞.
3277	IBM 3277 terminal without the APL feature. Uppercase and lowercase characters are permitted.
32771	IBM 3277 display station with the Data Analysis-APL feature code 1066.
3278	IBM 3278 without the APL feature.
32781	IBM 3278 with the APL feature code 1120.
3279	IBM 3279 terminal without the APL feature.
32791	IBM 3279 terminal with the APL feature code 1120.
3290	IBM 3290 terminal without the APL feature.
32901	IBM 3290 terminal with the APL feature.
3335	ASCII terminal without the APL feature.

Figure 5 (Page 2 of 2). Device Codes for the TERMCODE Option

Code	Description
33351	ASCII terminal with APL typewriter-pairing keyboard arrangement.
33352	ASCII terminal with APL bit-pairing keyboard arrangement.
3767	IBM 3767 without the APL feature.
37671	IBM 3767 with the APL feature.
8775	IBM 8775 without the APL feature.
87751	IBM 8775 with the APL feature.

TRACE(*nnn***)**

Provides system diagnostic output during your APL2 session. The option is intended for use by system programmers in diagnosing system or internal APL2 problems. It is not for use during normal APL2 operation and can significantly degrade APL2 performance; its use also interferes with normal terminal communication.

For a description of the TRACE option, see *APL2/370 Diagnosis Guide*.

WSSIZE(*size***)**

Specifies the amount of *contiguous* virtual storage in your virtual machine to be reserved for your active workspace.

size

Integer expressed in bytes, kilobytes, or megabytes, as in the following:

```
WSSIZE(1048576)

WSSIZE(1024K)

WSSIZE(1M)
```

The size can also be expressed as a percentage of the CMS virtual machine size or the TSO logon size. For example:

```
WSSIZE(40%)
```

Note: Under MVS/XA*, you can use numbers larger than 100% to obtain an extended region size larger than your TSO private region.

In CMS, the size can also be specified as a negative amount. In that case, the workspace allocated is the largest size possible greater than the specified amount:

```
WSSIZE(-1M)

WSSIZE(-1024K)

WSSIZE(-40%)
```

Minimum size: 16K bytes
Maximum size: APL supports up to 2047 megabytes under CMS and 1008 megabytes under TSO (except that the TSO VSAM library system imposes a limit of 128 megabytes). The available storage may be limited further by your system.
IBM-supplied default: 25% of CMS virtual machine size or TSO logon size

You can issue the)*QUOTA* system command after invoking APL2 to display the default size of the active workspace. The)*QUOTA* command is described in *APL2 Programming: Language Reference*.

XA(*nn*)

Identifies the address range from which working storage should be allocated on XA or ESA systems.

nn One of the following:

24 Working storage should only be allocated below the 16-megabyte boundary.

31 Working storage can be allocated either above or below the 16-megabyte boundary.

If no XA( ) value is specified, then working storage is allocated where ever it is available. On XA or ESA systems, this may be above the 16-megabyte boundary.

IBM does not supply a default setting for XA( ). Your installation can supply a default, or specify an overriding value.

If you are on a system that does not support XA or ESA mode addressing, specifying XA(31) causes invocation to fail.

Even when XA(31) is specified or defaulted, some pieces of working storage must be allocated below the 16-megabyte line due to macro or access method restrictions.

Session Termination under CMS

When you enter the system command)*OFF* or)*CONTINUE*, APL2 calls the AP2EXIT EXEC again and processes the commands that reset your terminal for the continuation of your CMS session. Control then returns to CMS or to the program or the EXEC that invoked APL2.

APL2 returns return code 0 if an)*OFF* or)*CONTINUE* system command successfully ended the session.

APL2 returns return code 4 if all sources of input are exhausted but no)*OFF* or)*CONTINUE* system command was encountered. This situation can occur, for instance, under CMS batch, when there is no system command to end the APL2 session in the input stream. APL2 saves the active workspace in *CONTINUE* before terminating if this situation does occur.

APL2 returns return code 8 if an error occurs that prevents APL2 from initializing. This situation can occur, for instance, if there is insufficient storage.

APL2 returns return code 16 if an error occurs after initialization that forces APL2 to terminate further processing abnormally. This situation can occur, for instance, when APL2 has part of its program storage over-written by another program.

Any input lines remaining on the AP 101 alternate input stack after `)OFF` or `)CONTINUE` are available to CMS, unless APL2 was invoked with TERMCODE(-1).

Session Interruption under CMS

If a system interrupt occurs during your session and you are disconnected, you can attempt to save any new work since your last `)SAVE` by reconnecting to your virtual machine. Do the following:

1. Log on to the same type of terminal you were using before. If you log on to a different type of terminal, APL2 may be unable to properly process your terminal input and output.
2. If you are reconnected to the control processor (CP), a message similar to the following appears:

   ```
   RECONNECTED AT 07:47:31 PDT MONDAY 03/30/92
   ```

   Your installation can impose a time limit on how long you can be disconnected. If you do not reconnect within the time limit, CP logs off your virtual machine, and you lose the contents of your active workspace. When logging back on, you see the usual initialization messages instead of the RECONNECTED message.
3. Restore the CP settings established by AP2EXIT EXEC before you attempt to restart APL2. (When you reconnect, AP2EXIT EXEC is not processed.)

   The CP settings in the AP2EXIT EXEC supplied with APL2 for a display terminal are:

   ```
   TERMINAL APL ON
   TERMINAL ATTN OFF (for non-3270 terminals)
   ```
4. Enter the CP command:

   ```
   BEGIN (or B)
   ```

   This command takes you from CP mode and returns you to your APL2 session.

Session Termination under TSO

When you issue the system command `)OFF` or `)CONTINUE`, control returns to TSO so that you can continue your TSO session. If APL2 was invoked from a CLIST, the CLIST may regain control if coded to do so.

When APL2 is successfully terminated, any unused lines on the AP 101 input stack are passed to TSO as if they formed a command procedure (CLIST).

If APL2 is restarted as a result of a stacked APL2 command, the input stack for the new session is empty, but any previously stacked lines following the APL2 command remain in the CLIST, which regains control when APL2 terminates again.

Session Interruption under TSO

A line interrupt, repeated attention signals, or a system or operator cancel can force termination of an APL2 session. An abnormal termination purges lines on the AP 101 input stack. During such termination, however, APL2 attempts to save your active workspace into a workspace named *CONTINUE* in your default library. Your default library is the one identified by the ID invocation parameter.

If you log on again and a *CONTINUE* workspace exists in your default library, APL2 displays a message such as the one below, instead of the *CLEAR WS* message:

```
SAVED 1994-03-27 18.22.57 (GMT-4)
```

Check the contents of the *CONTINUE* workspace using the)*NMS* command. If you want to save the data, you can use the APL2)*WSID* and)*SAVE* system commands to rename and save the *CONTINUE* workspace. If you do not save the data and another interrupt occurs, APL2 writes the contents of the active workspace into the *CONTINUE* workspace, replacing any previous data.

The *CONTINUE* workspace is autosaved into the user's default library, which can be either SAM or VSAM:

- If a W0 ddname (or W*nnn* ddname where *nnn* is the default library) was allocated, the *CONTINUE* is saved in that VSAM library.
- If the autosave into the user's VSAM default library fails, APL2 attempts to save the *CONTINUE* workspace into the user's default SAM library.
- Whether or not the first autosave is into the VSAM or SAM default library, APL2 attempts to drop the SAM library *CONTINUE* workspace, and tries the autosave again if the autosave into the SAM library fails.
- If the drop or the second autosave into the SAM default library fails, APL2 attempts to dump the active workspace (as DUMP*nnnn*) into the user's default SAM library.
- If none of these operations can be done, then APL2 terminates without saving an active workspace.

If the active workspace was either autosaved or dumped into the user's SAM default library, the user has to invoke APL2 with a different ID, or without allocating a VSAM default library in order to retrieve the data. You can allocate other VSAM libraries while retrieving the data. For example, if your ID was 1001 when the *CONTINUE* workspace was autosaved into the SAM library, you could allocate your normal default library as W1000. This allocation would permit you to resave the data in your normal default library by specifying library 1000.

When your TSO user ID is not the same as the PREFIX setting of the TSO PROFILE command, APL2 uses the ddname CONTINUE to access the *CONTINUE* workspace in your default SAM library. Because of this, you can prevent simultaneous access of the *CONTINUE* workspace by two TSO users sharing the same prefix, or by a batch job and your TSO session.

Allocate a data set to ddname CONTINUE. If you do not, and you attempt to access the *CONTINUE* workspace in your default SAM library, you receive the error message *IMPROPER LIBRARY REFERENCE*. If this happens during a forced session termination, your active workspace is dumped.

If you allocate the ddname CONTINUE and you use the)*LIB* system command to list the names of the workspaces in your default SAM library, the name *CONTINUE* appears as the first workspace in the library. All other names appear in alphabetic order following it. Only during)*LOAD* and)*COPY* operations does APL2 check whether the file allocated to the ddname CONTINUE actually contains a real APL workspace.

If you use the)*DROP* system command to drop the *CONTINUE* workspace in your default SAM library, APL2 empties the data set allocated to the ddname CONTINUE. The empty data set remains allocated, and the name CONTINUE appears in)*LIB* reports for your default library.

Chapter 3. The APL2 Session Manager

The APL2 session manager provides a full-screen interface to APL2 for users of the IBM 3270 family of display stations at installations where the Graphical Data Display Manager (GDDM) Licensed Program is installed. The APL2 session manager may be used to enter input and display output in immediate execution mode. It can be used to respond to □ and ⍞ prompts. With the line editor, it can also be used to control input and output in definition mode. The two APL2 editors are described in *APL2 Programming: Language Reference*.

Use of the session manager is optional and is controlled by the availability of GDDM and the value of the SMAPL invocation option: The APL2 session manager requires GDDM. The default setting for the SMAPL option is TRY (try to invoke the session manager). If GDDM is available, the session manager is invoked. If GDDM is not available, the session manager is not invoked.

If the session manager is not used, the terminal is under control of your system's standard input/output protocol.

When the session manager is used, a session manager profile is automatically loaded as part of APL2 invocation. The profile contains settings for function keys, specifications for the display of data, and default settings for copying lines of APL2 I/O to another file.

You can supply the name of the session manager profile to be loaded in the PROFILE option of the APL2 command. If you do not supply a name, the name DEFAULT is used. For more information about the session manager profile, see “PROFILE” on page 58. For more information on APL2 invocation, see Chapter 2, “APL2 Invocation and Termination” on page 8.

The APL2 session manager assumes GDDM is installed with EBCDIC code page 351 as the default code page. If your installation has customized GDDM so that some other code page is the default, then APL characters may not be handled properly. To correct this you must reset the default with a GDDM external defaults file. GDDM external defaults files are described in detail in *GDDM System Customization and Administration*.

Under CMS, you do this by creating a file on an accessed disk. By default, GDDM expects the name for this file to be PROFILE ADMDEFS, but this name can be customized at your installation.

Under TSO, you do this by creating a sequential data set with RECFM(F) and LRECL(80). Any data set name can be used, but the data set must be allocated as a specific FILENAME before APL2 is invoked. By default, GDDM expects the data set to be allocated as FILENAME(ADMDEFS), but this name can be customized at your installation.

In either case, the file or data set should contain the following record:

```
 ADMMDFT APPCPG=351
```

Note: The leading blank is required before ADMMDFT.

Features of the APL2 Session Manager

When an APL2 session is started with the session manager active, normal input and output are mediated by the session manager. Editor 2 and AP 126 are independent of the session manager.

If the line editor is invoked, the session manager works with the line editor to control input to the definition of a defined function or operator.

The full-screen interface to APL2 provided by the session manager allows you to:

- Create, modify, and reuse one or more lines of input or output as new lines of input in immediate execution mode in response to ☐ and ⍞ and in definition mode with the line editor.
- Define program function (PF) key settings to reduce the need to type commonly entered lines.
- Tailor the session manager profile to make the session manager more efficient for you.
- Create a log of APL2 input/output lines to be saved between sessions.
- Print portions of the log.
- Use the features of the IBM 3270 and 3290 series of terminals for scrolling and other display control facilities.
- Display the seven new APL2 characters on terminals that have the Programmed Symbol Set (PSS) feature (unavailable on IBM 3277 display stations). The complete APL2 character set is loaded into a PSS in your terminal when GDDM is activated. The new characters must be entered at a display terminal using the printable backspace character, which is activated by entering the APL2 system command `)PBS ON`.

  If the session manager is not used or if PSS is unavailable, the new characters are not displayed as expected. The difference in the display of the characters does not affect the execution of the objects containing new characters.

  Session manager keywords and messages can be translated.

The Session Manager Screen

The session manager replaces the standard CMS or TSO screen on your terminal with its own display screen. The size and position of the session manager screen in relation to the terminal's physical display screen can be controlled by the SIZE and ORIGIN operands of the session manager DISPLAY command, described under “DISPLAY” on page 50. The default size and position cause the session manager screen to completely fill the display screen so that the session manager alone controls the terminal screen. The session manager screen is illustrated in Figure 6 on page 38.

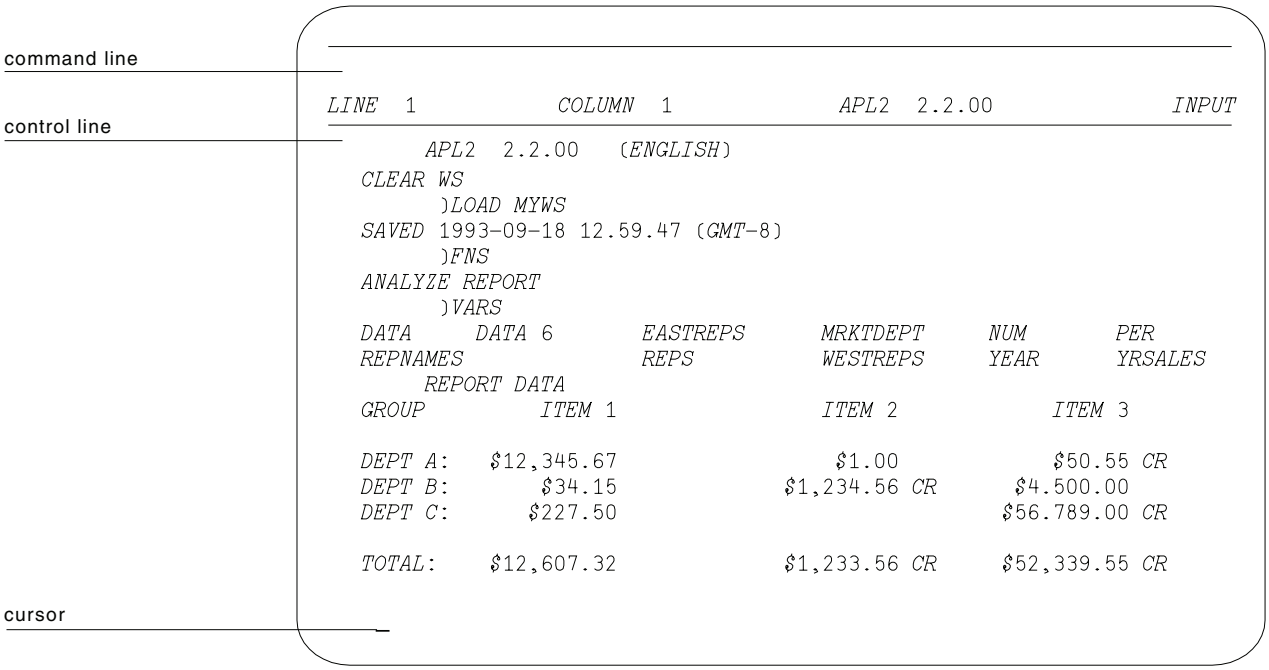


Figure 6. Session Manager Screen

Command Line: The top row of the screen is the *command line* at which you enter session manager commands. For a description of the session manager commands, see “APL2 Session Manager Commands” on page 43.

Control Line: The second line of the screen is the *control line*, which contains:

- LINE *nnnn*—the number of the session log line currently displayed on line 3 of the session manager screen (line 1 of the input/output area). You can enter any valid session manager LINE command operand after the word LINE.

  For valid operands, see “LINE” on page 54.

- COLUMN—displays either a number or WRAP:
  - A number indicates which column of the session manager screen is the leftmost column displayed on the terminal screen. For example, COLUMN 1 means that the first character of each line appears in screen column one. COLUMN 40 indicates that the 40th character of each line appears in column one of the session manager screen. Thus, only the portion of each line that fits within the current columns is visible, and each row contains data from a different line.
  - WRAP indicates that input or output longer than the width of the screen is wrapped onto multiple screen rows so that complete lines are displayed, regardless of whether they are longer than the display width. Thus, one line can occupy more than one row of the session manager screen.

  You can enter any valid operand of the COLUMN command over the current value displayed after the word COLUMN. For valid operands, see “COLUMN” on page 44.

  Relationship between ⎕*PW* and the COLUMN Field: The value of the system variable ⎕*PW*, printing width, determines the length of the line stored on the session manager log file. If the length of an input or output line is greater

than the value of □*PW*, the number of characters displayed on the terminal depends on the value of the COLUMN field.

  - If a number is the value of the COLUMN field, that number designates which position of each line is displayed in column 1. The width of each line displayed is either the width of the screen or the width of □*PW*, whichever is the lesser value.
  - If the value of the COLUMN field is WRAP, the length of the line displayed is equal to the value of □*PW*. Characters displayed beyond the width of the screen are wrapped to column 1 of the next line.

- *User field* initially displays the characters *APL2* and the release level you are using. You can type anything you want in this field. It remains displayed until you change it, remove it, or end the session.

  If the cursor is in this field when you press either Enter or a function key, the session manager processes the entry in the field as if it were made on the command line.

  In this field you can enter a session manager command that you want to use several times during the session. For example, to find repeated occurrences of the character string VALUE:

  1. Enter the following in the user field:

     ```
     FIND /VALUE/
     ```

  2. Leave the cursor in the user field and press Enter.
  3. If the character string exists in the current log file, the line containing the string is displayed on line 1 of the input/output area. To display the next occurrence of the string, repeat step 2—leave the cursor in the user field and press Enter.

  For a description of the session manager FIND command, see “FIND” on page 53.

- *Mode field* displays one of the following:

  MODE **MEANING**

  INPUT APL2 is waiting for input in the input/output (I/O) area of the session manager screen. If the cursor is not already in the I/O area, you can use the terminal's cursor movement keys to position the cursor in the input/output area, or you can press Enter once or twice, causing the session manager to position the cursor at the next available input line.

  The session manager indents the cursor six spaces to indicate that it is prepared for input. You can, however, enter input anywhere on the screen. You can even type over lines already displayed. For more information on input lines, see “Reusing Previous Lines in the Session Log” on page 41 and “Entering Multiple Lines of Input” on page 41.

  RUNNING APL2 is processing input. Processing can include writing to the screen lines of output generated by your input. You cannot enter additional input lines in RUNNING mode, but you can signal an interrupt during RUNNING mode. For more information about signalling interrupts, see Appendix H, “Summary of Terminal Information for APL2 Tasks” on page 394.

RUNNING mode is also displayed when your terminal is interlocked, because a shared variable specification or reference does not follow the protocol established by the access control for a variable. In this case, you must enter an interrupt to return to INPUT mode.

If AP 126, the GDDM processor, issued an ASREAD command while the session manager is active, RUNNING mode can indicate that AP 126 is waiting for input.

MORE This is displayed only if the command DISPLAY HOLD OFF was issued. A one-second delay is in effect before the next screen of output is displayed. This delay gives you the opportunity to signal an attention to halt further display.

HOLDING The input/output area of the screen is full and the session manager has more lines to display.

You must scroll the session manager screen to the newest lines in your log until there is room on the screen to display the lines. Provided that your cursor is not in the *user field* (see above), pressing Enter without changing any fields scrolls you immediately to the newest lines. You can also use the LINE or PAGE command or press a function key set to a scroll command. Under CMS (but not TSO), you can also use the CLEAR key to scroll to the newest lines.

You can prevent the session manager from entering HOLDING mode by entering the DISPLAY HOLD OFF command described under “DISPLAY” on page 50.

During HOLDING mode, you can select displayed lines for reuse as described in “Reusing Previous Lines in the Session Log” on page 41.

OFF You ended your APL2 session, but control has not yet returned to the host system.

Any input you enter during OFF mode is ignored.

Input/Output Area: The input/output (I/O) area (all lines of the session manager screen except the first two) is where APL2 work is done. You enter input to APL2, and APL2 displays its output. When you first begin your APL2 session, the I/O area normally displays two pieces of information:

1. The first identifies the release level of APL2 and the national language in use.

   You can change the national language by specifying `⎕NLT←'`*national language*`'`. For the available national languages and their appropriate spellings, see *APL2 Programming: Language Reference*.

2. The second indicates one of these:

   - `CLEAR WS` to indicate a clear workspace
   - `SAVED` *timestamp* to indicate that APL2 has loaded a workspace

     If a workspace is loaded, it may be one that was saved with the system command `)CONTINUE` at the end of a previous session, or it may be one that was loaded as part of APL2 invocation using the INPUT option.

   - Messages resulting from APL2 invocation options such as INPUT, PROFILE, or DEBUG

Entering Multiple Lines of Input

Under the session manager, you can reduce the response time required to process several lines of input by typing more than one input line before pressing Enter. The lines are processed in the order in which they appear on the screen (from top to bottom).

The Session Manager Log

The session manager creates a file, called the session log, containing the lines of input and output from the session. APL2 adds records to the session log as you process APL2 statements. During the session, you have online access to this file.

Each line of input and output during the session is recorded in the session log. Some long input and output lines can be wrapped so that they cover more than one physical row on the screen, but they create only one logical line in the log. The length of lines in the log is equal to the value of the system variable, $\square PW$. (See the discussion of the relationship between $\square PW$ and the value of the session manager COLUMN field on page 38.) The lines in the session log are implicitly numbered, and scrolling through the log is controlled by line number. For information about the wrapping of input and output lines and about scrolling, see the COLUMN, LINE, and PAGE commands later in this chapter.

The session manager command SUPPRESS can be used to suppress the display and logging of APL2 output. For a description of this session manager command, see “SUPPRESS” on page 61.

Note: The only output recorded in the session manager log file is output from APL2 itself. Output from CMS or TSO, AP 126 screens, and editing done with the Editor 2 are not recorded in the session log.

Reusing Previous Lines in the Session Log

Because you have online access to the session log, the session manager allows you to scroll to previous records (lines) and select them for reuse. You can reuse previous lines by completing two steps:

1. Do one or more of the following:
   - Retype a character or blank anywhere on the line
   - Key new data anywhere on the line
   - Change one or more characters by keying over displayed data
   - Insert one or more characters
   - Delete characters
2. Press Enter.

After Enter is pressed, the original line from the session log is restored, and the modified line, including columns not visible, is placed at the end of the session log and processed.

If you select several lines in the log before pressing ENTER, the lines are processed as new lines of input in the order in which they appear in the log, not in the order in which you select them.

Scrolling through the Session Log

Scrolling can be controlled by program function (PF) key settings and the LINE, PAGE, and COLUMN session manager commands described later in this chapter. Also, if you position the cursor on a line in the session log and then press ENTER without making any changes to the line, that line becomes the first line shown on the screen.

Controlling the Size of and Saving the Session Log

The session manager LOG command, described later in this chapter, determines the size of the session log and whether it is saved after each session.

As supplied by IBM, the session manager log defaults to a size of 8192 bytes (8K), or about 200 lines, depending on the length of the lines. During installation of APL2, your system administrator may have changed this default size. You can change the size using the session manager LOG command, which can be included in your session manager profile data set.

After changing the size of a permanent session log, you should process the *)PROFILE SAVE* command to store the size changes in your session manager profile. Failure to do so can damage the session log and lose all its data.

The session log is normally saved as a file in your default private library. This permanent log, and the input/output lines, are available the next time you invoke APL2; this allows you to review work and reuse lines from a previous APL2 session. If, during a session, you create more lines than your log file can contain, the oldest ones are discarded.

You can use the LOG command to specify that the session manager is to maintain only a temporary in-storage log file. An in-storage log contains input/output lines from the current session only. It is not available after *)OFF* or *)CONTINUE*.

When you switch from a permanent log to an in-storage log, the permanent log is deleted. When you switch from an in-storage log to a permanent log, lines in the in-storage log are not transferred to the permanent log. Therefore, whenever you switch from in-storage to permanent (or vice versa), the *first* entry in the log is the *next* input/output line of your session.

CMS: Two physical files are required for the session manager log. The session log files are contained in the private library. The two files both have a file name of @LOG1 or @LOG2 (the name switches when you change the log size). The file type and file mode depend on LIBTAB APL2.

TSO: The session log is kept in a VSAM cluster that can also contain files processed by AP 121, the APL2 Data File Processor. If your installation chose a VSAM library system for APL2 workspaces, the VSAM cluster also contains your default private APL2 library. For more information about APL2 VSAM files, see "APL2 Libraries, Workspaces, and Data Files Under TSO" on page 67.

APL2 Session Manager Commands

The APL2 session manager provides several commands that allow you to:

- Scroll through the session log by lines, pages, or columns
- Search for character strings with the FIND command
- Print (or write to a file) portions of the session log
- Control the size of the log and specify whether it is to be saved
- Set the function keys to provide either APL2 input or issue session manager commands
- Create or replace a session manager profile, or do both
- Control the display characteristics of the session manager
- Obtain help about the session manager commands

Session manager commands may be issued on the command line of the session manager screen, in the user field on the control line, or from within APL2-defined functions using AP 120, the session manager command processor (see Chapter 17, "AP 120—APL2 Session Manager Command Processor" on page 179).

Session Manager Command Summary

Figure 7 on page 44 lists each session manager command and its operands and summarizes the purpose of each command.

Note: All Session manager command name and operand keywords are shown in uppercase in the following descriptions, and fields to be substituted with user data are shown in lowercase. In actual usage, keyword characters may be entered in upper-, lower-, or mixed-case format. Some case restrictions exist for user-supplied values, and are described with the affected commands.

Command and Operand Abbreviations: Most of the session manager commands and operands that are entered in uppercase letters have valid abbreviations. The length of an abbreviation varies because of other commands or operands that begin with the same letters.

The shortest valid abbreviation for each command is given in Figure 7 on page 44 and in the command format figure when the command is described in detail. The shortest valid abbreviation for an operand is given as part of the detailed description of the operand.

Figure 7. Summary of APL2 Session Manager Commands

Command	Operands	Purpose
COLUMN COL	[*number* | *+number* | *-number* | WRAP]	Controls the display of lines longer than the width of the session manager screen.
COPY COP	[ON | OFF | SCREEN | *first last*] [ID *destination*] [CODE *n*]	Copies a portion of the session log.
DISPLAY DI	[ON | OFF CODE n EDS ON | OFF HOLD ON | OFF ORIGIN DEFAULT | *row col* SIZE DEFAULT | *rows cols*]	Controls how the session displays.
FIND FI	/*string*/	Locates a specific character string in the session log.
HELP HE		Displays a summary of session manager commands.
LINE LI	[*number* | *+number* | *-number*]	Scrolls forward or backward through the session log by line number.
LOG LO	[SIZE *n*] [RENUMBER]	Controls the session log.
PAGE PA	[*+number* | *-number*]	Scrolls forward or backward through the session log by a specified number of pages.
PFK PF	[*nn* [APL] IMMEDIATE [*text*] | DELAY]	Sets a function key to a specified value.
PROFILE PR	[LOAD [*name*] | SAVE [*name*]]	Loads, stores, creates, or replaces a session manager profile.
SUPPRESS SU		Prevents display and session log recording of output until APL2 needs further input.

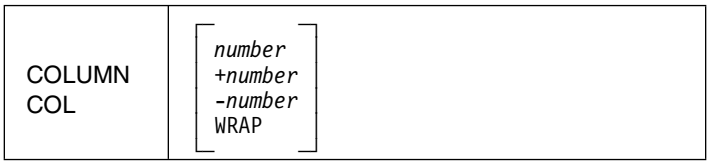
The session manager commands are described in detail below in alphabetic order.

COLUMN

The COLUMN command controls how the session manager displays lines of APL2 input and output that are longer than the width of the screen. The current setting is indicated in the column field in the session manager control line.

The setting of the COLUMN command is included in session manager profiles created with the PROFILE SAVE command, as described under “PROFILE” on page 58.

COLUMN Command Format



COLUMN Command Operands

COLUMN (no operands)
Displays the current COLUMN command setting in the format:

```
* COLUMN nn
   or
* COLUMN WRAP
```

number
An integer between 1 and 1024 indicating the specific column you want displayed on the leftmost edge of your session manager screen. When a number is specified, lines greater than the width of the screen are truncated on display, not wrapped.

+*number*
Number of columns to move the column number setting toward the right (higher column number).

If you specify a number that would take you beyond column 1024, the session manager sets the column field to 1024.

-*number*
Number of columns to move the column number setting toward the left (lower column number).

If you specify a number that would take you beyond column 1, the session manager sets the column field to 1.

Spaces between the sign and the number do not affect the command.

WRAP
W Specifies that lines longer than the width of the screen are displayed on multiple rows. Characters displayed beyond the width of the screen are wrapped to column 1 of the next line. For more information, refer to the discussion of the relationship between $\square PW$ and the COLUMN field following “Control Line” on page 38.

Entering these operands after the word COLUMN in the column field on row two of the session manager screen is equivalent to entering the COLUMN command and the operand on the command line (row one).

See the discussion of the relationship between the value of the $\square PW$ system variable and the value of the COLUMN field on page 38.

COPY

The COPY command enables you to write portions of the session log to either a file or a printer and to specify a code that determines how the copied data is translated. Log lines copied to a printer are folded to printer width. Using the COPY command you can obtain:

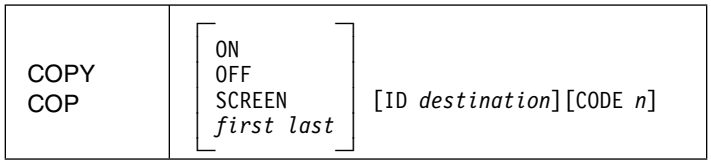
- Continuous copy of the APL2 session
- Copy of the terminal screen, including the session manager screen and any display created by sharing the terminal screen with AP 126, the GDDM processor
- Copy of a range of lines from the session log

The COPY command is included in session manager profiles created with the PROFILE SAVE command. Possible settings of the COPY command for the profiles are described later in this section.

If you are copying to a GDDM destination and are using a CODE 0, you can override the default GDDM print control options by using a GDDM nicknames file.

For information on file identifiers and on other copy destinations, see the separate CMS and TSO discussions following the description of the COPY command operands.

COPY Command Format



COPY Command Operands

COPY (no operands)
Displays the current setting of the COPY command on the command line in the format:

* COPY *operands*

ON
Starts a continuous copy of the APL2 session, beginning with the next line entered or displayed in the session manager input/output area.

Unless you have previously entered a COPY ID (see the ID operand below), the COPY ON command requires that you also specify ID *destination* to tell the session manager where to send the copy.

OFF
OF
Ends the continuous copy function.

SCREEN
SCR
Specifies that a copy of the current session manager screen is to be copied to the specified ID *destination*.

first last

Specifies beginning and ending numbers of a range of lines to be copied to the specified ID *destination*.

ID *destination*

Specifies where the copy is to be sent; *destination* differs for CMS and TSO, as described below under the appropriate system.

Copy output can be directed to a sequential data set or directly to a 328X printer. Output directed to a 328X printer is handled by facilities outside of APL2. For more information, see your system administrator.

If you specify only an ID and destination (and not SCREEN, *first last*, or ON), the destination is used for future COPY requests.

If you specify an ID destination (and SCREEN, *first last*, or ON), the specified destination is temporarily used. Subsequent copy requests then use the destination, if any, in your current session manager profile or the destination established by the COPY ID command.

On a query of the COPY command (entered without operands), a destination of asterisk (*) indicates that no destination has been defined.

CODE *n*
COD *n*

n is a number that determines how the copy output is translated. The value of *n* depends on whether the output is to be processed by GDDM or by the system, as specified by the ID *destination*. (See the CMS or TSO discussion below.)

- **GDDM Processed Output:** If the output is to be processed by GDDM, the data is translated on the basis of what GDDM knows about the type of output device. In some cases, GDDM has insufficient information about a device to translate the data properly. For example, an output device that combines APL characters with a special character set, such as Katakana or Canadian French, requires a special translate table to ensure that the data is translated properly.

  For a file that is to be processed by GDDM, *n* identifies a translate table that is device dependent. The number is passed to GDDM by the FSOPEN function.

  If this parameter is not specified, a default of 0 is assumed, and GDDM translates the data on the basis of what it can determine about the output device.

  For values other than the default of 0, see *GDDM Base Application Programming Reference* or the person at your installation responsible for the GDDM program product.

- **System Processed Output:** If the output is to be processed by the system and you specify a number, it must be in the range 0 to 9999.

  If *n* is other than 0, APL2 uses it to form a CMS file or TSO module name APLX*nnnn* containing a translate table to translate the EBCDIC data of APL2 to the desired code points for the output device you are using. *nnnn* is the number *n* right justified and preceded by 0's to fill four digits. APLX*nnnn* TEXT * is the name of the CMS file. APLX*nnnn* is the name of the TSO load module.

Note: If a translate table is not available, the session manager issues an error message with a return code of 1 54 from AP 126 (copy translate table unavailable).

Destination under CMS: Under CMS, the *destination* can be one of the following:

- File name of a GDDM format disk file to be printed later using the GDDM print utility, ADMOPUV
- File name of a CMS QSAM file identified by the FILEDEF command. The file can be associated with a particular output device, such as a printer or tape drive, or the file can default to disk.

GDDM Format Disk File: A GDDM file is created if the *destination* does not begin with the three characters APL. For example, either of the following commands creates the file SAMPLE ADMPRINT A:

```
COPY ON ID SAMPLE
COPY SCREEN ID SAMPLE
```

ADMPRINT is the GDDM file type for print files from the GDDM environments defaults module for VM/SP. Both sample commands use the default CODE 0. In the case of a continuous copy, the file does not appear in the disk directory until a COPY OFF or)*OFF* command is issued, at which time the file is implicitly closed.

After the APL2 session is ended, the file can be printed with the GDDM print utility.

Note: A COPY command for a GDDM file that already exists generates the error message:

```
COPY ID ALREADY EXISTS
```

A different *destination* must be specified, or the existing file must be erased.

CMS QSAM File: APL2 creates a CMS QSAM file identified by the FILEDEF command if the *destination* begins with the letters APL (in upper-, lower-, or mixed-case format). The IBM-supplied default for the file name is FILE. The *destination* becomes the file type. For example, either of the following commands creates the file FILE APLSAMPL A:

```
COPY ON ID APLSAMPL
COPY SCREEN ID APLSAMPL
```

If a FILEDEF command had been issued to associate APLSAMPL with a printer, the file would be spooled to your virtual printer. Files created by APL2 as a result of the COPY command contain EBCDIC data and can be printed using normal CMS procedures, such as the PRINT command, or be routed through the remote spooling communications subsystem (RSCS). APL characters in the log are preserved in the file.

If you use the COPY command to copy log lines and specify a *destination* of an existing file, the existing data on the file is overwritten and hence lost. To avoid overwriting a disk file, explicitly define the file with the CMS FILEDEF command and specify a disposition of MOD.

Print and tape files must be explicitly defined. To avoid overwriting lines stored on tape, specify DISP MOD in the FILEDEF command.

Data sets allocated to the ddname used in *destination* can have fixed-length or variable-length (not spanned) records. If you specify ASA print control in the FILEDEF, the output is single spaced. Disposition MOD should be specified so that subsequent COPY commands do not overwrite existing data.

Each line copied from the session log creates a record in this data set. Each record has a maximum length of 132 characters; session log entries longer than 132 characters create multiple records.

Destination under TSO: Under TSO, *destination* specifies either:

- *Logical terminal identifier* (LTID) for the printer where you want the output printed
- *ddname* of a previously allocated data set

The *destination* name you specify is passed to GDDM, using the FSOPEN call.

Logical Terminal Identifier (LTID): If *destination* specifies an LTID, GDDM dynamically creates the data sets necessary to pass the request to the GDDM output print utility (ADMOPUT). (For details on the GDDM output print utility, see *GDDM User's Guide*.)

ddname: If GDDM does not recognize or accept the *destination* as a printer defined when GDDM was installed by the system administrator, APL2 uses *destination* as the ddname for a data set. You must allocate the data set before issuing the COPY command, and you are responsible for printing and disposing of the data set.

Data sets allocated to the ddname used in *destination* can have fixed-length or variable-length (not spanned) records. If you specify ASA print control in the ATTRIBUTE command, the output is single spaced. Disposition MOD should be specified so that subsequent COPY commands do not overwrite existing data.

Each line copied from the session log creates a record in this data set. Each record has a maximum length of 132 characters; session log entries longer than 132 characters create multiple records.

For more information on the TSO ALLOCATE and ATTRIBUTE commands, see *OS/VS2 TSO Command Language Reference*.

COPY Command in Session Manager Profiles: The COPY command is included in session manager profiles created with the PROFILE SAVE command, as described in “PROFILE” on page 58. To create a COPY command for inclusion in a profile or to create a COPY command that can be used at another time, you can specify either of the following: `COPY ID` *destination* [`CODE` *n*] `COPY CODE` *n* [`ID` *destination*]

The first form allows you to enter COPY ON or COPY SCREEN and, optionally, CODE and number. APL2 uses the predefined *destination* to determine where to send the output.

The second form allows you to enter COPY ON or COPY SCREEN and a *destination*; thus you should always use the translate table identified by CODE *n*.

Note: If the *destination* identifies a file created without disposition MOD specified, the file is overwritten each time the COPY command is issued.

DISPLAY

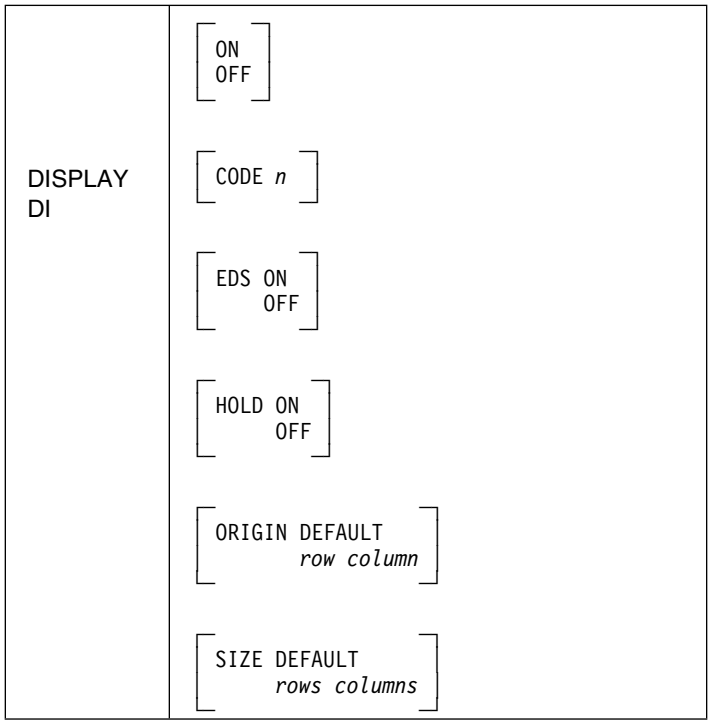
The DISPLAY command controls several aspects of the session manager screen configuration. Some of the DISPLAY command options are for experienced APL2 users who program applications that use the GDDM auxiliary processor (AP 126). These options are noted in the description of the operands.

With the DISPLAY command you can specify:

- Whether or not APL2 output is displayed
- That the session manager should automatically scroll to the newest line in the log rather than entering HOLDING mode
- The size, shape, and location of the session manager screen as it appears on the terminal screen
- A code telling GDDM how to translate data for your terminal

The DISPLAY command is included in session manager profiles. For more information on session manager profiles, see “PROFILE” on page 58.

DISPLAY Command Format



DISPLAY Command Operands

DISPLAY (no operands)
Displays the current settings of the DISPLAY command on the command line in the format:

```
* DISPLAY settings
```

ON

Lines of output are stored in the session log and displayed on the screen.

OFF
OF

Lines of output do not appear on the screen until input is required. Currently running APL2-defined functions and operators continue to process, and their output updates the session log, even though it does not appear.

When APL2 requires input that is not satisfied by alternative means (such as an input stack generated with AP 101), it issues the command DISPLAY ON and requests the input.

Output from AP 126, from the GDDM Processor, and from the system (both CMS and TSO) always appears, regardless of whether or not DISPLAY OFF is specified.

CODE *n*
COD *n*

Identifies the appropriate device-dependent translate table for the terminal. To identify the translate table, the value of *n* is passed to GDDM by the GDDM call ASTYPE.

This option is primarily for APL2 users who need access to graphics that are not required by and not normally available to APL2 (for example, Katakana and Canadian French).

If CODE is not specified, *n* defaults to 0. If the default is used or 0 is specified for *n*, GDDM attempts to determine the appropriate value to be used on the basis of what it knows about the terminal.

The values allowed depend on the GDDM alphanumerics defaults module defined for your system. Some of the valid values in the module as it is supplied by GDDM are listed in Figure 8. For a complete list of valid codes other than 0, see *GDDM Base Application Programming Reference* or the person responsible for GDDM at your installation.

Figure 8. Some of the Valid DISPLAY CODE Values as Supplied by GDDM

Code	Description
0	Use default GDDM code for the device (session manger default)
1	No additional translation. One-for-one EBCDIC used for Canadian French or *katakana* character sets
3277	IBM 3277 terminal with no APL feature
32771	IBM 3277 terminal with the Data Analysis-APL feature code 1066
3279	IBM 3276, 3278, 3279, or 3290 terminal with no APL feature
32791	IBM 3276, 3278, 3279, 3290, or 8775 display terminal with the APL feature

EDS ON|OFF
E ON|OFF

Maintained for compatibility with previous versions of APL.

This operand indicates the value to be used in the TYPE parameter of the GDDM FSPCRT call. ON indicates TYPE=3; OFF indicates TYPE=0. Under current releases of GDDM, these values are ignored.

The default is OFF.

HOLD ON|OFF
HO ON|OFF

Determines what the APL2 session manager does when there is data to be displayed but no space for it in the input/output area.

HOLD ON allows the terminal to enter HOLDING mode. You must scroll forward to make space in the input/output area.

HOLD OFF specifies that the session manager automatically scrolls to the newest line in the session log to make space for the output display. There is a one-second delay before the scrolling takes place (identified by MORE in the mode field) to give you an opportunity to signal Attention.

The default is ON.

ORIGIN DEFAULT|*row column*
OR DEFAULT|*row column*

Defines where the upper left corner of the session manager screen appears on the terminal screen.

The upper left corner of the terminal screen is row 1, column 1. This location is used when ORIGIN DEFAULT is specified.

row column specified must be a valid row and column for your device type. Also, see the restrictions below given after the description of the SIZE operand.

SIZE DEFAULT|*rows columns*
SI DEFAULT|*rows columns*

Defines the size of the session manager screen to be displayed on the terminal screen.

SIZE DEFAULT selects the largest size that can be used for the device type.

rows columns are numbers that specify the size of the session manager screen in rows and columns.

Restrictions: The SIZE of the screen must be at least 3 rows and 39 columns. ORIGIN must be such that all the defined rows and columns fit on the screen.

Note: On the 3270 and 3290 series display terminals, an attribute byte is appended to each row. You do not include this byte in the *column* specification, but your terminal screen must be at least one byte longer than the *column* specification. Attribute bytes are not included in the session log.

FIND

The FIND command searches for a specific string of characters in the session log.

FIND Command Format

FIND F	/*string*/

FIND Command Operand

/string/

Specifies the string of characters to be located. No monocasing of characters is done; they must be entered exactly as they exist in the session log.

The delimiting character, here represented by /, can be any nonalphanumeric character not included in the string. Invalid delimiting characters include blanks and national use characters. A complete list of national use characters is provided in *APL2 Programming: Language Reference*.

The leading delimiter is required. The terminating delimiter is optional. If the terminating delimiter is used, trailing blanks after the character string, but before the delimiter, are significant. For example, the command `FIND /ABC /` (two blanks) locates the result of `5↑'ABC'`, but not the result of `4↑'ABC'`. The command `FIND /ABC` locates either result.

The search initiated by the FIND command begins with the first line *previous* to the session log line currently displayed as the top line on the session manager screen. The search proceeds *backward* until a match is found.

If a match is found, the session manager automatically scrolls, and the line containing the match becomes the top line of the session manager screen.

Each logical line is searched for the string.

Note: If the command COLUMN WRAP is on, a logical line can occupy more than one physical row. A match may be found in a wrapped line that crosses the boundary of a physical row. Matching strings never cross logical line boundaries.

If the top (oldest) line of the session log is reached before a match is found, the search continues with the last (newest) line in the log until every line in the log is searched, except the line currently displayed on the top of the session manager screen.

If the string is not found, the message AP2AFIND437 TEXT NOT FOUND: is displayed on the control line. No scrolling occurs, and the FIND command remains displayed on the command line as entered, preceded by an asterisk (*).

HELP

The HELP command displays a list of session manager commands and their syntax.

HELP Command Format

HELP HE	

In contrast with most other session manager commands, the HELP command has no operands, and any that can be supplied are ignored.

The session manager portion of the terminal screen is displaced by a display of the session manager commands and their syntax. If the display is too wide for the session manager screen, the output is folded to fit the width of the screen. If there are more lines than can fit on the screen, you can review the additional lines by pressing ENTER.

To return to the session manager screen, press ENTER after reviewing the last line of the HELP display. Or, to return immediately to the session manager screen:

Under CMS: Press PA2.

Under TSO: Press PA1 or ATTN, depending on how your terminal is connected to TSO.

LINE

The LINE command controls scrolling backward and forward through the session log by line number. The LINE field on the control line of the session manager screen indicates the number of the line currently displayed as the top line of the session manager input/output area.

If the LINE command is issued through AP 120 and a line number or number of lines to scroll is included in the command, then the text of the line scrolled to is returned in the DAT variable if a 0 0 return code is returned in the CTL variable.

LINE Command Format

LINE LI	[*number* *+number* *-number*]

LINE Command Operands

number
Number of the line in the session manager log to be displayed on the top line of the session manager screen.

+*number*
Number of lines to scroll toward the newest line in the session log (scrolling forward).

-*number*
Number of lines to scroll toward the oldest line in the session log (scrolling backward).

Entering any of these operands after the word LINE on the control line is equivalent to entering the full LINE command on the command line.

LOG

The LOG command controls the use and size of a permanent session log.

LOG Command Format

LOG LO	[SIZE *n*] [RENUMBER]

LOG Command Operands

LOG (no operands)
Displays the current setting of the LOG command in the format:

```
* LOG SIZE n
```

SIZE *n*
SI *n*
n is the number of bytes to be allocated to the APL2 session log. If *n* is 1 or more, the session manager can round up the size by as much as 4K bytes. The APL2-supplied default, which may be changed during installation, is approximately 8K bytes.

SIZE 0 specifies an in-storage log, which means that the log is not saved at the end of the session. There is no way to control the size of the in-storage log.

Note: Once SIZE 0 has been specified, it is not possible to resume use of the permanent session log during that same APL2 session.

If *n* is greater than 0, a permanent session log is maintained, and the SIZE operand can be changed during the session. If you change it to a size smaller than the current log size, the oldest lines in the log are discarded so that new lines can be added to the log file. If you do not want a permanent session log, specify LOG SIZE 0 and the log is deleted.

After changing the size of a permanent session log, you should process the *PROFILE SAVE* command to store the size changes in your session manager profile. Failure to do so can damage the session log and lose all its data (error message AP2AINI439 Session log file replaced).

RENUMBER (R)
Specifies that the lines in the session log are to be renumbered. The oldest line in the log becomes line 0, and the remaining lines are renumbered sequentially.

SIZE and RENUMBER may be entered independently, or both may be part of the same LOG command. Either operand can be specified first.

If a log does not already exist at the start of a session, a permanent log is always created unless:

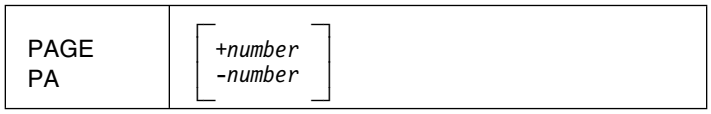
- LOG SIZE 0 is specified in the default VSAPLPR file
- Under CMS there is no writable private library disk
- Under TSO there is no allocated FO library
- The private library or disk is full
- An I/O error occurs trying to create the file

For more information on the session manager log under CMS and TSO, see "The Session Manager Log" on page 41 and also Chapter 4.

PAGE

The PAGE command controls scrolling backward and forward through the session log by a specified number of pages. A page is defined as the number of rows displayed in the input/output area of the session manager screen. (For information on defining the size of the session manager screen and where it is displayed, see "DISPLAY" on page 50.)

PAGE Command Format



PAGE Command Operands

+*number*
Number of pages to scroll toward the newest line in the session log (scrolling forward).

-*number*
Number of pages to scroll toward the oldest line in the session log (scrolling backward).

The *number* may be expressed with a decimal point. The *number* can be viewed as:

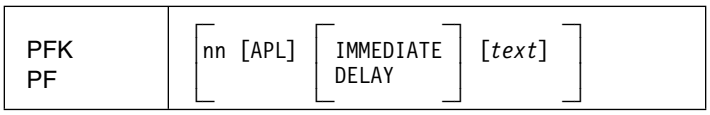
number × *rows in input/output area*

The number you specify is rounded to the nearest number of screen rows. At least one line is scrolled even if PAGE +0 is specified.

PFK

The PFK command controls the setting of the program function (PF) keys for the session manager. The PFK command for each function key can be changed at any time during the APL2 session. The PFK command for each function key is also saved in session manager profiles created with a PROFILE SAVE command. For information on session manager profiles, see “PROFILE” on page 58.

PFK Command Format



PFK Command Operands

PFK (no operands)

Displays the current setting of all 24 function keys.

The display shows settings for function keys 1 through 24, even when your terminal has only 12 function keys.

The output from the PFK command with no operands replaces the area of the terminal defined for the session manager screen. If the output is too wide for display in the area, the lines are folded; if there are more lines than can be displayed on the screen, the remaining lines can be reviewed by pressing ENTER.

To return to the session manager screen, press ENTER after reviewing the last line of the PFK display. Or, to return immediately to the session manager screen:

Under CMS: press PA2.

Under TSO: press PA1 or ATTN, depending on how your terminal is connected to TSO.

nn A number from 1 through 24, representing the function key to be queried or set by the command. PFK *nn* with no other operands displays the setting of the specified function key.

APL

A Indicates that the next operand sets input for APL2 to be activated by the function key specified by *nn*.

APL is required if the *text* operand specifies an APL2 input line.

If specified, it must be the first operand following the function key number.

IMMEDIATE

IM Indicates that the result of pressing the function key is to be as if the text were typed in the appropriate area and ENTER were pressed.

IMMEDIATE is the default.

DELAY

DEL

Indicates that the text for the function key is to be displayed in the appropriate input area with the cursor positioned following the text.

DELAY allows you to press ENTER to activate the entry without change, or to modify the text before pressing ENTER.

text

Input is to be entered when the specified function key is pressed.

Any line of APL2 input or any session manager command can be specified. No monocasing of characters is done; they are retained exactly as entered.

If an APL2 input line is specified, the APL operand must be specified as the first operand following the function key number.

To set a function key to move the cursor to the session manager input/output area, specify:

```
PFKnn APL DELAY
```

The following sets a function key to move the cursor to the session manager command line. (In the default session manager profile supplied by IBM, this command has been assigned to PF2.)

```
PFKnn DELAY
```

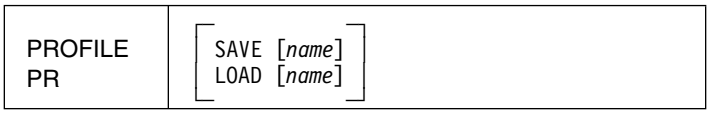
PROFILE

The PROFILE command allows you to load a session manager profile and to save the settings of the following session manager commands to create a new profile or to replace an existing one:

- COLUMN
- COPY
- DISPLAY
- LOG
- PFK 1 through 24

These commands are saved automatically by the command PROFILE SAVE, but any session manager command can be placed in a session manager profile by using a CMS or TSO editor to edit the file that contains the profile.

PROFILE Command Format



PROFILE Command Operands

PROFILE (no operands)

Displays the command settings that are currently in effect and that would be saved with the PROFILE SAVE command.

The output from the PROFILE command with no operands replaces the area of the terminal screen defined as the session manager screen. If the output is too wide for the area, the lines are folded; if there are more lines than can be displayed on the screen, the remaining lines can be reviewed by pressing ENTER.

To return to the session manager screen, press ENTER after reviewing the last line of the PFK display. Or, to return immediately to the session manager screen:

Under CMS: Press PA2.

Under TSO: Press PA1 or ATTN, depending on how your terminal is connected to TSO.

SAVE
SA

Stores a copy of the current settings of the COLUMN, COPY, DISPLAY, LOG, and PFK commands under the *name* specified.

LOAD
LOA

Retrieves a copy of the session manager profile identified in *name* and processes the commands in it.

name

Name of the file containing the settings to be saved or loaded. The name must begin with an alphabetic character and be no more than eight characters long. Alphabetic characters can be entered in upper-, lower-, or mixed-case format, and are treated as uppercase. If you do not specify a name, DEFAULT is assumed.

Under CMS: *name* is the file name of the CMS file containing the profile. The IBM-supplied default file type is VSAPLPR, but this file type may have been changed by your system administrator during installation of APL2.

When saving a profile, the session manager uses file mode A. For example, if the session manager command PROFILE SAVE MYPROFIL were issued, the full name of the CMS file containing the saved profile would be MYPROFIL VSAPLPR A.

A profile to be loaded can be on any accessed disk. When PROFILE LOAD *name* is specified, the disks are searched in standard CMS search order until a file with the proper file name and file type is found.

Under TSO: *name* specifies the second-level qualifier of the data set containing the profile to be stored or loaded. The IBM-supplied default third-level

qualifier of a profile data set is VSAPLPR, but your installation may have changed this name.

When saving a profile, the session manager uses your TSO profile-prefix as the first-level qualifier of the data set. For example, if your TSO profile-prefix (which may be the same as your TSO logon user ID) is GB02 and if you issue the session manager command PROFILE SAVE MYPROFIL, the fully-qualified name of the saved data set is GB02.MYPROFIL.VSAPLPR.

If the named profile is not found using your profile-prefix, a data set with the name *publib.name*.VSAPLPR is searched for, where *publib* is your installation's default public library workspace qualifier. The IBM-supplied default library identifier for workspaces in public libraries is AP2V2R02. The fully-qualified data set name for the profile in the example above is therefore AP2V2R02.MYPROFIL.VSAPLPR.

Normally, the session manager automatically processes a PROFILE LOAD command as part of APL2 invocation when the session manager is used. You can supply the name of the session manager profile to be loaded in the PROFILE option of the APL2 command. If you do not enter a PROFILE option, the profile name DEFAULT is used. If you enter the PROFILE option without a name, that is, PROFILE(), the initial PROFILE LOAD command is suppressed. For more information on APL2 invocation, see Chapter 2, “APL2 Invocation and Termination” on page 8.

Default Session Manager Profile: As supplied by IBM, the default session manager profile contains the commands and default settings listed in Figure 9 on page 61. Your system administrator may have changed the default profile during installation.

```
COLUMN WRAP
COPY OFF ID * CODE 0
DISPLAY ON EDS OFF HOLD ON ORIGIN DEFAULT SIZE DEFAULT CODE 0
LOG SIZE 8140
PFK1 IMMEDIATE HELP
PFK2 DELAY
PFK3 APL DELAY )LOAD
PFK4 IMMEDIATE COPY SCREEN
PFK5 IMMEDIATE SUPPRESS
PFK6 APL DELAY )SAVE
PFK7 IMMEDIATE PAGE -1
PFK8 IMMEDIATE PAGE +1
PFK9 APL DELAY →⎕LC
PFK10 IMMEDIATE COLUMN -40
PFK11 IMMEDIATE COLUMN +40
PFK12 APL DELAY →
PFK13 IMMEDIATE HELP
PFK14 DELAY
PFK15 APL DELAY )LOAD
PFK16 IMMEDIATE COPY SCREEN
PFK17 IMMEDIATE SUPPRESS
PFK18 APL DELAY )SAVE
PFK19 IMMEDIATE PAGE -1
PFK20 IMMEDIATE PAGE +1
PFK21 APL DELAY →⎕LC
PFK22 IMMEDIATE COLUMN -40
PFK23 IMMEDIATE COLUMN +40
PFK24 APL DELAY →
```

Figure 9. Default Session Manager Profile

SUPPRESS

The SUPPRESS command prevents the display of all output on the terminal screen and stops all output to the session log until the next input prompt that requires a response from the keyboard. For example, if you process an expression that generates more output than you care to look at, you can enter the SUPPRESS command when your screen goes into HOLDING mode; the remaining output is abandoned.

SUPPRESS Command Format

SUPPRESS SU	

The SUPPRESS command has no operands.

The DISPLAY OFF command is similar to the SUPPRESS command, except that DISPLAY OFF generates the output and records it in the session log, but does not display it on the screen. SUPPRESS cancels output completely.

Session Manager Messages

If an error is detected in the processing of a session manager command, the command in error is displayed on the command line preceded by an asterisk (*); the cursor is positioned below the asterisk. A message describing the error is displayed on the control line. To reissue the command, delete the asterisk, correct the entry, and press Enter. If you press Enter without deleting the asterisk, the session manager command and control lines are restored to the values that existed prior to the erroneous session manager command.

Session manager messages, their causes, and suggested responses are described in *APL2/370 Messages and Codes*.

Note: Under CMS, if you are using the session manager or another full-screen application such as AP 126 or an editor, CP messages may not be displayed immediately, but are instead displayed at the end of the session or when you press PA1.

Chapter 4. APL2 Libraries: Workspaces and Data Files

Workspaces that you or other APL2 users have saved, files used by the APL2 data file processors, and session manager log files are stored in *libraries* that are identified to APL2 by a *library number*. Library numbers can range from 1 to 9999999.

Workspaces files are created by the)*SAVE* system command to store the current workspace's contents. They are used by the)*LOAD*,)*COPY*, and)*PCOPY* system commands to retrieve the contents of previously saved workspaces.

APL2 data files are used by auxiliary processor 121 and Processor 12 to hold application data. They can contain arbitrary APL2 arrays.

Session manager log files are used if the session manager LOG command specifies a nonzero value. They provide a permanent record of APL2 sessions.

The remainder of this chapter is divided into two sections. The first section discusses APL2 libraries under CMS. If you are a TSO user, skip to the second section, “APL2 Libraries, Workspaces, and Data Files Under TSO” on page 67.

Appendix F, “APL2 Files and Data Sets” on page 381, lists files and data sets that are used for creating and modifying APL2 libraries.

APL2 Libraries, Workspaces, and Data Files under CMS

APL2 workspaces, data files, and session manager log files are CMS files. Each of these files is associated with an APL2 library number. Your default private library number is ↑⎕*AI*. Each APL2 library number is associated with a virtual disk. The association of a library number to a virtual disk is defined in a sequential CMS file with the file name and file type of LIBTAB APL2.

You should have access to a LIBTAB APL2 file, whether on your A disk or on a system disk. The system LIBTAB APL2 file is usually on the same disk that contains the APL2 executable modules. To use a workspace or an APL2 data file, the library number must be defined in LIBTAB APL2.

If you do not have access to a LIBTAB file, the system behaves as if you had the sole entry `PRIVATE 0 @ A` as your LIBTAB file. Each record in the LIBTAB APL2 file contains:

- Type of library (public or private)
- Numeric library identification (one library number or a range of library numbers)
- Disk address of each library

Each APL2 user can have available a LIBTAB APL2 file tailored for that user's exclusive use. This personal LIBTAB file could define libraries for a specific application whose universal access is undesirable.

To create your own LIBTAB file, follow this procedure:

1. Copy the LIBTAB APL2 file from your installation's system disk to one of your own disks.

2. Update the file to define new libraries (see “Updating LIBTAB APL2 to Create New APL2 Libraries”).
3. Ensure that the disk containing your own LIBTAB file is accessed before (in CMS search order) the system disk that contains the system LIBTAB file.

Updating LIBTAB APL2 to Create New APL2 Libraries

The format of the records in the LIBTAB file is:

```
library-type  library-number(s)  library-address
```

library-type
Is either PUBLIC or PRIVATE. It can be shortened to the first three characters (PUB or PRI).

library-number(s)
Identify one number or a range of numbers. If a range, the first and last numbers are specified in a pair of parentheses. The first number must be less than or equal to the last number; a specified 0 means the first element of ⎕*AI*.

library-address
Identifies the CMS disk on which the libraries reside. There are two different formats for the library address:

@ file mode
or
owner device [*link-mode*]

The components of the library address are:

@ file mode	The @ sign indicates a previously accessed disk. The *file mode* specifies the virtual disk on which the library resides.
owner device	CMS user ID of the *owner* of the disk, and the virtual *device* address of the disk in the owner's VM directory.
link-mode	The link-mode for writing, used in the CP LINK command. The default is "W".

Note: Both forms of the library address can be used for the PUBLIC and PRIVATE library types.

Figure 10 is a listing of a sample LIBTAB APL2 file.

```
PUBLIC (1 2) @ Y
PUBLIC (3 99) @ K
PUBLIC (1002 2000) JONES 393
PRIVATE 1001 @ A
PRIVATE 3001 @ Q
PRIVATE 2000 LEE 191 M
PUBLIC 314159 LOPEZ 192 W
```

Figure 10. Sample LIBTAB APL2 File

Accessing Libraries, Workspaces, and Data Files

To be able to access the contents of an APL2 library, the LIBTAB APL2 file must point to the disk on which the library is defined and APL2 must have the appropriate access (for example, read/write or read only) to the virtual disk that the LIBTAB file specifies as the residence of the library.

APL2 can access libraries in either of two ways, depending on their definition in LIBTAB APL2.

- If the LIBTAB entry indicates that the library is on a virtual disk accessed as a specific file mode, APL2 assumes that a disk with that file mode has already been accessed and merely reads or writes workspace records using the given file mode.
- If the LIBTAB entry specifies a specific disk address belonging to a specific user, APL2 dynamically links to the disk and accesses it with an unused file mode, reads or writes as necessary, and then releases the disk.

When you use the APL2 system commands, you must include the library number if you want to access a library other than the library specified (or defaulted to) in the ID option of the APL2 invocation command.

Because the file type for all private workspaces is the same (the default file type is APLWSV2), it generally serves no purpose to define more than one private library on the same virtual disk. You might inadvertently lose a workspace saved in one library that has the same name as a workspace saved in another library. The file name, file type, and file mode of the workspaces would be the same.

Public libraries can be shared by several APL2 users. Because workspaces and files in public libraries have file types that vary depending on the library number, many public libraries can be mixed on one disk.

Workspace Names

Each stored APL2 workspace is a CMS file on the disk defined by the LIBTAB APL2 file. A workspace file can be deleted, copied, renamed, and so forth through the facilities of CMS, but the file is unintelligible outside of APL2.

The file mode of a workspace file is defined in LIBTAB APL2.

The file name and file type are:

PRIVATE WORKSPACE:	*wkspcname* APLWSV2
PUBLIC WORKSPACE:	*wkspcname* **V***nnnnnnn*

wkspcname is the 1- to 8-character workspace name you specified through the)*SAVE* or)*WSID* system commands.

The last two characters (V2) of the 7-character file type for private libraries and the first character (V) of the 8-character file type for public libraries can be changed by your installation. The sixth character of the file type for private libraries is always the first character of the file type for public libraries. *nnnnnnn* is the library number, padded with leading 0s.

Data File Names

Each APL2 data file is a CMS file on the disk defined by the LIBTAB APL2 file. A data file can be deleted, copied, renamed, and so forth through the facilities of CMS, but the file is unintelligible outside of APL2.

The filemode of a data file is defined in LIBTAB APL2.

The file name and file type are:

```
PRIVATE DATA FILE: filename VSAPLFL
PUBLIC DATA FILE:  filename Fnnnnnnn
```

filename
Is the 1- to 8-character file name specified in AP 121 create command or the Processor 12 ⎕*NA* expression.

nnnnnnn
Is the library number specified and includes leading zeros to pad the name to eight characters.

Note: Your installation may have changed the default file types used for APL2 data files. Check with your system administrator.

Transfer Files under CMS

Data from an active APL2 workspace can be written to a *transfer file* through the)*OUT* system command. Data in a transfer file can be read into an active workspace through the)*IN* system command. Transfer files generally require less storage space than a workspace, but use more processing time while reading and writing.

When you use the)*OUT* command to write a transfer file, a sequential CMS file is created. If you specify only a file name in the command, the file type defaults to APLTF and the file mode defaults to A.

With the)*IN* command, the standard CMS disk search order is used to find the specified name.

You can specify the full file name, file type, and file mode of a file created or retrieved with the)*IN* and)*OUT* commands by separating the three components with periods (or dots).

For example, the command)*OUT MYFILE* creates a file whose complete name is MYFILE APLTF A. The command)*OUT MYFILE.MINE.L* creates a file with the name MYFILE MINE L (provided that the L disk was accessed, along with write access).

The)*IN* command, when specified with only a file name, retrieves a workspace with the default name of *file name* APLTF * (where * is whatever disk pertains to the standard CMS search order).

To retrieve a transfer file with a file type and file mode other than the default, enter the file's name, type, and mode separated by periods. For example, to retrieve transfer file MYFILE WKSP L, enter:

```
)IN MYFILE.WKSP.L
```

Library Passwords

You cannot protect individual workspaces or AP 121 files through the facilities of APL2. Instead, you must use the security features of VM by assigning passwords to the virtual disk containing the libraries you want to protect from unauthorized reading or writing. You can specify virtual disk passwords on the APL2 system commands, and the passwords are passed to VM for verification. If you do not specify a password when one is required, VM prompts you for it.

APL2 Libraries, Workspaces, and Data Files Under TSO

Libraries are divided into three groups according to their number. All libraries numbered below an installation defined maximum are treated as public libraries. Private libraries can be numbered either 1001 or ↑⎕*AI* depending on access method. All libraries numbered between the installation defined maximum and 9999999 (except for private libraries) are treated as project libraries. Users can obtain the installation defined maximum by using the)*USER* built-in command of AP 100 (its range is 1 to 32767). For more information on the)*USER* command, see Chapter 10, “AP 100—Host System Command Processor Under CMS” on page 107.

Under TSO, workspace libraries are handled separately from file libraries.

File libraries are always stored in virtual storage access method (VSAM) clusters. They are used for:

- Associated Processor 12 APL files
- AP 121, the APL2 Data File Processor, files
- Permanent session manager log files
-)*COPY* work files

The discussion here does **not** apply to files accessed by AP 123, the VSAM file processor. AP 123 does not manage files in terms of APL2 libraries.

Workspaces can be stored either in VSAM clusters or in individual sequential data sets. You must allocate a specific *file name* for each library that is to be accessed in a VSAM cluster. See “Accessing VSAM Libraries” on page 69, for more details. For any other library numbers, sequential (SAM) files are used. Your installation may have provided TSO logon PROCs or CLISTs that allocate files for the VSAM libraries you are using.

APL2 does not attempt to calculate workspace library information under TSO. Both current and limit values are returned as 0 in)*QUOTA*.

Virtual Storage Access Method (VSAM) Library System

When VSAM is being used, each APL2 library number is associated with one or two VSAM clusters. Workspaces are stored separately from files. Workspace library ddnames always begin with a W, while file library ddnames begin with an F. "Accessing VSAM Libraries" on page 69, describes these ddnames further.

For both workspace and file libraries, ↑⎕*AI* is used as a private library number. Private, project, and public libraries are distinguished by their capability to be shared:

- Private libraries can be read by others, but cannot be written into by any other user during your APL session or dynamically deallocated.
- Public libraries can be written into by multiple users (one at a time), but cannot be dynamically deallocated (TSO FREE)
- Project libraries can be written into by multiple users (one at a time) and can be dynamically deallocated

The additional sharing capabilities of public and project libraries result in additional overhead during their use, with project libraries having more overhead than public libraries. You should keep this in mind when choosing library numbers, and use public, or especially project, numbers only when sharing is required.

Your *private* file library is especially important to you if you intend to use the APL session manager, because this is where the session manager's permanent log is kept. If you do not have a private file library, your session manager log is maintained for the current session only.

You should normally allocate CPYSPILL and CPYSWAP files for work files created during)*COPY*,)*MCOPY*, and)*PCOPY* processing. If you have not done so, and you do have a private file library allocated, the system attempts to allocate work files there during copy operations.

Creating APL2 VSAM Libraries

Your installation may have supplied VSAM libraries for your use. If so, you need not be concerned with this section.

The DEFINE command of access method services must be used to create a VSAM cluster for each APL2 private, project, and public library. The file must have a key-sequenced organization, with key length 14, offset 0. The data set name can be any allowed by MVS. Figure 11 shows the format of the DEFINE command to create a VSAM cluster for an APL2 library.

```
DEFINE CLUSTER (NAME (dataset.name) VOLUME(volid)
                RECORDS(primary secondary) SHR(2) SPEED)
       INDEX(IMBED REPL)
       DATA (KEYS(14 0) CISZ (4096) RECSZ(1024 4088))
       CATALOG(name)
```

Figure 11. Access Method Services DEFINE Command for APL2 Libraries

You may be able to model a new library on an existing one. Figure 12 shows the format of a DEFINE command.

```
DEFINE CL(NAME (dataset.name)
          RECORDS(primary secondary)
          MODEL(old.dataset))
```

Figure 12. New Library Based on Access Method Services DEFINE Command

The *primary* and *secondary* fields shown in Figure 11 and Figure 12 tell access method services how much space to reserve initially for the cluster, and by how much to expand it, if it overflows. The unit of space is the first number after RECSZ. If, for example, you follow the above form and specify RECORDS(500 100), the cluster is created with 500K bytes of space, and expanded in increments of 100K bytes as necessary.

You need to estimate the amount of data you intend to store in the library. RECORDS(500 100) may be reasonable for a private *file* library if you do not have private AP 121 files, but it would probably not be large enough for a *workspace* library.

You need to specify SHR(2) because APL2 does not support SHR(3) or SHR(4). SHR(1) is supported, but unnecessarily restrictive. It prevents any update to public libraries that have been accessed by another APL2 user (until APL termination) and may prevent a user from accessing his own private library.

You may want to place password protection on the library. This can be done when the cluster is first defined or later changed by the ALTER command of access method services. Password protection can only be placed on the entire library, not on individual workspaces or files.

Accessing VSAM Libraries

Use the TSO ALLOCATE command to allocate a library to your TSO session. The allocation can be done in TSO mode or during an APL2 session. In most cases, the libraries you use should be allocated automatically by your TSO logon PROC or by a CLIST. In a batch TSO session, the JCL DD statement can also be used.

The ddname of the cluster must be specified as W*nnnnnnn* for workspaces, or F*nnnnnnn* for files, where *nnnnnnn* is the library number without leading 0s.

Note: This ddname should not be confused with the term “filename” as used in Chapter 18, “AP 121—APL2 Data File Processor” on page 182 and Chapter 30, “Processor 12—Files as Arrays” on page 352. A ddname describes a library that can contain many individual workspaces or AP 121 files.

As a special case, W0 and F0 can be used to name your private workspace and file libraries. On all private library references, APL2 first looks for a W0 or F0 ddname, whichever is appropriate. If that ddname is not found, it looks for the corresponding W*nnnnnnn* or F*nnnnnnn* ddname, where *nnnnnnn* is the first item of □*AI*. For an example of the ALLOC command see Figure 13.

```
ALLOC F(F0) DA(dataset.name) REU
```

Figure 13. ALLOCATE for Private Files and Session Log

After a library is allocated to your session, each time you)*SAVE* a new workspace or create an APL2 data file in the library you add records to the VSAM file. The 1- to 8-character workspace name you specify on an APL2 system command, or the file name you specify in an AP 121 command or Processor 12 ⎕*NA* expression, is recorded by VSAM to associate the new records with the correct workspace or file.

If the library is password protected, you must supply the password to access any file or workspace in it. VSAM passwords can be specified on the APL2 system commands, and are passed on to VSAM for verification. If you do not specify a password when one is required, or give an incorrect value, VSAM prompts you for it (unless your TSO PROFILE specifies NOPROMPT). If the library is RACF* protected, you must have access authorization.

Using ALLOCATE to associate a library number with any desired data provides a flexible design with both advantages and dangers. For example, it allows you to access another user's private library as a project library for your session, or to have a nonstandard set of public libraries.

If your installation is using the Hierarchical Storage Manager (HSM), you should be aware that VSAM clusters are staged as a unit. HSM cannot separate active workspaces or files from inactive ones within a library.

Sequential Access Method (SAM) Library System

This discussion applies only to workspaces. File libraries under TSO are always stored in VSAM clusters.

Under the sequential library system, each saved workspace is stored as an individual MVS sequential data set. The data set name is generated by APL2, and includes both the library number and workspace name. This means that project and public libraries are known globally to the system. Specifically, all users of the same MVS master catalog and the same APL2 load module have project and public libraries containing the same data.

The data set names for private library workspaces include the user's TSO PROFILE PREFIX. This gives each user, or group of users with a common prefix, a unique private library. It also means that, if you change your PROFILE PREFIX, your private library seem to disappear and be replaced with a different one.

Library 1001 is always your private library, regardless of ↑⎕*AI*. It allows you to share workspaces on the same basis as shared variables. That is, the left argument of ⎕*SVO* used to share a variable with a user is the same as the library number used to load a workspace that the user saved without qualification.

New libraries are created automatically by the first user to save a workspace in them. After it is created, a project library is "owned" under the TSO PROFILE PREFIX of the user who created it. No user can)*SAVE* workspaces in the library or)*DROP* them from it unless the PROFILE PREFIX matches. Ownership is retained, even if all workspaces in the library are)*DROP*ed, until the special system command)*DROP* ***libno*** *OWNERSHIP* is issued. Installations can set other controls, independent of PROFILE PREFIX, which limit an individual user's access to certain library numbers. The distinction among private, public, and project libraries is that:

- Your private library can only be accessed by someone having the same PROFILE PREFIX that you do, and can only be accessed as that person's private library.
- A project library is owned by a single PROFILE PREFIX, but can be used by others.)*SAVE* and)*DROP* authority can be extended to others in a limited group, but is normally limited to the owner.
- A public library is owned by the system. It is not normally modifiable except by a system administrator.

Workspaces can be password protected by the TSO PROTECT command. Passwords can be specified on the APL2 system commands, and are passed on to MVS for verification. If you do not specify a password when one is required, or give an incorrect value, MVS prompts you for it (unless your TSO PROFILE specifies NOPROMPT). If the workspace is RACF protected, you must have the appropriate access authorization.

Workspace Names

Fully-qualified names for all APL2 workspace data sets consist of three qualifiers. Different formats exist for workspaces in each type of library, but all library types (private, project, and public) have as the third-level qualifier the 1- to 8-character workspace name given on APL2 system commands.

Figure 14 shows the format of the data set name for each of the three types of libraries. In the figure:

- The letters **APL2** in the first-level index can be changed by your installation to a different qualifier.
- The letter **V** in the second-level index can be changed by your installation to another letter or letters.
- *nnnnnnn* is the library number of a project or public workspace. Because a user can have only one private library, the number of the private library is not a part of the private workspace second-level qualifier.

Figure 14. Data Set Names for Workspaces in a Sequential Library System

Library Type	Level 1	Level 2	Level 3
PRIVATE	*user's-profile-prefix*	**V**	*workspace-name*
PROJECT	*owner's-profile-prefix*	**V***nnnnnnn*	*workspace-name*
PUBLIC	**APL2**	**V***nnnnnnn*	*workspace-name*

CONTINUE workspace

In most cases the *CONTINUE* workspace is handled just like other workspaces in your default library. It can be)*LOAD*ed,)*SAVE*d, or)*DROP*ped explicitly, and is displayed as a part of that library by the)*LIB* command. It is, of course, handled specially at the beginning and end of your APL2 session, as described in *APL2 Programming: Language Reference*.

If, however, you are using TSO batch, or your TSO user ID is not the same as the **PREFIX** setting of the TSO **PROFILE** command, APL2 uses a *CONTINUE* ddname, which you must allocate, to access the *CONTINUE* workspace. This is done so that you can prevent conflicting usage of the *CONTINUE* workspace by two TSO users sharing the same prefix, or by a batch job and your interactive TSO session.

- Create a sequential file that can be used to contain future *CONTINUE* workspaces for batch sessions or your interactive sessions if you do not have a unique **PREFIX**. The easiest way to do this is online in a TSO APL2 session:

  ```
  )CLEAR
  )WSID 1001 CONBATCH
  )SAVE
  ```

  This normally creates a data set named *prefix*.V.CONBATCH, but you can verify the name by entering the command)*HOST APL WSNAME* 1001 *CONBATCH*. Actually, the data set name is arbitrary for most purposes, but using this technique makes it possible in an interactive session to load a *CONTINUE* that was saved in a batch session. The use of *CONBATCH* is completely arbitrary, but it is used in the examples provided.

  Note: If)*HOST APL WSNAME* does ***not*** return a data set name that ends with *CONBATCH*, then you probably have a VSAM library currently allocated as W1001, and the above sequence does not work.

- Either before or after invoking APL2, **ALLOCATE FI(CONTINUE) DA(V.CONBATCH)**. If you do not, and you attempt to)*LOAD*,)*SAVE*, or)*DROP CONTINUE*, error message *IMPROPER LIBRARY REFERENCE* is displayed. If your session ends with)*CONTINUE*, or MVS forces session termination, and you have no CONTINUE ddname allocated, APL2 attempts to save a **DUMP***nnnn* workspace, and the following message is displayed:

  YOUR WORKSPACE HAS JUST BEEN DUMPED TO WORKSPACE DATA SET DUMP*nnnn*

  In all of these cases,)*MORE* message **DDNAME CONTINUE REQUIRED BUT NOT ALLOCATED** is queued.

- You can issue)*DROP* against a CONTINUE ddname, but it does not delete or unallocate the data set. It, however, removes all data from the data set. If you later attempt to)*LOAD* from a CONTINUE that was “dropped” in this way (or have it allocated during APL2 invocation), error message *IMPROPER LIBRARY REFERENCE* is displayed.)*MORE* message **THE WORKSPACE DATA SET IS EMPTY** is queued.

- If you issue the)*LIB* command while a CONTINUE ddname is in effect, *CONTINUE* is reported as the first name, regardless of its normal alphabetical position. The name is reported so long as the CONTINUE ddname is allocated, even if)*DROP CONTINUE* has been issued.

Transfer Files under TSO

Data from an active APL2 workspace can be written to a *transfer file* by the)*OUT* system command. Data in a transfer file can be read into an active workspace through the)*IN* system command. Transfer files generally require less storage space than a workspace, but use more processing time while reading and writing.

The *file* specified in the)*IN* and)*OUT* commands, unless enclosed in quotation marks, are qualified with first- and second-level qualifiers to form a fully-qualified data set name of the form:

prefix.APLTF.*file*

where *prefix* is the user's TSO PROFILE PREFIX. Note that *file* can be itself simple or qualified. If the *file* is given in quotation marks, it is used without further qualification.

A data set transferred by the)*IN* command must be cataloged. APL2 does the allocation. A data set transferred through the)*OUT* command is created if one does not already exist.

Security and Integrity of APL2 Data

To protect your APL2 data, you can use the security features of any of the following:

- TSO PROTECT command
- RACF
- Password protection by Access Method Services

See your system administrator to find out what security features are available on your computing system.

Chapter 5. Named Editors

APL2 provides two built-in editors for APL2: a line editor known as Editor 1 and a full-screen editor known as Editor 2. APL2 also allows you to use a named system editor or named APL2 editor. You select these options by specifying the name of the editor in either a)*EDITOR* or)*EDITOR* 2 system command.

Regardless of the setting of)*EDITOR*, Editor 1 is used to process ∇-commands that include both a [□...] display request of some sort and a closing ∇. Such commands simply display all or part of a function without performing any actual editing.

If you specified the name of a system editor using the)*EDITOR* command, you can then use ∇ followed by the name of a function or a character vector or matrix to begin editing that object using your editor.

When you are using a named system editor, if the name following the ∇ is a function in the workspace, APL2 writes the canonical representation of the function to a CMS file or a TSO data set. If the name following the ∇ is a variable of rank two or less, APL2 writes the variable itself in a canonical representation. If the name following the ∇ does not exist in the workspace, APL2 assumes you want to create a function of that name and writes a single record containing the rest of your ∇ command. APL2 then invokes your editor through AP 100 with the name of the file or data set as your editor's argument.

If your editor ends with a return code of 0 and if the file or data set still exists when your editor returns to APL2, APL2 reads the file or data set and, based on its contents, reestablishes the object being edited in the workspace. APL2 then deletes the file or data set.

If you specified the name of an APL editor using the)*EDITOR* 2 command, you can then use ∇ followed by an expression to invoke the editor. APL2 associates a name with the specified editor and calls it passing the ∇ expression as a right argument. It is the responsibility of the APL editor to parse the expression and provide editing facilities.

Restrictions Using Named System Editors

*S*Δ stop vectors and *T*Δ trace vectors are not updated when lines are inserted into or deleted from a function or operator.

The '[...]' edit instructions are not allowed when a named editor was specified unless a closing ∇ is also present and the command is processed by Editor 1. When editing variables, some named editors may delete trailing columns that contain only blanks.

Using Named System Editors under CMS

The file written by APL2 has a file name equal to a number. The file type is AP2EDPGM for new names, functions, operators; AP2EDCHR for simple arrays; and AP2EDEVL for nonsimple arrays. The file mode is A. Your named editor can be any CMS command, module, EXEC, or any CP command. XEDIT is an example of a valid name. If you use XEDIT, you should consider the following:

- Your PROFILE XEDIT file should contain 'COMMAND SET APL ON', at least for file types of AP2ED*.
- For all names other than names of character matrixes, APL2 writes a file with an LRECL of 80 or the longest record, whichever is larger. If you want to add or extend lines longer than this, then the LOAD subcommand in your PROFILE XEDIT file should include a WIDTH option of a sufficient size.

  For names of character matrixes, APL2 writes a file with an LRECL equal to the width of the matrix or the length of the name, whichever is larger. When reading the file after the editor has terminated, APL2 does not strip trailing blanks from the file's records. Some editors provide facilities that can be used to change the LRECL and record format of the file. These facilities can be used to change the width of the matrix.

The PDF editor can also be used if APL2 is invoked under ISPF. To invoke ISPF editor under VM:

1. Write a CMS EXEC called ISPFAPL2 to invoke APL2 under ISPF:

   ```
   &TRACE OFF
   &COMMAND VMFCLEAR
   &SUBCOMMAND ISPEXEC SELECT CMD (APL2 &ARGSTRING AP(ISPAPAUX)) LANG(APL)
   &COMMAND VMFCLEAR
   ```

2. Write a CMS EXEC called PDFEDIT to invoke the PDF editor:

   ```
   &TRACE
   &COMMAND VMFCLEAR
   &SUBCOMMAND ISPEXEC EDIT FILE (&1 &2 &3)
   &COMMAND VMFCLEAR
   ```

3. Start ISPF:

   ```
   ISPFV2
   ```

4. Start APL from Panel 6:

   ```
   ISPFAPL2
   ```

5. Use)*EDITOR* and specify the EXEC that invokes the PDF editor:

)*EDITOR PDFEDIT*

Using Named System Editors under TSO

The file written by APL2 has a ddname of APL2EDIT. If that ddname was not preallocated, it is dynamically allocated. ATTR APL2EDIT is used for the dynamic allocation if such an ATTR exists. Otherwise the ATTR is also dynamically created. The dynamic allocation of the APL2EDIT ddname is to the data set name 'prefix.APL2.EDIT'. Anything dynamically allocated is freed or deleted when editing ends.

For all names other than names of character matrixes, APL2 allocates a data set with attributes of RECFM(V,B), LRECL(259), and BLOCK(263).

For names of character matrixes, APL2 allocates a file with an LRECL equal to the width of the matrix or the length of the name, whichever is larger. RECFM(F,B) is used. When reading the file after the editor has terminated, APL2 does not strip trailing blanks from the file's records. Some editors provide facilities that can be used to change the LRECL and record format of the file. These facilities can be used to change the width of the matrix.

To avoid the performance overhead of dynamic allocation, you can preallocate a data set for APL2 editing using the ddname and data set name described above. However, because APL2 does not have control of the file allocation at the time of entering edit, the file's LRECL is not adjusted to the object's width. If you preallocate a data set, you should use RECFM(V) or RECFM(VB), LRECL(259), and BLOCK(263).

Under TSO, the named system editor is invoked with a preceding '%', and thus must be a CLIST in one of the data sets allocated to the ddname SYSPROC. The argument to the CLIST is the positional parameter 'APL2.EDIT'. The CLIST or EXEC can invoke TSO commands such as EDIT or ISPF.

To invoke APL2 from ISPF (using panel 6, for example):

1. Define CLIST ISPFAPL2:

   ```
   PROC 0 AI() AP()........
   ISPEXEC SELECT CMD(APL2 AI(&AI) AP(&AP)....) LANG(APL)
   ```

2. Under ISPF panel 6, issue the command:

   ```
   %ISPFAPL2
   ```

If the ISPF AP is to be used under APL2, specify AP(ISPAPAUX...) in the APL2 command parameters. See the ISPF documentation for information regarding the ISPF AP and its use.

To use the ISPF editor from APL2 when APL2 is NOT running under ISPF:

1. Define CLIST ISPFEDIT:

   ```
   PROC 1 DSN
   ISPSTART CMD(%PDFEDIT &DSN)
   ```

2. Define CLIST PDFEDIT:

   ```
   PROC 1 DSN
   ISPEXEC EDIT DATA SET(&DSN)
   ```

3. Under APL2, issue the command:

)EDITOR PDFEDIT

4. Under APL2, when you open function FNCTNX for editing, FNCTNX is written to data set 'prefix.APL2.EDIT', and APL2 issues the command

)HOST %PDFEDIT APL2.EDIT

To use the ISPF editor from APL2 when APL2 is running under ISPF:

1. Define CLIST PDFEDIT:

   ```
   PROC 1 DSN
   ISPEXEC EDIT DATA SET(&DSN)
   ```

2. Under APL2, issue the command:

)EDITOR PDFEDIT

3. Under APL2, when you open function FNCTNX for editing, FNCTNX is written to data set 'prefix.APL2.EDIT', and APL2 issues the command

)HOST %PDFEDIT APL2.EDIT

The following notes apply to the use of ISPF under APL2:

- The terminal type must be properly set under ISPF option 0.1. The ISPF product includes terminal translate tables that support APL2 characters. Consult *APL2/370 Installation and Customization under TSO* for information about how to enable this support.
- If ISPF is running, CLISTs to be executed, specified directly to AP 100, or in the *)HOST* command (as clistname, %clistname, or EXEC clistname), are passed for execution to ISPF as though the command ISPEXEC SELECT CMD(clistname) were issued.
- If ISPF is not running, CLISTs to be executed, specified directly to AP100, or in the *)HOST* command, are executed by APL2. These CLISTs may not include the ISPEXEC command.
- CLISTs to be executed, specified directly to AP 100, or in the *)HOST* command (as TSO clistname), are passed for execution to the TSO CLIST processor.
- When CLISTs are passed to ISPF for execution, APL2 has issued the command *ISPEXEC CONTROL ERROR RETURN*; hence, return codes are passed back by ISPF.
- The minimum record length that APL2 allocates for rank 2 arrays is 10 characters to support PDF Edit's minimum LRECL restriction.

Chapter 6. Batch Processing

When you do not have a terminal available or when your processing need not be done online (for example, when you are printing a long-running report), APL2 can be processed as a batch job. You can continue to use APL2 interactively for other work while your batch job is running.

When APL2 is processed in batch mode, the session manager cannot be used. If your installation defaults to use of the session manager, make sure you code the APL2 command with the option SMAPL(OFF). To prevent a *CONTINUE* workspace from being loaded at the start of batch APL2, include the option INPUT( ) with the APL2 command.

The next section discusses batch jobs under CMS. If you are a TSO APL2 user, ignore the next section and continue with “Batch Jobs under TSO” on page 80.

Batch Jobs under CMS

You can submit a batch job containing APL2 expressions and commands from your terminal by spooling the card images to the virtual card reader of the CMS batch facility. Your installation probably provides one or more virtual machines (such as CMSBATCH) to run all batch jobs.

When submitting a job from the terminal, you spool the input deck to the CMS batch machine either as a CMS file or as virtual punched output from you.

CMS Batch Facility Input

The input must include:

1. Control statements for the CMS batch machine. For the statements required in your installation, contact your system administrator.
2. CP and CMS commands to give the CMS batch machine access to virtual disks required by APL2.

   The CP and CMS commands you use and the information you specify depend on what your batch job is intended to do.
3. The APL2 invocation command.
4. APL2 input statements.

   The APL2 statements you enter depend on the work you want done.
5. CP and CMS commands to route the results of your batch job to the desired output device.

Figure 15 contains a sample batch job to the CMS batch machine.

```
/JOB myuserid account# jobname ◄──────Job identification.
CP SPOOL CONSOLE TO myuserid
CP LINK myuserid 191 192 RR password
ACCESS 192 B/A
CP SPOOL PRT TO myuserid    ◄──────────Send report to your reader.
APL2 SMAPL(OFF)──────────┐
)LOAD MYWS            APL2 commands; use MYWS
REPORT QUOTAS      workspace to produce a report.
)OFF           ──────────┘
CP SPOOL PUN TO myuserid    ◄──────────CP commands to transfer any
DISK DUMP * APLWSV2 A                  error report to your reader.
/*
```

Figure 15. Sample Batch Job

Spool the file to the CMS batch machine card reader to submit the job. A sample spool and punch command to spool the file APLFILE BATCH are shown below:

```
CP SPOOL PUNCH TO batchid NOCONT
PUNCH APLFILE BATCH (NOHEADER)
```

CMS Batch Facility Output

The CMS batch machine output consists of all the input statements submitted together with all “terminal” output generated while processing the batch job. Unless you specify otherwise, all the CMS batch machine output is directed to the system printer.

To redirect the CMS batch machine output to your terminal, spool the output to your virtual machine by including the following command in your batch job:

```
CP SPOOL CONSOLE TO userid
```

userid is your user identifier. After the job finishes processing, the batch output is placed in your virtual card reader.

Other APL2 Considerations

The following considerations apply to CMS batch machine jobs using APL2:

- The host system command processor, AP 100, should not be used to issue commands that are prohibited by the CMS batch machine. For a discussion of CP and CMS command restrictions, see the appropriate user's guide for your system.
- The initial value of □*PW* is 120.
- If a password is required in an APL2 system command, you must specify the password in the card that contains the command. The CMS batch machine does not prompt for a missing password. For example, to load the workspace *TESTWKSP* in library 1234 that is protected by the password SECRET, enter:

)*LOAD* 1234 *TESTWKSP*:*SECRET*
- The word BATCH is passed to the AP2EXIT EXEC.

Batch Jobs under TSO

To submit a batch APL2 job, you can use the SUBMIT facility of TSO or any other submit technique, such as a job entry subsystem.

TSO Batch Input

Figure 16 on page 80 contains sample JCL for submitting a batch job to run APL2. Note the following information about the JCL:

- IKJEFT01 is the TSO program that allows TSO commands to be invoked in batch mode.
- SYSTSPRT is the ddname for non-APL2 output.
- APLPRINT is the ddname for output from the APL2 batch session. For information regarding this file, see "Directing APL2's Output" on page 83.
- SYSTSIN is the ddname for input to TSO.
- APLIN is the ddname for input to APL2. For information regarding this file, see "Providing Input to APL2" on page 82.

```
//APLBATCH JOB  job card information
//STEP     EXEC PGM=IKJEFT01,DYNAMNBR=50,REGION=4096K
//SYSTSPRT DD SYSOUT=A
//APLPRINT DD SYSOUT=A
//SYSTSIN  DD *                          <--------Commands issued to TSO
ALLOC FI(CONTINUE) DA(V.CONBATCH) OLD
APL2 SMAPL(OFF) INPUT() CODE(1)        <--------APL2 = CLIST name
LOGOFF                                 <--------Logoff when APLIN done
//APLIN DD *
)LOAD 1234 MYWORKSP:SECRET             <--------Data passed to APL2
  .
  .
)OFF
/*
```

Figure 16. Sample JCL for Submitting a Batch Job under TSO

TSO Batch Output

SYSTSPRT and APLPRINT output can be printed, sent to a file, directed to another network node, or held for viewing from your interactive TSO session. Your installation should provide information on SYSOUT classes and JES control statements to use.

An IBM 3800 printing subsystem can print the APL2 character set if your installation has installed the APL printer fonts in the MVS image library. On other printers, compound symbols and overstrike characters may not print correctly.

The CONTINUE ddname identifies a data set that can be used for any of the following purposes:

- Automatically loading a workspace at the beginning of the session
- Satisfying a)*CONTINUE* command at the end of the session
- Preserving your work if MVS forces a premature end of session

This ddname is ignored if a VSAM private library is allocated. See “Sequential Access Method (SAM) Library System” on page 70 for further details.

Chapter 7. Controlling APL2 Invocation

Several facilities are provided that can be used to control the invocation of APL2 and automatically invoke APL2 applications. This chapter discusses these facilities and highlights the interaction among them. Here are several reasons for automatically invoking applications as APL2 is invoked:

- To prepare an interactive session environment
- To run a specific APL2 application from within a non-APL program
- To run a specific APL2 application in batch

When controlling the invocation of APL2 and running an application, several types of information are of concern:

- How to provide input data
- Where to direct output data
- How to minimize screen disturbance
- How DBCS data is handled

The following options are used to control APL2 invocation:

INPUT	To provide input to be executed
RUN	To specify an application to run
TERMCODE	To control whether input and output files and in some cases the stack are used
QUIET	To control whether APL2 displays output data
SMAPL	To control whether the APL2 session manager is started or whether the cooperative interpreter interface is to be used

Providing Input to APL2

Several invocation options, and the stack on CMS, can be used to provide line input to APL2 at invocation. APL2 line input includes system commands and expressions to be processed. Expressions that are provided through any of these options can also use AP 101 to stack more line input. The input provided by the RUN and INPUT invocation options and the APLIN file, if TERMCODE(-1) is used, are stacked FIFO by APL2 in the following order:

1. RUN
2. INPUT
3. APLIN

The expressions provided by these options can use AP 101 to stack data in either LIFO or FIFO stack order. This can cause data stacked by an APL application to be processed before processing of data provided at invocation. For example, if the program invoked using RUN stacked data LIFO, this data would be processed before data provided through the INPUT option.

APLIN is a ddname defined before APL2 is started by using the CMS FILEDEF or TSO ALLOCATE command, or the MVS JCL DD statement. It is used to supply APL2 with input only if TERMCODE(-1) was specified (or defaulted for TSO batch.)

Consult the discussion of the TERMCODE option on page 28 for information about the requirements of the APLIN and APLPRINT files.

On CMS, TERMCODE(-1) also controls whether the CMS stack is used for input. If TERMCODE(-1) is specified, data on the CMS stack is ignored; any data in the stack at invocation is retained until exit from APL2. If TERMCODE(-1) is not specified, data on the CMS stack is used in place of APLIN.

On TSO, an APL stack is maintained independently of the TSO stack. The use of TERMCODE(-1) controls the disposition of data left on the APL stack at APL2 termination. If TERMCODE(-1) is used, data left on the APL stack is discarded at termination. If TERMCODE(-1) is not used, data left on the APL stack is placed on the TSO stack at termination. The TSO stack is never used to provide input to APL2.

After all these input facilities are exhausted, APL2 prompts for more input either using the session manager, the standard input protocol of the operating system in use, or through the cooperative interpreter interface if it is active. If TERMCODE(-1) is used, when all these input facilities are exhausted, and processing of the last expression is complete, APL2 terminates.

Directing APL2's Output

APLPRINT is a ddname defined before APL2 is started by using the CMS FILEDEF or TSO ALLOCATE command, or the MVS JCL DD statement. It is used for APL2 line output only if TERMCODE(-1) was specified (or defaulted for TSO batch.) APL2 line output includes data displayed by system commands, results of processed expressions, and error and other messages.

If TERMCODE(-1) is specified, APL2 line output is directed to the file allocated to the ddname APLPRINT. If the ddname APLPRINT is not allocated, APL2 discards line output. Any output data longer than the allocated record length is wrapped.

If TERMCODE(-1) is not specified, APL2 line output is either displayed using the session manager, displayed using the standard output protocol of the operating system in use, or transmitted through the cooperative interpreter interface.

Consult the discussion of the TERMCODE option on page 28 for information about the requirements of the APLIN and APLPRINT files.

When TERMCODE(-1) is used and an)*OFF* or)*CONTINUE* command was processed, the default return code is 0. If an)*OFF* or)*CONTINUE* command was not processed before the sources of input were exhausted, then a *CONTINUE* workspace is saved and the default return code is 4.

On TSO, when APL2 is invoked in a batch environment, a CONTINUE ddname must be preallocated if a)*CONTINUE* workspace is to be saved.

Controlling APL2's Use of the Screen

When invoking APL2 from an interactive program, it is frequently desirable to control APL2's use of the screen. This may be because you want to avoid disturbing data displayed by APL2's invoker or are invoking an application that uses AP 124, AP 126, or ISPF for displays and want to enter that display environment directly.

The SMAPL(OFF) invocation option can be used to prevent APL2 from starting the session manager. The QUIET(ON) invocation option can be used to prevent APL2 from displaying any data until the first input request is made.

It is often desirable to invoke APL2 quietly and conceal from the end-user the invocation of an application. Occasionally however, the application is written to use APL2's default display and prompt mechanisms rather than a full-screen interface. In these cases, QUIET can be too silent. QUIET suppresses all APL2 line output until input is requested. This could cause suppression of the lines explaining why input is needed.

The *OPTION* external function can be used to set the QUIET option either ON or OFF. In the scenario described above, the application could be invoked with QUIET(ON) and it could then use *OPTION* to set QUIET(OFF) before issuing its prompt.

DBCS and APLIN/APLPRINT Files

Many written languages such as Japanese Kanji use a character set with more than 256 characters, which is the most that can be encoded on a computer using a single 8-bit byte. To process such languages on a computer, a *double-byte character set* (DBCS) is used. In a DBCS, two bytes are used to encode each character, so tens of thousands of characters can be encoded.

DBCS in Other IBM Products

Many IBM products use a convention for storing characters that allows DBCS characters and characters from languages with one-byte codes to be mixed in strings or files.

In this convention, DBCS characters are distinguished from single-byte characters by enclosing strings of DBCS with a shift-out (SO) character and a shift-in (SI) character. The code point used for SO is X'0E'. The code point used for SI is X'0F'.

When this convention is used, a string of bytes must be interpreted in the following manner to understand what string of characters it represents:

- Until an SO character is found, each byte is interpreted as a single-byte character.
- After an SO character is found, and until a matching SI character is found, each pair of bytes is interpreted as one double-byte character.

DBCS in APL2

To support more than one DBCS concurrently, and to provide superior performance when character strings are being processed, APL2 uses a different convention with the APL workspace for storing and processing character strings containing DBCS.

In an APL workspace, DBCS characters are represented as extended characters and occupy 4 bytes of storage each. Each extended character contains a character set identifier (2 bytes) and the character itself (2 bytes). When DBCS characters are stored in extended character format in an APL workspace, the first 2 bytes of each character represents the character set identifier and the last 2 bytes represent the DBCS character. When non-DBCS characters are stored in extended character format in an APL workspace, the first 2 bytes of each character represents the character set identifier, the third byte is zero and the last byte represents the character. APL characters (those mapped by $\square AV$), have the character set identifier zero, and thus the first 3 bytes of each such character is zero.

Reading DBCS from APLIN

1. If the DBCS invocation option is OFF or was omitted, then the input is not checked for DBCS.
2. If DBCS is ON, TRY, or 0 then the input is scanned for a shift-out character. Until an SO is encountered the input is treated as single byte characters in $\square AV$.

   If an SO is encountered, then the input following is treated as a sequence of double-byte characters, up until a corresponding shift-in character is encountered. A matching SI character must exist in the same line of input, or an ENTRY ERROR is generated and the line is ignored. If data exists following the SI, then the SO scan starts again.

   Each double-byte character is verified to check that either the character is X'4040', or the first and second bytes are both in the range X'41' and X'FE'. If any character is not valid an ENTRY ERROR is generated and the line is ignored.

   If an ENTRY ERROR is not generated, then the double-byte characters are converted into extended APL2 characters with a character set ID of 0.
3. If DBCS is set to a specific number other than 0, then the input is scanned for double-byte characters as in step 2, and the number is used as the character set ID if any are found.

Writing DBCS to APLPRINT

1. If the DBCS invocation option is OFF or was omitted, then extended characters outside of $\square AV$ are output as the character 'ω'.
2. If DBCS is ON, TRY, or 0 then the last two bytes of extended characters outside of $\square AV$ are output as is, with sequences of these two-byte pairs enclosed by shift-out (X'1E') and shift-in (X'1F') characters.
3. If DBCS is set to a specific number other than 0, then the number is compared to the first two bytes of extended characters outside of $\square AV$. If the number matches, than the last two bytes are output as is, with sequences of two-byte pairs enclosed by shift-out and shift-in characters. If the number does not match then a 'ω' is output.

Chapter 8. Using APL2 across Systems

This chapter describes how to communicate between APL2 sessions on different systems, and how to transfer programs and data between systems.

Cooperative Processing

APL2 sessions can communicate either with each other or with other non-APL programs across a Transmission Control Protocol/Internet Protocol (TCP/IP) network.

There are four major facilities within APL2/370's support for cooperative processing:

- Cross-System Shared Variables

  This facility allows a user to share variables with other processors on a TCP/IP network using normal APL2 shared variable techniques. It provides APL2's most convenient program-to-program cross network communication path.

- Shared Variable Interpreter Interface

  This interface provides a set of protocols whereby an APL2 interpreter can be controlled through a shared variable. It provides a way for a program to control a remote session.

- Remote Session Managers

  The external function RAPL2 can be used to control a remote interpreter. The APL2 workstation products each include session managers that can also be used to control remote interpreters. These facilities allow the user to conduct an interactive session with a remote intepreter using the Shared Variable Interpreter Interface.

- TCP/IP Auxiliary Processor (AP 119)

  This processor allows users and applications to make direct requests to TCP/IP. It provides APL2's most flexible program-to-program cross network communication path. The interface can also be used for communication between APL2 and non-APL programs across a network.

Processor Network Identification

An APL2 session consists of a collection of processors. From the point of view of an APL2 program, each processor is identified by a single nonnegative integer. The APL2 user is identified with a processor number greater than 1000. Other processors in the session are called auxiliary processors (APs) and are normally identified with a processor number less than 1000.

A single integer is not enough to address processors in multiple sessions and processors in sessions on a network. Therefore a Processor Profile provides a cross reference between the single processor number used by APL2 programs and a processor network identification.

Under CMS, the processor profile is file AP2TCPIP APL2PROF. Under TSO, the processor profile is member AP2TCPIP in the data set pointed to by DDNAME APL2PROF.

The profile is used for both outgoing offers from a processor and for incoming offers from other processors. It is read for each offer and can be dynamically modified.

Every processor on the network has a unique name consisting of the following parts:

```
IP_address user_id processor_number[,parent[,grandparent]]
```

For example:

```
123.45.6.78 BROWN 1002
123.45.6.78 BROWN 127,1001
```

A processor named with only an IP address, user ID, and processor number is called an independent processor. In the first example above, 1002 is an independent processor. Normally, the APL2 interpreter runs as an independent processor.

A processor with a parent (or any ancestor) is called a dependent processor. A dependent processor is notified when its immediate ancestor signs off. In the second example above, 127 depends on 1001 and 1001 is independent. Normally, the APL2 interpreter runs as an independent processor with its auxiliary processors dependent on it. This scheme allows processor 127 to be informed (and normally to terminate itself) when the APL2 session ends.

A third level of dependency is defined if a processor is started with a grandparent processor number. This scheme allows APL applications to serve as dependent auxiliary processors, since they in turn need to use other dependent auxiliary processors. Longer sequences of ancestors would be meaningful but are not supported.

Processor Profile Structure

Each line in the processor profile can contain one or more tags and its associated data. Tags can be written in uppercase, lowercase, or mixed case. Any line starting with the character "*" is ignored.

Each processor entry must begin with either an :svopid tag or a :procauth tag and continues to the next occurrence of one of these tags or to the end of the profile. A processor entry beginning with an :svopid tag is known as an identification or ID entry. Here is an example of an ID entry that defines 33586 as a remote user signed on as ID 1002 under BROWN at 123.45.6.78.

```
* user BROWN at STLAPL
:svopid.33586
          :address.123.45.6.78
          :userid.BROWN
          :processor.1002
```

A processor entry beginning with a :procauth tag is known as an authorization entry. Here is an example of an entry that authorizes the remote processor identified by an svopid of 33586 to share with a local processor 100, which is a dependent of processor 1001:

```
* AP100 authorization
:procauth.100,1001
          :rsvopid.33586
```

Using the Port Server

APL2 includes a program called the port server that manages the establishment of communication links across TCP/IP networks. Each system in the network should have a port server running. When a user first attempts to use TCP/IP (either through cross-system sharing or AP 119), TCP/IP assigns the user a TCP/IP port number. This port number is registered with the user's local port server. When a cross-system share offer is made, APL2 contacts the port server at the partner's system to find out the partner's TCP/IP port number.

APL2 port servers also have port numbers. The default port number for the servers is 31415. However, some sites may choose to install the port server with a different port number. If APL2 cannot contact your partner's port server, a message is displayed stating this fact and the port number that was used in the attempt to make contact. In these cases, you need to specify the server's port number in your TCP/IP profile file. Consult your partner's system programmer to find out their system's port number.

It is also possible to share variables across systems even if one or both of the systems do not have a port server running. The AP 119 command GETLPORT is used to find out what your own port number is. The command SETLPORT is used to inform the cross-systems shared variable facility what your potential partner's port number is. For further details, please refer to “Listening Ports” on page 170.

Sending a Share Offer

When a shared variable offer is extended, the left argument of $\square SVO$ is first matched against the list of local processors that are signed on to the SVP. If a match is found, the offer is extended to that local processor. If no match is found, the number is matched against the data in the :svopid tags in the processor profile. If a match is found, the offer is extended to the processor described by the :processor, :address, and :userid values.

If no match is found in the processor profile, the offer is extended to the global processor identified by the left argument of $\square SVO$ whether or not it is presently signed on to the SVP.

Receiving a Share Offer

When receiving an offer to share, the processor profile serves to identify the remote processor and to authorize the share. First, the processor identification of the processor originating the share is matched against the :address, user ID, and :processor tags of each ID entry in the processor profile. Limited “wildcard” support is provided as discussed in “Processor Profile Syntax” on page 89.

If a matching entry is found, the svopid of this entry must be identified in the :rsvopid tag of an authorization entry for the local processor with whom the caller is trying to share. If this is true, the share is allowed to proceed.

Processor Profile Syntax

Each line in the processor profile can contain one or more tags and its associated data. Tags can be written in uppercase, lowercase, or mixed case. Any line starting with the character "*" is ignored.

Identification Entries

Each processor ID entry must begin with a :svopid tag and continues to the next occurrence of a :svopid or :procauth tag or to the end of the profile.

```
:svopid.id
```

This tag identifies the beginning of an entry and is required. It specifies the number to be used in the left argument of □SVO when sharing with the processor described by this entry. For incoming offers its value is returned by □SVQ. It must be a positive number. This value can be coded as 0 to allow incoming offers to be assigned a unique processor number. Such an entry could not, however, be used to initiate an offer.

```
:processor.id[,id[,id]]
```

This tag gives one, two, or three processor numbers separated by commas and is required. These numbers represent the actual procid, parent and grandparent of the share partner.

Because offers to processors with procids less than 1000 are considered to be offers to dependent processors, a profile ID entry is required to share with a processor running independently on the same machine and user ID. In this case, the svopid can be the same or different than the actual procid of the independent processor. Note that if the same number is used, it is impossible to share with a dependent processor using that number as its procid.

id can be coded as "*" in which case the entry identifies any remote processor with the corresponding :svopid.

```
:address.addr
```

This tag gives the IP address of the partners machine in IP "dotted decimal" notation consisting of four decimal numbers between 0 and 255 separated by periods and is optional.

addr can be coded as "*" in which case the entry identifies processors from any address with the corresponding :svopid.

| :userid.*userid*

This tag gives the character identification of the user ID of the partner and is optional. It can be one to eight characters and is case sensitive.

userid can be coded as "*" in which case the entry identifies processors from any user ID with the corresponding :svopid.

Here is an example of a processor entry that defines □*SVO* argument 33586 as a remote user.

```
* user BROWN at STLAPL
:svopid.33586
          :address.123.45.6.78
          :userid.BROWN
          :processor.1002
```

Authorization Entries

Each processor authorization entry must begin with a :procauth tag and continues to the next occurrence of a :procauth or :svopid tag, or to the end of the profile.

| :procauth.*id[,id[,id]]*

This tag identifies the procid, parent and pparent that is authorized to receive shares and is required. *id* can be coded as "*" in which case the entry serves to authorize all local processors.

| :rsvopid.*id[,id[,...]]*

This tag lists the svopid numbers that identify remote processors that are authorized to share with the processor named in the corresponding :procauth tag. Multiple numbers can be listed separated by commas.

id can be coded as "*" in which case the entry authorizes any remote processor to share with the corresponding :procauth.

Here is an example of an entry that authorizes the processor identified by :svopid.33586 to share with a local processor 100 dependent on processor 1001:

```
* AP100 authorization
:procauth.100,1001
          :rsvopid.33586
```

Communicating with Version 2 Release 1

The processor profile syntax has been changed from Version 2 Release 1. Version 2 Release 2 recognizes cross-system share requests from Version 2 Release 1. If you are initiating a share to a Version 2 Release 1 system, you must use the processor profile syntax defined for Version 2 Release 1. See *APL2/370 Programming: System Services Reference* for Version 2 Release 1 for syntax information.

Processor Profile Examples

The technical reference material you need to share variables between processors running on different machines was presented in “Processor Network Identification” on page 86 and “Processor Profile Structure” on page 87. This section provides examples of how to code processor profile entries for some typical application needs.

For the purposes of these examples, assume that there are three users running interpreters on three different machines. Each interpreter process is identified with a unique IP address, user ID, and processor number. The processor numbers correspond to ⍎⎕*AI* as reported by the interpreters themselves.

For clarity, these sample interpreters are referred to as Users 1, 2, and 3. You need the following information:

User	Address	User ID	⍎⎕*AI*
1	9.10.11.123	BARB	1001
2	9.10.11.222	jsmith	32739
3	123.45.6.77	djones	6666

User to User Shared Variables: Assume that User 1 wants to share variables with User 2. The information needed is as follows:

1. In order to offer to share a variable with another processor, you need to identify that processor with some number. A processor profile entry is then used to associate that number with the user's network information.

   Assume that User 1 wants to refer to User 2 with the number 777. The following entry is required in User 1's profile:

   ```
   :svopid.777
      :address.9.10.11.222
      :userid.jsmith
      :processor.32739
   ```

   This entry allows User 1 to extend an offer to User 2.

2. User 1 needs to authorize User 2 to share variables. User 2 is already identified as 777 with the :svopid tag, and it is known that User 1 is running as processor 1001, so the following entry can be used:

   ```
   :procauth.1001
       :rsvopid.777
   ```

   This authorizes remote processor 777 to share variables with local processor 1001.

Remember that there are always two sides to every share. User 2 also needs a number to use to refer to User 1. Assume that User 2 wants to use the number 3456. The following entry is required in User 2's profile:

```
:svopid.3456
   :address.9.10.11.123
   :userid.BARB
   :processor.1001
```

User 2 also needs to authorize User 1:

```
:procauth.32739
   :rsvopid.3456
```

Note that the :rsvopid values correspond to the :svopid values shown above, and the :procauth values correspond to the processor numbers of the interpreters as reported by ↑⎕*AI*.

If Users 1 and 3 also wanted to share variables, they would have to code identification and authorization entries just as Users 1 and 2 did. However, when coding the authorization entry, User 1 can take either of two approaches. First, User 1 could simply add another authorization entry. Assume User 3 has been identified as processor 8888:

```
:procauth.1001
   :rsvopid.8888
```

You can have as many authorization entries in your processor profile as you want, but there is another way. The :rsvopid. tag can provide a list of processors. So, User 1 could add User 3's number to the existing entry like this:

```
:procauth.1001
   :rsvopid.777,8888
```

This entry authorizes the remote users associated with the numbers 777 and 8888 to share variables with User 1's 1001 processor.

User 1 could authorize more users by adding more entries or by just adding numbers to the :rsvopid tag.

User to Auxiliary Processor Shared Variables: For another example, assume that User 1 wants to share a variable with AP 127, which is running under interpreter processor 6666 on User 3's machine. This might be useful if User 3's machine contained a database that User 1 needed access to.

User 1 needs to add another identification entry to associate a number with the remote processor 127. Assume that User 1 wants to use the number 9127 to refer to the remote processor. The entry is as follows:

```
:svopid.9127
   :address.123.45.6.77
   :userid.djones
   :processor.127,6666
```

Notice the :processor tag now lists two processor numbers. This indicates that AP 127 is a dependent of processor 6666.

User 1 also needs to authorize shares with the remote processor 127 so the authorization entry becomes:

```
:procauth.1001
   :rsvopid.777,8888,9127
```

If Users 1 and 3 are set up for user to user sharing, User 3 already has identification entry for User 1. However, AP 127 is a new processor on User 3's machine, and is sharing variables, so another authorization entry needs to be added for User 3. Assuming User 3 has identified User 1 as 2229:

```
:procauth.127,6666
   :rsvopid.2229
```

Like the :processor. tag shown above, the :procauth. tag in this authorization entry lists two numbers. In this case, however, it does not refer to two separate processors. Only a single processor (in this case 127, dependent of 6666) can be listed in a :procauth tag. The exception to this rule is the use of an asterisk as mentioned below.

Using Asterisks in Processor Profile Entries: The examples shown demonstrate how to identify specific remote processors and authorize them to establish shares with specific local processors. Sometimes however, it is not possible to identify all the potential share partners. Similarly, sometimes you want to give one or more share partners access to all your processors. Asterisks are used as wildcards in processor profile entries to provide general identification and authorization

Suppose that User 1 needs to identify and authorize shares from any processor running under a particular user ID. Rather than coding separate identification entries for each processor, an asterisk can be coded in the :processor tag. For example:

```
:svopid.1234
   :address.111.222.333.111
   :userid.JANE
   :processor.*
```

This entry identifies any processor on JANE's machine as processor ID 1234. Notice that because only one processor can be associated at a time with an :svopid tag, variables can only be shared with one processor on JANE's machine at a time when this technique is used.

Asterisks can also be coded in the :address and :userid tags. For example, if User 1 needs to identify any processor 127's running on the network, the following entry could be used:

```
:svopid.1234
   :address.*
   :userid.*
   :processor.127
```

Notice again that this technique requires that only one processor at a time can be associated with the :svopid 1234.

By using several identification entries each of which can identify several processors, you have now handled the problem of identifying multiple processors, and can now deal with authorization. An asterisk can also be used in the :rsvopid tag like this:

```
:procauth.1001
   :rsovpid.*
```

This entry authorizes any identified processor for shares with processor 1001.

Finally, assume that User 1 not only wanted to authorize remote users to share with processor 1001, but with any processor. The following entry can be used:

```
:procauth.*
   :rsvopid.*
```

This entry authorizes all remote processors, indicated by the asterisk in the :rsvopid. tag, to establish shares with any processor on the machine, indicated by the asterisk in the :procauth. tag.

Transferring Workspaces

Workspaces are easily transferred between APL2 systems. Transfer file formats have been defined to permit exchange of workspace objects among all IBM APL2 implementations. Additional tools are provided for migration of the older VS APL format to APL2.

Workspace Transfer between APL2 Systems

In general, APL2 workspaces must be sent to other APL2 systems as *transfer form* files. Transfer forms have the following default file naming conventions:

CMS	*filename* APLTF *
TSO	*prefix*.APLTF.*filename*
OS/2 or DOS	*path**filename*.ATF
AIX* or UNIX**	*path*/*filename*.atf

The APL2 commands used to create and read transfer form files are)*OUT*,)*IN*, and)*PIN*. To transfer a workspace, start APL2 on the system where the workspace resides, and issue the following commands:

```
)LOAD wsid
)SIC  (or )RESET)
)OUT filename
```

A transfer file is created by the)*OUT* command.

Once the transfer file is created, it then must be moved to the target APL2 system, and be saved with a name following the conventions of the target system. The techniques for physically moving files from one system to another can vary depending on the types of systems and what connections exist between them.

- One key issue is that some systems (for example MVS/TSO and VM/CMS) use an EBCDIC character encoding, while others (for example OS/2 and AIX/6000) use an ASCII encoding. Both ASCII and EBCDIC transfer file formats are defined, and all IBM APL2 systems accept both formats. No data conversion should be attempted within the file itself when transferring it from one system to another. The receiving APL2 system performs any necessary conversion. If the transfer is done electronically through a network connection, the programs controlling that transfer must be told that this is a “binary” rather than “character” file. (The exact terminology used may vary depending on the system and network control programs being used.)
- Some systems use “record-oriented” files while others use stream files. If stream files are being transferred to a system that expects record-oriented files,

an arbitrary record length may be used, but the existing record separators ("LF" or "CR/LF") must be retained. Conversely, separators should not be inserted when record-oriented files are being transferred to a system that expects stream files. Again, the receiving APL2 system adjusts to these differences.

- Within these constraints, standard data transmission commands appropriate to the system such as "ftp put," "SEND," "SENDFILE," or "TRANSMIT" can be used for network transmission, with corresponding commands such as "ftp get" or "RECEIVE" as appropriate to the receiving system.
- Because the receiving APL2 system performs all necessary conversions, it is also possible to use shared DASD, remote file systems, removable media, or other such facilities to transport the data.

When the file has been transferred to the target system, it can then be read into APL2 and saved as a workspace:

```
)CLEAR
)IN filename
)SAVE wsid
```

Migration of TryAPL2 Workspaces

Workspaces saved under TryAPL2 can be read by APL2/2, APL2/6000, and APL2 for Sun Solaris. The function *TRYLOAD* in the FILE workspace can be used to read these files. Once migrated to one of the workstations, the)*OUT* and)*IN* processes can be used to migrate to the mainframe.

Migration of VS APL Workspaces

VS APL workspaces can be migrated to the mainframe platforms using the)*MCOPY* command. For more information, see *APL2 Migration Guide*.

Transferring AP 211 Files

Files created by AP 211 are portable between APL2/370, APL2/2, APL2/6000, and APL2 for Sun Solaris. The files must be transferred in binary mode. The receiving APL2 system performs all necessary conversions of data. Files to be uploaded to the mainframe must be uploaded as fixed format files, with a record length equal to the AP 211 blocksize for the file. The blocksize can be obtained by issuing an AP 211 'USE' command against the file.

In addition, files created by AP 211 on APL2/PC can be read by APL2/2, APL2/6000, and APL2 for Sun Solaris. Writing back to these files is not allowed. The function *REBUILD*211 in the public workspace 2 FILE can be used to permanently convert the APL2/PC file to the new format if desired. Once migrated to one of the workstations, the file can then be uploaded to the mainframe.

Part 2. Auxiliary Processors

Chapter 9. Summary of Auxiliary Processors Distributed with APL2 . . . 102
Using Auxiliary Processors . . . 103
Using the Share-Offer Utilities . . . 104
Suggestions for Use of Auxiliary Processors . . . 105

Chapter 10. AP 100—Host System Command Processor Under CMS . . . 107
Associated Workspace . . . 107
Shared Variable Overview . . . 107
Initial Value . . . 108
Communication Procedure . . . 108
Querying the Operating System . . . 109
CMS Communication and IMPCP and IMPEX Settings . . . 109
Cautions . . . 110
Return Codes . . . 110

Chapter 11. AP 100—Host System Command Processor Under TSO . . . 112
Associated Workspace . . . 112
Shared Variable Overview . . . 113
Communication Procedure . . . 113
Querying the Operating System . . . 114
Return Codes . . . 114
AP 100 Built-In Commands . . . 116

Chapter 12. AP 101—Alternate Input (Stack) Processor . . . 127
Associated Workspaces . . . 127
Shared Variable Overview . . . 128
Data Formats . . . 128
Initial Values . . . 128
Communication Procedure . . . 129
AP 101 Commands . . . 129
Using AP 101 within a Defined Function . . . 130
Disposition of Data on the Stack . . . 130
Exiting and Returning to APL2 within a Defined Function . . . 131
Using the INPUT Invocation Option in a Stacked APL2 Command . . . 131
Using AP 101 with the TSO Fence Option . . . 131
Fence Commands . . . 132
Cautions . . . 132
Return Codes . . . 133

Chapter 13. AP 102—Main Storage Access Processor . . . 134
Shared Variable Overview . . . 134
Commands . . . 135
Communication Procedure . . . 135
Formatting the Result from AP 102 . . . 136
Cautions . . . 136
Return Codes . . . 137

Chapter 14. AP 110—CMS File Processor . . . 138
Associated Workspace . . . 138
Shared Variable Overview . . . 138

Initial Values . . . 139
Communication Procedure . . . 140
Record Variable . . . 141
Control Variable . . . 141
Specifying the Control Variable . . . 141
Cautions . . . 143
Return Codes . . . 144

Chapter 15. AP 111—QSAM File Processor . . . 146
Associated Workspaces . . . 146
Shared Variable Overview . . . 146
Initial Values . . . 147
Communication Procedure . . . 148
Cautions . . . 150
Return Codes . . . 151
Undiagnosed Errors . . . 152

Chapter 16. AP 119—Socket Interface Processor . . . 154
Shared Variable Overview . . . 154
The APL2 Socket Application Program Interface . . . 154
IUCV Paths and Sockets . . . 155
AP 119 and TCP/IP Commands Summary . . . 155
Definition of TCP/IP Terms . . . 156
Blocking . . . 158
Using AP 119—The TCPIP Commands . . . 158
ACCEPT . . . 158
BIND . . . 159
CLOSE . . . 159
CONNECT . . . 159
FCNTL . . . 160
GETHOSTID . . . 160
GETHOSTNAME . . . 161
GETPEERNAME . . . 161
GETSOCKNAME . . . 162
GETSOCKOPT . . . 162
LISTEN . . . 163
READ . . . 163
RECV . . . 164
RECVFROM . . . 164
SELECT . . . 165
SEND . . . 166
SENDTO . . . 166
SETSOCKOPT . . . 167
SHUTDOWN . . . 167
SOCKET . . . 168
WRITE . . . 168
Using AP 119—The AP Commands . . . 168
The APL2 Port Server . . . 169
PSLIST—Send LIST Command to the Port Server . . . 169
PSCLEAR—Send CLEAR Command to the Port Server . . . 170
PSSHUTD—Send SHUTDOWN Command to the Port Server . . . 170
UNREGSTR—Send an UNREGISTER Command to the Port Server . . . 170
Listening Ports . . . 170
GETLPORT—Get Listening Port . . . 171

SETLPORT—Set Listening Port . . . 171
Starting AP 119 . . . 171
Sample AP 119 Session Using the APL2 Socket API . . . 172
Return codes . . . 175

Chapter 17. AP 120—APL2 Session Manager Command Processor . . . 179
Shared Variable Overview . . . 179
Data Formats . . . 180
Specification . . . 180
Reference . . . 180
Communication Procedure . . . 180
Return Codes . . . 181

Chapter 18. AP 121—APL2 Data File Processor . . . 182
Associated Workspaces . . . 182
Shared Variable Overview . . . 182
Access Control Considerations . . . 183
APL2 Data Files . . . 183
File Identification . . . 183
APL2 Data File Organization . . . 184
Communication Procedure . . . 185
Commands . . . 185
Opening a File . . . 187
Open for Sequential Write (SWC or SW) . . . 187
Open for Sequential Read (SR) . . . 187
Open for Direct Read (DR) . . . 187
Open for Direct Update (DUC or DU) . . . 188
Checking for End of File . . . 190
APL2 Data File Maintenance . . . 190
Library Query . . . 190
Space Requirements for Storing APL2 Variables . . . 191
Size Limitations . . . 191
Cautions . . . 191
Return Codes . . . 192

Chapter 19. AP 123—VSAM File Processor . . . 195
Associated Workspaces . . . 195
Shared Variable Overview . . . 195
VSAM Files—General Information . . . 195
File Identification . . . 196
File Formats and Keys . . . 196
Commands . . . 197
Communication Procedure . . . 198
Opening a VSAM File . . . 199
Processing a VSAM File . . . 200
Reading a File . . . 200
Writing a File . . . 200
Replacing a Record . . . 201
Erasing (Deleting) a Record . . . 201
Obtaining the Key of the Last I/O Operation . . . 201
Positioning the Record Pointer . . . 202
Specifying Character Conversion . . . 203
Closing a VSAM File . . . 203
Cautions . . . 203

Return Codes 204

Chapter 20. AP 124—Text Display Auxiliary Processor 207
Shared Variable Overview 207
Understanding Screen Management 208
Logical Screens 208
Screen Fields 208
Field Attributes 209
Communications Procedure 209
Screen Management Commands 210
Delayed Clear of the Screen 210
Formatting the Screen 211
Immediate Write of Data to Screen 214
Read and Wait 214
Writing to the Screen 216
Getting Data 217
Modifying Field Attributes 217
Returning Screen Information 218
Reading the Screen Format 218
Sounding the Alarm 219
Setting the Cursor 219
Modifying Input Field Attributes 220
Erasing the Screen 220
Return Codes 221

Chapter 21. AP 126—GDDM Processor 222
Associated Workspaces 222
Licensed Program Requirements 223
Shared Variable Overview 223
Data Formats 224
Returned Values 224
Communication Procedure 226
GDDM Calls 228
Restrictions 228
AP 126 Commands 228
Query GDDM Calls 229
Set Error Threshold 230
Set Protection Key 231
Set EBCDIC Translation 231
Set Default Buffer Size 232
Set AP 126 Options 232
Query AP 126 Options 233
Query Subset of Fields for Modifications 233
Query Current Hard-Copy Destination 234
Issue CHART Call 234
Obtaining Copies through AP 126 237
GDDM FSOPEN Request or DSOPEN, DSUSE Sequence 237
Alternating Paths 237
Implications of Multiple Data Paths 237
Page Sharing with the APL2 Session Manager 238
Guidelines for Sharing with the Session Manager 238
Handling Attentions 239
APL2/370 and GDDM EBCDIC Code Page Differences 240
GDDM Error Diagnosis 241

Return and Reason Codes 241

Chapter 22. AP 127—SQL Processor 244
Shared Variable Overview 245
Communication Procedure 245
AP 127 Commands 246
Return Codes 247

Chapter 23. AP 210—BDAM File Processor (TSO Only) 248
Associated Workspace 248
Shared Variable Overview 248
BDAM File Requirements 249
DCB Attributes Provided by AP 210 249
Communication Procedure 249
Initial Values 249
Formatting a Direct File Using AP 210 250
BDAM File Processing Procedure 251
Cautions 253
Return Codes 253
Data Management Error Codes 254

Chapter 24. AP 211—The APL2 Object File Processor 255
Shared Variable Overview 255
Commands Accepted by AP 211 255
CREATE 255
DROP 256
USE 257
RELEASE 257
SET 258
GET 258
RENAME 258
ERASE 258
LIST 259
Return Codes 259

Chapter 25. APL2 Shared Variable Interpreter Interface 261
Shared Variable Interpreter Interface Protocols 261
Shared Variable Overview 262
Interpreter Input Data 262
Interpreter Output Data 263

Chapter 9. Summary of Auxiliary Processors Distributed with APL2

An auxiliary processor is an autonomous program that performs services for APL2 users. Such services include access to files and databases, control of the user's display screen, interaction with other subsystems, and inspection or modification of the system and application environment.

Auxiliary processors may be supplied with APL2 or they may be written by your system programming staff. Information on writing auxiliary processors can be found in *APL2/370 Programming: Processor Interface Reference*.

Figure 17 lists, in sequence by numeric identifier, the auxiliary processors supplied with APL2. With the exceptions noted below, all processors are available under both CMS and TSO.

- AP 110 is available only under CMS.
- AP 210 is available only under TSO.

These two processors provide comparable file access facilities appropriate to their respective host systems.

In the **Protocol** column of Figure 17, if the processor requires a pair of variable names that start with *CTL* and *DAT*, the names *CTL* and *DAT* are so identified. Otherwise, if no naming conventions are required, the number and type of required variables are indicated without a specific name.

Figure 17 (Page 1 of 2). Auxiliary Processors Supplied with APL2

Processor	Purpose	Protocol
AP 100 Host System Command Processor	Issue commands to the host system; execute user-written programs.	One variable.
AP 101 Alternate Input (Stack) Processor	Create a stack of programmable input to APL2 (or to the host system after ending an APL2 session).	One variable.
AP 102 Main Storage Access Processor	Obtain the contents of specified areas of main storage.	Two variables—*CTL* and *DAT*.
AP 110 CMS File Processor	Sequential or direct access to CMS files. *CMS Only*	One or two variables (record and control)
AP 111 QSAM File Processor	Sequential access to QSAM files.	One or two variables (record and control). Control variable is optional.
AP 119 Socket Interface Processor	Controls the TCP/IP socket interface and cross-system shared variables.	One variable.
AP 120 Session Manager Command Processor	Issue session manager commands from within a defined function.	One or two variables—*CTL* and *DAT*. *DAT* variable is optional.

Figure 17 (Page 2 of 2). Auxiliary Processors Supplied with APL2

Processor	Purpose	Protocol
AP 121 APL2 Data File Processor	Stores and retrieves APL arrays by object number within a special APL-formatted file.	One variable—*CTL*—for sequential access. Two variables—*CTL* and *DAT*—for direct access.
AP 123 VSAM File Processor	Access VSAM data.	Two variables—*CTL* and *DAT*.
AP 124 Text Display Processor	Controls the screen of an IBM 3270 Information Display System terminal.	Two variables—*CTL* and *DAT*.
AP 126 GDDM Processor	Use the facilities of the Graphical Data Display Manager.	Two variables—*CTL* and *DAT*.
AP 127 SQL Processor	Use the facilities of the Structured Query Language.	One variable—data variable.
AP 210 BDAM File Processor for TSO	Relative-record access to fixed-length, unkeyed BDAM data sets. *TSO Only*.	Two variables—record and control.
AP 211 APL2 Object File Processor	Stores and retrieves APL arrays by name in an APL-formatted file.	One variable.
Shared Variable Interpreter Interface	Control APL2 interpreter through shared variable	One variable named *APL2*.

Using Auxiliary Processors

To use an auxiliary processor requires a shared variable, or for some auxiliary processors, a pair of shared variables, to pass commands and data between the APL2 application and the auxiliary processor. Since auxiliary processors can run asynchronously with the APL2 interpreter, applications must follow a proper Shared Variable Processor (SVP) protocol when establishing shared variable communication with an auxiliary processor, to avoid potential timing problems with the two processes running in parallel. For a complete description of the SVP system functions, refer to *APL2 Programming: Language Reference*.

The examples given for most auxiliary processors in the following chapters show □*SVO* and □*SVC* followed immediately by a specification of the first request in the variable and a reference to get a return code. This approach works well if the auxiliary processor is known to be available, but if the auxiliary processor is not responding, the APL2 session hangs on the reference, and you must interrupt it. An alternative approach is to wait until the variable is fully coupled before making the first request. This can be accomplished using □*SVE* and □*SVO* in a loop, and the loop can be written to time out if it appears the auxiliary processor is not responding. This has the disadvantage that the time chosen may not be long enough for some situations, but the advantage that the application can recover without your intervention.

Using the Share-Offer Utilities

To simplify the establishment of fully-coupled shares, and to ensure that the necessary access control is set for typical communication with an auxiliary processor, two functions are distributed in the library 1 UTILITY workspace for application developers. The two functions are *SVOFFER* and *SVOPAIR*. *SVOFFER* is for use with auxiliary processors employing a single shared variable interface. *SVOPAIR* is used for auxiliary processors such as AP 124 and AP 210 that require a control and a data variable for communication.

The *SVOFFER* function must return a "degree of coupling" of `2` for each variable offered, before the shared variable can be used to pass commands and data. This indicates that the auxiliary processor has accepted the share offer. An indeterminate amount of time is required for the auxiliary processor to accept the offer. Typically, an auxiliary processor accepts the shared variable offer immediately, but the *SVOFFER* function queries the degree of coupling for a maximum of 15 seconds before exiting with a result of `1` indicating that the auxiliary processor has not matched the offer.

The *SVOPAIR* function is used for auxiliary processors that support a two-variable interface, where the control variable name begins with "*CTL*" and the data variable name begins with "*DAT*." *SVOPAIR* waits up to 15 seconds for all control variable offers to be accepted. It returns the final degree of coupling for all variables offered. The expected coupling for the control variables is 2 (fully coupled), and the data variables can properly return either 1 or 2, depending on the auxiliary processor.

Prior to sending commands to an auxiliary processor, shared variable access control should be set to ensure that the SVP maintains the necessary sequencing of sets and references of the shared variable by both the APL2 application program and the auxiliary processor. *SVOFFER* and *SVOPAIR* set the necessary access controls for typical auxiliary processor communication. *SVOFFER* sets access control on all of the variables offered, and *SVOPAIR* sets access control only on the variables with names starting with the letters "*CTL*" (that is, control variables only—no access control is applied to data variables). The access control applied is `1 0 1 0`, which prevents two successive sets of the variable by the application without an intervening access by the auxiliary processor, and also ensures that the auxiliary processor sets a new value in the variable between successive uses by the APL2 application. This is the most common access protocol used for shared variable communication with the auxiliary processors.

Note: The *SVOPAIR* function was written for use with the workstation 2-variable auxiliary processors, AP 124 and AP 210. It is included with the mainframe APL2 for compatibility and is useful for writing portable code that uses AP 124. However, the mainframe 2-variable auxiliary processors are written to an older interface. Variables are matched immediately, if matched at all, and there might be different access control requirements for the data variable. The logic of the *SVOPAIR* function is not necessary or appropriate for all these auxiliary processors.

Example 1

```
      ⍝ Single offer to host auxiliary processor
           100 SVOFFER 'CMD'
      2

      ⍝ Offer multiple variables to one AP
           100 SVOFFER 'V1' 'V2'
      2 2

      ⍝ Offer multiple variables to multiple APs
           100 211 SVOFFER 'V100' 'V211'
      2 2

      ⍝ Check degree of coupling for multiple variables
           SVOFFER 'V100' 'V211'
      2 2

      ⍝ Invalid shared variable offer
           211 SVOFFER 'BAD+NAME'
      0

      ⍝ Offer and trap errors
           ⎕ES (2∨.≠AP SVOFFER VARS)/'Share offer unaccepted by AP',⍕AP
```

Example 2

```
      ⍝ Offer a set of variables to the fullscreen processor
           124 SVOPAIR 'CTL124' 'DAT124'
      2 2

      ⍝ Offer using surrogates
           124 SVOPAIR 'Control C' 'Data D'
      2 2

      ⍝ Note: Access control set for control, not data
           ⎕SVC¨ 'Control' 'Data'
       1 0 1 1  0 0 0 0

      ⍝ Check degree of coupling
           SVOPAIR 'Control' 'Data'
      2 2
```

Suggestions for Use of Auxiliary Processors

The display of a shared variable constitutes a use of the variable. To avoid being interlocked when you need to use the returned value more than once, first assign the variable to another variable, and then use that variable instead of the shared variable.

To ensure that an interface has been established with your intended partner, use ⎕*SVO* to query an offer.

It is good programming practice to share local variables. The localized shared variable is automatically retracted even if the function aborts.

When using localized names or surrogates, it is possible to share two variables with the same name. Ensure that your functions are coded such that you can avoid a mismatched variable when using an auxiliary processor that can share either one variable or a pair of variables. Otherwise, use of duplicate names can cause a data variable to be paired with the wrong control variable. To avoid this problem,

always offer to share a pair when using an auxiliary processor with a 2-variable protocol, even when you do not need a pair.

Chapter 10. AP 100—Host System Command Processor Under CMS

The CMS version of AP 100, the host system command processor, can be used within an APL2 session to pass commands to CP or CMS, to process user-written programs, or to process CMS EXECs.

Associated Workspace

APL2 is distributed with the CMS workspace. The workspace contains cover functions that can be used to communicate with AP 100. For information on these functions, type *DESCRIBE*, *HOW*, or *ABSTRACT* after loading the CMS workspace.

Shared Variable Overview

Figure 18 provides an overview for sharing variables with AP 100 under CMS.

Figure 18. Shared Variable Overview for AP 100 under CMS

SV Protocol	AP 100 Conventions
General	One variable. Each specification passes a command to CP or CMS. Each reference obtains the return code from the most recent specification.
Maximum Number of Shared Variables	14
Name	Any valid APL2 variable name not exceeding 77 characters.
Initial Value	See “Initial Value” on page 108.
Subsequent Values	Specify a character vector not exceeding a length of 200. Reference a scalar integer or a character vector.
Data Types Supported	Character vectors.
Access Control	0 0 0 1

Initial Value

The initial value of a variable offered to AP 100 specifies whether subsequent specifications are CP or CMS commands and indicates whether conversion is to be applied to the commands.

The format of the initial value is:

```
CMS100←'target(conversion)'
```

where the options are:

target Target system for commands subsequently passed through the variable. Valid targets are:

- *CMS* (default)
- *CP*
- *SUBCOM* (for CMS subcommands)

conversion Conversion to be applied to the commands. Valid conversion options for AP 100 under CMS are:

- *BCD* (default)
- *EBCD*

If no initial value is specified or if the initial value is empty or invalid, defaults of *CMS* and *BCD* are assumed.

The default conversion is BCD for compatibility with existing applications coded under previous implementations of APL. However, it is recommended that new applications using AP 100 specify the EBCD option because the internal encoding of APL2 is EBCDIC. Standard CMS translation as controlled by the CMS SET INPUT command is applied, so with the conversion option EBCD commands can be entered in mixed uppercase and lowercase. With the conversion option BCD, commands can be entered in mixed uppercase and underbarred case.

Note: The)*HOST* command uses the EBCD translation option.

Communication Procedure

Figure 19 on page 109 shows a sample APL2 session to communicate with AP 100. In the sample, AP 100 is being used to issue the FILEDEF command for a QSAM file, query and set IMPEX, and process a user-written CMS EXEC.

```
      CMS100←'CMS(EBCD'          ⍝ Optional Initialization
      100 □SVO 'CMS100'          ⍝ Offer to Share
2                                ←──────────Degree of coupling OK
      1 0 1 0 □SVC 'CMS100'      ⍝ Set Access Control
1 0 1 1
      CMS100
0                                ←──────────Return code OK
      DCB←'(RECFM F LRECL 60)'
      CMS100←'FILEDEF QSAMFIL DISK USERFILE TEST ',DCB
      CMS100                     ⍝ Check Return Code from FILEDEF
0
      CMS100←'QUERY IMPEX'       ⍝ Display IMPEX Setting
IMPEX = OFF                      ←──────────Display from CMS, not APL2

      CMS100
0                                ←──────────Return code OK

      CMS100←'SET IMPEX ON'      ⍝ Set IMPEX for Executing EXEC
      CMS100
0
      CMS100←'USEREXEC'          ⍝ Execute User-Written EXEC
      .
      .
      .
```

Figure 19. Sample CMS APL2 Session to Communicate with AP 100

To cancel output from a CP command, press the PA1 key or the attention key. To cancel output from a CMS command, enter the HT command (halt terminal output) from the terminal.

Querying the Operating System

AP 100 returns a character string containing the name of the operating system if a null character vector is specified. This allows applications that need to run on multiple systems to determine dynamically which environment they are running in.

```
      CMS100←''
      CMS100
CMS
```

CMS Communication and IMPCP and IMPEX Settings

When the target in the initial value of the shared variable is *CMS* (the default) or *SUBCOM*, the settings of two operands of the CMS SET command (IMPCP and IMPEX) affect the way you issue a CP command or process an EXEC using AP 100.

On issuing a CP command:

- If IMPCP is set OFF, you must precede a CP command with the letters CP. For example:

  ```
  CMS100←'CP QUERY FILES'
  ```

- If IMPCP is set ON, you may omit the letters CP. For example:

  ```
  CMS100←'QUERY FILES'
  ```

User-written CMS EXECs can be processed through AP 100 by specifying the name of the EXEC in the shared variable.

- If IMPEX OFF is the setting, you must precede the name of the EXEC with the characters EXEC. For example:

  ```
  CMS100←'EXEC USEREXEC'
  ```

- If IMPEX ON is in effect, only the name of the EXEC need be specified. For example:

  ```
  CMS100←'USEREXEC'
  ```

For information on IMPCP and IMPEX, see *VM/SP CP Command Reference for General Users*.

Cautions

Entry of any CMS command, EXEC, or module that requires or resets OS storage or that overlays CMS storage may cause an abrupt termination of an auxiliary processor, an entry defined using ⎕NA, or even APL2 itself. These include commands such as EXECOS, LOAD, LOADMOD, and START.

On the 3270 family of terminals when running the session manager or AP 126, CP messages may not be displayed immediately, but rather are displayed at the end of the session, when you press the PA1 key, or when you press the key assigned using the CP TERMINAL BRKKEY command.

Return Codes

After the share offering is complete, unless specified with a null character string, each reference of the variable shared with AP 100 returns a scalar integer. This return code can originate from several sources:

- AP 100
- CP
- CMS
- User-written program

Return Codes from AP 100: Figure 20 contains the possible numeric values that AP 100 can generate.

Figure 20. Return Codes Issued by AP 100 under CMS

Code	Description
¯100	CMS command ended abnormally. A CMS message containing an ABEND code is displayed at the user's terminal. The failing command may have damaged the integrity of CMS or APL2.
0	No error exists; the initial value was accepted or the command completed successfully.
1	Unknown CP command or improper initial value.
444	Invalid value assigned to the shared variable. It is the wrong shape, size, or data type.

Return Codes from CMS: CMS return codes may indicate that the command could not be found or processed. They may also be returned when the command finishes processing.

For information about CMS return codes, see *VM/SP System Messages and Codes*.

Return Codes from CP: The CP codes returned through AP 100 are extracted from the numeric portion of the CP message id. For example, the CP message id DMKCQG020E generates a return code of 20.

For definitions of CP messages, see *VM/SP System Messages and Codes*.

Return Codes from User Programs: CMS EXECs and user-written programs can generate any return code. The meanings of these codes must be supplied by the author of the program.

Chapter 11. AP 100—Host System Command Processor Under TSO

The TSO version of AP 100, the host system command processor, can be used within an APL2 session to process TSO commands or CLISTs (including ISPF CLISTs), or to call special AP 100 built-in commands.

TSO commands are normally processed in an environment separate from APL2. That is, task libraries associated with APL2 or with any tasks that started APL2 are ***not*** searched for the commands, and authorized commands can be used even though APL2 is not authorized. There is a special TSO built-in command described at the end of this chapter that permits commands to be issued within the APL2 environment.

CLISTs to be processed should normally be preceded by EXEC or %. If this is not done, any command of the same name is used in preference to the desired CLIST. You must *always* use the EXEC or % form to process a CLIST in an ISPF environment that invoked APL2. Except for the ISPF case, CLISTs are executed in an environment separate from APL2, as described for commands above. AP 100 can be used to process CLISTs and REXX execs. They can be processed either directly or through the built-in commands EXEC and TSO. However, if ISPF is active, commands that are prefixed with either % or that are run using the built-in command EXEC are passed to ISPF for processing. In these cases, ISPF is usually able to detect changes that may have been made to the screen, and refresh it if necessary. When processing commands that can invoke ISPF display services, either the command should be prefixed with % or the EXEC built-in command should be used. Otherwise, an ISPF CONTROL DISPLAY REFRESH service request should be issued to avoid unpredictable screen results.

The)*HOST* system command can be used interactively to invoke any command, CLIST, or built-in command with exactly the same support provided by AP 100, except that the return code or result is displayed rather than assigned to a variable.

Associated Workspace

The APL2 Licensed Program is distributed with the TSO workspace, which your installation may have placed in public library 1 or 2. The workspace contains cover functions that can be used to communicate with AP 100. For information on these functions, type *DESCRIBE*, *HOW*, or *ABSTRACT* after loading the TSO workspace.

Shared Variable Overview

Figure 21 provides an overview for sharing variables with AP 100.

Figure 21. Shared Variable Overview for AP 100 under TSO

SV Protocol	AP 100 Conventions
General	One variable. Each specification passes a command. Each reference obtains the return code from the command or data returned from an AP 100 built-in command.
Maximum Number of Shared Variables	14
Name	Any valid APL2 variable name not exceeding 77 characters.
Initial Value	*BCD* or *EBCD* conversion option. (Default is *EBCD*.)
Subsequent Values	Specify a character vector not exceeding a length of 32765. Reference a vector of integers or a character array.
Data Types Supported	Character and integer scalars, vectors, and matrixes.
Access Control	0 0 0 1

Communication Procedure

Figure 22 shows a sample APL2 session to communicate with AP 100. The sample uses AP 100 to pass the ATTRIB and ALLOCATE commands to TSO for a QSAM file that can be subsequently processed using AP 111.

Using AP 100, commands are passed with 370 or EBCD conversion.

To cancel output from a TSO command or CLIST, signal attention.

```
      TSO100←''                  ⍝Set Initial Value
      100 ⎕SVO 'TSO100'          ⍝Offer to Share
2                        ←──────────Degree of Coupling OK
      1 0 1 0 ⎕SVC 'TSO100'      ⍝Set Access Control
1 0 1 1
      A←'ATTR ATTRNAME DSORG(DA) '
      TSO100←A,'LRECL(80) BLKSIZE(800) RECFM(F B)'
      TSO100                     ⍝Check Success of ATTR Command
0                        ←──────────Return Code OK
      A←'ALLOCATE DDNAME(QSAMFIL) DSN(AP111FIL) '
      A←A,'SPACE(200,5) BLOCK(800) NEW '
      TSO100←A,'USING (ATTRNAME)'
      TSO100                    ⍝Check Success of ALLOCATE Command
0                        ←──────────Return Code OK
      TSO100←'EXEC USERCLST'     ⍝Execute CLIST Immediately
```

Figure 22. Sample TSO APL2 Session to Communicate with AP 100

Querying the Operating System

AP 100 returns a character string containing the name of the operating system if a null character vector is specified. This allows applications that need to run on multiple systems to determine dynamically which environment they are running in.

```
      TSO100←''
      TSO100
TSO
```

Return Codes

Each reference of the variable shared with AP 100 returns a vector of integers or a character vector or matrix. The variable can be specified by several sources:

- AP 100, while trying to analyze the request
- The requested built-in command (see "AP 100 Built-In Commands" on page 116)
- The TSOLNK command invocation facility
- The requested command or CLIST
- Operating system ABEND codes

Figure 23 and Figure 24 identify the return codes that AP 100 controls.

Note: A command or CLIST can create any possible return code, but there is little ambiguity if it follows the MVS standard of creating only return codes that are multiples of 4 between 0 and 4090.

Figure 23. TSO Return Codes Issued by AP 100 and Related Routines under TSO

Code	Description
0	The TSO command completed successfully.
1	Invalid TSO command or unauthorized use of the command.
2	You terminated the command with an attention interrupt.
3	Insufficient virtual storage to process command.
4	Insufficient shared storage to return the result. This may happen if you have specified one of the AP 100 built-in commands that return a large result value. **Action:** Reference other shared variables or reinvoke APL2 with a larger SHRSIZE value.
444	An invalid data type was set into the shared variable. It is the wrong size, shape, or type.

Figure 24. Additional Return Codes from AP 100 DDI, DSI, and PDSI Built-In Commands

Code	Description
1	Invalid syntax in command argument, or data set not available for PDSI command.
3	Insufficient free storage to process command.
5	Insufficient free storage for result.
8	Data set or ddname not found, or, for the PDSI command, the data set has been migrated or is not a valid PDS.

Return Codes from AP 100: Only return code 444 is normally generated by AP 100 itself. This return code is used if the APL workspace has set the shared variable to anything other than a character vector (or scalar) whose length is 1 to 256. It is also used if the APL ATTACH built-in command returns an invalid array.

AP 100 does generate return code 4 if the result produced by a built-in command is too large to fit in available shared storage.

Return Codes from a Built-In Command: As indicated in Figure 23 on page 114, these commands may issue return code 4 or 8. The context should show which command was issued, and that it was a built-in command.

Return codes from TSOLNK: The TSOLNK facility provides both a return code and a reason code, as documented in *TSO Extensions User's Guide*. Most of the return codes deal with actions by AP 100 or the attached command or CLIST, and are treated separately here. The TSOLNK return code indicating that attention was signalled is mapped into return code 2. TSOLNK return code 20 deals with a number of problems, more precisely defined by the reason code. These problems are mapped into return code 1, but a *)MORE* message is queued that lists the specific reason code:

PROCESSOR 100 ERROR *reason*

The most likely *reason* code is 44, a syntax error in the command name.

Return Codes from the Command or CLIST: This includes three cases:

- The contents of **&LASTCC** when a CLIST terminates
- The contents of register 15 when a command terminates normally
- The code associated with any ABEND issued by a command

In the third case a *)MORE* message is also queued:

TSO COMMAND *name* **ABEND - SYSTEM CODE - 000 - USER CODE -** *code*

Return Codes from an Operating System ABEND: The system abend code value (in decimal) is multiplied by 4096 to yield a very large return code value. An 80A abend, for example, is really X'80A000', and becomes 8429568. A *)MORE* message is also queued, listing the original code:

TSO COMMAND *name* **ABEND - SYSTEM CODE -** *hex* **- USER CODE - 0**

AP 100 Built-In Commands

With the TSO version of AP 100, you can display several types of data that describe the processing environment in which APL2 is running. The built-in commands provided by AP 100 allow you to obtain accounting information for the session, reset terminal translation options, reset debugging options, suppress terminal output, and more. Figure 25 lists and summarizes the available built-in commands. These commands are detailed following the figure.

Figure 25. AP 100 Built-In Commands

Command	Description
AI	Return TSO-oriented accounting information
ATTACH	Attach user-written modules
CODE	Obtain and set terminal translation options
DEBUG	Obtain and set debugging options
DDI	Return descriptive information on a file
DSI	Return descriptive information on a data set, or list data set names associated with an index level
LIB	Return a list of workspace names in a given APL2 library
LIBS	Return a list of APL2 public and project library numbers
NOMSG	Process a TSO command as if from a CLIST with CONTROL NOMSG
PDSI	Return descriptive information on the members of a partitioned data set
QUIET	Suppress APL2 terminal output until next terminal read request
QUOTA	Return a 10-item vector of integers
USER	Return information about the system environment in which APL2 is running
WSID	Return the name of the active workspace and the date last saved
WSNAME	Return the TSO data set name for a given workspace or library
EXEC	Process a TSO command procedure
TSO	Process a TSO command

Built-in Command Format: When you specify a built-in command in the variable shared with AP 100, you precede the command either with the characters APL and a space or with a right parenthesis. For example, either of the following specifications is valid for the AI accounting information built-in command:

```
TSO100←'APL AI'

TSO100←')AI'
```

There are two exceptions to this format—the EXEC and TSO built-in commands. These are entered without a leading *APL* or).

APL AI

The APL AI command returns a 10-item vector of integers containing TSO-oriented accounting information. The information supplements the information obtained from the APL2 system variable ⎕*AI*. The returned vector is summarized in Figure 26 on page 117.

Note: All items are numeric, and APL2 suppresses leading zeros on numeric displays.

Figure 26. Data Returned with the APL AI Command

Item	Example	Description
1	92087	The MVS system date The last three digits are the day of the year (for example, 087=MARCH 27) and the preceding digits are the year relative to 1900 (for example, 92=1992 and 100=2000).
2	1657241	Time of day HHMMSST MVS system clock time (hours, minutes, seconds, tenths)
3	1240	Session time HHMMSST Total connect time for this TSO session, including time not spent in APL2 (hours, minutes, seconds, tenths)
4	1888467	Session processor time MMMM Total processor time recorded for this TSO session, including time not spent in APL2 (micro-seconds)
5*	4698	MVS service units NNNN Total number of MVS service units recorded for this TSO session, including time not spent in APL2. Service unit numbers (NNNN) are established by MVS and installation parameters.
6*	31168	MVS transaction active time Total transaction active time for all transactions.
7*	10	Total number of MVS transactions recorded for this session, including time not spent in APL2.
8	8	Total number of TSO terminal input requests for this session, including time not spent in APL2.
9	31	Total number of TSO terminal output requests for this session, including time not spent in APL2.
10	11	MVS EXCP requests Total number of I/O requests recorded by the System Management Facility (SMF) for allocated direct access data sets. Data sets that were deallocated by the TSO FREE command are not counted.

Note:

- * Units displayed are those returned by system event code X'26'. See *OS/VS2 MVS System Programming Library Initialization and Tuning Guide*.

APL ATTACH modulename *[parameter-list]*

The ATTACH built-in command can be used to attach other subtasks during an APL2 session. It is for use by experienced system programmers.

One blank must separate the command name and the 1- to 8-character name of the module to be attached. An optional parameter list to be passed to the module can follow the module name.

If the LOADLIB option or DDNAME was specified when APL2 was invoked, the associated data sets are used as a TASKLIB for the module specified in the ATTACH command.

The attached module can pass results back as an APL2 formatted shared variable. It is recommended that all modules specified in the APL ATTACH command be invoked from APL2 functions that can ensure the proper data format for data passed.

APL CODE *[code]*

The APL CODE built-in command enables you to query and define the type of terminal you are using. Issued without an operand, the CODE built-in command returns the current setting of the terminal device code in effect. This code is the same as that set by the TERMCODE (or CODE) invocation option. These codes are listed in Figure 5 on page 30 in Chapter 2, “APL2 Invocation and Termination.” Your installation may have added other terminal device codes.

When APL2 is invoked by another program with the TERMCODE(-1) invocation parameter, the APL CODE 2 built-in command can be used to provide for TN translation of output to the file allocated with the DDNAME APLPRINT. The APL CODE 1 built-in command can be used to return to no translation of output. See the description of the TERMCODE invocation parameter in Chapter 2, “APL2 Invocation and Termination” on page 8.

APL DDI *name*

The APL DDI built-in command returns a matrix of descriptive information about the file entered in the *name* operand. The *name* must be a currently allocated ddname (or FILE name in an ALLOCATE statement). Figure 27 on page 119 shows the data that is returned from the DDI built-in command. The information returned is the same as that from the DSI command when a data set name is entered with that command.

```
DSNAME(  'fully.qualified.name' )
DDNAME(  ddname                 )
VOLUME(  volume                 )
UNIT(    name                   )
DIR(     nnn    /*  nnn   USED*/)
SPACE(   nnn                    )
/*UNIT*/ alloc_unit
/*DISP*/ use     disp
MEMB(    nnn    /*MEMBER COUNT*/)
SPACE(   nnn    TRACKS /*ALLOC*/)
SPACE(   nnn    TRACKS /*USED */)
/*EXT */ extents
DSORG(   org                    )
RECFM(   fmt attr               )
LRECL(   nnn                    )
BLKSIZE( nnn                    )
/*CRDT*/ dd-mm-yy.ddd
/*RFDT*/ dd-mm-yy.ddd
/*EXDT*/ dd-mm-yy.ddd
/*SEC */ auth
```

Figure 27. Descriptive Data Returned with the DDI and DSI Built-In Commands

alloc_unit *TRACKS*, *CYLINDERS*, or *BLOCKS*
use *SHR*, *OLD*, *NEW*, or *MOD*
disp *KEEP*, *CATALOG*, or *DELETE*
org *PS*, *PO*, *DA*, or *VS*
fmt *F*, *V*, or *U*
attr *B* and/or *S* and/or *M* and/or *A*
auth *RACF*, *PASSWORD WRITE*, *PASSWORD R/W* or blank.

APL DSI *name*

The APL DSI built-in command, depending on the value of the *name* operand, returns either a character matrix containing descriptive information on the data set entered as the *name* operand, or a list of data set names associated with the *name* operand.

- If a fully- or partially-qualified data set name is specified in the *name* operand, the DSI built-in command returns the descriptive data shown in Figure 27.
- If an incomplete name is entered as the *name* operand, the command returns a list of data set names associated with the incomplete name.

The relationship between the value specified in the *name* operand and the matrix that is returned is detailed in Figure 28.

Figure 28. Relationship of DSI Name Operand and Data Returned

Value of Name Operand	**Action by AP 100**
Partially-qualified (or simple) name using TSO conventions	User's profile-prefix is used to form a fully-qualified name. Matrix of data set descriptive information is returned.
Fully-qualified data set name enclosed in single quotation marks.	Matrix of data set descriptive information is returned.
Incomplete name (with or without single quotation marks)	List of qualified data set names beginning with the specified incomplete name. One data set per row.

Figure 27 on page 119 shows the 20 rows of descriptive information returned from the DSI command when specified with a partially- or fully-qualified data set name. The matrix is the same as that returned from the DDI built-in command. If a blank ddname is returned, the data set is not currently allocated to your user ID. If the volume serial number is MIGRAT, APL2 assumes that the Hierarchical Storage Manager (HSM) has migrated the data set, and no additional information is returned.

For offline devices, APL2 can return only the data set name. The device a data set resides on must be online in order for APL2 to find it. For MSS, use a TSO ALLOCATE statement to force the device online if possible. If the data set resides on MSS and is not allocated, MVS treats it as if the device is offline.

When the DSI command is entered with a *name* operand of one or more index levels, the number of rows in the returned matrix depends on the number of data sets associated with the specified level(s). Each data set is one row of the matrix.

If the output is too much for the program to handle, partial contents are output with the last row of the matrix being all dots.

The number of columns in the matrix depends on the size of the data set name. The variable information starts in column 10. To eliminate the display of the keywords, use:

```
0 9↓0 ¯1↓DATA
```

APL DEBUG [*code*]

The APL DEBUG built-in command enables you to dynamically query and set the debug options in effect for the session. Issued without an operand, the DEBUG built-in command returns the current setting of the debug option. To set the debug option, specify one of the codes discussed with the DEBUG invocation option in Chapter 2, “APL2 Invocation and Termination” on page 8. Unlike the DEBUG invocation option, only a single code can be specified. The specified code replaces the current DEBUG settings.

This command is retained for compatibility with previous releases. See also option 'DEBUG' defined in *APL2/370 Programming: Using the Supplied Routines*. OPTION 'DEBUG:' behaves more like the invocation option, and is compatible across environments.

APL LIB [*n*]

The APL LIB built-in command returns a character matrix of workspace names, found in the library, specified as *n*. Each workspace name is one row of the 8-column matrix.

If *n* is omitted, the workspace names in your private library are returned. If *n* is an invalid library number, the command returned is either *IMPROPER LIBRARY REFERENCE* or *INCORRECT COMMAND*.

APL LIBS

The APL LIBS built-in command returns a character matrix of both public and project library numbers.

- All currently *allocated* VSAM public and project library numbers are displayed.
- All currently defined SAM public and project library numbers, including empty project libraries, are displayed.

  Note: There must be enough free storage, defined by the FREESIZE invocation option, to invoke Access Method Services, which requires 250K bytes. If sufficient storage is not found, the return code is 3.

The column width of the returned matrix is eight characters, as shown in the following example:

```
      TSO100←'APL LIBS'
      □←LIBNOS←TSO100
00000001
00000002
00001980
```

APL NOMSG *commandname commandparms*

The APL NOMSG built-in command processes *commandname* as if APL NOMSG had not been specified, except that:

- Messages normally controlled by the CONTROL NOMSG command within a TSO CLIST are suppressed for the duration of the command. When the command is completed, message display is restored to its prior state.
- The command cannot be another built-in command.
- The command cannot be an ISPF EXEC. (But the ISPEXEC command can be used to invoke such an EXEC indirectly.)

APL PDSI *name*

The APL PDSI built-in command returns a character matrix containing descriptive information on the members of a partitioned data set entered as the name operand. The format of this information consists of one row for each member and 77 columns corresponding in format to the display provided by the ISPF browse facility. If AP 100 cannot allocate enough storage for the complete result, the number of columns is 8 and only member names are returned.

In the following examples, column headings are shown for orientation. They are not part of the character matrix returned by the command.

Libraries edited by ISPF appear as follows:

```
   NAME              VV.MM  CREATED     CHANGED      SIZE  INIT   MOD   ID
----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+--
CADD                 01:01 91/08/26 91/08/26 08:54    10    10     0 L460581
CENV                 01:02 91/02/19 91/08/26 08:32    21    18     0 L460581
CSUB                 01:02 91/08/26 91/10/01 15:14    10    10     0 L460581
```

Load module libraries appear as follows:

```
 NAME                SIZE   TTR   ALIAS-OF AC --------- ATTRIBUTES ---------
----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+--
CMEMBER            001288 00022E          00 FO
CRDRL              000408 00020A          00 FO NX          RN RU
CRL                001178 000214          00 FO             RN RU
```

APL QUIET

The APL QUIET built-in command enables you to suppress messages from APL2 until the terminal opens for input. The effect is the same as if the session manager SUPPRESS command had been issued.

The APL QUIET command is provided for compatibility with previous releases. New applications should use the more general OPTION 'QUIET' as defined in *APL2/370 Programming: Using the Supplied Routines*.

APL QUOTA

The APL QUOTA built-in command returns a 10-item vector of integers. The first six items contain the same information displayed by the APL2 system command)*QUOTA*; however, the first two items are 0 because LIB and FREE are not measurable in TSO. (See *APL2 Programming: Language Reference* for information on)*QUOTA*.) The last four items are:

SVMAX	Maximum permitted size of a shared variable
REGION	Region size of your TSO session
FREESIZE	Amount of free space available in your TSO region
AISIZE	The size of the alternate input stack

Except for the workspace size, these figures do not reflect extended size for MVS/XA systems.

APL USER

You can use the APL USER built-in command to obtain miscellaneous information about the TSO environment in which APL2 is running. The information returned is an 8-column character matrix. Rows of the matrix are described in Figure 29.

Figure 29 (Page 1 of 2). Data Returned from the AP 100 APL USER Built-In Command

Row	Example	Description
1	APLUSER	TSO user ID (7 characters)
2	1001	Workspace library number
3	UTILITY	Workspace name
4		(Reserved)
5	32791	Terminal type code
6	24 80	Terminal screen height and width (two 4-character fields)
7	APL2	APL2 module name
8	2.2.00	APL2 licensed program release level
9	TSO/E	Subsystem type
10	2.04.0	Subsystem (TSO) release level
11	MVS/SP4	Operating system name
12	SP4.3.0	System Product Name
13*	ACF/VTAM*	TP access method name
14*	2.0	TP access method level
15*	GDDM	Graphical Data Display Manager
16*	V3R1.0	GDDM level (can be obtained more reliably by using call 122 of AP 126)
17*	RACF	Resource Access Control Facility
18*	1.09.2	RACF level
19*	HSM	Hierarchical Storage Manager
20*	2.2.1	HSM level
21.0	B	If batch processing
21.1	.J	If user has job submit authority
21.2	..O	If user has operator authority
21.3	...V	V if VTAM*
21.4	....ABCD	Sysout classes available
22	REMOTE1	Sysout remote destination

Figure 29 (Page 2 of 2). Data Returned from the AP 100 APL USER Built-In Command

Row	Example	Description
23.0	3090*	Processor model number
23.4	....SYSA	SMF processor identification
24	APL4	TSO project number (account)
25	APLOGON	Logon procedure name
26	APLUSER	Profile prefix (from TSO PROFILE command)
The following four fields are used only with a SAM library system.		
27	SYSDA	UNIT name for data set allocations
28	APLUNIT	UNIT name for new workspaces
29	APLVOL	VOLSER for new workspaces
30*	999	Maximum public library number
31-33		Reserved for IBM use
34-35	*****	Information from user field in the TSO user attribute data set (UADS). The information contained in these rows is installation dependent. Consult your system administrator regarding the data content.
36-nn*	*****	Reserved for installation fields.

* For the rows marked with asterisks, the values shown in the **Example** column are supplied as defaults with the licensed program. The values can be changed by your installation when APL2 is installed.

APL WSID

The APL WSID built-in command can be used to return the active workspace identification. The result is a 3-row character matrix containing:

- Name of the active workspace (WSID).
- Date and time stamp when the workspace was last saved. The format of this information is set by the DATEFORM invocation option. For information on DATEFORM, see Chapter 2, “APL2 Invocation and Termination” on page 8.
- The TSO user ID that saved the workspace, if under a SAM library system. Under a VSAM library system, the third row is blank.

This command is used only to query the active workspace identification. It cannot be used to change the workspace identifier. If any nonblank characters follow the APL WSID command, the return code is 1.

APL WSNAME [*libno*] *wsid*

The data returned from the APL WSNAME built-in command depends on whether the identified library is in SAM or VSAM format.

- With a SAM library system, the APL WSNAME command returns the fully-qualified data set name of the specified workspace (*prefix.libno.workspace-name*) if the specified library was defined. If the library number specified does not exist, no information is returned.
- With a VSAM library system, the APL WSNAME command returns the fully-qualified data set name of the VSAM cluster (the APL2 library) containing the specified workspace.

If the specified workspace does not exist, the command indicates only the name that would be used if the workspace existed.

EXEC *clistname*

The EXEC built-in command invokes a command procedure (CLIST) specified by *clistname*. The name must conform to TSO data set naming conventions.

TSO *commandname*

This built-in command processes *commandname* much as if TSO had not been specified, but the following differences:

- The present tasklib concatenation (including LOADLIB) is searched for the command and passed as a tasklib to it. Without the TSO prefix only the TSO session joblib/steplib is used.
- The command is invoked nonauthorized. Authorized commands fail.
- Certain APL2 tasks can continue to run asynchronously while the command is processing.

Formally, TSO *commandname* attaches the command as a *subtask* of APL2; without the TSO prefix it would be attached using the TSOLINK facility.

Note: TSO *clistname* is an invalid command. However, TSO *%clistname*, and TSO EXEC *%clistname*, are valid commands. In earlier APL2 systems, such CLISTs were not processed until APL2 terminated. They now process immediately, but as if TSO had not been specified. See EXEC above.

Chapter 12. AP 101—Alternate Input (Stack) Processor

The alternate input (stack) processor (AP 101) is used to create a stack of programmable input to either APL2 or the host system (CMS or TSO). Commands to the host system are stacked for processing after a programmed AP 101 exit from the APL2 environment. Under CMS, you can also stack input to a CMS command or EXEC.

Conceptually, the alternate input stack is a character vector of vectors. Each item is one stacked input line of variable length. The value of the AISIZE invocation option (TSO only) determines the number of characters that can be stacked.

Although any one user can maintain only one stack, several shared variables can be used to add entries to the stack. Entries can be added in either first in, first out (FIFO) or last in, first out (LIFO) order. One variable can add FIFO entries while another adds LIFO entries.

An input stack is normally created within a defined function. The top entry (the first item) in the stack is used when the terminal opens for input. This occurs when:

- An APL2 statement prompts for user input (⍞ or ⎕).
- The APL2 line editor, invoked by `)EDITOR 1`, prompts for input.
- APL2 opens the keyboard for immediate execution.
- APL2 execution is suspended as a result of an error or stop control (`S∆`).

Under TSO, you can set a *fence* within the stack. Data behind the fence is not passed to APL2 until the keyboard is opened for immediate processing. Establishing a fence in the input stack can be useful in providing recovery from interrupts or errors from APL2 or TSO. Only one fence is allowed, and it is allowed only under TSO.

Associated Workspaces

The CMS workspace (under CMS) and the TSO workspace (under TSO) contain a cover function that can be used to generate user-defined functions that add stack entries. For more information about this function, type `DESCRIBE`, `HOW`, or `ABSTRACT` after loading these workspaces.

Shared Variable Overview

Figure 30 provides an overview for sharing variables with AP 101.

Figure 30. Shared Variable Overview for AP 101

SV Protocol	AP 101 Conventions
General	One variable. Each specification adds an entry to the stack (or, under TSO, sets a fence option). Each reference obtains the return code from the most recent stack operation.
Maximum Number of Shared Variables	14
Name	Any valid APL2 variable name.
Initial Value	Stacking order—FIFO (default) or LIFO.
Subsequent Values	Character vectors not exceeding a length of 256, or integer scalars or vectors. Exception: TSO fence requests are numeric scalars.
Data Types Supported	Character vectors or scalars.
Access Control	**CMS:** 0 0 0 1 **TSO:** 0 0 1 1

Data Formats

With the exception of the fence options available under TSO (see below), all specifications of the variables shared with AP 101 must be character vectors or character scalars.

Initial Values

You indicate the order in which lines are to be stacked by specifying an initial value in the variable before it is shared. Stack order can be FIFO (the default) or LIFO. For compatibility with previous versions of APL, FIFO and LIFO can be specified as END and BEG, respectively. Stacking order can also be changed later (see "AP 101 Commands" on page 129).

The format of the initial value is:

```
SHR101←'[([LIFO|FIFO][)]]'
```

If the shared variable has no initial value or contains an empty vector, the default is FIFO.

Input lines are always removed from the stack, starting at the top. When you specify FIFO stacking, new lines are added to the bottom of the stack. When you specify LIFO stacking, new lines are added to the top of the stack.

If the initial value is invalid, a return code of 1 is set and continues to be set for all subsequent requests until a valid initial value is provided.

Communication Procedure

The steps to build a stack of input are summarized below.

1. Initialize the variable to be shared with the type of stack order you want. You can omit this step if you accept the default of FIFO and the variable you are sharing is empty or has no value. See "Initial Values" on page 128.
2. Offer the variable to AP 101 and ensure that the degree of coupling is 2.
3. Set the access control.
4. Check the return code from the initial value.
5. Specify the shared variable with the data you want stacked or a numeric command as described in "AP 101 Commands." Check the return code. Repeat this step for each entry to the stack.

   Under TSO, you can set a fence in the stack. The optional fence is discussed in "Using AP 101 with the TSO Fence Option" on page 131.
6. Stack entries can be used any time after being stacked. There can be several cycles of alternately filling and using the stack.
7. Retract the shared variable when your stack processing is finished.

Although AP 101 is ordinarily used in a defined function, Figure 31 shows a sample APL2 session that illustrates the communication procedure. Note that the data assigned to *SHR*101 is immediately processed, because the terminal returns to input mode immediately after the specification.

```
      101 ⎕SVO 'SHR101'          ⍝ offer to share
2             <-----------------⍝ degree of coupling is ok
      1 0 1 0 ⎕SVC 'SHR101'      ⍝ set access control
1 0 1 1
      SHR101←')WSID'
CLEAR WS       <-----------------⍝ )WSID executed immediately
      SHR101←')SAVE TEST101'
1992-03-27 10.53.15 (GMT-8) <---⍝ )SAVE executed immediately
```

Figure 31. Sample APL2 Session to Communicate with AP 101

AP 101 Commands

In addition to assigning character vectors to the variable to add lines to the stack, numeric codes can be used to purge the stack of all entries or to control stacking order.

- *SHR*101←0 — Purges the entire stack.
- *SHR*101←10 — Queries the state of the stack (LIFO/FIFO)
- *SHR*101←10 1 — Sets the stack to LIFO

- *SHR*101←10 ¯1 — Sets the stack to FIFO (the default)

See also the additional codes described in “Using AP 101 with the TSO Fence Option” on page 131.

Using AP 101 within a Defined Function

AP 101 is most useful when used in a defined function. Otherwise, as illustrated in Figure 31, each row of stacked data is immediately processed as it is stacked. For a sample defined function that uses AP 101, see Figure 32 on page 130.

To try this sample function, replace *CMD* with a name you have already defined, and specify the command or EXEC you normally issue to invoke APL2.

```
     ∇
[0]    Z←QUEUE STRING              ⍝ utility function for
[1]    SHR101←STRING               ⍝ queue data
[2]    Z←SHR101
     ∇
     ∇
[0]    Z←CHK N
[1]    Z←(0≠↑N)/0                  ⍝ return empty if ok
[2]    →(0=⍴Z)/0                   ⍝ else return zero
[3]    'AP 101 ERROR:' N           ⍝ after printing message
     ∇
     ∇
[0]    CMD AP101SAMPLE INVOC;SHR101;Z
[1]    Z←101 ⎕SVO 'SHR101'         ⍝ offer to share
[2]    →(2≠101 ⎕SVO 'SHR101')/0    ⍝ if coupling not ok, exit
[3]    Z←0 0 1 1 ⎕SVC 'SHR101'     ⍝ set access control
[4]    →CHK SHR101                 ⍝ if rc not ok, then return
[5]    →CHK QUEUE ')CONTINUE'      ⍝ save cont. ws;return on error
[6]    →CHK QUEUE CMD              ⍝ execute cmd;return on error
[7]    →CHK QUEUE INVOC            ⍝ restart APL2; return on error
     ∇
```

Figure 32. Sample APL2 Defined Function That Uses AP 101

Disposition of Data on the Stack

When data is stacked by a defined function using AP 101, the top entry on the stack is treated as the response to an input request when the user's terminal opens for input. The entry is then deleted from the stack.

The entire stack is deleted if:

- An attention signal is issued; if the TSO fence option is in use, the stack is deleted up to the fence.
- A stack overflow condition occurs (TSO only). See return code 12 in Figure 33 on page 133.

Exiting and Returning to APL2 within a Defined Function

Whatever remains in the stack after an)*OFF* or)*CONTINUE*, is passed to the host system (CMS/TSO). If the stack contains the command to reenter the APL2 environment, any input stacked after the APL2 command is:

CMS Processed by APL2.

TSO Not passed to APL2. Use the INPUT invocation option (described next) to automatically run expressions upon return to APL2.

Using the INPUT Invocation Option in a Stacked APL2 Command

When the APL2 command is stacked to return to APL2 from the host system, you can include the INPUT invocation option in the command. With this option you can specify a string of input to be processed upon reentering APL2. For example:

```
101 ⎕SVO 'SHR101'
1 0 1 1 ⎕SVC 'SHR101'
SHR101←')OFF'
INP←' INPUT(''')LOAD MYWS''' '''RESTART''')'
SHR101←'APL2 WSSIZE(500K)',INP
```

When the stacked APL2 command is processed, the MYWS workspace is loaded and restarted.

The INPUT invocation option is also discussed in Chapter 2, “APL2 Invocation and Termination.”

Using AP 101 with the TSO Fence Option

To understand the purpose of the TSO fence option, you need to understand the difference between user-initiated input requests and system-initiated input requests. Data behind a fence is used only for APL2 system-initiated input requests. Data added after a fence was set (data in front of the fence) is used for either user-initiated or system-initiated input requests.

User-initiated input requests include prompts for user input (⎕ or ⍞) from a defined function.

System-initiated input requests are those issued by APL2 in immediate execution mode. When a fence is set, all data currently on the stack is placed behind the fence, regardless of how it was stacked (FIFO or LIFO). Entries behind the fence are not processed until the terminal returns to immediate execution mode.

Returning to immediate execution mode clears the fence, and the stacked data is passed to APL2 (one line per input request). For example, if two lines of data are stacked behind the fence and the first invokes a function, a ⎕ or ⍞ prompt from the function is satisfied by the second stacked line, unless the function first sets a new fence.

Returning to immediate processing mode can occur for any one of the following reasons:

- The line editor prompts for input.
- The function you are running terminates normally.

- A function has a stop control ($S\Delta$) set for a function line. (Note that stop control is not honored in a nonsuspendable function or in functions called from a nonsuspendable function.)
- An error occurs (such as a $VALUE\ ERROR$) during function processing.
- A terminal or line interrupt occurs.

Fence Commands

There are two additional commands you can specify in the shared variable under TSO. They are:

1—Set or move the fence.
2—Purge the stack up to the fence.

When you specify the shared variable with a 1, all current entries on the stack are placed behind the fence. If a fence already exists in the stack, the fence is moved to the top of the stack.

Command 2 deletes all entries on the stack except those behind the fence, assuming a fence was set. If no fence exists, all entries on the stack are deleted. A command of 2 can be used in a function that conditionally prompts for input.

Command 0 (see "AP 101 Commands" on page 129) deletes all entries on the stack—including those behind the fence. It can be used when you have set an error recovery trap behind the fence, but it did not have to be processed.

Note: Even if you set a numeric value in the shared variable (requests to set a fence or purge stack entries), you should check the return code. The stack is not cleared if a nonzero return code results. AP 101 may be indicating that the initial value supplied was invalid. All subsequent specifications of the shared variable are ignored until a valid initial value is supplied. No additional data can be stacked until you obtain a return code of 0.

Cautions

If an interrupt is signaled or if a system error occurs as the stack is being processed, the entire stack (including a TSO fence and the data following it) is deleted and the terminal returns to immediate execution mode.

Under CMS, the two CMS immediate commands HT (halt terminal output) and RT (resume terminal output) cannot be stacked for deferred execution. These commands are executed when you stack them.

Under TSO, you cannot use the DATA/ENDDATA statements of TSO CLISTS to supply APL2 input.

By selecting characters from $\Box AV$, you can generate characters that cannot normally be entered from a terminal. If you stack lines that include characters generated in this way and if the stacked lines are read by a program not prepared for these characters, unexpected results can occur. For a stack read by APL2, this includes characters other than the blank character or displayable APL2 characters.

Return Codes

After a variable is first shared and after each specification of the shared variable, a simple scalar is returned with the next reference of the variable. This return code indicates whether or not the last specification of the variable was successful. Return codes from AP 101 are listed and explained in Figure 33.

Figure 33. Return Codes from AP 101

Code	Description
0	Success. The initial value or the last user specification of the shared variable was successful.
1	Invalid initial value. Subsequent specifications of the variable are interpreted as attempts to assign a valid initial value. **Action:** Specify a valid initial value. Input is not stacked until a return code of 0 is returned from the initial value.
12	**TSO Only:** Stack overflow. You have specified a data value that was too large to fit into the remaining space on the stack. The stack is purged up to the fence if a fence is present; otherwise, the entire stack is purged. **Action:** If there is no program logic error on your part, restart APL2 with a larger value specified in the AISIZE option. For more information on the AISIZE operand and the default stack size for AP 101, see Chapter 2, “APL2 Invocation and Termination” on page 8.
444	Invalid value specified in the shared variable. It is the wrong size, shape, or data type. **TSO only**: The stack is purged up to the fence if a fence is present; otherwise, the entire stack is purged. **CMS only**: The invalid specification is not used.

Chapter 13. AP 102—Main Storage Access Processor

This chapter contains Product-Sensitive Programming Interface and Associated Guidance Information.

The main storage access processor is used by system programmers to obtain the contents of specified areas of main storage. It is used primarily as a tool for monitoring the systems environment in which APL2 runs.

The storage accessed with this processor can only be read, not updated. Storage areas that can be accessed include:

- **CMS:** Any part of the user's virtual machine in addition to any currently-loaded shared segments.
- **TSO:** All addressable storage in the user's storage protect key. Under MVS/XA, this includes storage addresses above 16 megabytes. A storage block that is fetch-protected cannot be accessed.

There are no significant differences between CMS and TSO in the use of this processor.

Shared Variable Overview

Figure 34 provides an overview for sharing variables with AP 102.

Figure 34. Shared Variable Overview for AP 102

SV Protocol	AP 102 Conventions
General	Two variables—control and data. Each specification of the control variable passes a service request to the auxiliary processor; each reference obtains the return code from the most-recent request. Any specification of the data variable is ignored. Each reference, however, returns the result of the request specified in the control variable.
Maximum Number of Shared Variables	Six pairs.
Names	Must start with *CTL* and *DAT*. Suffixes pair the variables. Names cannot exceed 12 characters.
Initial Values	Same as subsequent values. If a CTL initial value is provided, DAT should be offered before CTL; otherwise, the return code is 2.
Subsequent Values	*CTL*: Specify a service request (scalar integer or one- to three-item vector of integers). Reference a return code (scalar integer). *DAT*: Reference a vector of one or more integers.
Data Types Supported	Nonnegative integers.
Access Control	*CTL*: 0 0 0 1 *DAT*: 0 0 0 0

Commands

Two types of information can be returned in the data variable:

- Address of the active workspace

  Specify the control variable as a scalar integer (or a one-item integer vector) with a value of 0.

- Contents of a specified area of primary storage

  Specify the control variable as a two- or three-item vector:

  Item 1 = 1 (Service request: required.)

  Item 2 = *address* (Displacement, in decimal, from storage location 0: required.)

  Item 3 = *length* (Number of bytes to be returned; optional—default is four bytes.) The processor rounds the specification up to a multiple of four.

Figure 35 summarizes the formats for the control and data variables.

Figure 35. AP 102 Commands

Request	*CTL* **Specification**	**Result of** *DAT* **Reference**
Obtain address of active workspace.	*CTL*`102←0`	One-item integer vector.
Obtain contents of storage.	*CTL*`102←1` *address* [*length*]	Vector of integers; each item represents four bytes of storage as an integer between ¯2147483648 and 2147483647.

Communication Procedure

Figure 36 on page 136 shows a sample APL2 session that uses AP 102 to obtain the SHRSIZE and WSSIZE limits for the session. The PERTERM address accessed in the figure is an internal control block for your terminal. Mapping macros for it are provided with the product, but the offsets within it are not guaranteed for future releases.

```
      102 ⎕SVO¨'CTL102' 'DAT102'    ⍝ offer to share
2 2
      1 0 1 0 ⎕SVC 'CTL102'         ⍝ set access control
1 0 1 1
      CTL102←0                      ⍝ request workspace address
      CTL102
0                          ←──────Return code OK
      GWS←DAT102                    ⍝ save workspace address
      CTL102←1,GWS+4                ⍝ request PERTERM address
      CTL102
0                          ←──────Return code OK
      PTH←DAT102                    ⍝ save PERTERM address
      CTL102←1,(PTH+20),16          ⍝ request SHRSIZE and WSSIZE
      CTL102
0
      DAT102
32768 0 0 1048576        ←──────SHRSIZE, [unused], [unused], WSSIZE
      ⎕SVR ¨ 'CTL102' 'DAT102'      ⍝ retract shared variables
2 2
```

Figure 36. Sample APL2 Session to Communicate with AP 102

Formatting the Result from AP 102

The value returned in the data variable is always a vector of integers. This is useful for following storage chains, but, if the data is character, you may want to convert it to character or hexadecimal format. Use the following expression to convert the data variable to character format:

```
⎕AF,⍉(4⍴256)⊤DAT102
```

The *UTILITY* workspace contains the function *HEXDUMP*, which can be used to format the result returned in the data variable into hexadecimal format.

Cautions

APL2 control blocks and workspace formats may be changed, without notice, by program temporary fixes (PTFs) or future releases of APL2. Consequently, applications that use AP 102 to access information from these areas should be designed in such a way that they can easily be modified if the internal structure of APL2 changes.

Return Codes

The return code issued by AP 102 is a scalar integer. Figure 37 describes the AP 102 return codes.

Figure 37. Return Codes from AP 102

Code	Description
0	Successful completion of requested function.
2	The request specified in the control variable cannot be satisfied because you did not offer a data variable. The AP offered the data variable to you, but you did not match that offer. **Action:** Offer the data variable, ensure the degree of coupling is 2, and respecify the request.
3	Invalid service request. The first item specified in the control variable was not a 0 or 1.
4	Invalid number of parameters for specified service request.
5	Invalid APL2 data specified in the control variable. It is the wrong size, shape, data type, or outside the range -2^{31} to $2^{31}-1$.
6	Invalid address specified in the control variable. The value is negative or represents an address that is outside the range of accessible storage.
7	Invalid length specified in the control variable. The value is negative or zero.
8	Insufficient storage available for AP 102 to return the contents of storage in the data variable. **Action:** Either specify a smaller length in the control variable or restart APL2 with a larger size specified in either the SHRSIZE or FREESIZE option.

Chapter 14. AP 110—CMS File Processor

AP 110, the CMS file processor, is used to sequentially or directly access records on a CMS file. These records must be on a disk that is under the control of the CMS file system. CMS files can be created, updated, or merely read with AP 110.

AP 110 is not available under TSO. BDAM files, however, can be similarly processed under TSO using AP 210, the BDAM file processor.

Associated Workspace

The CMS workspace contains several cover functions that can be used to access CMS files with AP 110. For information on the functions and how to use them, type *DESCRIBE*, *HOW*, or *ABSTRACT* after loading the CMS workspace.

Shared Variable Overview

Figure 38 provides an overview for sharing variables with AP 110.

Figure 38. Shared Variable Overview for AP 110

SV Protocol	AP 110 Conventions
General	One or two variables. **Sequential Access:** Only a record variable is required. Each specification writes a record; each reference reads a record. **Direct Access:** A control variable and a data variable are required. • Each specification of the control variable sets the read/write pointer and/or the number of records to be read/written; each reference obtains return codes from the read or write operation. • Each specification of the record variable writes the record(s) identified in the control variable. Each reference reads the record(s) requested in the control variable.
Maximum Number of Shared Variables	40
Names	Any valid APL2 variable names not exceeding a length of 77 characters.
Initial Value	The name of the file you want to access must be specified in the initial value of both variables. Matching file names pair the control and record variables. Optionally, you can specify: • Fixed-length file • Conversion option • Type of access
Subsequent Values	**Record:** Depends on conversion option **Control:** One- to four-item vector of integers
Data Types Supported	Any valid type for the conversion option specified
Access Control	**Record:** 0 0 1 1 **Control:** 0 0 1 1

Initial Values

The format of the initial value for the record and control variables is shown below. The *fileid* is required for both variables. The option *CTL* is required for the control variable.

If the initial value is invalid (return code other than 0), subsequent specifications of the variable are treated as attempts to assign a valid initial value.

Initial values are specified as follows:

```
REC110←'fileid ([FIX][access][conversion][])]

CTL110←'fileid ( CTL[ )]'
```

where:

fileid Specifies the file name, file type, and file mode of the CMS file to be accessed. The default file mode is A1. A file mode specified as an asterisk (*), indicating that the file mode can be anything, can be used only for an existing file.

In general, it is a good idea to explicitly state the file type. It defaults to VMAPLCF for the record variable (where **'c'** represents the conversion option), but it always defaults to VMAPLVF for the control variable. Therefore, it is essential that the file types match when specifying a conversion option. See the *conversion* parameter described below.

AP 110 converts the fileid with the 370 option. The fileid can consist of characters that convert to uppercase and lowercase letters, numbers, the at sign (@), number sign (#), and dollar sign ($). If you want to access a file whose converted fileid contains lowercase letters, you must specify underbarred letters (such as A) for the lowercase letters.

FIX Indicates that the new file being created is to contain fixed-length records; the record length is determined from the first specification of the record variable after its initial value is accepted. All subsequently written records must be this length.

If the *FIX* parameter is omitted, the file is created with variable-length records. If the file already exists, the existing record format is used, and this parameter is ignored.

access Indicates the type of access to be associated with the file. Valid values of the access options are:

U The file can be read or written. This is the default.

R The file can only be read. Any specification of the record variable (except for its initial value) is ignored unless you set the access control vector to control your specifications, in which case your terminal is locked when you specify the record variable.

W The file can only be written.

conversion Indicates the conversion to be applied to the processed data. (Valid conversion options are listed below.)

If no conversion option is specified and if no file type is entered for the file name, *conversion* defaults to *VAR* and the file type defaults to VMAPLVF.

The sixth position of the file type of the format VMAPLCF relates to the conversion option as follows:

- If the file type is specified in the VMAPLCF format, the conversion option defaults to the matching option specified in the sixth position (c) of the file type.
- If a conversion option but not file type is specified, the file type defaults to VMAPLCF, with the sixth position being the matching code for the conversion option in the table below.

The conversion options and matching default file type (sixth position) are:

APL	'*A*'
BIT	'*B*'
BYTE	'*Y*'
CDR	'*C*'
*COD*1	'*X*'
DBCS	'*D*'
EBCD (or 192)	'1'
VAR	'*V*'
BCD (or 370)	'3'

This auxiliary processor provides no conversion assistance in accessing files with character encoding other than EBCDIC or APL (either APL2 or VS APL) internal code. Applications that use the *BYTE* option to translate other encodings into APL internal characters should continue to do so. However, these applications must be modified to run under APL2.

For a description of all conversion options, see Appendix D, “Auxiliary Processor Conversion Options” on page 370.

CTL Indicates that this variable is a control variable. If the *CTL* parameter is omitted, the variable is assumed to be a record variable.

Communication Procedure

CMS files can be processed with AP 110, using either sequential or direct access. Only the record variable is required to process a file sequentially, but, with only one variable, you cannot check the return code of read/write operations and can read or write only one record at a time.

When you share a pair of variables with AP 110, you can process a CMS file by using direct access, check the return codes, and read or write multiple records for each request.

When a pair of variables is shared with AP 110, the record variable must be offered before the control variable.

Record Variable

The record variable is used to transmit the data content of records between the active workspace and a CMS file. The record variable is always required.

The first reference of the variable after sharing returns the return code for the initial value. Subsequent references obtain records from the file if the access option specified in the initial value was U (update—read or write) or R (read only). Each specification of the record variable after the initial value writes one or more records on the file.

The value of the control variable determines which and how many records are read, and which and how many records are written. If the control variable is not used, processing is done sequentially, one record at a time.

Control Variable

When shared, the control variable is a one- to four-item vector of integers used to control I/O operations. The four items are:

1. *CTL*110[1]—Return code of the previous I/O operation. (See "Return Codes" on page 144.)
2. *CTL*110[2]—Next record to be read (initial default = 1).
3. *CTL*110[3]—Next record to be written (initial default = end of file + 1).
4. *CTL*110[4]—Number of records to be read or written (blocking factor; initial default = 1).

Specifying the Control Variable

You have the option of specifying the control variable before each input or output operation.

- If you specify a scalar or a one-item vector, the read pointer is reset.
- If you specify a two-item vector, the read and write pointers are reset.
- If you specify a three-item vector, the read pointer, the write pointer, and the blocking factor are reset.
- If you specify a four-item vector, the first item is ignored, and the last three items reset the read and write pointers and the blocking factor.

If you specify a 0 for any of the items in the control variable vector, the respective item is left unchanged.

An indexed specification of the control variable can also be used to change one or more of the values of the control vector (*CTL*110[*I*]←value). Considerations for specifying the read pointer, the write pointer, and the blocking factor are given below. Note, however, that:

- If you specify a 0 or a negative number as any item, the corresponding value in the control variable remains unchanged.
- If you specify an unacceptable value (that is, a noninteger, a vector with more than four elements, or a matrix), the entire control variable remains unchanged.
- Specifying the return code (*CTL*110[1]) has no effect.

Specifying the Read Pointer (*CTL*110[2]): The read pointer can be set to any positive integer. If it exceeds the number of records in the file, an end-of-file (EOF) condition occurs on the next input (read) operation (return code 12).

Specifying the Write Pointer (*CTL*110[3]): If the write pointer is set to the number of an existing record, that record is replaced by the next output (write) operation.

If the write pointer is set to one more than the number of existing records, the next output (write) operation appends one or more records to the end of the file.

If the write pointer is set to a value greater than end of file (EOF) + 1 and the file contains fixed-length records, the next output (write) appends empty records to fill the file until the specified output record is written to the file.

If the write pointer is set beyond EOF+1 and the file contains variable-length records, an error occurs (return code 7) on the next output (write) operation.

Specifying the Blocking Factor (*CTL*110[4]): The blocking factor is always 1 (and any change is ignored) when:

The file contains variable-length records.

Conversion option *CDR*, *VAR*, or *APL* is specified.

When records are fixed length, and the conversion option is not *CDR*, *VAR*, or *APL*, you can decrease the time it takes to access a file by increasing the blocking factor.

If the blocking factor is set to a value greater than 1, that number of records is read as a matrix; each row is one record.

If the blocking factor is set greater than 1 and the data to be written is a matrix, each row is written as one record; the number of rows in the output matrix cannot exceed the blocking factor.

If you specify a blocking factor too large for the processor's buffer, it is ignored. The acceptance of a large blocking factor by AP 110 does not ensure that a sufficient amount of shared storage is available when the record variable is actually transmitted.

Figure 39 on page 143 shows how to create a CMS file to store APL2 objects. This file can later be read or updated, using either sequential or direct access.

```
      REC110←'AP110FIL VMAPLCF (FIX CDR)'  ⍝ initialize REC var.
      CTL110←'AP110FIL VMAPLCF (CTL)'      ⍝ initialize CTL var.
      110 ⎕SVO¨'REC110' 'CTL110' ⍝ offer record variable first
2 2                              ←──────────Degree of coupling OK
      (⊂1 0 1 0) ⎕SVC ¨ 'REC110' 'CTL110'  ⍝ set access control
 1 0 1 1  1 0 1 1
      REC110                        ⍝ check return codes
0 1 1 1                          ←──────────Pointers set to 1st record
      CTL110
0
      REC110←80↑'THE FIRST RECORD ESTABLISHES MAX RECORD LENGTH'
      CTL110
0 1 2 1                          ←──────────Write pointer set to 2nd
      REC110←1 2 3                  ⍝ specify a numeric vector
      CTL110
0 1 3 1                          ←──────────Return code OK
      REC110←90⍴'ABC'               ⍝ specify too long a vector
      CTL110
15 1 3 1                         ←──────────Incorrect length code
      REC110←2 3⍴⍳6                 ⍝ specify a matrix
      CTL110
0 1 4 1                          ←──────────Return code OK
      REC110                        ⍝ read first record
THE FIRST RECORD ESTABLISHES MAX RECORD LENGTH

      CTL110                        ⍝ check return code and pointers
0 2 4 1                          ←──────────Read pointer at 2nd record
      REC110                        ⍝ read second record
1 2 3                            ←──────────Second record
      CTL110                        ⍝ check return code
0 3 4 1
      REC110                        ⍝ read third record
1 2 3
4 5 6
      CTL110                        ⍝ check return code
0 4 4 1
      CTL110[3]←2                   ⍝ prepare to update 2nd record
      CTL110                        ⍝ display read/write pointers
0 4 2 1
      REC110←'REPLACED SECOND RECORD'
      CTL110                        ⍝ check return code
0 4 3 1
      REC110                        ⍝ read next sequential record
                                 ←──────────Empty vector returned
      CTL110
12 4 3 1                         ←──────────End of file confirmed
```

Figure 39. Example of Use of AP 110 to Create a CMS File

Cautions

If you replace a variable-length record with a record of different length, the remainder of the file is deleted.

VM/SP does not normally rewrite a disk directory until all files on that disk are closed. For APL2, this may not occur until you end your APL2 session. APL2 attempts to force directory rewrites whenever an output file is closed, but it cannot guarantee that the directory is rewritten. If APL2 does not terminate normally, all data written during the session could be lost.

Return Codes

Return Codes from Initial Values: If the initial value of the record variable is invalid, a code of 1 is returned in the record variable. Otherwise, if the initial value is valid, the record variable contains a four-item vector whose first item is 0.

If the initial value of the control variable is invalid, a code of 1 is returned in the control variable. Otherwise, the control variable contains a 0.

If you reference the control variable again, AP 110 returns a 4-element vector as described below.

Return Codes from I/O Operations: After each read or write operation, the control variable, if any, contains a four-item vector, with the first item containing the return code of the most recent I/O operation. If no control variable is shared with this processor, it is not possible to check the return codes from I/O operations.

Figure 40 lists the return codes that can be returned using AP 110. Codes 0, 1, and 443 through 445 are returned by the auxiliary processor. All other codes are returned by the CMS FSREAD and FSWRITE macros. The codes listed in Figure 40 are some of the more frequently issued codes. For a complete list, see *VM/SP: CMS Command and Macro Reference*. For CMS limits exceeded when code 6, 10, 17, or 19 is returned, see the appropriate user's guide for your system.

Figure 40 (Page 1 of 2). Return Codes Using AP 110

Code	Description
0	No error exists
1	Attempt to read a nonexistent file, or improper initial value
3	Permanent I/O error
4	First character of file mode is invalid
5	Attempt to read more records than the maximum allowed by CMS
6	Attempt to write too many records in a CMS file
7	Attempt to write past the end of a variable-length file
8	Attempt to read a record with incorrect record length
10	Attempt to create a file when you already have the maximum allowed by CMS
12	End-of-file read or attempt to write on a read-only disk
13	Attempt to write on a full disk
14	Attempt to write on an unformatted disk
15	Attempt to write a record with incorrect length into a file with fixed format
17	Attempt to write a record that is too large into a variable-length file
19	Attempt to write in a file already containing as many data blocks as CMS allows
22	Virtual storage capacity exceeded
25	Insufficient storage for CMS file system
26	Item number is invalid
27	Attempt to replace a variable-length record with one of different length

Figure 40 (Page 2 of 2). Return Codes Using AP 110

Code	Description
443	Insufficient free storage for input/output buffers **Note:** The amount of free storage allocated is based on the specified shared storage size. **Action:** Restart APL2, increasing the value of the SHRSIZE invocation option. If the resultant workspace is too small, restart CMS with more virtual storage.
444	The value assigned to the shared variable is invalid. It is the wrong shape, size, or data type
445	Attempt to read a record larger than shared storage. If the blocking factor was greater than 1, then check that enough shared storage is available to handle the number of records read or written as a matrix. **Action:** Try again with a reduced blocking factor value or restart APL2 and increase the value of the SHRSIZE invocation option. If the resultant workspace is too small, restart CMS with more virtual storage.

Chapter 15. AP 111—QSAM File Processor

The QSAM file processor, AP 111, is used to process sequential data files on I/O devices supported by the operating system. These devices include printers, punches, readers, disks, and tapes. Before using this processor, the file must be allocated with either ALLOCATE or FILEDEF. The definition or allocation can be done at any time, provided it is done before the shared variables for the file are offered to AP 111.

Associated Workspaces

The TSO workspace contains cover functions that can be used to access QSAM files from within an APL2 session. Type *DESCRIBE*, *HOW*, or *ABSTRACT* after loading this workspace for information on the functions and how to use them.

Shared Variable Overview

Figure 41 provides an overview for sharing variables with AP 111.

Figure 41. Shared Variable Overview for AP 111

SV Protocol	AP 111 Conventions
General	One or two variables. Control variable is optional. Any specification is ignored; each reference obtains the return code from the most recent read/write operation. Record variable is required to pass the content of data records. Each specification writes a record; each reference reads a record.
Maximum Number of Shared Variables	14
Names	Any valid APL2 variable names not exceeding a length of 77 characters.
Initial Value	The ddname of the file you defined or allocated for access must be specified in the initial value of both variables. Matching ddnames pair the control and record variables. Optionally, you can specify a conversion option.
Subsequent Values	**Record:** Depends on the conversion option. See Chapter 18, “AP 121—APL2 Data File Processor” on page 182 as well as Appendix D, “Auxiliary Processor Conversion Options” on page 370 for more information. **Control:** Reference a scalar integer.
Data Types Supported	Record variable: Any valid type for the conversion option specified. The default is VAR.
Access Control	**Record:** 0 0 1 1 **Control:** 0 0 1 1

Initial Values

The initial values for the control and record variables must include the ddname (or file name) specified in the FILEDEF (CMS) or ALLOCATE (TSO) command. The ddname pairs the two variables.

In the record variable, you can, optionally, specify a conversion option. Under TSO, you can also specify the type of access allowed on the file (read only, write only, or both).

The initial value of the control variable, if used, must contain the letters *CTL*.

The formats for the initial values of the record and control variables are:

```
REC111←'ddname ([access][conversion][])'
CTL111←'ddname (CTL[ )]'
```

ddname Is the ddname of the file or device to be accessed. It must be the ddname specified in a FILEDEF command already issued to CMS, or the ddname (FILE name) specified in an ALLOCATE command already issued to TSO.

access Indicates the type of processing you want to do.

Note that this option is only available under TSO.

Valid values of the access option are:

R The data set can only be read. Any specification of the data variable or an attempt to open the data set for output results in a return code of 440.

W The data set can only be written. Any reference of the data variable or an attempt to open the data set for input results in a return code of 441.

U The data set can be either read or written. If no access option is specified, this is the default.

conversion Indicates the conversion to be applied to the file data. Valid conversion options with this processor are:

APL
BCD (or 370)
BIT
BYTE
CDR
*COD*1
DBCS
EBCD (or 192)
TN (TSO only)
VAR (default)

For a description of conversion options, see Appendix D, “Auxiliary Processor Conversion Options” on page 370.

CTL Specified only for the control variable. If *CTL* is not specified, the offered variable is assumed to be a record variable.

The control variable is paired with the most recently offered record variable whose ddname matches that specified in the control variable.

Any conversion or access option specified in the control variable is ignored.

Communication Procedure

The procedure for communicating with AP 111 to process a sequential file is:

1. Define (with a FILEDEF command in CMS) or allocate (with an ALLOCATE command in TSO) the file you want to access. AP 100 or the)*HOST* command can be used to do this from within your APL2 session.

   If the file contains variable length records, you must specify a maximum record length and/or block size in the CMS FILEDEF or TSO ALLOCATE command.
2. Initialize the variable(s) you intend to share (see “Initial Values” on page 147).
3. Offer the variables to AP 111 and ensure that the degree of coupling is 2.

   If you are sharing both control and record variables, offer the record variable before the control variable.
4. Set the access control.
5. Check the return codes from the initial values of the control and record variables.
6. Process the file.

After checking the return codes as described in step 5, the record variable can be referenced or specified.

At any given time, the file can be open for input or output depending on whether the record variable is referenced or specified.

The first reference of the record variable causes the file to be opened for input and its first record to be returned. Each subsequent reference of the record variable returns the next sequential record read from the file.

The first specification of the record variable causes the file to be opened for output. Unless a disposition of MOD was specified in the FILEDEF or ALLOCATE command, specification of the record variable causes records to be written sequentially to the file beginning with the first record in the file. If MOD was specified as the disposition, specification of the record variable causes records to be written sequentially after the last existing record in the file.

Access is changed from input to output when the record variable is first specified after it has been referenced. Access is changed from output to input when the record variable is first referenced after it has been specified. When access is changed with one of these operations, the file is first closed and then reopened for input or output as appropriate. When the file is reopened in this way, read and write operations begin with the first record in the file, unless MOD was specified, in which case, write operations cause records to be written sequentially after the last record in the file.

If CDR or VAR was specified as the conversion option for the file, each logical record in the file contains a single APL array (of any shape or rank within the constraints of the logical record size for the file). For files with fixed-length records, (RECFM = F or FB), arrays whose size is less than the logical record length of the file are padded transparently to match the record length when the record variable is specified. More information on this subject is provided in Appendix D, “Auxiliary Processor Conversion Options” on page 370 under the descriptions of the CDR and VAR options. For conversion options other than CDR or VAR, no padding of short records occurs (except with option BIT where records are padded with binary zeros to a byte boundary); if the file has RECFM F or FB, records written must match the file's logical record length in size.

If a conversion option other than CDR, VAR, or BIT was specified, each logical record in the file contains a character vector. For conversion option BIT, each logical record in the file contains a bit vector.

When the record variable is referenced, a single logical record is read from the file and returned. For conversion options other than CDR, VAR, or BIT, a character vector representing that record is returned. For conversion option BIT, a bit vector representing that record is returned. For conversion options CDR or VAR, an APL array is returned.

When the record variable is specified, one or more logical records are written sequentially to the file. For conversion options CDR or VAR, specification of the record variable causes a single logical record to be written to the file. For a conversion option other than CDR or VAR, specification of the record variable with a scalar or vector causes a single logical record to be written to the file. Specification of The record variable with a matrix causes each row of the matrix to be written as a single logical record in the file. Note that the file's blocking factor (typically BLKSIZE divided by LRECL) does not limit the number of rows in a matrix that can be written with a single specification of the record variable; more than one block can be written when a matrix is specified.

- To obtain the return code from a read or write, reference the control variable after the reference or specification of the record variable.
- To close the file, retract the record variable first. Check the return code in the control variable.

Figure 42 on page 150 shows a sample APL2 session that creates a fixed-length file with AP 111. It includes the use of AP 100 to issue a FILEDEF command to CMS for the file being created. (Under TSO, the ATTRIB and ALLOCATE commands would be issued instead of the FILEDEF command.) The file name, type, and mode are AP 111FIL TEST A1. The ddname is QFILE. The file contains 60-byte fixed-length records.

```
      100 ⎕SVO 'CMS100'      ⍝ offer to share with AP 100
2
      1 0 1 0 ⎕SVC 'CMS100'  ⍝ set access control
1 0 1 1
      CMS100←'FILEDEF QFILE DISK AP111FIL TEST (RECFM F LRECL 60)'
      CMS100                 ⍝ check return code from FILEDEF
0
      REC111←'QFILE (EBCD)'  ⍝ initialize record variable
      CTL111←'QFILE (CTL)'   ⍝ initialize control variable
      111 ⎕SVO ¨ 'REC111' 'CTL111'  ⍝ offer to share with AP 111
2 2
      (⊂0 0 1 1) ⎕SVC ¨ 'REC111' 'CTL111' ⍝ set access control
 0 0 1 1  0 0 1 1
      REC111                 ⍝ check return codes
0
      CTL111
0
      REC111←60↑'FIRST RECORD' ⍝ write first record
      CTL111                 ⍝ check return code from 1st write
0
      REC111←60⍴'HELLO '     ⍝ write second record
      CTL111                 ⍝ check return code from 2nd write
0
      REC111                 ⍝ implicit close; reopen for read
FIRST RECORD
      REC111                 ⍝ continue reading
HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO HELLO
      REC111
                      ←──────────Empty vector returned
      CTL111
12                    ←──────────End of file confirmed
      REC111←60↑'NEW FIRST REC'  ⍝ implicit close; reopen for
      CTL111                 ⍝ write from the beginning
0
      REC111                 ⍝ implicit close; reopen for read
NEW FIRST REC
      REC111
                      ←──────────Original 2nd record gone; empty vector returned
      CTL111
12                    ←──────────End of file confirmed
      ⎕SVR 'REC111'          ⍝ retract to explicitly close file
2
      CTL111                 ⍝ check return code
0
```

Figure 42. Sample Creation of QSAM Disk File Using AP 111

Cautions

CMS: CMS does not normally rewrite a disk directory until all files on that disk are closed. For APL2, this may not occur until you end your APL2 session. APL2 attempts to force directory rewrites whenever an output file is closed, but there is no guarantee that the directory is rewritten. Therefore, if APL2 does not terminate normally, all data written during the session could be lost.

AP 111 files must be closed before the file definition is cleared. Otherwise, you may lose your APL2 session, lose the file, and have to reinitialize CMS. Unless you are sure that the file is closed, do not issue a FILEDEF *ddname* CLEAR command from within your active workspace.

When AP 111 is active you must be careful about the CMS commands you specify. CMS QSAM simulation uses GETMAIN storage for buffers, and many CMS commands release all GETMAIN storage.

Files containing variable-spanned records (RECFM=VBS or VS) cannot be read or written.

TSO: AP 111 files must be closed (by implicit or explicit retraction of the record variable) before the file allocation is freed. Otherwise, you may lose your APL2 session and your file. Unless you are sure that the file is closed, do not issue the FREE FILE(*ddname*) command from within your active workspace.

Files containing variable-spanned records (RECFM=VBS or VS) can be read but not written.

Return Codes

Return Codes from Initial Values: If the initial value of the record or control variable is invalid, a code of 1 is returned in the first reference of the variable. Otherwise, a 0 is returned.

When the return code from the initial value is a 1, you must retract the variables, specify a valid syntax for the initial value and reoffer the variables.

A return code of 0 on the initial offer means that the syntax of the initial value was correct. No check is made that the specified ddname is correctly defined or allocated.

Return Codes from I/O Operations: When you reference or specify the data variable, the control variable contains the return code associated with the QSAM data set GET/PUT operation. When you retract the data variable, the control variable contains the return code associated with the QSAM data set CLOSE operation.

Figure 43 lists the return codes returned by AP 111. For other return codes you can receive, see “Undiagnosed Errors” on page 152.

Figure 43 (Page 1 of 2). AP 111 Return Codes

Code	Description
0	The initial value in the shared variable offer was accepted; or successful completion of the requested I/O operation.
1	The control variable was specified and no matching data variable could be found, or one of the two variables contains an invalid initial value: • No initial value was specified. • The value is not a character vector. • The syntax is incorrect. **Action:** Retract the variables offered, specify a valid syntax for the initial values, and reoffer the variables.
12	End of file. The value in the record variable is empty. The last record from the data set was passed in the previous read request. **Action:** To close the data set, retract the record variable.
15	Wrong length record on fixed-length output. When RECFM=F or FB, all records must be exactly equal to the record length specified in the FILEDEF or ALLOCATE command.

Figure 43 (Page 2 of 2). AP 111 Return Codes

Code	Description
17	Record too large for output. This error occurs when a record to be written exceeds the LRECL specified for a data set with variable or undefined record lengths (RECFM=U, V, or VB).
440	Error in open for output. This error occurs under the following conditions: • **CMS or TSO:** The open routine for the data set has failed. System error messages accompany this return code. It usually means you did not properly define or allocate the data set. • **TSO Only:** The R access option was specified in the initial value of the shared variable and you specified the data variable, indicating output (write) processing. • **TSO Only:** Your installation prohibited your access to the data set. Additional MVS error messages are displayed. **Action:** Verify your allocation, initial values, and authorization. Verify the data set attributes on direct access by using the CMS LISTFILE or the TSO LISTDS command. Verify the options you have selected, using the CMS FILEDEF or TSO ATTRIB command to see whether there is a conflict.
441	Error in open for input. This error can occur in the following situations: • You did not properly allocate the data set using the appropriate options. • The data set was opened for output, and you referenced the data variable. CLOSE processing failed. • **TSO Only:** Your installation prohibited your access to the data set with security mechanisms. **Action:** Verify your allocation, initial values, and authorization. Compare the data set attributes on direct access by using the CMS LISTFILE or TSO LISTDS command with the options you selected in the CMS FILEDEF or TSO ATTRIB command to see whether there is conflict.
443	Insufficient space to process output data. **Action:** Try reinvoking APL2, increasing the value of the FREESIZE invocation option.
444	Invalid data type or shape. There is a conflict between the conversion option you selected and the APL value you specified in the record variable.
445	Insufficient shared storage for input data. **Action:** Reinvoke APL2 and specify a larger value in the SHRSIZE invocation option. **Note:** If you are using the *VAR* or *CDR* conversion option and you think the SHRSIZE invocation option is set properly, verify that the record you are trying to read has a valid *APL* internal form. Functions are available in the *UTILITY* workspace to read a record from your data set (specified with the *BYTE* conversion option) to verify the *APL* internal format.

Undiagnosed Errors

If a return code other than any of those listed in Figure 43 is returned, it is a decimal value that represents either an ABEND code or sense and status bytes returned by QSAM as a result of an I/O error. When sense and status are returned, a)*MORE* message is also queued that contains additional information provided by the operating system.

The decimal value can be converted to its 4-byte hexadecimal representation in 0-origin by the following expression:

```
'0123456789ABCDEF'[⎕IO + 2 4⍴(8⍴16)⊤CTL]
```

If the second row of the resulting matrix is `0`, the result is an ABEND code. If the second row is nonzero, the first row contains status bytes and the second row contains sense bytes. For sense and status byte information, see *MVS/DFP: Using Data Sets* or the status information in the macro instruction manual appropriate to your system:

- *OS/VS Data Management Macro Instructions*.
- *MVS/XA Data Administration: Macro Instruction Reference*
- *MVS/ESA Data Administration: Macro Instruction Reference*

Chapter 16. AP 119—Socket Interface Processor

The socket interface processor, AP 119, is used to pass requests to the Transmission Control Protocol/Internet Protocol (TCP/IP) product. TCP/IP provides communication facilities across networks.

AP 119 also provides commands to control how APL2's cross-system shared variable interface uses TCP/IP.

Shared Variable Overview

Figure 44 provides an overview for sharing variables with auxiliary Processor 119.

Figure 44. Shared Variable Overview for AP 119

SV Protocol	AP 119 Conventions
General	One variable for commands and responses
Maximum number of shared variables	128 (includes cross-system shared variables)
Name	No restrictions
Initial Values	None. Initial values are ignored by AP 119.
Subsequent values	Specify a command, reference a 3 element array: AP return code, subsystem return code, and data
Access Control	`0 0 1 1`

AP 119 supports access to up to 64 sockets through a single shared variable.

The APL2 Socket Application Program Interface

The APL2 Socket API gives an APL application access to applications on other computers on a network using TCP/IP. The calls used in the API are similar to the C socket interface calls.

A *socket* is an endpoint for communication. It is represented by a socket number (usually 3 or greater). A socket number is allocated by the SOCKET call. Before a socket number can be used in other calls it must be associated with a port number and an IP address. This association is accomplished with the BIND call.

Port numbers are used to distinguish individual processes within a system. Some applications always use the same port on every system. These port numbers are called “well-known ports” and include applications such as FTP (File Transfer Protocol), NFS (Network File System) and TELNET. Other ports are arbitrary numbers assigned to a user by TCP/IP.

An IP address consists of a dotted decimal number such as `'123.45.678.9'`. Each IP address on a TCP/IP network must be unique. A computer can have more than one address if it is attached to more than one network.

Once a socket is bound to a port and IP address, the LISTEN call can be used to make that port/address combination available for connections. Another process

can then use the CONNECT call to attempt to form a connection. The ACCEPT call is then used by the listening process to complete the connection. The ACCEPT call allocates a new socket number that is associated with the new connection. The original socket remains in listening mode.

Once a connection is completed, one or both sides can read and write data with the SEND, RECV, READ, and WRITE calls.

Other socket calls return information about the system (GETHOSTID, GETHOSTNAME) or a connection (GETPEERNAME,GETSOCKNAME). FCNTL and IOCTL are used to change the characteristics of a socket. Finally, SHUTDOWN and CLOSE are used to terminate one or both sides of a connection. For details about the meaning and use of the socket calls, consult: *TCP/IP Version 2.0 for VM: Programmer's Reference* or *TCP/IP Version 2.0 for MVS: Programmer's Reference*.

IUCV Paths and Sockets

The APL2 Socket API uses IUCV to communicate with TCP/IP. Initially, one IUCV path is used for cross-system shared variables and one path is used for each variable shared directly with AP 119. Under CMS, the number of concurrent IUCV paths is controlled by the MAXCONN parameter in the user's directory entry. This may need to be changed if a large number of variables are to be shared with AP 119.

A maximum of 64 sockets are available for each IUCV path. For the cross-system shared variable path, 2 sockets are always used for a listening connection and communication with the port servers. This leaves a maximum of 62 sockets available for use. One socket is used for each processor number specified in a shared variable offer. Each variable shared with AP 119 can allocate up to 64 unique sockets to be used in other APL2 Socket API calls.

AP 119 and TCP/IP Commands Summary

To use AP 119, the user shares a variable with the AP and passes vectors of vectors that request various actions. The first element of the value assigned to the variable determines which of two types of commands is being issued:

- Commands to TCP/IP - '*TCPIP*'
- Commands to AP 119 - '*AP*'

The general form of the result is a three-element vector:

1. An AP 119 return code
2. A TCP/IP return code
3. Data returned by the command

For example, to issue the TCP/IP command GETHOSTID, you assign to the shared variable:

```
SV119←'TCPIP' 'GETHOSTID'
(AP119_RC TCPIP_RC DATA)←SV119
```

Figure 45. Auxiliary Processor 119 Commands

Command	
TCP/IP Commands	**Blocking**
'TCPIP' 'ACCEPT' socket	Yes
'TCPIP' 'BIND' socket local_port local_addr	No
'TCPIP' 'CLOSE' socket	No
'TCPIP' 'CONNECT' socket remote_port remote_addr	Yes
'TCPIP' 'FCNTL' socket command data	No
'TCPIP' 'GETHOSTID'	No
'TCPIP' 'GETHOSTNAME'	No
'TCPIP' 'GETPEERNAME' socket	No
'TCPIP' 'GETSOCKNAME' socket	No
'TCPIP' 'GETSOCKOPT' socket level option	No
'TCPIP' 'LISTEN' socket backlog	No
'TCPIP' 'READ' socket type	Yes
'TCPIP' 'RECV' socket flags type	Yes
'TCPIP' 'RECVFROM' socket flags type	Yes
'TCPIP' 'SELECT' num_sockets read_mask write_mask exception_mask timeout	No
'TCPIP' 'SEND' socket flags type data	Yes
'TCPIP' 'SENDTO' socket flags type data family remote_port remote_addr	Yes
'TCPIP' 'SETSOCKOPT' socket level option optvalue1 [optvalue2]	No
'TCPIP' 'SHUTDOWN' socket how	No
'TCPIP' 'SOCKET'	No
'TCPIP' 'WRITE' socket type data	Yes
AP Commands	
'AP' 'GETLPORT'	
'AP' 'SETLPORT' processor_id listening_port	
'AP' 'PSLIST' password	
'AP' 'PSCLEAR' password	
'AP' 'PSSHUTD' password	
'AP' 'UNREGSTR'	

Definition of TCP/IP Terms

address. An IP address represented as a character string containing four decimal numbers separated by dots (such as '11.222.3.444')

backlog. The maximum length for the queue of pending connections on a listening socket.

data. The data sent or received.

exception mask. A Boolean vector use to identify socket numbers for which notification of exceptional conditions is desired.

family. Defines the addressing family used for IP addresses. For the APL socket interface, use 2 (AF_INET).

flags. An integer representing options to the SEND and RECV commands.

hostname. A character vector representing the name of the host processor.

how. Defines the condition of a SHUTDOWN command.

IP address. An address assigned to a host connected to TCP/IP network. A host has one IP address for each network connection.

length. Number of characters sent or received.

level. The level of communication specified on a GETSOCKOPT or SETSOCKOPT command. When using the APL socket interface, the level must be 6 (SOL_SOCKET).

listening port. A port available for a connection.

local address. The IP address of the host on which you are running.

number of sockets. The number of sockets to check in a SELECT command.

option. An integer identifying an option for the GETSOCKOPT and SETSOCKOPT commands.

option value. Value assigned to a particular option by the SETSOCKOPT command.

password. The password for the local port server. You are prompted for this password when starting the port server. It is also required on the AP commands PSLIST, PSCLEAR, and PSSHUTD.

port. Number used to distinguish your process from others on the host. This number can be selected by you or assigned by TCP/IP.

read mask. A Boolean vector use to identify socket numbers for which notification of ready-to-read conditions is desired.

remote address. The IP address of the host with which you are communicating.

socket. An endpoint for communication.

socket number. An integer used to represent a socket. For compatibility with the C socket interface, socket numbers 0, 1, and 2 are not used.

type. A character scalar that identifies translation options for data sent or received.

write mask. A Boolean vector use to identify socket numbers for which notification of ready-to-write conditions is desired.

Blocking

Some socket calls may not return control until a condition is satisfied. For example, the READ and RECV calls may not return control until data is available. The default state of a socket is blocking mode, which means that these calls do not return control immediately.

Using the APL2 socket interface with sockets in the default mode, AP 119 does not receive control back from TCP/IP until blocking calls are complete. Since AP 119 does not have control, it is not able to set the shared variable until the blocking condition is satisfied. A reference of the variable causes a shared variable interlock until the blocking call completes.

A socket can be set to nonblocking mode with the FCNTL socket call. If this is done, AP 119 receives control on subsequent calls and returns the EWOULDBLOCK return code.

Using AP 119—The TCPIP Commands

The TCPIP commands provide a means to make calls to TCP/IP. The AP 119 TCP/IP interface closely matches the TCP/IP C socket interface. The following sections describe the APL2 syntax used for making TCP/IP calls through AP 119.

ACCEPT

Accepts a connection request. This call accepts the first connection on its queue of pending connections. A new socket number is returned for the connection and the original socket remains available to accept more connection requests. This call blocks if there are no pending connections, and if the socket is in blocking mode.

```
      SV119←'TCPIP' 'ACCEPT' sn
      (APRC TCPIPRC CMDRC)←SV119
      (ns rp ra)←CMDRC
```

Where:

sn Is the socket number.

ns Is the new socket number assigned by TCP/IP.

rp Is the port number of the remote process that connected with you.

ra Is the IP address of the remote process that connected with you.

Example:

```
      SV119←'TCPIP' 'ACCEPT' 3
      (APRC TCPIPRC CMDRC)←SV119
      CMDRC
4 1023  9.113.12.92
```

BIND

Associates a local IP address and port with a socket number.

```
      SV119←'TCPIP' 'BIND' sn lp la
      (APRC TCPIPRC CMDRC)←SV119
```

Where:

sn Is the socket number.

lp Is the local port number. If this number is 0, TCP/IP assigns an unused port number.

la Is the local IP address. If this address is '0.0.0.0', the socket can be used with any local network.

CMDRC Is 0.

Example:

```
      SV119←'TCPIP' 'BIND' 3 1023 '9.112.12.92'
      SV119
0 0 0
```

CLOSE

Shuts down a socket. If the socket is associated with an open TCP connection, the connection is closed.

```
      SV119←'TCPIP' 'CLOSE' sn
      (APRC TCPIPRC CMDRC)←SV119
```

Where:

sn Is the socket number.

CMDRC Is 0.

Example:

```
      SV119←'TCPIP' 'CLOSE' 4
      SV119
0 0 0
```

CONNECT

Completes the binding necessary for a socket if *BIND* was not issued and establishes a connection to a socket in listening mode. If the socket is in blocking mode, this call blocks until the connection is complete or an error is returned.

```
      SV119←'TCPIP' 'CONNECT' sn rp ra
      (APRC TCPIPRC CMDRC)←SV119
```

Where:

sn Is the socket number.

rp Is the remote port number.

ra Is the remote IP address.

CMDRC Is 0.

Example:

```
      SV119←'TCPIP' 'CONNECT' 3 1002 '9.113.14.90'
      SV119
0 0 0
```

FCNTL

Allows an application to change the operating characteristics of a socket.

```
      SV119←'TCPIP' 'FCNTL' sn cmd cdata
      (APRC TCPIPRC CMDRC)←SV119
```

Where:

sn Is the socket number.

cmd Is the command (see below).

cdata Is the data associated with the command (see below).

CMDRC Is 0 (if setting the status) or the status flags (if getting the status).

Note: If the *cmd* is 'F_GETFL', the, *cdata* parameter is ignored.

The possible values for *cmd* are *'F_GETFL'* and *'F_SETFL'*.

The possible values for *cdata* are 0 and *'FNDELAY'*.

Example:

```
      SV119←'TCPIP' 'FCNTL' 3 'F_SETFL' 'FNDELAY'
      SV119
0 0 0
```

GETHOSTID

Returns the IP address for the host. If the host has more than one IP address, the primary one is returned.

```
      SV119←'TCPIP' 'GETHOSTID'
      (APRC TCPIPRC ia)←SV119
```

Where:

ia Is the primary host IP address.

Example:

```
      SV119←'TCPIP' 'GETHOSTID'
      (APRC TCPIPRC CMDRC)←SV119
      CMDRC
9.113.12.92
```

GETHOSTNAME

Returns the name of the host processor on which the user is running.

```
      SV119←'TCPIP' 'GETHOSTNAME'
      (APRC TCPIPRC hn)←SV119
```

Where:

hn Is the name of the host.

Example:

```
      SV119←'TCPIP' 'GETHOSTNAME'
      (APRC TCPIPRC CMDRC)←SV119
      CMDRC
STLVM20
```

GETPEERNAME

Returns the family, port, and IP address of a peer connected to a given socket.

```
      SV119←'TCPIP' 'GETPEERNAME' sn
      (APRC TCPIPRC CMDRC)←SV119
      (fm rp ra)←CMDRC
```

Where:

sn Is the socket number.

fm Is the family (always 2 - AF_INET).

rp Is the remote port.

ra Is the remote IP address.

Example:

```
      SV119←'TCPIP' 'GETPEERNAME' 3
      (APRC TCPIPRC CMDRC)←SV119
      CMDRC
2 1002  9.113.14.90
```

GETSOCKNAME

Returns the family, port, and IP address of a given socket.

```
      SV119←'TCPIP' 'GETSOCKNAME' sn
      (APRC TCPIPRC CMDRC)←SV119
      (fm lp la)←CMDRC
```

Where:

sn Is the socket number.

fm Is the family (always 2 - AF_INET).

lp Is the local port.

la Is the local IP address.

Example:

```
      SV119←'TCPIP' 'GETSOCKNAME' 3
      (APRC TCPIPRC CMDRC)←SV119
      CMDRC
2 1023  9.113.12.92
```

GETSOCKOPT

Gets options associated with a socket.

```
      SV119←'TCPIP' 'GETSOCKOPT' sn lv op
      (APRC TCPIPRC ov)←SV119
```

Where:

sn Is the socket number.

lv Is the communication level.

op Is the option name.

ov Is the option value.

Figure 46 provides the values and levels that are defined for the *GETSOCKOPT* and *SETSOCKOPT* calls.

Figure 46. TCP/IP Socket Options

Option	Level
SO_BROADCAST	SOL_SOCKET
SO_DONTROUTE	SOL_SOCKET
SO_ERROR	SOL_SOCKET
SO_LINGER	SOL_SOCKET
SO_OOBINLINE	SOL_SOCKET
SO_REUSEADDR	SOL_SOCKET
SO_TYPE	SOL_SOCKET

Example:

```
      LEV←'SOL_SOCKET'
      OPT←'SO_BROADCAST'
      SV119←'TCPIP' 'GETSOCKOPT' 4 LEV OPT
      SV119
0 0 1
```

Note: If the option specified is SO_LINGER, the third item is a vector of two integers, representing the timeout value in seconds and microseconds, respectively. For all other options, the third item is a single integer.

LISTEN

Completes the binding necessary for a socket if *BIND* has not been issued, and creates a connection request queue.

```
      SV119←'TCPIP' 'LISTEN' sn bl
      (APRC TCPIPRC CMDRC)←SV119
```

Where:

sn Is the socket number.

bl Is the length of the request queue.

Example:

```
      SV119←'TCPIP' 'LISTEN' 3 5
      SV119
0 0 0
```

READ

Reads data from a given socket. If the socket is in blocking mode and no data is available to read, this call blocks.

```
      SV119←'TCPIP' 'READ' sn type
      (APRC TCPIPRC CMDRC)←SV119
```

Where:

sn Is the socket number.

type Is one of:

- `'B'` To receive data as is
- `'E'` To receive EBCDIC character data
- `'A'` To receive ASCII character data

CMDRC Is the data read from the socket.

Example:

```
      SV119←'TCPIP' 'READ' 4 'B'
      (APRC TCPIPRC CMDRC)←SV119
      CMDRC
(data sent by partner)
```

RECV

Receives data from a given socket. If the socket is in blocking mode and no data is available to receive, this call blocks.

```
      SV119←'TCPIP' 'RECV' sn flg tp
      (APRC TCPIPRC CMDRC)←SV119
```

Where:

sn Is the socket number.

flg Is receive option parameters (see below).

tp Is the type and is one of:

- `'B'` To receive data as is
- `'E'` To receive EBCDIC character data
- `'A'` To receive ASCII character data

Possible flag values are 0, 1 (MSG_OOB) or 2 (MSG_PEEK)

Example:

```
      SV119←'TCPIP' 'RECV' 4 0 'B'
      (APRC TCPIPRC CMDRC)←SV119
      CMDRC
(data sent by partner)
```

RECVFROM

Receives data from a socket and identifies the source of the data.

```
      SV119←'TCPIP' 'RECVFROM' sn flg tp
      (APRC TCPIPRC CMDRC)←SV119
      (dat add)←CMDRC
      (fam rp ra)←add
```

Where:

sn Is the socket number.

flg Is the recvfrom option parameters (see RECV above).

tp Is the type and is one of:

- `'B'` To receive data as is
- `'E'` To receive EBCDIC character data
- `'A'` To receive ASCII character data

dat Is the data received from the socket.

fam Is the family (always 2 - AF_INET).

rp Is the remote port.

ra Is the remote IP address.

Example:

```
      SV119←'TCPIP' 'RECVFROM' 4 0 'B'
      (APRC TCPIPRC CMDRC)←SV119
      CMDRC
(data sent by partner) 2 1003 9.113.14.90
```

SELECT

Monitors activity on a set of sockets as specified by three masks. The first argument indicates how many sockets to check. This is normally 1 plus the largest socket number allocated. The masks must be at least as long as this value. A 1 in the mask specifies a corresponding socket to check. Three masks are returned as soon as something happens on a selected socket. For example, the read mask has a 1 if the corresponding socket has data ready to read. If the connection to a listening socket is completed, the read mask is set to indicate that a connection has been made to the socket and an ACCEPT call can be made. A socket is normally always ready to write. The except mask is set if out of band data is received. The timeout value specifies the number of seconds to wait for the call to be completed. A value of zero means to wait indefinitely.

```
      SV119←'TCPIP' 'SELECT' ns rm wm xm to
      (APRC TCPIPRC CMDRC)←SV119
      (rm wm xm)←CMDRC
```

Where:

ns Is the number of sockets to check.

rm Is a read mask.

wm Is a write mask.

xm Is an exception mask.

to Is the timeout value.

Example:

```
      R_MASK←0 0 0 1 1
      W_MASK←0 0 0 0 0
      X_MASK←0 0 0 0 0
      SV119←'TCPIP' 'SELECT' 5 R_MASK W_MASK X_MASK 0
      (APRC TCPIPRC CMDRC)←SV119
      DATA
0 0 0 1 0   0 0 0 0 0   0 0 0 0 0
```

SEND

Transmits data to a remote user whose remote address and port were bound to the socket. If the socket is in blocking mode, this call blocks until TCP/IP can send the data.

```
      SV119←'TCPIP' 'SEND' sn flg tp dat
      (APRC TCPIPRC CMDRC)←SV119
```

Where:

sn Is the socket number.

flg Is the send option parameters (0 or 1 - MSG_OOB).

tp Is the type and is one of:

- '*B*' To send data as is
- '*E*' To send EBCDIC character data
- '*A*' To send ASCII character data

dat The data to be sent.

CMDRC Is the number of characters sent.

Example:

```
      SV119←'TCPIP' 'SEND' 3 0 'B' 'CHARACTERS'
      (APRC TCPIPRC CMDRC)←SV119
0 0 10
```

SENDTO

Transmits data to a remote user whose remote address and port are specified in the command. For APL2, the family is normally 2.

```
      SV119←'TCPIP' 'SENDTO' sn fl tp dat fm rp ra
      (APRC TCPIPRC CMDRC)←SV119
```

Where:

sn Is the socket number.

fl Is the sendto option parameters (0 or 1 - MSG_OOB).

tp Is one of:

- '*B*' To send data as is
- '*E*' To send EBCDIC character data
- '*A*' To send ASCII character data

dat The data to be sent.

fm Is the family (always 2 - AF_INET).

rp Is the remote port number.

ra Is the remote IP address.

CMDRC Is the number of characters sent.

Example:

```
      (FAM R_PORT R_ADDR)←2 1002 '9.113.12.92'
      DATA←'These are characters.'
      SV119←'TCPIP' 'SENDTO' 3 0 'B' DATA FAM R_PORT R_ADDR
      SV119
0 0 23
```

SETSOCKOPT

Sets options associated with a socket.

```
      SV119←'TCPIP' 'SETSOCKOPT' sn lv op o1 [o2]
      (APRC TCPIPRC CMDRC)←SV119
```

Where:

sn Is the socket number.

lv Is the level of communication.

op Is the option name.

o1 Is the option value.

o2 Is the optional second option value. This is used only if *op* is SO_LINGER.

Figure 46 on page 162 provides the values and levels that are defined for the *GETSOCKOPT* and *SETSOCKOPT* calls.

Example:

```
      LEV←'SOL_SOCKET'
      OPT←'SO_BROADCAST'
      SV119←'TCPIP' 'SETSOCKOPT' 4 LEV OPT 1
      SV119
0 0 0
```

SHUTDOWN

Shuts down all or part of a duplex connection.

```
      SV119←'TCPIP' 'SHUTDOWN' sn how
      (APRC TCPIPRC CMDRC)←SV119
```

Where:

sn Is the socket number.

how Is the type of shutdown (0, 1, or 2).

Example:

```
      SV119←'TCPIP' 'SHUTDOWN' 4 1
      SV119
0 0 0
```

SOCKET

Creates an endpoint for communication. A socket number is allocated for use in other socket calls.

```
      SV119←'TCPIP' 'SOCKET'
      (APRC TCPIPRC CMDRC)←SV119
```

Where:

CMDRC Is the socket number allocated.

Example:

```
      SV119←'TCPIP' 'SOCKET'
      SV119
0 0 3
```

WRITE

Writes data to a given socket.

```
      SV119←'TCPIP' 'WRITE' sn tp dat
      (APRC TCPIPRC CMDRC)←SV119
```

Where:

sn Is the socket number.

tp Is one of:

- `'B'` To send data as is
- `'E'` To send EBCDIC character data
- `'A'` To send ASCII character data

dat Is the data to be sent.

CMDRC Is the number of characters written.

Example:

```
      SV119←'TCPIP' 'WRITE' 3 'B' 'Many characters'
      SV119
0 0 15
```

Using AP 119—The AP Commands

The AP commands provide a means to talk to the AP itself. They support using the cross-system shared variable interface without port servers and issuing commands to the local port server.

The APL2 Port Server

APL2's cross-system shared variable facility provides communication across TCP/IP networks through APL2 shared variables.

TCP/IP's addressing is designed to support server machines, not individual user IDs. TCP/IP talks to port numbers that users, and servers, must acquire. Standard servers are assigned standard ports on all TCP/IP systems. A server's standard port is called its “well-known” port number. Port numbers from 1 to 255 are reserved for this purpose. For example, the FTP file transfer server is always on port 21. Other applications can use port numbers between 256 and 65535. TCP/IP provides no way for APL2 to know the port that an individual on a given system may be using.

To provide this ability, APL2 includes a program called the port server that manages the communication of individual port numbers across the network. However, users need to be able to contact APL2's port server. APL2 has not been assigned a well-known port number. Therefore, APL2's cross-system shared variable facility assumes a nonstandard port is available on all systems. For this purpose, APL2 uses the number 31415 as its “not-so-well-known” port number. The APL2 port server is a program that listens on TCP/IP port number 31415.

The server can be started with a different port number if 31415 is not available, for debugging purposes or for use between host systems that agree to use the different number. All users who want to perform cross-system shares from or to a system using a different port number must be informed of the number in use. The users should specify the port number in their TCP/IP profile files.

An APL2 port server should be run on each machine that has users who want to use cross-system shared variables. However, it is possible to use the interface without having a port server running.

The port server has three functions:

- Accept requests to register users on the same system. This function tells the server which port number a given user is using to accept connections from other users. This port number is arbitrarily assigned to the user by TCP/IP.
- Accept requests to unregister users. This notifies the server that a given user is no longer accepting communication. This is automatically issued when the user's APL2 session ends.
- Accept requests from remote users who want to know the port number that has been registered by a user.

PSLIST—Send LIST Command to the Port Server

This command causes the port server to list the registration entries (if any) that are active. This list occurs on the console of the port server only, no data is returned to the user issuing the command.

```
      SV←'AP' 'PSLIST' password
      SV
0 0 0
```

Where:

password Is the password used when starting the port server.

PSCLEAR—Send CLEAR Command to the Port Server

This command causes the port server to clear all registration entries (if any) that are active.

```
      SV←'AP' 'PSCLEAR' password
      SV
0 0 0
```

Where:

password Is the password used when starting the port server.

PSSHUTD—Send SHUTDOWN Command to the Port Server

This command causes the port server to shut down.

```
      SV←'AP' 'PSSHUTD' password
      SV
0 0 0
```

Where:

password Is the password used when starting the port server.

UNREGSTR—Send an UNREGISTER Command to the Port Server

This command causes the port server to erase the current registration entry for the user.

```
      SV←'AP' 'UNREGSTR'
      SV
0 0 0
```

Listening Ports

The APL2 cross-system shared variable interface lets users share variables across systems connected by TCP/IP networks. It uses AP 119 to communicate with port servers across the network. AP 119 provides two commands through which a user can use cross-system shared variables even if the partner's port server is not available.

APL2's port server manages a list of the ports in use by individuals on the system on which it is running. When you make an offer to share a variable across systems through a TCP/IP network, the interface uses AP 119 to contact the port server on the remote system and requests the port number of your partner.

If there is no port server running on the remote system, you need a way to give the cross-system shared variable interface the listening port your partner is using. Likewise, if there is no port server running on your machine, you need a way to find out what port you are using (to tell your partner.) The GETLPORT and SETLPORT AP 119 commands provide these functions.

GETLPORT—Get Listening Port

Acquires a listening port (if one is not already in use) and returns the port number. The number can then be given to a remote user to contact you even if your system does not have a port server running.

```
      SV←'AP' 'GETLPORT'
      (APRC TCPIPRC PORT)←SV
```

Where:

APRC Is the AP 119 return code.

TCPIPRC Is the TCPIP return code.

PORT Is your listening port number.

SETLPORT—Set Listening Port

Lets you inform AP 119, and thereby the cross-system shared variable interface, the listening port number that is in use by a session on a remote system. If the SETLPORT command was issued for a specific processor ID, AP 119 uses the specified port number rather than attempting to contact the port server on the remote system. This allows you to then share a variable with the remote user even if there is no port server running on the remote system.

```
      SV←'AP' 'SETLPORT' PROC_ID PORT
      SV
0 0 0
```

Where:

PROC_ID Is the processor ID number by which you need to refer to the remote partner. This number must be defined in the TCP/IP profile file.

PORT Is the listening port number that your remote partner is using. Your partner can determine this number by using the GETLPORT command.

Starting AP 119

Auxiliary Processor 119 automatically starts when APL2 is invoked unless the EXCLUDE option is used to prevent it. The following options can be specified with the APNAMES parameter:

SERVPORT(nnn)

Normally, the local port server is listening on port number 31415. If the port server is using a different port number, this parameter must be

used to allow AP 119 to communicate with the server. Allowable values are 256 to 65535.

LISTEN(nnn)
AP 119 does not normally open a listening connection until a variable is offered to a remote processor. If you want AP 119 to open a listening connection on startup, use this parameter. AP 119 attempts to use the number specified as its listening port. If this number is unavailable, the listening connection is not started. Allowable values are 256 to 65535, or 0 to let TCP/IP assign an arbitrary number.

TCPID('ccc')
AP 119 expects the name of the local TCP/IP machine (under CMS) or started task name (under TSO) to be TCPIP. If the name is different, this parameter must be used to allow AP 119 to successfully communicate with TCP/IP.

This is an example of using the startup parameters:

```
APL2 APNAMES(AP2X119(SERVPORT(1234) LISTEN(2345) TCPID('TCPTEST')))
```

Figure 47. Sample AP 119 startup parameters

If no options are specified, the IBM-supplied default for AP2X119 is:

```
SERVPORT(31415) TCPID('TCPIP')
```

This may have been changed by your installation.

Sample AP 119 Session Using the APL2 Socket API

In this example, one user shares the variable *A* and the other user shares variable *B*.

```
      ⍝ User 1 shares a variable with AP 119
      119 ⎕SVO'A'
1
      0 0 1 1 ⎕SVC 'A'
0 0 1 1
      ⍝ User 1 allocates a socket
      A←'TCPIP' 'SOCKET'
      A
0 0 3
```

The return code shows that socket number 3 was allocated. This is a stream socket that is allocated to the user but not bound to a particular port or address and is not connected.

```
      ⍝ User 1 binds socket to a port
      A←'TCPIP' 'BIND' 3 1023 '0.0.0.0'
      A
0 0 0
```

Notice that a zero IP address was specified. If a machine is connected to more than one network (and therefore has more than one IP address), you can bind to a particular network, or specify '0.0.0.0' as the address meaning that you accept a connection to any network. Using '0.0.0.0' as the IP address helps maintain the portability of your application.

Port number 1023 is an arbitrary number agreed upon by both users. If the port number is being used by anyone else on the local system, an EADDRINUSE error is returned and the BIND is unsuccessful.

```
      ⍝ User 1 listens for a connection
      A←'TCPIP' 'LISTEN' 3 5
      A
0 0 0
```

```
      ? User 2 shares a variable with AP 119
      119 ⎕SVO 'B'
1
      0 0 1 1 ⎕SVC 'B'
0 0 1 1
      ⍝ Allocates a socket
      B←'TCPIP' 'SOCKET'
      B
0 0 3
      ⍝ Binds it to user 1's port and address
      B←'TCPIP' 'BIND' 3  1055 '0.0.0.0'
      B
0 0 0
```

Although user 2 also gets socket number 3, this has no relationship with the socket allocated to user 1.

```
      ⍝ User 2 connects to user 1
      B←'TCPIP' 'CONNECT' 3 1023 '9.113.14.90'
      B
0 0 0
```

```
      ⍝ User 1 accepts the connection
      A←'TCPIP' 'ACCEPT' 3
      A
 0 0   4 1055 9.113.14.90
```

When user 1 does an ACCEPT, a new socket is allocated (4 in this case) and the connection is completed using the new socket. The original socket (3 in this case) remains listening for new connections.

There is now an established connection between the two users. The result from the ACCEPT call has as its third item the new socket number allocated and user 2's port number and IP address.

```
      ⍝ User 1 sends data to user 2
      A←'TCPIP' 'SEND' 4 0 'E' 'SOME DATA'
      A
0 0 9
```

The 'E' means that EBCDIC characters are being sent. In this case, specifying type 'E' is the same as type 'B' since both sides are already using EBCDIC. If the receiving side was ASCII-based then type 'A' on the SEND would cause translation to ASCII before the data is sent.

Note that you can send noncharacter data so long as you and your partner agree on formats. In this case, you should always use the 'B' type option. When the data is received by the partner, it is received as characters and the external function RTA can be used to restore the data to the expected format.

```
      ⍝ User 2 receives the data
      B←'TCPIP' 'RECV' 3 0 'E'
      B
 0 0 SOME DATA
```

In this case, all the data was received at once but this may not always be the case. Stream socket protocol does not guarantee that data that was sent is received in one RECV call. It is up to the users to agree on a convention for saying how much data is sent so that the partner can tell when all data has been received.

```
      ⍝ Since a user cannot predict when data
      ⍝ will arrive, it is not known when a
      ⍝ receive should be done.  User 1 issues
      ⍝ a SELECT that requests that he be
      ⍝ informed when data arrives.
      R_MASK←0 0 0 0 1
      W_MASK←0 0 0 0 0
      X_MASK←0 0 0 0 0
      A←'TCPIP' 'SELECT' 5 R_MASK W_MASK X_MASK
```

Because the largest socket is 4, you must specify 5 as the number of of sockets and provide three length 5 masks. The one in the first mask says that user 1 wants to be informed when socket 4 is ready to read. You can specify more than one socket by specifying more than one 1.

Since RECV is a blocking call, if user 1 now references 'A', processing stops until data arrives. Alternately user 1 can do other computing. The user can use *⎕SVS 'A'* and check for 0 1 0 1 to see if AP 119 has assigned data to the variable. The user can use *⎕SVE* to wait for an event on any of his shared variables or until a specified amount of time has passed, whichever comes first.

```
      ⍝ User 2 sends some data
      B←'TCPIP' 'SEND' 3 0 'E' 'DATA BACK TO YOU'
      B
0 0 0
```

```
      ⍝ User 1 notices that data is available
      ⎕SVS 'A'
0 1 0 1
      A
 0 0   0 0 0 0 1  0 0 0 0 0  0 0 0 0 0
```

The 1 in the read mask means that socket 4 is ready to read. The masks returned from the SELECT may have zero or more 1 bits on but never more than originally specified in the SELECT call.

```
      ⍝ User 1 reads the data
      A←'TCPIP' 'RECV' 4 0 'E'
      A
 0 0 DATA BACK TO YOU
      ⍝ User 1 closes all his connections
      A←'TCPIP' 'CLOSE' 4
      A
0 0 0
      A←'TCPIP' 'CLOSE' 3
      A
0 0 0
```

```
      ⍝ User 2 closes his connection
      B←'TCPIP' 'CLOSE' 3
      B
0 0 0
```

Return codes

The result passed back to the user in the shared variable is always a three item vector: AP 119 return code, subsystem (AP, TCP/IP or IUCV) return code, and data.

Figure 48 (Page 1 of 2). AP 119 Return Codes

Code	Definition
0	Success
1	Incorrect command
2	Wrong type

Figure 48 (Page 2 of 2). AP 119 Return Codes

Code	Definition
3	Wrong rank
4	Wrong shape
5	Item is wrong type
6	Item is wrong rank
7	Item is wrong shape
8	Item data is wrong
10	AP 119 subsystem support error. Second element of result is defined as follows: **1** No more sockets available through this variable. **2** Insufficient storage available to process the command.
11	TCP/IP error occurred. Second element of result is TCP/IP return code.
12	IUCV error occurred. Second element of result is IUCV return code.

AP codes 2, 3, 4, 5, 6, 7, and 8 return in the second item of the return code the index to the item being processed when the error was detected. The first item is numbered 0.

If the first item of the return code is 11, the second item is set to one of the character values listed under *Errno* in Figure 49. The numeric *code* is used in messages AP2...304 and AP2...305.

Figure 49 (Page 1 of 3). TCP/IP Return Codes

Code	Errno	Definition
0		Success
1	EPERM	Not Owner
2	ENOENT	No such file or directory
3	ESRCH	No such process
4	EINTR	Interrupted system call
5	EIO	I/O error
6	ENXIO	No such device or address
7	E2BIG	ARG list too long
8	ENOEXEC	EXEC format error
9	EBADF	Bad file number
10	ECHILD	No children
11	EAGAIN	No more processes
12	ENOMEM	Not enough core
13	EACCES	Permission denied
14	EFAULT	Bad address
15	ENOTBLK	Block device required
16	EBUSY	Mount device busy
17	EEXIST	File exists
18	EXDEV	Cross-device link
19	ENODEV	No such device
20	ENOTDIR	Not a directory
21	EISDIR	Is a directory
22	EINVAL	Invalid argument

Figure 49 (Page 2 of 3). TCP/IP Return Codes

Code	Errno	Definition
23	ENFILE	File table overflow
24	EMFILE	Too many open files
25	ENOTTY	Not a typewriter
26	ETXTBSY	Text file busy
27	EFBIG	File too large
28	ENOSPC	No space left on device
29	ESPIPE	Illegal seek
30	EROFS	Read-only file system
31	EMLINK	Too many links
32	EPIPE	Broken pipe
33	EDOM	Argument too large
34	ERANGE	Result too large
35	EWOULDBLOCK	Operation would block
36	EINPROGRESS	Operation now in progress
37	EALREADY	Operation already in progress
38	ENOTSOCK	Socket operation on nonsocket
39	EDESTADDRREQ	Destination address required
40	EMSGSIZE	Message too long
41	EPROTOTYPE	Protocol wrong type for socket
42	ENOPROTOOPT	Protocol not available
43	EPROTONOSUPPORT	Protocol not supported
44	ESOCKTNOSUPPORT	Socket type not supported
45	EOPNOTSUPP	Operation not supported on socket
46	EPFNOSUPPORT	Protocol family not supported
47	EAFNOSUPPORT	Address family not supported by protocol family
48	EADDRINUSE	Address already in use
49	EADDRNOTAVAIL	Cannot assign requested address
50	ENETDOWN	Network is down
51	ENETUNREACH	Network is unreachable
52	ENETRESET	Network dropped connection on reset
53	ECONNABORTED	Software caused connection abort
54	ECONNRESET	Connection reset by peer
55	ENOBUFS	No buffer space available
56	EISCONN	Socket is already connected
57	ENOTCONN	Socket is not connected
58	ESHUTDOWN	Cannot send after socket shutdown
59	ETOOMANYREFS	Too many references: cannot splice
60	ETIMEDOUT	Connection timed out
61	ECONNREFUSED	Connection refused
62	ELOOP	Too many levels of symbolic links
63	ENAMETOOLONG	File name too long
64	EHOSTDOWN	Host is down
65	EHOSTUNREACH	No route to host

Figure 49 (Page 3 of 3). TCP/IP Return Codes

Code	Errno	Definition
66	ENOTEMPTY	Directory not empty
67	EPROCLIM	Too many processes
68	EUSERS	Too many users
69	EDQUOT	Disk quota exceeded
70	EVDBAD	RVD related disk error

Chapter 17. AP 120—APL2 Session Manager Command Processor

The APL2 session manager command processor, AP 120, allows you to issue APL2 session manager commands through shared variables. For a description of session manager commands and their syntax, see Chapter 3, “The APL2 Session Manager” on page 36.

AP 120 requires the Graphical Data Display Manager (GDDM).

Shared Variable Overview

Figure 50 provides an overview for sharing variables with AP 120.

Figure 50. Shared Variable Overview for AP 120

SV Protocol	AP 120 Conventions
General	Two variables—control and data. The control variable is used to pass APL2 session manager commands to AP 120 and to pass return codes from AP 120. The control variable must be a simple character scalar or vector. The data variable is used to pass data from AP 120 to the workspace. The data variable is a character matrix.
Maximum Number of Shared Variables	14 pairs
Names	Must start with *CTL* and *DAT*. Suffixes pair the variables. Names cannot exceed 11 characters.
Initial Values	No special requirements. If an initial value exists for the control variable when it is shared, the value is treated as the first specification of the variable.
Subsequent Values	*CTL*: Required. Character vector containing the command to be processed. There is no limit on the length of the command, except as noted below. Reference a return code from the request (two-item numeric vector) before referencing DAT. *DAT*: Optional. Reference a character matrix returned from AP 120. If the data variable is not shared, you cannot retrieve any results returned by session manager commands. Command processing, however, is unaffected. **Note:** The SHRSIZE invocation option or installation default may limit the size of a shared variable.
Data Types Supported	Simple character vector (*CTL* only). Character matrix (*DAT* only).
Access Control	*CTL*: 1 1 1 1 *DAT*: 0 0 0 0

Data Formats

Specification

The control variable (*CTL*) must be specified with a character vector that represents the session manager command to be processed.

AP 120 specifies the data variable (*DAT*). If you specify it, your specification is ignored.

Reference

AP 120 returns:

- Two-item numeric vector return code in the *CTL* variable. The return codes and their meanings are listed in Figure 52 on page 181.
- Character matrix in the optional *DAT* variable. The character matrix contains one of the following:
  - Text that is normally returned on the command line of the screen when the command is issued from the session manager itself.
  - Text that is normally returned from the command as a series of screens of information (for example, the HELP command or the PROFILE command with no operands).
  - Error message.
  - Text of the line, if the LINE command was successfully processed and you had supplied a line number or a number of lines to scroll.
  - Empty character matrix if the command does not normally return text (for example, COLUMN +*number* command) or if the command was not passed to AP 120 (return code 1 *nn*).

Communication Procedure

The steps to issue an APL2 session manager command through AP 120 are outlined below.

1. Offer to share the control variable and, if appropriate, the data variable with AP 120.
2. Assign a character vector containing the command to the control variable.
3. Check the control variable for a return code of 0 0, indicating successful completion of the request and processing of the command.
4. Reference the contents of the data variable, if necessary.
5. Repeat steps 2 through 4 until you have issued all the commands you require.
6. Retract the control and data variables.

Although AP 120 is normally used within a defined function, Figure 51 shows a sample APL2 session that illustrates the communication procedure.

```
      120 ⎕SVO ¨ 'CTL120' 'DAT120' ⍝ offer to share
2 2                          ←──────────Degree of coupling OK
      CTL120←'COPY ON ID APLAP120' ⍝ assign Sess. Mgr. command
      CTL120                 ⍝ check return code
0 0
      DAT120                 ⍝ copy command returns no data
      CTL120←'COPY'          ⍝ request copy settings
      CTL120                 ⍝ check return code
0 0
      DAT120                 ⍝ data variable contains
COPY ON ID APLAP120 CODE 0   ←──────────Current COPY setting
      ⎕SVR ¨ 'CTL120' 'DAT120'  ⍝ retracts shared variables
2 1
```

Figure 51. Sample APL2 Session to Communicate with AP 120

Return Codes

Figure 52 lists the return codes from AP 120, their descriptions, and suggested actions.

Figure 52. AP 120 Return Codes

Code	Description
0 0	Normal return. Command was successful.
1 11	Rank error on control variable. Rank is greater than 1.
1 13	Domain error on control variable. Control variable contains noncharacter data.
1 53	Required storage is unavailable. **Action**: Restart APL2, specifying a larger value for the FREESIZE option.
1 56	Shared variable size exceeded.
1 60	The APL2 session manager is not available.
4 *nnn*	Command was passed to the session manager, and an error was encountered. *nnn* is the numeric part of the session manager error message identifier. **Action**: Examine the *DAT* variable; it contains the text of the error message.

Chapter 18. AP 121—APL2 Data File Processor

AP 121, the APL2 data file processor, allows you to read and write APL2 arrays stored outside the active workspace in a sequential or direct access file. For example, you can use this processor to store large arrays that otherwise consume excessive space in the workspace. This processor writes arrays in their APL2 internal form, and, when it reads arrays, it expects their internal form.

APL2 data files can also be created and accessed using associated Processor 12. That Processor provides asynchronous access to data files using APL2 syntax rather than through auxiliary processor commands.

Under CMS, the arrays are stored in a CMS file in a library defined in the LIBTAB APL2 file.

Under TSO, the arrays are stored in a VSAM library, which must be allocated to your session before you use AP 121. This library can be the same library used for your session manager log file.

Associated Workspaces

Functions in the two public workspaces *APLDATA* and *VAPLFILE* can be used to read and write APL2 arrays to data files and to create, drop, and change file sizes. For information on the contents of these workspaces and where they can be found see *APL2/370 Programming: Using the Supplied Routines*.

Shared Variable Overview

Figure 53 provides an overview for sharing variables with AP 121.

Figure 53 (Page 1 of 2). Shared Variable Overview for AP 121

SV Protocol	AP 121 Conventions
General	One or two variables. **Sequential Access:** Only a control variable is required. Each specification passes a service request or an array to be written. Each reference obtains a return code or an array read. **Direct Access:** A control variable and a data variable are required. • Each specification of the control variable passes a service request; each reference obtains the return code from a requested operation. • A specification of the data variable contains an array to be written; a reference returns the array requested in the control variable.
Maximum Number of Shared Variables	14 pairs
Names	Must start with *CTL* and *DAT*. Suffixes pair the variables. Names cannot exceed 11 characters.
Initial Value	None required. However, if specified, it is treated as if it were the first assignment to the variable after sharing.

Figure 53 (Page 2 of 2). Shared Variable Overview for AP 121

SV Protocol	AP 121 Conventions
Subsequent Values	*CTL*: Specify a service request (character vector). Reference a return code (one-item numeric vector). *DAT*: Specify and reference any valid APL2 array that can be manipulated within the active workspace.
Data Types Supported	All valid APL2 data types.
Access Control	*CTL*: 1 1 1 1 *DAT*: 0 0 0 0

Access Control Considerations

AP 121 sets no access control on the data variable. If you choose to control the processor's specification of the data variable (`0 1 0 0 ⎕SVC 'DAT121'`), you must reference the data variable each time you receive a 0 return code for a successful direct read operation. If you do not reference the data variable, the return code from the next direct read request is 42.

If you choose to control your own access to the data variable (`0 0 1 0 ⎕SVC 'DAT121'`), you should not reference DAT when the return code from a direct read request is anything other than 0. If you do, your terminal is interlocked, and you must enter a strong interrupt to resume your APL2 session.

APL2 Data Files

Under CMS, APL2 data files created and processed using AP 121 are managed by the CMS file system. Each file is associated with an APL2 library number defined in the LIBTAB APL2 file, just as are workspace libraries. Workspaces and AP 121 files can coexist in the same library.

Under TSO, an APL2 data file resides in a VSAM library. The library must be defined as a VSAM key-sequenced data set with key length 14, offset 0. If the session manager is used, AP 121 files can reside in the same VSAM library as the session manager log file. Before any service request to AP 121 is issued, the VSAM library must be allocated to your TSO session.

To define and allocate VSAM libraries, see pages 68 through 70.

File Identification

An APL2 data file is identified to AP 121 as follows:

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

where:

libno Is the number of the private, project, or public library that contains, or is to contain, the APL2 data file. If this option is omitted, the library number defaults to the first item of ⎕*AI* (the user's APL2 account number). Unless changed by your installation, or unless you specify a different number in the ID option when you invoke APL2, the library number defaults to 1001.

CMS: The LIBTAB APL2 file associates the library number with the virtual disk on which the library resides.

TSO: The library number is associated with a ddname of F*nnnnnnn* for the VSAM file (leading 0s are suppressed). F0 can be used for the default library number.

Note: This ddname should not be confused with the term "filename." A ddname describes a library that can contain many individual files.

file name Is the name of the file. The name must be one to eight alphanumeric characters, the first being alphabetic. It can contain neither national use characters (such as @, $, or #) nor the extended APL characters ∆, ⍙, ¯, _, and underbarred letters.

CMS: AP 121 assigns a file type and file mode to files created with AP 121. For private-library files, the file type defaults to:

VSAPLFL

For files in public libraries, the file type defaults to *Fnnnnnnn*, where *nnnnnnn* is the library number and includes leading zeros to pad the name to eight characters.

Note: Your installation may have changed the default file types this processor assigns. Check with your system administrator.

The file mode assigned to the file is the same as that defined for the library in the LIBTAB APL2 file.

password Is an optional password associated with:

- **CMS:** Virtual disk on which the file resides.

  For read requests, the password is the virtual disk read password. For all other requests, it must be the virtual disk write password.

- **TSO:** VSAM cluster.

  For read requests, the password is the file's VSAM password at read-level or higher. For all other requests, it must be the file's VSAM password at update-level or higher.

If specified, the password must be preceded by a colon (:).

If you do not supply a password when one is required, you may be prompted for the password. If you supply an incorrect password, the auxiliary processor returns a return code of 34.

APL2 Data File Organization

An APL2 data file is organized as either a sequential file or a direct access file. Both organizations require the file to be created sequentially and extended sequentially (new arrays are added to the end of the file).

The record length of a direct file is specified in the service request to create the file. You can choose to use either limited or arbitrary data length. The maximum limited record length is 4070 bytes. Array sizes can be changed (up to the maximum) during update. To create a direct file with arbitrary data length you must specify 0 in the length field.

Arbitrary data length direct files are more flexible, easier to use, and better protected from program or system failures. But they require more space to store, may be slower to access, and are incompatible with APL Licensed Programs prior to APL2 Version 1 Release 3. If you decide to create *limited* data length direct files,"Space Requirements for Storing APL2 Variables" on page 191 discusses how to determine the size of various types of APL2 arrays.

Communication Procedure

You can open and process multiple files in the same APL2 session with AP 121. The number of files that can be processed concurrently depends on the available space in your virtual machine (CMS) or address space (TSO), and on the limit of number of shared variables. The same control and data variables can be used to pass requests and data for multiple files. However, if you only have one pair of shared variables, you can process (read, write, or update) only one file at a time. After a file is opened by a shared variable, that variable remains dedicated to the same file until it is used to close the file.

Commands

AP 121 commands are specified in the control variable. Figure 54 on page 186 lists the service requests that are available with AP 121.

Communicating with AP 121 is summarized as follows:

1. Offer to share a control and data variable. Although sequential processing does not require a data variable, it is recommended you share one to avoid the possibility of a mismatched pair if another function localizes and reoffers variables with the same names.

   Either the control variable or the data variable can be offered first. However, the data variable is not coupled until after a matching control variable is offered.

2. Create the file. If the file already exists, omit this step.

   If the file does not yet exist, specify the control variable with the service request to create either a sequential (*S*) or a direct (*D*) file.

   You use the create (*C*) request for both files that are accessed by VS APL and for files that contain data types new to APL2.

3. Open the file for one of the following types of processing:

Sequential write:	*CTL*←'*SWC fileid*[*length*]'
Sequential write:	*CTL*←'*SW fileid* [*length*]'
Sequential read:	*CTL*←'*SR fileid*'
Direct read:	*CTL*←'*DR fileid*'
Direct update:	*CTL*←'*DUC fileid*'
Direct update:	*CTL*←'*DU fileid*'

   If you are creating a file, you must open it for sequential write. If you create a direct file, you must specify a data length in the sequential write open request. It can be either an arbitrary length by indicating 0 in the length field or a limited length that cannot exceed 4070 bytes.

   Using the *SWC* and *DUC* open requests, you open files that allow any valid APL2 data types to be written or updated. You should use *SWC* and *DUC* when you are enhancing VS APL applications for use in APL2 or when creating new APL2 applications.

Using the SW and DU open requests, you open files that are compatible with VS APL. Such files allow data to be written and updated in a format that is compatible with VS APL, but they cannot contain any data types new to APL2.

"Opening a File" on page 187 provides more information on each open request.

4. Process the file sequentially or directly, depending on the file's organization and the open request.
5. Close the file.

   When you are finished reading, writing, or updating an APL2 data file, you should explicitly close the file by specifying the control variable with an empty vector.

   If you want to change from sequential reading to writing (or vice versa), you must close the file and reopen it.

   See "Checking for End of File" on page 190.
6. After a file is closed, the variable (or pair) is available for reuse to create or open another file. Repeat the procedure.

Two commands allow you to drop (delete) an APL2 data file or change its file size. For information on these two operations, see "APL2 Data File Maintenance" on page 190.

Library query command is described in "Library Query" on page 190.

Command	Operation
'*C fileid S*'	Create a sequential file
'*C fileid D*'	Create a direct file
	Open for sequential write
'*SWC fileid* [*length*]'	APL2-only
'*SW fileid* [*length*]'	VS APL-compatible
'*SR fileid*'	Open for sequential read
'*DR fileid*'	Open for direct read
	Open for direct update
'*DUC fileid*'	APL2-only
'*DU fileid*'	VS APL-compatible
0,*recnum*	Direct read
1,*recnum*	Direct write
'' or ⍳0	Close a file
'*D fileid*'	Drop a file
'*FS fileid size*'	Change file size limit
'*Q libno* [:*password*]'	Library query

Figure 54. Commands for APL2 Data File Operations Using AP 121

Opening a File

This section describes how to open a file for:

- Sequential write
- Sequential read
- Direct read
- Direct update

Open for Sequential Write (SWC or SW)

When you open a file for sequential write:

- The file can be either sequential or direct organization.
- For direct files, you must specify a data length or zero in the first *SW* request. For a new file, see Step 3 under "Commands" on page 185 for details.

  If you want to write records on an existing direct file that already contains records, do not specify a record length in the *SW* request. If you do, the length must equal that specified in the original *SW* request when the first array was written.
- Each subsequent specification of the control variable writes the next sequential array to the file. (Arrays are always added to the end of the file.)
- The file must be closed before it is reopened for any other type of processing in the same APL2 session.

  Note: Results are unpredictable if a sequential write (*SW*) is active through one shared variable at the same time a sequential read (*SR*) is active through another shared variable for the same file.

Open for Sequential Read (SR)

When you open a file for sequential read:

- The file can be organized either directly or sequentially.
- The first reference of the control variable (after the return code from the read request is obtained) returns the first array of the file. Subsequent references return arrays sequentially.
- If you reach the end of the file (return code 5) on a read operation, the processor accepts another open request for the file without an intervening close request.
- No return code is provided unless an error (including end of file) is detected, in which case the processor first assigns an empty vector to the control variable and then returns an error return code.

Open for Direct Read (DR)

When you open a file for direct read:

- The file must be of a direct organization.
- A data variable must be shared with the auxiliary processor to pass the contents of arrays read.

  To read an array, you must:

  –Specify the control variable as:

  CTL←0 **,** *recnum*

where *recnum* is the array number (beginning with 1 for the first array in the file) of the array you want read into your active workspace.

–Reference the data variable. The data variable contains the specified array.

Open for Direct Update (DUC or DU)

When you open a file for direct update:

- The file must have been created with a direct organization.
- A data variable must be shared.

You can read and write arrays in any sequence you want. You specify the array you want to read or write by relative array number (the first array is array 1).
If you want to read an array:

–Specify the direct read request as:

CTL←0 **,** *recnum*

–Reference the data variable.

If you want to replace an array:

–Specify the data variable with the APL2 array you want to write.

DAT←**'** *variable* **'**

–Specify the control variable with the direct write request:

CTL←1 **,** *recnum*

Note: When using direct update, it is possible to write an empty vector to the file. Should this occur, it may be impossible on a sequential read to detect whether an error occurred or whether end of file was encountered.

Example:: Figure 55 on page 189 shows a sample APL2 session that creates a direct file, initially loads it with arrays, and processes it for direct update.

```
      121 ⎕SVO¨ 'CTL121' 'DAT121'    ⍝ offer to share a pair
2 2                                  ←──────────Degree of coupling OK
      CTL121←'C DIRFIL1 D'           ⍝ create a direct file
      CTL121
0                                    ←──────────Return code OK
      CTL121←'SWC DIRFIL1 80'        ⍝ open for sequential write
      CTL121
0                                    ←──────────Return code OK
      CTL121←'FIRST ARRAY'           ⍝ begin writing sequentially
      CTL121
0
      CTL121←'SECOND ARRAY'
      CTL121
0
      CTL121←''                      ⍝ close file
      CTL121
0
      CTL121←'SR DIRFIL1'            ⍝ reopen for sequential read
      CTL121
0
      ⎕←X←CTL121                     ⍝ begin reading
FIRST ARRAY
      ⎕←X←CTL121
SECOND ARRAY
      ⎕←X←CTL121
                                ←──────────Empty vector returned?
      ⍴X
0                               ←──────────Yes
      CTL121                         ⍝ end of file?
5                               ←──────────Yes
      CTL121←'DUC DIRFIL1'           ⍝ reopen for update
      CTL121
0
      CTL121←0,2                     ⍝ point to second array
      CTL121
0
      DAT121                         ⍝ read second array
SECOND ARRAY
      DAT121←'NESTED' 1⍽2
      CTL121←1,2                     ⍝ replace second array
      CTL121
0
      CTL121←''                      ⍝ close file
      CTL121
0
```

Figure 55. Sample APL2 Session of Direct File Processing with AP 121

Checking for End of File

On a sequential read, an empty vector is returned when end of file is reached or when an error occurs. The next reference of the control variable returns the return code.

- If end of file was reached, the return code is 5.
- If an error has occurred, the return code is other than 0 or 5.

An array that is a simple empty vector can be written only to a direct access file when an array is replaced (otherwise, the file is closed when you specify an empty vector).

APL2 Data File Maintenance

Two service requests exist to allow you to:

- Drop (delete) an APL2 data file
- Change the total size allowed for an APL2 data file

An APL2 file can be deleted from a library by the drop file service request:

CTL←'*D fileid*'

To change the total size allowed for an APL2 file in a library, issue the following service request:

CTL←'*FS fileid size*'

where *size* is the maximum size in bytes to be allowed for the file. *size* can be specified as more or less than that currently allocated to the file, but it cannot be less than that currently occupied by the file.

The file size is the total size required for the file, not the maximum data length for a single array.

Library Query

A list of file names contained in a specific library can be obtained through the query service request:

'*Q libno* [**:***password*]'

The result is a matrix of file names returned in the DAT variable. The matrix always contains 9 columns with trailing blanks as needed. If the library is empty, `0 9 ⍴ ''` is returned.

Note: When library number 0 is specified, the query is performed on the user's current private library.

Space Requirements for Storing APL2 Variables

The only restrictions on the size of APL2 arrays written to sequential files or direct files that use an arbitrary data length are those imposed by available storage. (See "Size Limitations.") Each array in a direct file using limited data length, however, is limited in size to the number of bytes specified on the initial sequential write request when the file was loaded ('*SWC* fileid length'). Also, the largest data length you can specify for a limited length direct file is 4054 bytes.

The encoding of APL2 variables used by AP 121 is common data representation (CDR). Different types of APL2 variables require different amounts of space. In addition, each variable requires space for an APL2 variable descriptor, or the CDR header. To calculate the space required for a variable written in CDR, use the attributes ⎕*AT* system function with the integer 4 as a left argument. ⎕*AT* returns a two-item integer vector:

```
      DAT121←2 2⍴'VOLUME' 1021 'CHANGE' ¯23
      4 ⎕AT 'DAT121'
80 20
```

The first item is the total number of bytes required by the variable in CDR format. Four bytes must be added to this to obtain the AP 121 data length.

Note: If the file was created with SW rather than *SWC*, and your data consists only of integers between 0 and 255, the values returned by ⎕*AT* are too small.

For a further description of ⎕*AT*, see *APL2 Programming: Language Reference*.

Size Limitations

The APL2 arrays you write to a file can be any that are small enough to be:

- Manipulated within your active workspace
- Processed by the shared variable processor (SVP)
- Buffered by AP 121

You can control the size of your active workspace, the space allotted to the SVP, and the dynamic buffer space used by APL2 by specifying the APL2 invocation options WSSIZE, SHRSIZE, and FREESIZE, respectively.

Cautions

CMS: An APL2 data file resides on a virtual disk. The maximum size of the file is restricted to the available space on the disk. For more information on file sizes, see the appropriate user's guide for your system.

CMS does not normally rewrite a disk directory until all files on that disk have been closed. For APL2, this may not occur until you end your APL2 session. APL2 attempts to force directory rewrites whenever an output file is closed, but there is no guarantee that the directory is rewritten. Therefore, if APL2 does not terminate normally, all data written during the session could be lost.

TSO: AP 121 files reside in VSAM data sets. Depending on VSAM resource availability, you may be put in a wait queue if you try to access VSAM libraries used by other APL2 users. This can happen when your AP 121 file is opened for write (*SW*

or *SWC*) or update (*DU* or *DUC*). Repeated attention procedures may cause you to exit from APL2.

Always explicitly close a VSAM file by issuing the close service request (`''` or `⎕IO`) and check the return code. If you allow the file to close implicitly (the control variable being retracted), you cannot check the return code from the close, and the integrity of your data is exposed. If, for example, an I/O error occurs on the last buffer written, your file would be incomplete and you would not know it.

Return Codes

Return codes from AP 121 are passed in the control variable as a one-item numeric vector. Figure 56 lists the possible return codes from AP 121.

Figure 56 (Page 1 of 3). Return Codes from AP 121

Code	Description
0	Operation successful.
1	Array not found for a direct read or update request. **Action:** Specify an existing array number and reissue the request.
2	**CMS Only:** File on a virtual disk blocked 800 contains maximum number of records. As a result, the file is closed. Maximum file size is 64K. **Action:** Move the file to a disk that is not formatted with 800-byte blocks.
3	Attempted to write an APL2-only object with SW or DU. **Action**: Use the *SWC* and *DUC* requests if you want to write or update APL2 objects in a file; or, reformat the data so it is compatible with VS APL.
4	Either an internal logic error or the file being read contains invalid data. Action: Notify your system administrator that the file is unusable.
5	End of file. As a result, the file is closed. Also, for sequential read operations, an empty vector is assigned to the control variable.
6	You attempted a direct read or direct update on a sequential file.
7	Record length error. The buffer supplied to receive data is not large enough, or the record provided is too large for the file. When AP 121 is being used, buffer size problems are normally resolved automatically by the AP, even though a queued message may appear. Such queued messages may also appear (and be automatically resolved) during system command processing on TSO when a VSAM library is used.
9	**CMS:** Virtual disk is full. As a result, the file is closed. **TSO:** VSAM library is full. As a result, the file is closed.
10	**TSO:** User interrupted the operation with an attention key. If a write or drop was in progress, the array or file can be in an unusable state, and later may be deleted implicitly by the system or explicitly by another drop request.
11	Defined file size limit exceeded, or request to reduce file size limit would cause data loss. As a result, the file is closed. **Action:** Increase the file size limit (via the FS service request), reopen the file, and retry.

Figure 56 (Page 2 of 3). Return Codes from AP 121

Code	Description
12	Invalid service request, or syntax error in request. This code is returned for several error conditions associated with a service request: • Control variable does not contain a character vector when a service request is required. • An invalid service request code was specified. • A library number or the file size was not specified as a number, or it was too large a number. • File name or password is too long or has invalid characters. • A required field for the service request was omitted. • Unexpected fields were found. • Invalid direct read or update request. It was not a two-item numeric vector with the first item being either zero or one.
15	File not open. An attempt is being made to issue read or write requests against a file that is not open.
23	You have tried to replace an array on a direct file, but the file is not opened for direct update.
24	Improper library reference. **CMS Action:** Check LIBTAB APL2 for correct library numbers. **TSO Action:** Make sure the library is allocated.
25	File already exists.
26	File does not exist. **CMS Only**: This can also indicate that you were trying to write to a file that is stored on a read-only disk.
28	File in use by others. Your shared or exclusive file request is blocked. **Action:** Retry later. Be sure you do not already have the file open through another variable.
32	Either no space for I/O buffer or SHRSIZE is too small to contain the data. **Action:** Restart APL2 with more free space (by getting a larger virtual machine or region or specifying the FREESIZE invocation option described in Chapter 2, “APL2 Invocation and Termination” on page 8), or a larger SHRSIZE.
33	You attempted to replace an array on a direct file, but you did not specify the data variable before you issued the write service request.
34	Incorrect or missing password. **TSO Only:** This return code also can indicate that you were trying to write to a file that is available to you only for read access.
35	Hardware I/O error or APL2 product failure. As a result, the file is closed. Also, for sequential read operations, an empty vector is assigned to the control variable. **CMS Action:** Retry request. If problem persists, notify your system administrator. **TSO Action:** Verify that the library is a VSAM cluster defined with KEYS(14 0). Retry request. If problem continues, notify your system administrator.
36	**CMS Only:** Named file is not an APL2 data file. **TSO Only:** Named Data Set is an invalid VSAM cluster.
37	File size limit was reduced below the previous file size limit. This code is issued as a warning only; the FS operation is successful. There is still space for all arrays currently on the file.
38	Invalid length specified on an *SW* or *SWC* service request. The number is not within the valid range of 0 through 4070, is more than 8 characters long, or does not agree with the value given when the file was first written. If this return code appears in response to an FC Processor Service call, it indicates that the value is neither 0 nor a positive integer from 5 through 4074.

Figure 56 (Page 3 of 3). Return Codes from AP 121

Code	Description
39	Required record length is not specified in the sequential write (SW) open request to load records on a newly-created direct file.
42	Data variable has not been referenced.
50	Invalid request for CMS/TSO, or the file system is not operational. **Action:** See your system administrator.

Chapter 19. AP 123—VSAM File Processor

AP 123, the VSAM processor, allows you to perform file operations on entry-sequenced, key-sequenced, or relative-record VSAM files. Although AP 123 cannot create VSAM data sets, it can be used to read, write, update, and delete records on existing files.

Associated Workspaces

Public library 2 normally contains two workspaces that can be used with AP 123. Functions in the *VSAMDATA* workspace can be used to pass service requests to this processor. Functions in the *UTILITY* workspace are available to convert key and record data to System/370* data types. For information on these functions and how they are used, type *DESCRIBE*, *HOW*, or *ABSTRACT* after loading this workspace.

Shared Variable Overview

Figure 57 provides an overview for sharing variables with AP 123.

Figure 57. Shared Variable Overview for AP 123

SV Protocol	AP 123 Conventions
General	Two variables—control and record. The control variable is used to pass service requests to the AP and return codes from requested operations. The record variable is used to pass the content of records between the workspace and the VSAM file.
Maximum Number of Shared Variables	14 pairs.
Names	Must start with *CTL* and *DAT*. Suffixes pair the variables. Names cannot exceed 24 characters.
Initial Values	None. Initial values are ignored by the processor.
Subsequent Values	*CTL*: Specify a service request (character string). Reference a return code (two-item integer vector). *DAT*: Specify and reference a character scalar or character vector. **Note:** The SHRSIZE invocation option or installation default may limit the size of a shared variable.
Data Types Supported	Simple character vectors or scalars.
Access Control	*CTL*: 0 0 0 1 *DAT*: 0 0 0 0

VSAM Files—General Information

Before a VSAM file can be processed with AP 123, the file must have been previously created using the DEFINE command of Access Method Services. Under CMS, the DLBL command for the file must be issued before the APL2 session is initiated. Under TSO, the ALLOCATE command for the file must be issued at any time before the file is opened with AP 123.

File Identification

With AP 123, a VSAM data set or file is identified as follows:

```
ddname[:password]
```

ddname Is the ddname specified for the VSAM file in the CMS DLBL command or the FILE name specified in the TSO ALLOCATE command.

For more information on the DLBL command, see the CMS Command Reference manual for your VM system.

For more information on the TSO ALLOCATE command, see *TSO/E Command Reference*. The file can also be allocated to your TSO session by a DD statement in your logon procedure. For information on DD statements, see the JCL reference for your MVS system.

password Is an optional password assigned to the VSAM file by the DEFINE or ALTER command of Access Method Services. The password can consist of up to eight arbitrary characters entered immediately after a colon.

If you try to access a file that is password protected and you either do not supply a password or supply the wrong password, you receive an error return code. Under TSO, you are first prompted for the password, unless you disabled the prompt for the TSO session. If NOPROMPT is specified in your current profile, you receive the error return code.

File Formats and Keys

A VSAM file is defined as one of three formats:

- An *entry-sequenced* file contains variable-length records that are organized sequentially. New records are written at the end of the file. Existing records can be read or rewritten either sequentially or directly. Direct write operations may not change the length of a record.

  When an entry-sequenced file is accessed directly, the byte offset (relative byte address, or RBA) of a record is specified as the key. The RBA must be specified as a character vector containing up to 15 digits. The RBA is returned in the data variable when a key feedback (KF) request is specified in the control variable. The RBA is returned as a 10-character vector.

- A *relative-record* file contains fixed-length records that can be read or written either sequentially or directly. When accessed directly, the relative-record number (RRN) of the desired record must be specified as the key. This key is not contained in the record itself, but is derived solely from the position of the record within the file. Empty slots, where records were erased or not yet written, are counted when determining relative-record numbers. The first record in the file has an RRN of 1.

  The key feedback (KF) request returns the key of a relative-record file as a 10-character vector. You may want your program to manipulate this value. The APL2 execute (⍎) primitive function can be used to convert character numbers to numeric data, and the format (⍕) function can be used to convert numeric data to character.

- A *key-sequenced* file contains variable-length records and is organized according to a key field contained in each record. Key-sequenced files can be read or written either sequentially or directly. The length of an existing record can be changed when it is rewritten.

  When accessed directly, a record's key is specified as a character vector of any length up to the key length specified in the Access Method Services DEFINE command when the file was created. Keys can be recorded as System/370 packed decimal, binary, or other data formats. Keys not encoded in EBCDIC, however, must be translated to EBCDIC before they are specified in the control variable.

VSAM alternate index support is available for entry-sequenced and key-sequenced files when processed by AP 123.

Commands

Requests for file processing are made by specifying the control variable. Figure 58 contains a description of each possible command and the corresponding value to specify in the control variable.

Command	Operation
'*OR ddname*[*:password*]'	Open file for read
'*OW ddname*[*:password*]'	Open file for write
'*OU ddname*[*:password*]'	Open file for update
'*OC ddname*[*:password*]'	Open and clear reusable file
'*R*'	Sequential read
'*RU*'	Sequential read for update
'*R*:*key*'	Direct read
'*RU*:*key*'	Direct read for update
'*W*'	Write a record
'*W*:*key*'	Write new record (relative record)
'*E*:*key*'	Erase a record (key sequenced or relative)
'*PO*[:*key*]'	Position record pointer
'*KF*'	Key feedback
'*C*'	Close file
'*T*'	Transfer data without translation (default)
'*T*1'	Translate to or from VS APL
'*T*2'	Translate to or from EBCDIC

Figure 58. Commands for VSAM File Operations

Communication Procedure

Figure 59 shows a sample session to read and write records on a VSAM file.

```
      123 ⎕SVO¨ 'CTL123' 'DAT123' ⍝ offer to share
2 2                  ←──────────Degree of Coupling OK
      0 0 1 1 ⎕SVC 'CTL123'       ⍝ set access control
0 0 1 1
      CTL123←'OC MYFILE'          ⍝ open and clear reusable file
      CTL123
0 0                  ←──────────Return code OK
      DAT123←'FIRST RECORD'
      CTL123←'W'                  ⍝ write first record
      CTL123
0 0                  ←──────────Return code OK
      DAT123←'SECOND RECORD'
      CTL123←'W'                  ⍝ write second record
      CTL123
0 0                  ←──────────Return code OK
      CTL123←'C'                  ⍝ close file
      CTL123
0 0                  ←──────────Return code OK
      CTL123←'OU MYFILE'          ⍝ reopen for update
      CTL123
0 0                  ←──────────Return code OK
      CTL123←'RU'                 ⍝ read for update
      CTL123
0 0                  ←──────────Return code OK
      HOLDAREA←DAT123             ⍝ save record contents
      .
      .              ←──────────Manipulate HOLDAREA
      .
      DAT123←HOLDAREA
      CTL123←'W'                  ⍝ rewrite record
      CTL123
0 0                  ←──────────Return code OK
      CTL123←'C'                  ⍝ close file
      CTL123
0 0                  ←──────────Return code OK
      ⎕SVR¨ 'CTL123' 'DAT123'     ⍝ retract variables
2 2
```

Figure 59. Sample Session to Communicate with AP 123

The steps below summarize the procedure for communicating with AP 123.

1. Offer to share a pair of variables. Either variable can be offered first. However, AP 123 does not match your DAT offer until you offer a corresponding control variable.

   Verify the degree of coupling, but do not try to obtain a return code. Because neither partner has specified a value yet, you are interlocked if you try to use the control variable.

2. Set the access control.

3. Open the VSAM file.

   A file can be opened for read, write, update, or clear. Check the return code from the open request.

4. Process the file.

   Records can be read, written, erased, or replaced—either sequentially or directly. How you process the file depends on how the file was opened (see Figure 60). Check the return code after each service request is issued.

5. Close the file.

   The file should be closed explicitly by a close service request. Check the return code.

   Note: Although the file can be closed implicitly by retracting the control variable, data integrity is exposed if you do so. You cannot check the return code from an implicit close, which may indicate an I/O error on the last buffer written.

6. Steps 2, 3, and 4 can be repeated with the same file or a different file without retracting and reoffering the pair of variables.

Opening a VSAM File

A VSAM file can be opened for read only, for reading and writing, or for updating (reading, writing, replacing, and erasing records). Figure 60 shows the file processing requests allowable with each OPEN option.

If you do not specify a password and one is required, you are prompted, perhaps more than once, for one.

Figure 60 (Page 1 of 2). Cross-Reference of OPEN and Processing Requests for AP 123

OPEN Option	Possible Record Processing
'*OR ddname*[*:password*]' Open file for reading only.	'*R*[**:***key*]' Read sequentially; or, if a key is specified, read directly.
'*OW ddname*[*:password*]' Open file for reading or writing new records.	'*R*[**:***key*]' Read file sequentially; or, if a key is specified, read directly. '*W*[**:***key*]' Write a record. • A key must be specified (except during update) to write a record to a relative-record data set. • A key must *not* be specified to write a record to an entry-sequenced or key-sequenced data set.
'*OU ddname*[*:password*]' Open file for update (reading, writing, erasing, and replacing records).	'*R*[**:***key*]' Read sequentially [or directly]. '*W*[**:***key*]' Write a record. (Same conditions as '*W*[**:***key*]' above.) '*RU*[**:***key*]' Read for update. (The next WRITE operation replaces this record.) Either the next sequential record is read, or the record belonging to the specified key is read. '*E***:***key*' Erase the record whose key is specified.

Figure 60 (Page 2 of 2). Cross-Reference of OPEN and Processing Requests for AP 123

OPEN Option	Possible Record Processing
'*OC ddname*[:*password*]'	'*W*[**:***key*]'
Open and clear contents of reusable file.	The only service request allowed with an OPEN request of *OC* is to write records in ascending sequence. A file opened with the *OC* request must have been defined with the REUSE attribute.

Processing a VSAM File

Processing a VSAM file includes reading, writing, updating, and erasing records from the file. The processing options available with this processor are included in Figure 60. The processing options are dependent upon how the file was opened.

Reading a File

All three types of VSAM files can be read sequentially or directly. To read sequentially, you merely specify the control variable with a value of '*R*'. To read directly, you specify a key, as '*R*:*key*'. The procedure consists of three steps:

1. Specify the control variable.
2. Reference the control variable (obtain the return code).
3. Reference the data variable.

Note: When you read a VSAM file, you must reference the data variable if the return code from the read request was 0 0. If you omit the reference of the data variable, the return code from the next read request is 1 42. To recover, merely reference the data variable before the next read request.

If the return code from a read request is other than 0 0 or 1 42, do not reference the data variable. If you do, you probably get a record from an earlier read request.

If the internal encoding of the data records in the file is other than EBCDIC, you may need to convert the data so that APL2 can process it. The public library workspace UTILITY contains defined functions to make these conversions. Also, translate options are available as service requests (see *T*, *T*1, and *T*2).

Writing a File

New records can be written to all three types of VSAM files. Note that, when you write to a VSAM file, you must specify a value in the data variable before specifying the write request in the control variable. If you fail to specify the data variable first, the return code from the write request is 1 33. To recover, merely specify the data variable and reissue the write request.

The format of the write service request is:

*CTL*123←'*W*:[*key*]'

You do not specify a key if the file is key sequenced or entry sequenced. You must specify the key if the file is relative-record, because the key is not embedded in the record itself.

The procedure to write a record consists of three steps:

1. Specify the data variable with the record contents.
2. Specify the control variable with the write request.
3. Reference the control variable to obtain the return code.

Replacing a Record

Records can be replaced in any type of VSAM file with AP 123 by opening the file for update (*'OU ddname'*) and issuing the read-for-update service request (*'RU*[:*key*]*'*). Omit the key for sequential update; specify the key for direct update. The procedure consists of five steps:

1. Specify the control variable with the *OU* request.

   Reference the control variable for the return code.

2. Specify the control variable with the *RU* request.

   Reference the control variable for the return code.

3. Reference the data variable to read the record.

4. Specify the data variable with new record contents.

5. Specify the control variable with the *'W'* write request without a key.

   Reference the control variable for the return code.

Restriction on Entry-Sequenced or Relative-Record Files: When replacing a record on an entry-sequenced or relative-record VSAM file, the record that is written must be the same length as the record that was retrieved.

Erasing (Deleting) a Record

Records can be erased from a VSAM file with AP 123, provided that the file is not an entry-sequenced file. The file must be opened for update (*'OU* ddname*'*). You must specify the key of the record to be erased in the service request. The format of the request to erase a record in a key-sequenced or relative-record data set is:

```
CTL123←'E:key'
```

Check the return code from the erase request.

Obtaining the Key of the Last I/O Operation

This service request, which is specified as:

```
CTL123←'KF'
```

can be issued only after a successful read, write, or erase request. The file can be open for read, write, update, or clear.

The key is returned in the data variable. For entry-sequenced and relative-record data sets, the key is returned as a character vector of 10 integers. For a key-sequenced data set, you must convert the key to EBCDIC if it is not already in that encoding.

Positioning the Record Pointer

When you read a file sequentially, a pointer is automatically set, pointing to the next record to be read. You can explicitly set the record pointer to another point in the file by issuing this service request:

```
CTL123←'PO[:key]'
```

When a key is specified and found in the file, the record pointer is positioned at the specified record. When a key is omitted, the pointer is positioned at the beginning of the file.

Key-Sequenced Data Set: If the specified key is shorter than the key length of the VSAM file, the pointer is positioned at the first record whose key starts with the value entered as the key.

If no match for the specified key is found in the file:

- The return code is 8 16 (record not found).
- The pointer is positioned at the first record whose key is higher than the specified key.

If the specified key is higher than any in the file, the return code in the control variable is 8 4 (end of file).

You may need to convert numeric keys to System/370 data format. Conversion functions are available in the public workspace *UTILITY*.

The example below uses the function *IO* to position the record pointer to the record whose key is defined as a 4-byte binary number, 50.

```
CTL123←'PO:',4 IO 50
```

Entry-Sequenced Data Sets: Zero (0) is the relative byte address (RBA) of the first record. The only efficient way for you to determine the RBA of subsequent records is to issue a key feedback (*KF*) request.

Relative-Record Data Sets: The key of the first record of a relative-record data set is 1. The key of the second record is 2; that of the third record is 3. To position the pointer to record 8, issue this request:

```
CTL123←'PO:8'
```

Specifying Character Conversion

If the data in a file is encoded in EBCDIC, you need no character conversion to process records under APL2, but you may want to specify a translate request of `'T2'` for records you read or write with AP 123.

AP 123 provides three translate options. A translate service request for a shared variable pair can be issued at any time; it remains in effect for that pair until a new translate request is issued or the control variable is retracted. The options are:

`'T2'` Keys and data are assumed to be encoded in EBCDIC, the same encoding used by APL2.

This translate option is the same as the EBCD conversion option described in Appendix D, "Auxiliary Processor Conversion Options" on page 370.

`'T1'` Keys and data are assumed to be encoded in VS APL internal format. They are translated to and from EBCDIC when records are read and written.

This translate option is the same as the COD1 conversion option.

`'T'` This is the default option. Data is transferred byte-for-byte without translation.

This translate option is the same as the BYTE conversion option.

Your VSAM file can require some other character conversion. The public library workspace UTILITY contains several data conversion functions that can be used to convert keys and data to various System/370 data types.

Closing a VSAM File

When you complete processing a VSAM file with AP 123, close the file by doing either of the following:

- Specify the control variable with the close service request.

  ```
  CTL123←'C'
  ```

  Check the return code from the explicit close.
- Retract the control variable.

Cautions

Always explicitly close a VSAM file by issuing the close service request (`'C'`), and check the return code. If you allow the file to close implicitly (the control variable being retracted), you cannot check the return code from the close and the integrity of your data is exposed. If, for example, an I/O error occurs on the last buffer written, your file would be incomplete and you would not know it.

CMS: Under CMS, VSAM is supported as it is under DOS. The CMS/DOS and APL2 environments impose the following restrictions:

1. AP 100 commands must be acceptable to CMS/DOS.
2. SET DOS ON, which must be set to issue the DLBL command, and must not be issued from within your APL2 session. To do so causes your session to abort, and you have to re-IPL CMS.
3. SET DOS OFF must not be issued while VSAM files are open. You lose your data.

   If the command is issued from within APL2, no VSAM files can be accessed after the setting takes effect.
4. After the first use of CMS VSAM, you must issue the ASSGN command whenever you issue a DLBL command.
5. When your session ends normally, AP 123 issues the CMS command DMSVSR to reset the CMS VSAM environment.

For more information about CMS/DOS and VSAM, see the appropriate user's guide for your system.

TSO: Depending on VSAM resource availability, you may be put on a wait queue if you try to access VSAM libraries used by other TSO users. This can happen when your AP 123 file is opened for write (*OW*) or update (*OU*). Repeated attention procedures may cause AP 123 to be unavailable for the remainder of your APL2 session.

Return Codes

Because initial values of the shared variables are ignored by AP 123, a return code is not returned in these variables as a result of the offer. However, after a service request is issued, the control variable contains a return code consisting of a two-item integer vector.

- If both items are 0, the service request has successfully completed.
- If the first item is 1, the return code was issued by AP 123. The second item indicates the type of error. Figure 61 lists the codes and descriptions returned by the auxiliary processor.
- If the first item is not 1, the return code was issued by VSAM. Figure 62 on page 206 lists some of the more common VSAM return codes.

  For more information about VSAM return codes, see *OS/VS Virtual Storage Access Method (VSAM) Programming Guide*.

Figure 61. Return Codes from AP 123

Code	Description
1 12	Syntax error in the value specified in the control variable.
1 13	You issued a service request to open the file. The file is already open. No action is required. Processing continues.
1 15	The *RU*, *W*, or *E* service request specified is not permitted by the current open mode. *RU* and *E* require an *OU* open. *W* requires *OW*, *OU*, or *OC*.
1 16	The service request is not valid for the type of VSAM file being processed. ERASE was specified for an entry-sequenced data set.
1 17	Length of record key specified in a service request was too long, or it was too short on a *RU key* request. Note that blanks are included in the count of the key length.
1 18	VSAM MODCB error.
1 19	VSAM SHOWCB error.
1 20	Something other than a character vector was specified in the data variable.
1 21	Data variable was specified with an incorrect record length.
1 22	You issued a service request to process a file that is not currently open.
1 27	VSAM TESTCB error.
1 32	Insufficient free storage for I/O buffers. **Action:** Restart the APL2 session with a larger value in the FREESIZE invocation option. If the resultant freespace size or workspace size is too small: • **CMS:** In CP mode, define a larger virtual machine size; re-IPL CMS; and restart APL2. • **TSO:** Logoff; log on with a larger region size specified in the LOGON command; and restart APL2.
1 33	No value has been assigned to the data variable for a write request.
1 42	You set access control on the data variable, but you have not yet used it for an earlier request. (The first read request resulted in a return code of 0 0.)
1 45	VSAM GENCB error.
1 48	Invalid sequence of service requests. • A key feedback (*KF*) request followed an open request or an error. • A relative-record write without a key (*W*) was requested without a preceding read-for-update request (*RU*).

Figure 62. Selected VSAM Return Codes Applicable to AP 123

Code	Description
0 8	More than one record has the specified key. The first is returned.
4 116	The file was not properly closed after the last usage. **Action:** Use the Access Method Services VERIFY command to check the status of the file.
8 4	End of file; or specified key is greater than the highest key on the file.
8 8	Duplicate key.
8 16	Record not found.
8 20	Record is in use by another user.
8 28	Data set is full.
8 32	Invalid relative byte address.
8 40	Insufficient virtual storage.
8 88	Sequential read requested without prior positioning.
8 96	Attempt to change key of record.
8 100	Attempt to change length of record on an entry-sequenced or relative-record data set.
8 110	Attempt to open an empty file for read or update.
8 128	Attempt to open file that has not been properly allocated.
8 136	Insufficient virtual storage.
8 152	Password error.
8 168	The data set is not available in the mode requested. It is in use by another job.
8 192	Invalid relative record number.

Chapter 20. AP 124—Text Display Auxiliary Processor

Use the Text Display Auxiliary Processor (AP 124), through an APL application program, to control the screen of an IBM 3270 Information Display System terminal. It allows your application to:

- Define a logical screen
- Format the logical screen into text fields
- Write text to the fields
- Prompt for input, and determine the type of input
- Read text from the fields

AP 124 provides only text support. It does not provide graphics support. AP 126 does provide graphics support for 3270's, but requires the GDDM program product.

Shared Variable Overview

Figure 63 provides an overview for sharing variables with AP 124.

Figure 63. Shared Variable Overview for AP 124

SV Protocol	AP 124 Conventions
General	Two variables—control and data. The control variable is used to pass operation codes and some parameters to the AP, and to pass return codes back from the AP. The data variable is used to pass additional information related to the operation codes. The type of the data and whether it is passed to the AP or received from it depends on the operation code.
Maximum Number of Shared Variables	6 pairs
Names	Must start with *CTL* and *DAT* or with *C* and *D*. Suffixes pair the variables. To avoid ambiguity, AP 124 does not match offers of variables whose names begin with *CAT* or *DTL*. Names cannot exceed 12 characters.
Initial Values	No special handling. Initial values, if specified, are treated like any other operation.
Subsequent Values	Control variable: required. Specify one AP 124 operation as described below. Some operations require that information be provided in the data variable first. Data variable: may be required depending on the operation. May return data depending on the operation.
Data Types Supported	Control variable: simple numeric scalar or vector. Data variable: depends on the operation.
Access Control	Control variable: 1 0 1 0 Data variable: 0 0 0 0

Your APL application requests screen management services by assigning to the control variable a numeric scalar or vector that specifies the requested action. In response, the auxiliary processor issues a return code in the control variable indi-

cating whether the requested action was successful. If data is to be sent to or from the screen, it is transmitted in the data variable.

When a screen management service request is issued, the IBM 3270 enters full-screen mode under control of AP 124. When this happens, your application is in full control of the screen and can issue additional screen management requests. When full-screen mode is interrupted, the IBM 3270 is returned either to host terminal (CMS or TSO) line by line mode, or to session manager full-screen mode. In this mode, the screen is under control of CMS or TSO or the session manager, and the standard formatted screen is displayed.

Full-screen mode is interrupted if:

- An interrupt signal is issued (by pressing the PA1 key)
- A normal (nonscreen management) input/output request is issued
- An error message is issued

When the IBM 3270 leaves full-screen mode and returns to Host terminal mode, the current screen image is saved. It is restored when the next screen management service request is issued.

Since APL2 issues a normal input/output request when a defined function completes processing, the duration of the full-screen image can be extremely short. Your application must provide an intervening delay (such as a screen management read request) to extend the duration of the full-screen image.

Understanding Screen Management

To use AP 124, certain general information about the screen and its attributes is required. This information is provided in the discussions below.

Logical Screens

AP 124 is capable of handling multiple logical screens, each of which is defined to be exactly the size of the physical screen. Each logical screen is accessed through a pair of shared variables. When you share a pair of variables, a logical screen is created, and upon retraction of the variables, the logical screen is destroyed. Each pair of variables is kept logically separate from the other variables, and the limit to the number of logical screens that may be open at any time is mainly controlled by the amount of free storage available to the auxiliary processor.

Screen Fields

AP 124 logically views the screen of an IBM 3270 display device in terms of rectangular areas called screen fields. It is only in these areas that data can be entered or displayed. Each screen field has a starting position, a width, and height the application defines when it formats the screen. The starting position of each field is the row and column address of the upper left-hand character in the field. (The upper left-hand position of the screen is row 1 column 1.)

Field Attributes

Each screen field has associated with it certain field attributes that qualify its content. For instance, a field attribute can indicate that a field is to contain alphabetic or numeric data or that its characters are to be displayed in a particular color. In addition, input fields can have additional attributes that are not meaningful for output-only fields. Your application can set the attributes for a field through various screen management service requests. If attributes are not explicitly set, the auxiliary processor supplies default values.

The IBM 3270 display system notes the attributes of a screen field by preceding the field with a column of attribute characters. These characters appear as blanks and are not considered part of the field. If a screen field begins in column 1, its attribute characters are wrapped around the screen; that is, the characters occupy column 80 in the previous row. A screen field that begins in row 1, column 1, has an attribute character in column 80 on the bottom row of the screen.

It is generally good practice to leave at least one column for attribute characters between adjacent fields. Otherwise, the column of attribute characters preceding the right-hand field obscures any data written to the last column of the left-hand field.

Communications Procedure

The following discussions describe the operations that can be made through AP 124. The discussions frequently indicate various representative items as follows:

- *cvar* represents the name of the control variable.
- *dvar* represents the name of the data variable.

These items should be replaced with the appropriate names when you issue your service requests.

Because multiple pairs of variables are allowed to be shared with AP 124, the processor must have a way to associate the pairs of variables. This is accomplished by having a rigid naming convention for the variables:

- Control variables can begin with the letters *CTL*. If one does, AP 124 associates it with a data variable beginning with the letters *DAT* and having the same letters following the first 3 (with the exceptions noted below.)
- Control variables can also begin with the letter *C*. If one does, AP 124 associates it with a data variable beginning with the letter *D* and having the same letters following the first (with the exceptions noted below.)
- To avoid possible ambiguity between the first two rules, control variables cannot begin with *CAT*, and data variables cannot begin with *DTL*.
- In any case, names cannot exceed 12 characters.

For instance, the following variables are paired together: *CTLXYZ* and *DATXYZ*; *CTL* and *DAT*; and *C124* and *D124*.

Before any useful operations can be issued with AP 124, both a control and data variable must be shared. The discussions assume that sharing has been completed for the control and data variables.

Screen Management Commands

Figure 64 summarizes the valid commands that can be specified to the control variable to request service from AP 124. Figure 64 shows the values that should be specified in both the control and data variables.

Figure 64. Screen Management Commands

cvar variable	*dvar* variable	Description
0 or 0,*n*		Delayed clear of screen
1	*format*	Format the screen
1,*fieldnum*	*format*	Reformat selected fields
2,*fieldnum*	*data*	Immediate write to screen
3 or 3,0		Read and wait
4,*fieldnum*	*data*	Buffered write to screen
5,fieldnum		Get data
6,*fieldnum*	*type*	Change field type
7,*fieldnum*	*color*	Change field color or intensity
8,2		Get device information
8,*n*		(Ignored for n omitted or not 2)
9		Read screen format
11 or 11,0		Sound alarm (delayed)
11,1		Sound alarm (immediate)
11,2		Cancel delayed alarm
12	*position*	Set the cursor
16,*fieldnum*	*attribute*	Change input field attributes
20		Erase the screen

Delayed Clear of the Screen

A request for a delayed clearing of the screen is made through the following assignment:

```
cvar ← 0
```

or

```
cvar ← 0 n
```

where *n* is an integer that is allowed for compatibility with other implementations of AP 124 but that is ignored.

The screen is cleared on the next operation requiring a screen update. In CMS, this operation eliminates the `MORE...` state before the full-screen appears.

Formatting the Screen

Format: This operation is used to describe the position and size of all screen fields.

A request to format the display screen is made through the following assignments:

```
dvar ← format
cvar ← 1
```

where *dvar* is assigned a four-, five-, or six-column numeric matrix with one row for each field to be formatted. If only one field is to be formatted, it can be a numeric vector, but is treated as a one row matrix. The first or only row of the matrix defines the first field, the second row defines the second field, and so on. The first screen field becomes field number 1 for subsequent screen management operations. The second screen field becomes field number 2, and so on.

Each row of *format* consists of the following fields:

row col height width [*type*] [*color*]

where

row Indicates the display screen row (origin 1) at which the field begins.

col Indicates the display screen column (origin 1) at which the field begins.

height Indicates the height (number of rows) of the field.

width Indicates the width (number of columns) of the field.

type Indicates the type attribute of the field.

- `0` Character input/output allowed
- `1` Character output allowed, numeric and uppercase input allowed

  On terminals with the numeric protection feature, this field accepts only numeric and uppercase input. On terminals that do not have the numeric protection feature, any characters can be entered, and type `1` is thus equivalent to type `0`.
- `2` Character output only
- `3` Character output/light pen interruptible
- `4` Character output/light pen selectable

The default field type is `2`—character output.

color Indicates the color or intensity of the field. Values from 0 through 255 are accepted.

- `0` Off or do not display
- `1` Normal intensity
- `2` Highlighted intensity

On terminals without color support, values larger than 2 but less than 256 are accepted and treated as either normal or intensified in groups of 8. If the quotient of the number divided by 8 is even, normal display is used. For odd quotients, intensified display is used.

Most 3270 terminals with color support provide 7 character colors and do not permit background control except in graphics mode. On this

class of terminals the remainder of dividing the color number by 8 is used to select a color as follows:

0 Device default color, either normal or intensified
1 Blue
2 Green
3 Turquoise
4 Red
5 Pink
6 Yellow
7 White

The quotient from the division by 8 produces normal display if even, or reverse video display if odd (default intensified display if the quotient is odd and the remainder is zero).

3270 architecture defines interfaces for up to 14 character colors as well as the same number of background colors. On any devices or emulators where this full capability is implemented, AP 124 provides a mapping for values greater than 2 based on the quotient and remainder when divided by 16. (But values of 0, 1, or 2 continue to be treated as default intensities as described above.) The quotient specifies the background color, and the remainder specifies the character color. Both are interpreted as follows:

0 Black
1 Dark Blue
2 Green
3 Turquoise
4 Red
5 Purple
6 Yellow
7 Grey
8 Grey
9 Blue
10 Light Green
11 Light Turquoise
12 Pink
13 Purple
14 Orange
15 White

The default display attribute is 1—normal intensity.

The starting position (upper left-hand corner) of a field must be a valid screen position. The enclosed area described by the starting position and field height and width must not extend beyond the screen boundaries. Fields with height 1 can wrap to succeeding rows.

A zero in any of the first 4 elements in any row effectively undefines a field. This removes the field from the formatted screen area but does not change the field numbers associated with the remaining fields.

When a display screen is formatted, the new screen format overlays the previous screen format. The data in all fields are initialized to blanks. The new screen format is transmitted to the display screen on the next operation requiring a screen update.

After a pair of control and data variables are shared, but before a successful format operation is made, a default format is used by the AP. This format has one field that starts in screen position row 1 column 1, and has a height and width equal to the physical screen dimensions.

The following example shows how to do simple formatting. Here, two screen fields are defined. The first field is to begin at row 3 column 5; it is to have a height of 10 rows and a width of 15 columns. The second field begins at row 3 column 30; it is to have a height of 12 rows and a width of 25 columns. Start by defining the format matrix MATX:

```
      MATX←2 4⍴3 5 10 15, 3 30 12 25
      MATX
3  5 10 15
3 30 12 25
```

Now use MATX as an argument to a function FORMAT, in which the appropriate formatting service requests are issued:

```
      ∇ FORMAT MAT;X;C
[1]   ⍝ SHARE VARIABLES WITH AP 124 AND TEST
[2]   X←124 ⎕SVO ¨ 'CTL' 'DAT'
[3]   →(2≠X)/FAIL
[4]   DAT←MAT
[5]   ⍝ ISSUE REQUEST AND TEST RETURN CODE
[6]   CTL←1
[7]   →(0=C←CTL)/0
[8]   'REQUEST NOT HONORED, CODE IS ',⍕C
[9]   →0
[10]  FAIL:'OFFER TO SHARE VARIABLE NOT MATCHED'
[11]  ∇
      FORMAT MATX
```

Reformatting Selected Fields: This operation is used to redefine selected screen fields. A request to format the display screen is made through the following assignments:

```
dvar ← format
cvar ← 1, fieldnum
```

where *format* is a four-, five-, or six-column numeric matrix, each row of which contains the field definition for the corresponding field number in *fieldnum*. *fieldnum* is a numeric vector of one or more field numbers. Each field number represents a screen field to be reformatted.

The format of the format matrix is the same as in the Format operation described above. When a display screen is reformatted, the new screen format definitions are merged with the pre-existing format definitions. The new screen format is transmitted to the display screen on the next operation requiring a screen update.

The reformat operation cannot be used to format additional field numbers beyond the pre-existing format definition. For instance, if in the original format screen definition, 17 fields were defined, the reformat operation may not be used to reformat a field number 18. If this effect is desired, then on the original format definition, the format matrix should contain some additional undefined rows (that is, one or more zero elements) at the end of the format matrix. The reformat operation can then subsequently be used to reformat these undefined fields to valid field definitions.

If a screen is formatted and then subsequently reformatted so that all fields are undefined, the screen format reverts to the default format. The AP ensures that the screen is never in a totally undefined state.

Immediate Write of Data to Screen

A request to write to the screen is made through the following assignments:

```
dvar ← data
cvar ← 2, fieldnum
```

where *data* is a matrix of characters, each row of which contains the data for the corresponding field number in *fieldnum*. *fieldnum* is a numeric vector of one or more field numbers. Each field number represents a screen field to be written and corresponds to the respective row in the formatting operation matrix that defined that field.

When a write operation is processed, the auxiliary processor maps the data in the first row of *data* into the screen field represented by the first element of *fieldnum*, the second row of *data* into the screen field represented by the second element in *fieldnum*, and so on. The auxiliary processor automatically skips over any unformatted areas of the screen. If a field number in *fieldnum* is larger than the number of rows specified in the format operation matrix, or if the field number does not currently apply to a defined field, that field number and the corresponding row of data are ignored. Similarly, if *data* has more rows than field numbers in *fieldnum*, the extra rows are ignored. Finally, if more field numbers are provided in *fieldnum* than there are rows in *data*, the extra fields are filled with nulls or blanks.

Each line of a screen field is filled with data from left to right, starting at the beginning of the field. Any trailing blanks padding out a field line are replaced with nulls on the screen, subject to the setting of the field attributes by the field attribute operation. If too much or too little data is specified in a row of *data*, the data is respectively truncated or extended with nulls or blanks to fit the full field area.

All non-3270 characters are displayed as double quotation marks (").

Erasing: The write request also doubles as an erase operation, where each erased field is filled with null characters. If *data* is empty, all fields identified in *fieldnum* are erased. If *data* contains fewer rows than fields in *fieldnum*, the extra rows are erased. Fields not identified in *fieldnum* are not affected by this operation.

Read and Wait

This step serves three purposes: first, it directs the auxiliary processor to wait for the user to complete the current input operation; second, it directs the auxiliary processor to return information about the current screen; third, based on how the user completed input, it can direct the auxiliary processor to read all the defined fields on the screen into an internal data area. Data is read only if the user completed the input operation using the ENTER key or a function key. (Note that modified data is not available with PA keys and the CLEAR key.)

An application requests this operation by specifying:

```
cvar ← 3
```

or

```
cvar ← 3 0
```

The two forms are equivalent.

Check the return code in the control variable, then reference the data variable to determine how the input was completed:

```
code ← cvar
data ← dvar
```

In response, *data* contains a vector of one or more numbers that indicate:

- The type of action that completed the current input operation. This is returned no matter how input was completed.
- A modifier for the type of action. This information is returned if the user completed the input operation using a PF or PA key or the ENTER key. For a PF or PA key it is the number of the key used. For the ENTER key, 0 is always returned.
- The current cursor position. This information is returned if the user completed the input operation using the ENTER key or a function key.
- The field numbers of the modified fields. A field is modified when the user enters data into it and stays modified until the application writes to the screen. This information is returned if the user completed the input operation using the ENTER key or a function key.

The possible values in this step are shown in Figure 65.

Figure 65. Data Variable Returned by Read and Test Requests

	Vector Elements					
	1	2	3	4	5	6...
User Action	Completion Code	Modifier	Cursor Field[1]	Cursor Row[2]	Cursor Column[2]	Modified Fields
Enter Key	0	0	*fldnum*	*row*	*column*	*fldnums* [3]
Function Keys	1	1-24	*fldnum*	*row*	*column*	*fldnums* [3]
PA Keys	4	1-3 [4]	-	-	-	-
Clear Key	5 [5]	-	-	-	-	-
No Input	6	-	-	-	-	-

Notes:

1. Indicates the field number of the field containing the cursor when input was completed. If this element is 0, the cursor was not found in a defined field. Elements 4 and 5 are then physical (relative to row 1, column 1) position indicators.
2. Indicates the position of the cursor when input was completed. The position is indicated by the row and column relative to the first row and column position in the field.
3. Indicates one field number for each field modified since the last preceding write operation. If no fields were modified, this element is not returned.
4. Pressing the PA1 key generates a weak interrupt signal. The function is suspended, and consequently the screen is returned to Host Terminal Mode. For CMS: The auxiliary processor enters CP mode when the PA1 key is pressed. The CP command BEGIN can be used to resume full-screen mode without completing input.
5. If the Clear key was used, the current screen format and field attributes are reestablished at the next read or write operation. For TSO: Some OS terminal access methods may not pass back the Clear key attention identifier to the APL2 full-screen manager. Consequently, PA2 forces a regeneration of the entire screen image, independent of the APL application program.

Writing to the Screen

To avoid unnecessary flickering in an interactive application, the display is not updated when this type of write request is issued. When requesting this type of write request, the screen is updated when the auxiliary processor encounters the next request to read and wait for input.

A request to make a delayed write to the screen is made through the following assignments:

```
dvar ← data
cvar ← 4, fieldnum
```

where: *data* is a matrix of characters, each row of which contains the data for the corresponding field number in *fieldnum*. *fieldnum* is a numeric vector of one or more field numbers. Each field number represents a screen field to be written and corresponds to the respective row in the formatting operation matrix that defined that field, following the same rules as for an immediate write of data to the screen.

Getting Data

This step completes the read operation. It obtains data from one or more screen fields (usually the modified fields indicated in the read status vector) after a read and wait has been performed.

To request this step the application specifies:

```
cvar ← 5, fieldnum
code ← cvar
data ← dvar
```

where *fieldnum* is a numeric vector of one or more field numbers. Each field number represents a screen field to be read and corresponds to the respective row in the formatting operation matrix that defined that field.

When the operation is complete, *data* contains a matrix of characters, each row of which is the data for the corresponding field in *fieldnum*. If an undefined field number is specified in *fieldnum*, a row of blanks is returned. The matrix is padded to the right with blanks so that the number of columns is equal to the length of the longest field.

Note that the data returned in this step might be altered if the application issued any intervening write or format requests.

Modifying Field Attributes

These operations are used to explicitly set the attributes of one or more screen fields. Two kinds of attributes can be set:

- The type of data permitted in the field
- The color or display intensity of the field

These attributes are set as described below. In each of these descriptions the term *fieldnum* is a numeric vector of one or more field numbers. Each field number represents a screen field whose attributes are to be set and corresponds to the respective row in the format operation matrix that defined that field.

Modifying Field Type: This operation is used to indicate what type of data is permitted in a field. It takes effect at the next screen management read or write operation and applies until changed explicitly or until the screen is reformatted.

Your application can request this operation by specifying:

```
dvar ← type
cvar ← 6, fieldnum
```

where *type* is a numeric vector of field type indicators. Each value in *type* indicates the data type for the corresponding field in *fieldnum*. If type is a single value, it is the field type indicator for all the fields specified in *fieldnum*.

The field type indicators are:

0 Character input/output allowed

1 Character output allowed, numeric and uppercase input allowed

On terminals with the numeric protection feature this field accepts only numeric and uppercase input. On terminals that do not have the numeric protection

feature, any characters may be entered, and type 1 is thus equivalent to type 0.

2 Character output only

3 Character output/light pen interruptible

4 Character output/light pen selectable

The default field type indicator is 2—character output.

Modifying Display Attribute: This operation sets the color or intensity of one or more fields. It takes effect at the next screen management read or write operation and applies until changed explicitly or until the screen is reformatted.

Your application can request this operation by specifying:

```
dvar ← color
cvar ← 7, fieldnum
```

where *color* is a numeric vector of values between 0 and 255. Each item in *color* indicates the color or intensity of the corresponding field in *fieldnum*. If *color* is a single item, it is the color or intensity attribute for all the fields specified in *fieldnum*.

The color or intensity attribute values are as defined for the format request.

Returning Screen Information

AP 124 provides an easy way of determining certain information about the screen:

```
cvar ← 8 2
code ← cvar
data ← dvar
```

When the operation is complete, *data* contains the following information:

```
data[1] - 1 if the display supports APL, else 0
data[2] - 0
data[3] - 1 if the display supports color, else 0
data[4] - 1 if an alarm request is pending, else 0
data[5] - 0
data[6] - 0
```

For compatibility with other implementations of AP 124, you can also issue the operation:

```
cvar ← 8 n
```

where n is omitted or not 2. The only effect is that 0 is returned in the *cvar*.

Reading the Screen Format

AP 124 provides an easy way of determining what format matrix it is currently using. A request to display the current format matrix is made by the following assignment:

```
cvar ← 9
code ← cvar
format ← dvar
```

When the operation is complete, *format* contains the current format matrix. This matrix contains 1 row for each field up to the highest valid field defined and is 6 columns wide. If a new format matrix is pending, the new format matrix is returned.

After a pair of control and data variables have been shared, but before a successful format request has been made, a default format is used by the AP. This format has one field that starts in screen position row 1 column 1, and has a height and width equal to the physical screen dimensions. Reading the default format before formatting the screen allows you to tailor your application to the actual screen size available.

If a screen is formatted and then subsequently reformatted so that all fields are undefined, the screen format reverts to the default format. The AP ensures that the screen is never in a totally undefined state.

Sounding the Alarm

In your screen operations, you may find it useful at times to sound an audible alarm. For instance, you might want an alarm sounded when a field of particular importance is filled or to generate a warning to the user of an application. Several operations are available to control the sounding of the alarm.

To request that the alarm be sounded the next time the screen is updated, specify:

```
cvar ← 11
```

or

```
cvar ← 11 0
```

These are requests for a delayed alarm, which takes effect at the next screen management read or write request. The two forms are equivalent. To find out whether an alarm is pending, specify `cvar ← 8 2` and examine the fourth element returned in *dvar*.

To request that the alarm be sounded immediately, specify:

```
cvar ← 11 1
```

To cancel a request for a delayed alarm, specify:

```
cvar ← 11 2
```

Setting the Cursor

This operation positions the cursor during the next screen update operation. Your application can request this operation by specifying:

```
dvar ← position
cvar ← 12
```

where *position* is a three-element vector whose first element indicates the number of the screen field in which the cursor is to be positioned. The second and third elements contain respectively the relative row and column position of the cursor in the field. If the first element is zero, the second and third elements are interpreted as the row and column position of the cursor from the upper left-hand position on the screen.

If either of the last two elements of *position* is zero or negative, the position of the cursor is not changed in the next read, write, or erase operation. (This too, is the default condition until you issue this service request.)

When the auxiliary processor is first invoked, the cursor appears on the screen in position row 1 column 1. The cursor defaults to this position after any new screen format takes effect.

Modifying Input Field Attributes

This operation is used to modify additional attributes for one or more fields. It takes effect at the next screen management read or write operation and applies until changed explicitly or until the screen is reformatted. The additional attributes control the handling of (1) trailing nulls, and (2) the auto-skip feature. These attributes are only useful for input fields, but can be specified for any field.

Your application can request this operation by specifying:

```
dvar ← attribute
cvar ← 16, fieldnum
```

where *attribute* is a numeric vector of attribute indicators. Each value in *attribute* indicates the attribute of the corresponding field in *fieldnum*. If *attribute* is a single value, it is the attribute indicator for all the fields specified in *fieldnum*.

The attribute indicators are:

0 No autoskip or trailing blank processing
1 Autoskip, but no trailing blank processing
2 No autoskip, but trailing blank processing
3 Autoskip and trailing blank processing

The default input field attribute is 3—autoskip and trailing blank processing.

If autoskip is in effect, the cursor automatically jumps to the beginning of the next input field when the user types a character in the last position of the current field.

If trailing blank processing is specified, then all trailing blanks in data written to the field are converted to nulls upon presentation to the physical screen. This allows the terminal user to use the INSERT MODE key on the 3270 to insert data into the field. If this option is turned off, the user's data is not modified upon presentation to the 3270 (that is, trailing blanks remain true blanks on the screen).

Erasing the Screen

A request to erase the screen immediately is made by the following assignment:

```
cvar ← 20
```

The screen is erased as soon as the request is processed. Note that the delayed clear (operation 0) and erase screen operations apply only to the physical screen. They *do not* change the contents of any logical fields. This is in contrast to the write and immediate write operations that erase the screen by erasing the data that is in the logical fields.

Return Codes

Figure 66 lists and describes the codes returned in the control variable in response to a screen management service request. The codes are scalar integers. The auxiliary processor returns a code each time a value is specified for the control variable.

Figure 66. Return Codes from AP 124

Code	Description
0	Normal return—request is successful.
11	Control variable rank error.
12	Control variable length error.
13	Control variable domain error.
14	Invalid operation.
15	Request to position cursor in an undefined field.
21	Data variable rank error.
22	Data variable length error.
23	Data variable domain error.
24	Data variable not shared.
30	Invalid field number.
32	Defined field extends beyond the screen.
33	Reference outside field definition.
35	Light pen field starts in column 1.
36	Light pen field not contained in one physical screen line.
37	Invalid field type.
38	Invalid field color or intensity.
41	Data variable was not specified in the correct sequence.
42	Data variable was not referenced in the correct sequence.
52	Device is not a 3270.
53	Required storage unavailable -- increase the SHRSIZE parameter or decrease the WSSIZE parameter when invoking APL2.
89	Unknown shared variable return code.
91	Physical field table overflow.
92	Physical field table error; interrupt field not found.
94	Device not available.
95	Unexpected I/O error.
96	Chained CCW string not complete.
97	Bad 3270 orders in output data.
98	Full-screen support not available.
99	Unknown 3270 device error.

Chapter 21. AP 126—GDDM Processor

Use the GDDM processor, AP 126, to pass requests to the Graphical Data Display Manager (GDDM) Licensed Program. GDDM enables you to develop interactive APL2 applications that accept input and display output on full-screen panels. It also provides the means to create pictures and other graphics or to print information on many of IBM's display terminals and printers. *Graphical Data Display Manager General Information* gives examples of pictures and alphanumeric applications you can create with GDDM and contains a list of the terminals and printers supported by GDDM.

The GDDM auxiliary processor handles two types of requests:

- GDDM call requests, which direct GDDM to process one of the calls available through GDDM's application programmer interfaces (APIs). Each GDDM call has a corresponding numeric AP 126 code that must be used when you issue the GDDM call request.
- AP 126 commands, which change certain AP 126 options or issue queries about the processing.

  "AP 126 Commands" on page 228 describes these commands and their codes.

Associated Workspaces

The following APL2-supplied workspaces are associated with AP 126:

GRAPHPAK
FSM
*FSC*126
GDMX
CHARTX

The *GRAPHPAK* workspace contains functions that create standard graphics, such as pie charts, the APL apple, and bar graphs. In addition, functions are available that aid you in defining your own graphic displays.

The use of the *GRAPHPAK* workspace functions is described in *APL2 GRAPHPAK: User's Guide and Reference*.

The *FSM* workspace contains functions that facilitate use of GDDM, including its presentation graphics feature (PGF). *FSM* contains a defined function corresponding to each of the GDDM subroutines for full-screen management of graphics input and output.

For more information, type *DESCRIBE*, *HOW*, or *ABSTRACT* after loading the *FSM* workspace.

The *FSC*126 workspace contains functions to aid in the design of full-screen panels and in the definition of functions that present the panels to the user and retrieve the user's responses.

For more information, type *DESCRIBE*, *HOW*, or *ABSTRACT*, and process the *HOWSD* and *PRINTHOWSD* functions in the *FSC*126 workspace. For further information, see *APL2/370 Programming: Using the Supplied Routines*.

Licensed Program Requirements

Use of AP 126, the GDDM processor, requires:

- GDDM, Version 2 Release 3, or later. Base GDDM provides a low-level interface for alphanumerics and graphics.

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

- Presentation Graphics Feature (PGF) of the GDDM Program Product, if you want to use the higher-level GDDM calls available with this feature.

  For a detailed description of the PGF Program Product, see *GDDM-PGF Programming Reference*.

Shared Variable Overview

Figure 67 provides an overview for sharing variables with AP 126.

Figure 67. Shared Variable Overview for AP 126

SV Protocol	AP Conventions
General	Two variables—control and data. The control variable is used to pass requests and numeric parameters to AP 126 and to pass return codes and numeric parameters from resulting AP 126 and GDDM operations. The control variable must be a simple numeric scalar or vector. The data variable is used to pass character parameters from the workspace to GDDM and data from GDDM to the workspace. The data variable must be a simple character scalar or vector.
Maximum Number of Shared Variables	7 pairs
Names	Must start with *CTL* and *DAT*. Suffixes pair the variables. Names cannot exceed 11 characters.
Initial Values	No special handling. Initial values, if specified, are treated like any other request.
Subsequent Values	*CTL*: Required. Simple numeric vector. Specify one or more AP 126 commands or GDDM calls and associated numeric parameters. Reference a return code and numeric parameters returned by the request (a simple vector of at least five items). *DAT*: Specify a simple character vector as required by the GDDM call(s). Size of the character vector is not limited except as noted below. Reference character data returned as a result of the GDDM call(s) or AP 126 command(s). **Note:** The SHRSIZE invocation option or installation default and the amount of free space for AP 126 buffers may limit the size of a shared variable.
Data Types Supported	Simple numeric vector (*CTL* only). Simple character vector (*DAT* only).
Access Control	*CTL*: 1 1 1 1 *DAT*: 0 0 0 0

Data Formats

Each AP 126 request consists of:

- A required control variable (simple numeric scalar or vector). The first item is the request code; subsequent items are numeric input parameters for the request, concatenated to the request code. Whether numeric input parameters are required depends on the individual request.

  All codes for AP 126 commands are less than 0. All codes for GDDM calls are greater than 0. A request code of 0 causes no call to GDDM and no action by AP 126 (a no-op).

- Data variable (simple character scalar or vector) contains character input, if any, required by the request. Some requests do not require character data passed by the data variable. When the data variable is required, it must be specified before the control variable is specified.

Multiple requests can be made at one time by first catenating the character items (if any), assigning them to the data variable, then catenating the numeric items, and assigning them to the control variable.

Parameters for GDDM calls must be in the order required by GDDM. Any parameters in matrix form (for example, a table that defines screen fields) must be ravelled before they are passed to GDDM. Ravelling allows data for multiple parameters to be combined in a way that is easy for AP 126 to process.

GDDM requires a length parameter for all string and array data of variable length. AP 126 uses the length data to determine the parameters in the control and data variables.

Returned Values

When a single request is made, the value returned in the control variable is a return code vector (*rc-vector*) in the following form:

```
hrc rc rs nnp lcd np
```

hrc Highest return code from GDDM, or 20 if AP 126 found an error and did not pass the request to GDDM.

If a single request was made, this item is usually the same as the return code (*rc*) for the request.

rc Return code for the request. It indicates:

- 0 Request successfully processed.
- 1 AP 126 detected an error associated with the request. These return codes and their associated reason codes are described in Figure 76 on page 242.
- 2 Abend in APL2 outside of AP 126, but not in GDDM. *rs* is the reason code.

4, 8, 12, 16 Severity code for an error detected by GDDM.

For 4, 8, or 12, *rs* contains the GDDM return code corresponding to the GDDM message number.

For 16, GDDM abnormally terminated and *rs* indicates the abnormal termination (ABEND) code.

rs Reason code associated with the return code (*rc*) for the request. For the meanings of the return and reason codes, see Figure 76 on page 242.

nnp Number of numeric parameters. These parameters are returned as the last items of the vector, as indicated by *np*.

lcd Length of character data returned in the data variable for the request.

np Numeric parameters. There may be none. *nnp* is a count of their number.

For example, Figure 68 shows the return code vector from the GDDM call ASREAD (request code 101).

```
      ⍝ ISSUE AN ASREAD CALL (REQUEST CODE 101)

      126 ⎕SVO ¨ 'CTL126' 'DAT126'
2 2
      CTL126←101
      CTL126
0 0 0 3 0 0 0 0
      ⎕SVR¨'CTL126' 'DAT126'
2 1

      ⍝ RETURN CODES HRC, RC, AND RS ARE 0
      ⍝ 3 ITEMS OF NUMERIC PARAMETERS, NO CHARACTER DATA
      ⍝ ASREAD ISSUED FROM WITHIN THE SESSION MANAGER
      ⍝     CAUSES A BLANK SCREEN TO REPLACE THE
      ⍝     SESSION MANAGER SCREEN
      ⍝  THE THREE NUMERIC PARAMETERS RETURNED ARE ALL ZERO
      ⍝  ALSO SEE "Handling Attentions" on page 239
```

Figure 68. Return Code Vector from a Single AP 126 Request

When more than one request was made with a single specification of the control variable, a reference of it yields:

- *hrc* as the first item.
- *rc*,*rs*,*nnp*,*lcd*,,*np* (if any) for each request in the order the requests were specified.

Character data, if any, is concatenated in the data variable.

For example, Figure 69 shows the return code vector from a query of the AP 126 options and the hard-copy destination. "Query AP 126 Options" on page 233 describes AP 126 command ¯7, and "Query Current Hard-Copy Destination" on page 234 describes the request associated with ¯9.

```
      ⍝ QUERY AP 126 OPTIONS (¯7,4)
      ⍝ AND QUERY HARDCOPY DESTINATION (¯9)

      CTL126←¯7,4,¯9
      CTL126
20 0 0 4 0 8 0 1 512 1 66 0 0
      DAT126

      ⍝ HRC = 20
      ⍝ QUERY AP 126 OPTIONS REQUEST (¯7,4)
      ⍝   RC,RS = 0 0
      ⍝     NNP = 4 (4 NUMERIC PARAMETERS RETURNED)
      ⍝     LCD = 0 (LENGTH 0 CHARACTER DATA)
      ⍝      NP = NEXT 4 ITEMS (8 0 1 512)
      ⍝ QUERY HARDCOPY DESTINATION REQUEST (¯9)
      ⍝    RC, RS = 1 66  NO HARDCOPY DESTINATION DEFINED
      ⍝       NNP = 0
      ⍝       LCD = 0
      ⍝        NP = NONE
```

Figure 69. Return Code Vector from Multiple AP 126 Requests

Communication Procedure

In general, when you use the GDDM processor, you follow the steps summarized below:

1. Share a control and a data variable with AP 126.
2. Assign character data for the next request, if any, to the data variable.
3. Assign the request code and associated numeric parameters, if any, to the control variable.
4. Retrieve the return code and numeric data, if any, from the control variable.
5. Retrieve character data, if any, from the data variable.
6. Repeat steps 2 through 5, as necessary, for additional requests.
7. When finished with AP 126, retract the control and data variables.

If not already started for the session manager, GDDM initializes when you make the first AP 126 request. If the APL2 session manager is not active, GDDM terminates when the last control variable is retracted.

Before issuing a GDDM call through AP 126, you must be familiar with the parameters required by the GDDM request and the format required by AP 126. For the detailed information about the GDDM requests, see *Graphical Data Display Manager User's Guide* and *Presentation Graphic Feature User's Guide*.

Although AP 126 is usually used in a series of defined functions that make up an application, Figure 70 on page 227 shows a sample APL2 session that illustrates the communication procedure. The sample session shows the formatting of a screen and the retrieval of data from that screen. Similar coding can be used within a defined function.

```
      ⍝ SHARE WITH AP 126.  CONFIRM COUPLING OF 2.
      126 ⎕SVO ¨ 'CTL126' 'DAT126'
2 2
      ⍝ DEFINE A FORMAT MATRIX FOR TWO ALPHANUMERIC FIELDS.
      FORMAT←3 6 ⍴ 1 4 11 1 33 2 2 7 11 1 6 2 3 7 18 1 33 0
      FORMAT
1 4 11 1 33 2
2 7 11 1  6 2
3 7 18 1 33 0
      ⍝ THE MEANING OF THE MATRIX IS DEFINED IN GDDM BASE APPLICATION
      ⍝     PROGRAMMING REFERENCE DESCRIPTION OF THE CALL ASDFMT.
      ⍝     FOR EXAMPLE, ROW 1 DEFINES FIELD NUMBER 1 AS
      ⍝     STARTING IN ROW 4 COLUMN 11, 1 ROW DEEP,
      ⍝     33 COLUMNS WIDE, PROTECTED ALPHANUMERIC (INPUT
      ⍝     FROM TERMINAL NOT ALLOWED WHEN GDDM DISPLAYS SCREEN).
      ⍝     ROW 2 DEFINES A SIMILAR, BUT SHORTER FIELD ON ROW 7
      ⍝     OF 6 COLUMNS.
      ⍝     ROW 3 DEFINES A THIRD FIELD ON ROW 7 BEGINNING
      ⍝     IN COLUMN 18, 33 COLUMNS WIDE, BUT UNPROTECTED (INPUT
      ⍝     ALLOWED FROM THE TERMINAL).
      ⍝ ISSUE GDDM CALL ASDFMT USING THE MATRIX.
      ⍝     ASDFMT DEFINES THE THREE FIELDS AND
      ⍝     SUPPLIES SIX ATTRIBUTES PER FIELD.
      CTL126←402 3 6 ,,FORMAT
      CTL126
0 0 0 0 0

      ⍝ PREPARE LITERALS FOR WRITING TO THE SCREEN.
      MESSAGE←'ENTER YOUR NAME(1ST MID LAST)'
      FLDNAME←'NAME:'
      ⍝ ASSIGN MESSAGE TO DATA VARIABLE.
      ⍝ ISSUE ASCPUT TO WRITE MESSAGE TO THE SCREEN
      DAT126←MESSAGE
      CTL126←424 1,⍴MESSAGE
      CTL126
0 0 0 0 0

      ⍝ ASSIGN FLDNAME TO DATA VARIABLE.
      ⍝ ISSUE ASCPUT TO WRITE FLDNAME TO THE SCREEN
      DAT126←FLDNAME
      CTL126←424 2,⍴FLDNAME
      CTL126
0 0 0 0 0

      ⍝ ISSUE ASFCUR TO POSITION THE CURSOR TO FIELD 3.
      CTL126←430 3 1 1
      CTL126
0 0 0 0 0

      ⍝ ISSUE ASREAD TO TRANSMIT THE SCREEN TO THE TERMINAL
      ⍝     AND WAIT FOR A RESPONSE.  (NOTE: YOUR SCREEN WILL
      ⍝     GO BLANK MOMENTARILY AND THEN YOU WILL SEE THE
      ⍝     MESSAGE 'ENTER YOUR NAME'.)
      CTL126←101
      CTL126
0 0 0 3 0 0 0 1

      ⍝ ISSUE ASCGET TO RETRIEVE THE CONTENTS OF FIELD 3.
      CTL126←422 3 33
      CTL126
0 0 0 0 33
      DAT126
HERBERT R. POCKET

      ⍝ RETRACT THE SHARED VARIABLES
      ⎕SVR¨'CTL126' 'DAT126'
2 1
```

Figure 70. Sample Communication Procedure for AP 126

GDDM Calls

Most GDDM calls can be issued through AP 126. Those that cannot are listed in "Restrictions."

For detailed information about the GDDM calls, their APL2 equivalent codes, and the format of the input and output, see *GDDM Base Application Programming Reference* and *GDDM-PGF Programming Reference*.

Restrictions

The GDDM call CHART can be issued only indirectly through the AP 126 command ¯10. You cannot issue the following calls:

ESPCB
FSEXIT
FSINIT
FSRNIT
FSTERM
SPINIT

The last two calls, FSTERM and SPINIT, are automatically issued by AP 126, as necessary. The GDDM call FSQERR can be used to accomplish the operation provided by FSEXIT (see "GDDM Error Diagnosis" on page 241). ESPCB is not supported because it is only meaningful under the IBM Information Management System/Virtual Storage (IMS/VS). It has no meaning under CMS or TSO.

The PGF calls can be used only if the presentation graphics feature (PGF) of GDDM is installed.

Some PGF calls allow parameters with a maximum length of eight characters, a minimum abbreviated length of four characters, and no length parameters. AP 126, however, requires that these parameters have four characters.

AP 126 Commands

AP 126 provides 10 services that allow you to query and set certain AP 126 options and to issue the GDDM call CHART. All AP 126 command codes are negative integers. Return and reason codes for the options are described in Figure 76 on page 242.

Figure 71 summarizes the codes corresponding with the AP 126 commands. Each of the commands is then described in detail.

Figure 71 (Page 1 of 2). AP 126 Commands

Code	AP 126 Command
¯1	Query GDDM calls
¯2	Set error threshold
¯3	Set protection key
¯4	Set EBCDIC translation
¯5	Set default buffer size
¯6	Set AP 126 options
¯7	Query AP 126 options

Figure 71 (Page 2 of 2). AP 126 Commands

Code	AP 126 Command
¯8	Query subset of fields for modifications
¯9	Query current hard-copy destination
¯10	Issue GDDM CHART call

Query GDDM Calls

```
   CTL←¯1
   CTL
rc-vector,request-codes
   DAT
list of mnemonic GDDM call names
```

This request (¯1) returns a list of the available GDDM calls and the corresponding AP 126 request codes. You can use this request to determine whether the presentation graphics feature (PGF) is installed, because the codes returned are those actually available. (Calls with AP 126 request codes in the 700s are examples of PGF calls.)

The *request codes* are concatenated to the return code vector in the control variable.

GDDM call names are returned in the *DAT* variable as a simple character vector. GDDM call mnemonics are extended with blanks to eight characters. This extension makes it possible to define a pair of functions that allow you to retrieve the corresponding request code for a GDDM mnemonic.

Figure 72 on page 230 illustrates two functions: The first (*INITGDDM*) makes the connection with GDDM through AP 126, issues AP 126 request code ¯1, and assigns the results to two variables—*NAMES* and *CALLS*. The second (*GDDMNUM*) assumes that the first has been processed and returns the AP 126 request code for the GDDM mnemonic presented as an argument.

```
      ∇
[0]   INITGDDM;RC;CTL126;DAT126
[1]   ⍝ FUNCTION: SHARE CTL126 AND DAT126 WITH AP 126
[2]   ⍝           ASSIGN NAMES AND CALLS
[3]   ⍝
[4]   RC←126 ⎕SVO¨'CTL126' 'DAT126'     ⍝ SHARE WITH AP 126
[5]   →(2∈RC)/SHAREOK                   ⍝ BRANCH IF COUPLED
[6]   'SHARED VARIABLE OFFER NOT ACCEPTED BY AP 126'
[7]   RC←⎕SVR¨'CTL126' 'DAT126'         ⍝  RETRACT OFFER
[8]   →0
[9]   SHAREOK:CTL126←¯1                 ⍝ QUERY GDDM CALLS
[10]  →(0=1↑RC←CTL126)/QUERYOK          ⍝ BRANCH IF QUERY OK
[11]  'NON-ZERO RC FROM AP 126, CALL WAS ¯1, RC WAS ',⍕RC
[12]  RC←⎕SVR¨'CTL126' 'DAT126'         ⍝ RETRACT OFFER
[13]  →0
[14]  QUERYOK:CALLS←5↓RC                ⍝ ASSIGN CALL NUMBERS
[15]  NAMES←((1↑3↓RC),8)⍴DAT126         ⍝ ASSIGN CALL NAMES
      ∇
      ∇
[0]   N←GDDMNUM NAME;I
[1]   ⍝ FUNCTION:  RETURN AP 126 CALL CODE FOR GDDM CALL 'NAME'.
[2]   ⍝ (THIS FUNCTION ASSUMES EXECUTION OF 'INITGDDM'.)
[3]   ⍝ RETURNS 0 IF NAME IS NOT A VALID, SUPPORTED CALL
[4]   I←(NAMES∧.=8↑NAME)⍳1
[5]   N←(CALLS,0)[I]
      ∇  13.45.46  7/15/83 (GMT-4)
      INITGDDM
      GDDMNUM 'ASTYPE'
111
      GDDMNUM ¨ 'ASTYPE' 'ASREAD' 'CHFINE'
111 101 799
```

Figure 72. Retrieving Request Codes for GDDM Calls

Set Error Threshold

```
   CTL←¯2,threshold
   CTL
rc-vector
```

This request (¯2) allows you to establish a threshold for GDDM error severities—4 8 12 16.

When a GDDM error is encountered that has a severity greater than or equal to the threshold value, AP 126 terminates processing of the current string of requests.

This threshold applies only to GDDM requests. An error from AP 126 (*rc*=1) automatically terminates processing of the current string of requests.

threshold is any nonnegative integer. The default threshold value is 8, which allows processing to continue when a warning is encountered.

Set Protection Key

```
   CTL←¯3,key
   CTL
rc-vector
```

The ¯3 service request has no effect in APL2. It is maintained for compatibility with previous versions of APL. This request allowed you to use pages normally reserved for the APL2 session manager. With APL2, you no longer need to issue this call to gain access to those pages.

key values for this request are:

0 = Default, no access to reserved pages.

1 = Access to reserved pages permitted.

Set EBCDIC Translation

```
   CTL←¯4,translation-value
   CTL
rc-vector
```

Under most conditions, AP 126 need not translate character data in a request before passing it to GDDM, because APL2/370 and GDDM use very similar EBCDIC code pages for character data. See "APL2/370 and GDDM EBCDIC Code Page Differences" on page 240.

However, when migrating character data from VS APL, the)*MCOPY* system command translates all character data from VS APL character codes to EBCDIC. This translation is appropriate for all character data passed to AP 126, with the following exceptions:

- Symbol set values—the data parameter in SSREAD, SSWRT, PSDSS, and GSDSS.
- Translate table values—input and output table value parameters in ASDTRN.
- Graphics Data File (GDF) data processed by GSGET and GSPUT.
- Image data processed by GSIMG and GSIMGS.

These values are scrambled by the translation.

There are two ways to resolve this problem: use the EBCDIC translation option or bypass the translation option by migrating the ⎕*AV* indexes of the characters in the symbol sets, translation tables, GDF, and image data.

Using the Translation Option: To unscramble the symbol set and other data values, this request (¯4) provides an option to make your VS APL workspaces containing symbol sets and translation tables compatible with AP 126 under APL2.

translation-value can be either:

0 = special EBCDIC translation off (default)

1 = special EBCDIC translation on

By inserting the AP 126 request ¯4 in the appropriate functions, you can ensure that the necessary translation takes place.

Circumventing the EBCDIC Translation Option: You can perform the translation of VS APL symbol sets, translation tables, and GDF and image data only once. Follow the steps outlined below.

1. In VS APL, retrieve the ⎕*AV* indexes of the characters in the symbol set, translation table, or GDF ⎕*AV*, assigning the indexes to a temporary variable. For example:

```
TEMPVAR ← ⎕AV ⍳ SYMSET
```

2. Use the)*MCOPY* command to migrate the temporary variable to an APL2 workspace. For example:

```
)MCOPY 1001 VSAPLWS TEMPVAR
```

3. Use bracket indexing of ⎕*AV* to re-create the original variable in APL2. For example:

```
SYMSET ← ⎕AV[TEMPVAR]
```

When completed successfully, this procedure eliminates the need for the use of the EBCDIC translation option (code ¯4) for symbol sets, translation tables, and GDF.

Set Default Buffer Size

```
   CTL←¯5,buffer-size
   CTL
rc-vector
```

The size of the AP 126 buffer is normally determined by the auxiliary processor itself, but this request (¯5) enables you to specify the default buffer size for the data path associated with a shared variable pair.

Regardless of how the buffer size is determined, if a request requires a larger buffer than is currently available, AP 126 dynamically obtains the necessary buffer space for the request and frees it after the request has completed. The buffer size is then reestablished to its default size.

buffer-size is specified in number of bytes and is rounded up to the next 8-byte multiple.

Set AP 126 Options

```
   CTL←¯6,count,options-vector
   CTL
rc-vector
```

This request (¯6) allows you to set, in one request, the AP 126 options described above and controlled by individual request codes ¯2 through ¯5.

You must specify a one- to four-item vector of options. Each item in the vector corresponds to one of the four options requests in the following order:

1. Error threshold
2. Protection key
3. EBCDIC translation
4. Default buffer size

A one-item vector specifies only the error threshold; a two-item vector specifies error threshold and protection key; and so forth.

You retain the current value of the option by entering ¯1 in the corresponding position of the options vector. Other values are the same as for the individual options.

count is the length of the options vector being passed.

Query AP 126 Options

```
   CTL←¯7,count
   CTL
rc-vector,options-vector
```

This request (¯7) allows you to query the AP 126 options described above.

The result of the request is a numeric vector that contains items corresponding to the options in the following order:

1. Error threshold
2. Protection key
3. EBCDIC translation
4. Default buffer size

count is the number of options to be queried. 1 for *count* returns the setting of the error threshold; 2 returns the settings of the error threshold and protection key, and so forth.

Query Subset of Fields for Modifications

```
   CTL←¯8,count,field-id-vector
   CTL
rc-vector,modf-vector, tlength vector, ilength vector
```

This request (¯8) allows you to query a specified subset of fields to determine which members of that subset were modified.

It is recommended that you use this request when sharing the page of the session manager.

count is the number of field identifiers in the subset of fields.

field-id-vector is a vector of the field identifiers that are to be queried. This vector is assumed to be a subset of fields.

modf-vector is a list of any fields from the *field-id-vector* that were actually modified. These fields are returned in the same order as modified fields are returned by the GDDM request ASQMOD, not necessarily in the order specified by the *field-id-vector*. The vector is of length *count* with trailing, unused entries in the vector set to 0.

tlength-vector is a list of total lengths, in the corresponding order and padded with 0s, for each member of *modf-vector*.

ilength-vector is a list of input lengths, in the corresponding order and padded with 0s, for each member of *modf-vector*.

As the subset of modified fields is returned, the fields become no longer modified, just as with ASQMOD.

Query Current Hard-Copy Destination

```
   CTL←¯9
   CTL
rc-vector,open-options-vector
   DAT
destination-name
```

This request (¯9) allows you to query the current hard-copy destination.

open options vector contains the options specified on the FSOPEN call for the destination.

destination name, returned in the data variable, identifies the current hard-copy destination name, padded with blanks to a length of eight characters, if necessary.

If destination name is all blanks, the destination was selected through the DSOPEN and DSUSE calls and not through FSOPEN. Issue DSQUSE, DSDROP, and DSUSE to retrieve the destination name.

Using this service request, an application can save the vector of FSOPEN options before changing the destination and restore them at a later time.

Issue CHART Call

```
   DAT← chart-control,keys,labels,heading
   CTL←¯10,len-chart-control,
            len-data-control,data-control,
            len-x,x,len-y,y,
            len-keys,len-labels,len-heading
   CTL
rc-vector
```

Because AP 126 is unable to determine the lengths of the parameters to the GDDM call CHART from the call descriptor tables provided by GDDM, a special AP 126 service request is required to allow you to issue the CHART request.

The GDDM CHART call has seven parameters:

chart-control (character input)
data-control (numeric input)
x (numeric input)
y (numeric input)
keys (character input)
labels (character input)
heading (character input)

This service request (`¯10`) requires you to provide a length with each CHART parameter, yielding the following 14 parameters:

len-chart-control (numeric input)
chart-control (character input)

len-data-control (numeric input)
data-control (numeric input)

len-x (numeric input)
x (numeric input)

len-y (numeric input)
y (numeric input)

len-keys (numeric input)
keys (character input)

len-labels (numeric input)
labels (character input)

len-heading (numeric input)
heading (character input)

Applying the standard AP 126 rules for passing character and numeric data yields the vectors for the data variable and the control variable, as specified in the list above. Figure 73 on page 236 shows a defined function that issues this service request.

You process this function by entering the name CHART. The CHART function invokes the interactive chart utility of PGF.

The *IO* function is contained in the *UTILITY* workspace, which is provided with APL2.

```
      ∇CHART[⎕]∇
[0]   RC←CHART;CHRTCTL;CTL;DAT;DATACTL;HEADING;KEYS;LABELS;X;Y;⎕IO
[1]   ⎕IO←1
[2]   →(2∧.=RC←126 ⎕SVO 2 3⍴'CTLDAT')/SHAREOK
[3]   'AP 126 NOT SHARING'
[4]   →0
[5]   SHAREOK:⍝ INIT CHART-CTL TO BLANKS
[6]   CHRTCTL←76⍴' '
[7]   ⍝ LEVEL 0
[8]   CHRTCTL[⍳4]←4 IO 0
[9]   ⍝ DISPLAY 2
[10]  CHRTCTL[4+⍳4]←4 IO 2
[11]  ⍝ HELP 0
[12]  CHRTCTL[8+⍳4]←4 IO 0
[13]  ⍝ ISOLATE 0
[14]  CHRTCTL[12+⍳4]←4 IO 0
[15]  ⍝ FORMNAME '*'
[16]  CHRTCTL[17]←'*'
[17]  ⍝ DATANAME '*'
[18]  CHRTCTL[25]←'*'
[19]  ⍝ PAIRING 0
[20]  CHRTCTL[32+⍳4]←4 IO 0
[21]  ⍝ NG 2
[22]  CHRTCTL[36+⍳4]←4 IO 2
[23]  ⍝ NE 3
[24]  CHRTCTL[40+⍳4]←4 IO 3
[25]  ⍝ KEYL 4
[26]  CHRTCTL[44+⍳4]←4 IO 4
[27]  ⍝ LABELL 6
[28]  CHRTCTL[48+⍳4]←4 IO 6
[29]  ⍝ HEADINGL 7
[30]  CHRTCTL[52+⍳4]←4 IO 7
[31]  ⍝ PRTNAME '*'
[32]  CHRTCTL[57]←'*'
[33]  ⍝ PRTDEP 0 PRTWID 80 PRTCOPY 2
[34]  CHRTCTL[64+⍳12]←,4 IO 0 80 2
[35]  ⍝ NO DATACTL DATA
[36]  DATACTL←0⍴0
[37]  ⍝ X 1 2 3
[38]  X←1 2 3
[39]  ⍝ Y 2 1 2.5 3 2 1
[40]  Y←2 1 2.5 3 2 1
[41]  ⍝ KEYS 'KEY1KEY2'
[42]  KEYS←'KEY1KEY2'
[43]  ⍝ LABELS 'LABEL1LABEL2LABEL3'
[44]  LABELS←'LABEL1LABEL2LABEL3'
[45]  ⍝ HEADING 'HEADING'
[46]  HEADING←'HEADING'
[47]  DAT←CHRTCTL,KEYS,LABELS,HEADING
[48]  CTL←¯10,(⍴CHRTCTL),(⍴DATACTL),DATACTL,(⍴X),X,(⍴Y),Y,(⍴KEYS),
      (⍴LABELS),⍴HEADING
[49]  RC←CTL
      ∇ 1983-07-09  9.23.40 (GMT-8)
```

Figure 73. Sample Function that Issues the CHART Service Request

Obtaining Copies through AP 126

You can copy a screen image from AP 126 through the GDDM FSOPEN request or a sequence of the DSOPEN, DSUSE calls for alternate devices.

GDDM FSOPEN Request or DSOPEN, DSUSE Sequence

The GDDM call FSOPEN (request code 604) or the sequence of calls DSOPEN, DSUSE for alternate devices can be used to open either an APL2 or GDDM copy destination.

If the FSOPEN request fails, you should issue the AP 126 service request to query the current hard-copy destination (request code ¯9) and save the result. If the query (¯9) does not return a destination name, the destination was used by the DSUSE call and was not opened by FSOPEN. In such a case, issue DSQUSE to query the destination rather than the AP 126 service request.

Note: While the AP 126 application's destination is open, any continuous copy lines generated from the session manager go to that destination rather than to the session manager destination.

If an AP 126 application uses the destination specified in the session manager profile, no FSOPEN call is necessary if the session manager already has the destination open for continuous copy. If the destination was specified in the profile, but the destination is not in use, the application can find the current session manager COPY ID and COPY CODE values through AP 120. It can then issue an appropriate AP 126 FSOPEN request based on those values.

If the session manager continuous copy is turned off while AP 126 is using the hard-copy destination, the AP 126 destination is closed and must be reopened.

Alternating Paths

To use AP 126 and to take full advantage of the available facilities, you need to be aware of several aspects of path coordination:

- Implications of multiple data paths
- Page sharing with the APL2 session manager

Implications of Multiple Data Paths

A *data path* is established for each control variable used to request AP 126 functions. APL2 tracks the current primary device and partition set by data path. If the partition set is the default partition set, APL2 also tracks the current page.

Before issuing a call for a given data path, APL2 checks the current device and partition set. If the device and partition set are not the same as the path, APL2 switches to the device and partition set of the path.

If the partition set is the default partition set, APL2 also checks the current page and switches pages if necessary.

The result is that paths are somewhat insulated from one another.

The alternation of devices, partition sets, and pages on the display screen follows the same rules when multiple AP 126 paths are contending for the screen as it

does when a single AP 126 path is contending for the screen with the session manager.

Page Sharing with the APL2 Session Manager

An application can share the session manager page by specifying the same page number. The APL2 session manager currently uses the default primary device, the default partition set, and page 1001. The application should not access any fields that are being used by the session manager.

Guidelines for Sharing with the Session Manager

Follow the guidelines below when designing an application that shares a page with the session manager.

Size of the Session Manager Screen: Limit the session manager to only a portion of the terminal screen by issuing the session manager commands:

- DISPLAY SIZE *rows columns*
- DISPLAY ORIGIN *row column*

When you change DISPLAY SIZE or DISPLAY ORIGIN, all alphanumeric fields are deleted. Therefore, you should set these two commands before defining any application fields. Chapter 3, “The APL2 Session Manager” on page 36 contains descriptions of the session manager DISPLAY command and its SIZE and ORIGIN operands.

Page Selection and Formatting: You can select a session manager page within an application, but you must not delete it. You cannot create a session manager page through an application.

Within the application, the page must never be formatted completely (as with ASDFMT); all format requests must be either ASDFLD (code 402) or ASRFMT (code 405).

Field Definitions: Avoid using field numbers 1000 through 1999, which are reserved for the session manager. If an application redefines a field being used by the session manager, unpredictable results can occur. The application can use any other numbers for its own fields.

Within an application, you cannot define new alphanumeric or graphic fields in the portion of the terminal screen being used by the session manager. The portion of a graphic field that overlaps the session manager's part of the screen does not appear.

Input/Output Requests: You can choose to restrict the application to output requests when addressing the session manager page through AP 126. Any simple command input can be handled by □ or □, which causes the session manager to issue an ASREAD GDDM request, thus displaying all the contents of the page. If the output needs to be updated on the screen only at times when the session manager schedules I/O, the application need not request the FSFRCE call.

If the application handles input from the session manager page through AP 126, the following guidelines apply:

1. An ASREAD request returns the total number of modified fields on the page, whether they are session manager or application fields. The ASQMOD request also returns the modified fields.

   In addition, ASQMOD resets the status of each field to unmodified regardless of whether it is a session manager field.

   The session manager does not know that data was entered in its fields, although the data continues to display if the ASREAD or ASQMOD had been issued by AP 126. Likewise, AP 126 does not know that data in its fields was modified if the session manager issues the ASREAD or ASQMOD.

   Given the effects of ASREAD and ASQMOD, it follows that the AP 126 service request to query a subset of fields for modifications (request code `¯8`) should be used rather than ASQMOD when a page is shared with the session manager. This service request allows you to query from within the application the modification status of a restricted list of fields. Provided this list does *not* include any session manager fields, the status of these fields is the same after the request as before.

2. An ASREAD from AP 126, which is completed with a function key, must be handled by the application. When pages are shared, the session manager profile function key definitions do not apply.

3. If the session manager unexpectedly issues an I/O request (for example, because of a syntax error in a function), it may be difficult for the terminal operator to determine whether a read request is handled by the application or the session manager.

4. If the application uses cursor positioning prior to issuing an AP 126 ASREAD request, it must position the cursor after any session manager terminal I/O request (most of which move the cursor).

Handling Attentions

If AP 126 issues an ASREAD that completes with an attention key (ATTN, PA1, or PA2, depending on the environment), the APL2 session receives either:

- Attention if no previous attention was signaled since the last APL2 processing
- Interrupt if an attention was just processed

AP 126 also returns the appropriate PA key completion (as returned by GDDM) to the application in response to the ASREAD request. The key completions can be affected by various PROCOPT settings, such as `'TSOINTRP, PA1'`.

If attention is signaled when the AP 126 page is displayed, but no ASREAD was issued, the APL2 session merely receives the appropriate attention signal.

APL2/370 and GDDM EBCDIC Code Page Differences

Although APL2/370 and GDDM both use EBCDIC-based character sets, there are some differences between the APL2 EBCDIC character set and the EBCDIC character set used by default by GDDM.

The APL2 EBCDIC character set, EBCDIC code page 293, is described in *APL2 Programming: Language Reference*. The GDDM default EBCDIC character set, EBCDIC code page 351, is described in *GDDM Base Application Programming Reference*, in the description of the GDDM ASTYPE call.

Of the characters for which specific graphics are defined in the APL2 EBCDIC character set, all but five are at the same code points in the GDDM default EBCDIC code page.

The characters that are in different code points are described by Figure 74.

Figure 74. Code Page Differences

	APL2 Code Point		GDDM Code Point	
Character	**Hex**	**⎕AF**	**Hex**	**⎕AF**
⍰	73	115	FA	250
⍳	74	116	26	38
∈	75	117	3C	60
⊢	76	118	2E	46
⊣	77	119	2F	47

For practical purposes, this means if you read a field of input from GDDM in which someone might haved typed the characters "⍰⍳∈⊢⊣," and you want them to be treated as "⍰⍳∈⊢⊣" within APL2, then you must translate any instances of the GDDM code points to the APL2 code points. The converse is true when writing APL2 characters out through GDDM.

This can be done simply as follows. Initialize a translate table to ⎕*AV*.

```
⎕IO←0                    ⍝ SO ⎕AF VALUE WILL INDEX ⎕AV
TT←⎕AV
```

For each APL2 codepoint to be translated, substitute the GDDM code point:

```
TT[ 115 116 117 118 119 ] ← ⎕AV[ 250 38 60 46 47 ]
```

For each GDDM codepoint to be translated, substitute the APL2 code point:

```
TT[ 250 38 60 46 47 ] ← ⎕AV[ 115 116 117 118 119 ]
```

Character strings from GDDM can now be translated for APL2:

```
APLIN←⎕AV[ TT ⍳ GDDMIN ]
```

and APL2 character strings can be translated for output to GDDM:

```
GDDMOUT←⎕AV[ TT ⍳ APLOUT ]
```

GDDM Error Diagnosis

If you receive a GDDM error return code (*rc*≠1), you can retrieve further information about the error by issuing the FSQERR call immediately after GDDM returns an error. The data variable returns the text of the message associated with the GDDM error return code. Figure 75 illustrates a sequence of entries that create a GDDM error and retrieve the text of the GDDM error message.

```
      126 ⎕SVO ¨ 'CTL126' 'DAT126'
2 2
      ⍝ FORMAT VARIABLE SPECIFIES OVERLAPPING FIELDS.
      FORMAT ← 3 6 ⍴ 1 2 3 1 20 2 2 4 3 1 25 2 3 4 8 1 6 0
      FORMAT
1 2 3 1 20 2
2 4 3 1 25 2
3 4 8 1  6 0
      ⍝ ATTEMPT TO FORMAT THE SCREEN RETURNS GDDM ERROR.
      CTL126←402 3 6,,FORMAT
      CTL126
8 8 206 0 0
      ⍝ RC=8, RS=206, GDDM MESSAGE NUMBER.
      ⍝ FSQERR CALL RETRIEVES THE MESSAGE FROM GDDM.
      ⍝ DATA VARIABLE CONTAINS THE MESSAGE TEXT.
      CTL126←107,160
      CTL126
0 0 0 0 160
      20↓100↑DAT126
ADM0206 E ALPHANUMERIC FIELD 3 OVERLAPS ALPHANUMERIC FIELD 2
```

Figure 75. Retrieving a GDDM Error Message

Return and Reason Codes

The return (*rc*) and reason codes (*rs*) are returned as items of the return code vector (*rc-vector*) in the control variable. (See “Returned Values” on page 224.)

When AP 126 detects an error, the first item in the return code vector is 20. The *rc* and *rs* items for the request that failed then contain a 1 and a reason code, respectively. The 20 appears even if only a single request has been passed.

The codes, their descriptions, and suggested responses are summarized in Figure 76 on page 242. The first item in the Code column of the table corresponds to the *rc* in the return code vector; the second item is returned as the *rs* item in the return code vector.

Figure 76 (Page 1 of 2). Return Codes Issued by AP 126

Code	Description
0 0	Normal return.
1 6	Copy destination unsupported.
1 7	Data variable is wrong length. **Action**: Supply data of correct length in the data variable, then respecify the control variable.
1 9	Copy destination full or print limit exceeded.
1 11	`1<ρρCTL` **Action**: Ravel the value in the control variable, and respecify it.
1 12	Syntax error in request. **Action**: Reconstruct the value in the control variable.
1 13	Control variable contains character data, complex numbers, or a mixed or nonsimple array.
1 14	Invalid request code in the control variable.
1 21	`1<ρρDAT` **Action**: Ravel the value in the data variable.
1 22	Copy destination disabled or closed.
1 23	*DOMAIN ERROR* on data variable; it contains numeric data or a mixed or non-simple array.
1 26	Copy destination not defined.
1 28	Copy destination enquiry failed or enquiry lost due to sync point.
1 30	Authorization check for copy destination failed.
1 35	Copy destination I/O error.
1 41	Data variable not specified. **Action**: Assign the required data to the data variable and respecify the control variable.
1 53	Required storage is not available. **Action**: If requests were concatenated, reduce the number of requests in one specification. Or try to retract other shared variable pairs to free up storage.
1 54	Copy translation table unavailable.
1 56	Insufficient space in shared variable quota. **Action**: If requests were concatenated, reduce the number of requests in one specification.
1 60	GDDM not available.
1 61	Invalid parameter for service request.
1 62	Invalid count, code, or length value on request.
1 63	Hard-copy translation table not available.
1 65	GSCOPY attempted to an APL destination. **Action**: Use GDDM destination rather than an APL destination.
1 66	Hard-copy destination not available. If this occurs on a request issued after a successful open, the destination was closed. **Action**: Issue an FSOPEN request for the destination.
1 67	A hard-copy destination is already open. **Action**: Query the hard-copy destination (request code `¯9`); if no name is returned, issue DSQUSE instead; issue FSCLS for that destination; issue FSOPEN for the new destination.

Figure 76 (Page 2 of 2). Return Codes Issued by AP 126

Code	Description
1 96	Invalid request. **Action:** Notify your system administrator; this code indicates an APL2 or system problem.
2 *nnn*	Abend occurred in APL2 outside of AP 126 and GDDM. *nnn* is the ABEND code.
ss nnn	A return code greater than 2 indicates a GDDM warning or error. *ss* is the severity of the error. *nnn* is the return code or ABEND code from GDDM.
4 *nnn*	Warning. *nnn* is the return code from GDDM.
8 *nnn*	Error. *nnn* is the return code from GDDM.
12 *nnn*	Severe error. *nnn* is the return code from GDDM.
16 *nnn*	ABEND. *nnn* is the ABEND code issued by the failed GDDM module. **Action:** Refer to *GDDM User's Guide*; consult GDDM request made and message ADM0*nnn*; issue FSQERR to retrieve the message text.

Chapter 22. AP 127—SQL Processor

Through AP 127, the SQL processor, APL2 provides access to the database management systems (IBM program products) Structured Query Language/Data System (SQL/DS) under CMS and IBM DATABASE 2 (DB2) under TSO.

Under CMS, AP 127 requires SQL/DS Version 3 Release 3 or higher. Under TSO, AP 127 requires DB2 Version 2 Release 3 or higher.

SQL is a high-level language that uses the relational data model. A *relation* in the relational data model can be thought of as a simple two-dimensional table—a matrix in APL2 terms. SQL provides access to the tables through SQL statements. SQL is a query, data manipulation, data definition, and authorization language.

The APL2/SQL interface consists of:

- The auxiliary processor AP 127, which accepts requests by shared variables, transforms them into standard run-time SQL requests, and passes them on to SQL/DS or DB2.

  AP 127 shared variable use is characterized by operation codes based on dynamic SQL, straightforward protocol, and simple, consistent syntax.

- The APL2 workspace named SQL, containing:
  - *Data access functions*, which pass SQL requests to AP 127
  - *User support functions*, which create common sequences of requests and pass them to AP 127
  - *Task control functions*, which allow you to manage the APL2/SQL interface environment
  - A *defined operator*, $UNTIL$, which creates a derived function that processes a stack of requests to AP 127.

  The facilities in the SQL workspace ease the task of communicating with the auxiliary processor. They allow matrix processing and command stacking with error recovery. In this way, they provide compatibility with the array processing capabilities of APL2 without compromising procedural control.

For information on how to use the APL2/SQL interface, see *APL2 Programming: Using Structured Query Language*.

The remainder of this chapter presents an overview of AP 127.

Shared Variable Overview

AP 127 allows multiple share offers but processes transactions for only one shared variable at a time.

Figure 77 provides an overview for sharing variables with AP 127.

Figure 77. Shared Variable Overview for AP 127

SV Protocol	AP 127 Conventions
General	One variable. The variable is used to pass requests, value-lists, and options-lists to AP 127 and to pass return codes and result data from AP 127 and SQL operations. The variable is assigned the operation code and any parameters required by that operation code.
Maximum Number of Shared Variables	10. AP 127 allows multiple share offers but processes transactions for only one shared variable at a time. Up to ten shared variables can be coupled at one time, but the *active* variable is the first one specified. It becomes inactive when it is retracted. This capability allows multiple applications to offer variables. The shared variable limit is *not* related to the limit of 40 simultaneously prepared cursors. If a variable specified or referenced is not the one currently active, an error message is returned.
Names	Any valid APL2 variable name of up to 255 characters. The variable DAT is shared by the SQL workspace.
Initial Values	Ignored.
Subsequent Values	Depend on the requested AP 127 operations. Reference a result vector that contains a return code and a data item that can contain values or can be empty.
Access Control	1 0 0 1 Note that for effective use, the user or the workspace function should convert this to 1 0 1 1. This prevents a reference before the auxiliary processor has specified a result.

Communication Procedure

1. Offer to share a variable with AP 127.
2. Wait for a degree of coupling of 2.
3. Set the access control.
4. Use a specification statement to assign an AP 127 operation to the variable.
5. Reference the variable to ensure that the operation was completed successfully and to retrieve any data returned by AP 127.

   A 5-item vector of zeros is returned if the operation was successful. Whether data is returned from AP 127 depends on the particular operation performed.
6. When you finish using AP 127, request retraction of the shared variable.

AP 127 Commands

Figure 78 shows the syntax of the AP 127 commands in which *DAT* represents a variable that is shared with AP 127.

Figure 78. Summary of AP 127 Commands

Syntax	Description
DAT←'*CALL*' *name* [*values*]	Processes defined SQL statement *name* that was previously processed by a *PREP* operation. AP 127 substitutes each item of *values* for the column index in *name*.
DAT←'*CLOSE*' *name*	Closes cursor (SELECT) statement *name*.
DAT←'*COMMIT*' ['*RELEASE*']	Makes permanent all of the changes you made to the database since you shared the variable or since the most recent *COMMIT* or *ROLLBACK* operation.
DAT←'*CONNECT*' [*id password*] [*database*] *DAT*←'*CONNECT*' [*database* | '*RESET*']	Specifies user ID or database name (SQL/DS) Specifies database name (DB2)
DAT←'*DECLARE*' *name*['*HOLD*' | '*NOHOLD*']	Declares a cursor name with hold attributes
DAT←'*DESCRIBE*' *name* [*options*]	Returns information about an open statement. *option* can be '*NAMES*', '*LABELS*', '*ANY*', or '*BOTH*'.
DAT←'*EXEC*' *statement*	Immediately processes *statement*.
DAT←'*FETCH*' *name* [*options*]	Returns new result table data as the second item of the shared variable result.
DAT←'*GETOPT*'	Returns the values of the AP 127 options.
DAT←'*ISOL*' [*level*]	Sets or returns the isolation level setting.
DAT←'*MSG*' *rcode*	Returns the error message text associated with return code *rcode*.
DAT←'*NAMES*'	Returns the names of all statements known to AP 127.
DAT←'*OPEN*' *name* [*values*]	Opens previously prepared statement *name*.
DAT←'*PREP*' *name statement*	Prepares SQL statement *name* for later processing by a *CALL* or *OPEN* operation.
DAT←'*PURGE*' *name*	Removes statement *name* from the list of active names in AP 127; if *name* is empty, all statements are removed from the list.
DAT←'*PUT*' *name values*	(CMS only.) Sends data to SQL/DS for INSERT
DAT←'*ROLLBACK*' ['*RELEASE*']	Removes all of the changes you made to the database since you shared the variable with AP 127 or since the most recent *COMMIT* or *ROLLBACK* operation.
DAT←'*SETOPT*' *options*	Sets the values of the AP 127 options.
DAT←'*SQLCA*'	Returns current contents of SQLCA control block.
DAT←'*SQLSTATE*'	Returns current value of SQLSTATE variable.
DAT←'*SSID*' [*name*]	(TSO only.) Sets or queries the DB2 subsystem name.
DAT←'*STATE*' *name*	Yields the current state of SQL statement *name*.
DAT←'*STMT*' *name*	Yields SQL statement *name*.
DAT←'*TRACE*' *(n1 n2)*	Yields an event trace of functions within AP 127; *n1* specifies the AP 127 module number and *n2* specifies the trace level.

Return Codes

Figure 79. Return Codes from AP 127

Code	Description
0 0 0 0 0	Normal return. All operations completed. Table retrieved by a *FETCH* request is complete.
0 1 0 *n msgn*	Normal return, but a warning message was issued. For example, cursor is beyond end of table on a *FETCH* request or *DELETE* statement deletes nothing. ***n*=1** Indicates a warning from AP 127. ***n*=2** Indicates a warning from SQL/DS or DB2.
0 0 1 0 0	Normal return, but a result table cannot have been completely retrieved.
1 1 0 *n msgn*	Transaction backout. All changes made to tables since the last *COMMIT* or *ROLLBACK* were discarded. Application must restore processing to point of last *COMMIT* or *ROLLBACK*.
1 0 0 1 *msgn*	Error in AP 127. *msgn* is the number of the AP 127 error message.
1 0 0 2 *msgn*	Error detected in DB2 or SQL/DS. *msgn* gives the value of DB2 or SQL/DS return code.
1 0 0 3 *msgn*	Error detected in an SQL workspace function. *msgn* gives the message number.

Chapter 23. AP 210—BDAM File Processor (TSO Only)

AP 210, the BDAM file processor for TSO, provides relative record access to fixed-length, unkeyed disk files through the Basic Direct Access Method (BDAM). Files can be read, written, or updated with AP 210.

This processor is not available under CMS. CMS files, however, can be similarly processed using AP 110, the CMS file processor (see Chapter 14, “AP 110—CMS File Processor” on page 138).

Associated Workspace

The TSO workspace contains several cover functions that can be used in the allocation procedure for BDAM files. For information on these functions and how they are used, type *DESCRIBE*, *HOW*, or *ABSTRACT* in the *TSO* workspace.

Shared Variable Overview

Figure 80 provides an overview for sharing variables with AP 210.

Figure 80. Shared Variable Overview for AP 210

SV Protocol	AP 210 Conventions
Protocol	Two variables—control and record. The record variable must be offered before the control variable. Specify the control variable to position the record pointer. Reference it to obtain the return code from the most-recent read/write operation. Specify the record variable to write a record. Reference it to read a record.
Maximum Number of Shared Variables	40
Names	Can be any APL2 variable names not exceeding a length of 77 characters.
Initial Values	Control and record variables are paired by a matching ddname in their initial values. (See “Initial Values” on page 249.)
Subsequent Values	**Record:** Specify or reference a character scalar or character vector not exceeding the LRECL of the file. (See “BDAM File Requirements” on page 249.) **Control:** Specify a relative record number (scalar integer). Reference a return code (one-item integer vector).
Data Types Supported	All types.
Access Control	**Record:** 0 0 1 1 **Control:** 0 0 1 1

BDAM File Requirements

Before using this processor on a BDAM file, you must allocate the file to the TSO session. The allocation can be done any time before AP 210 is invoked to perform some action on the file.

In addition, before you can read or write data on the file, the file must be formatted, or loaded, with representative records—one for each record the file contains. The file can be formatted in any one of three ways:

- By an MVS program written in a language other than APL
- By the QSAM auxiliary processor, AP 111, using fixed-length, unblocked records
- By the separate formatting protocol of AP 210 described in "BDAM File Processing Procedure" on page 251

DCB Attributes Provided by AP 210

Figure 81 lists DCB attributes the processor automatically assigns to the BDAM file. When the FMT option is specified in the initial value of the record variable, you are using AP 210 to format the file.

Figure 81. DCB Attributes Provided by AP 210

With FMT Option	Without FMT Option
DSORG=PS	DSORG=DA
RECFM=F	RECFM=F
MACRF=WL	MACREF=(RIC,WIC)

Communication Procedure

This section discusses the communication procedure for AP 210.

Initial Values

The initial values of each pair of record and control variables associate the pair with the *ddname* of the file to be processed.

The formats for the initial value of the record (*REC*210) and control (*CTL*210) variables are:

```
REC210←'ddname ([FMT] [access ] [conversion])'

CTL210←'ddname  (CTL)'
```

The options following the ddname can be entered in any order.

ddname Identifies the ddname of the previously allocated file to be processed.

FMT Specifies that you are formatting (or reformatting) the file. (See "Formatting a Direct File Using AP 210" on page 250.)

access Specifies the type of access to be used on the file. The access options are:

U The file can be read, written, or updated. This is the default.

R The file can only be read. This option is mutually exclusive with the FMT option.

W The file can only be written.

conversion Specifies the type of conversion applied to the data when records are transferred between the physical file and the active workspace. Available conversion options are:

BCD (370)
BIT
BYTE
CDR
*COD*1
DBCS
EBCD
TN
VAR (default)

Note: With conversion options *VAR* and *CDR*, the record variable can be specified with values that are less than or equal to the LRECL of the file. With all other conversion options, each specification of the record variable must be exactly equal to LRECL (for the BIT option, the length must be 8xLRECL).

See "Calculating the Length of APL2 Variables" on page 252. All conversion options are explained in Appendix D, "Auxiliary Processor Conversion Options" on page 370.

CTL Indicates that this variable is a control variable. FMT, access, and conversion options, if specified, are ignored.

If the *CTL* parameter is omitted, the variable is assumed to be a record variable.

Formatting a Direct File Using AP 210

If the file you want to process using AP 210 does not yet contain records, you can format it by following the first five steps of "BDAM File Processing Procedure," described next. To format with AP 210, you must provide the record count for the file and specify typical record contents for all records in the file.

You can reformat a file that already contains records using the AP 210 formatting procedure. You can also format using AP 111 or other non-APL formatters. Reformatting a file removes any data in the file.

BDAM File Processing Procedure

The following procedure is used to process a BDAM file with AP 210. Figure 82 on page 252 contains a sample APL2 session that illustrates this procedure.

1. Initialize the record and control variables.

   Both variables contain the ddname of the file as the first positional parameter. See "Initial Values" on page 249.

2. Offer the initialized variables (the record variable first), and check the degree of coupling.
3. Set the access control.
4. Check the return codes from the offer.
5. To format the file (omit this step if the file is already formatted to your satisfaction):
   a. Specify the control variable with the number of records you want the file to contain.
   b. Specify the record variable to initialize the file with typical record contents.
   c. Check the return code from the format request.
6. Process the file sequentially.

   Sequential processing starts with the relative record number last specified in the control variable, or from the beginning of the file, if the control variable was never specified with a relative record number.

   To process the file sequentially:
   a. Specify or reference the record variable.
   b. Reference the control variable for the return code from the read or write operation.
7. To process the file directly:
   a. Specify the control variable with the relative record number of the record you want to access.
   b. Specify the record variable to write the specified record, or reference the record variable to read the specified record.
   c. Reference the control variable for the return code from the read or write operation.
8. Retract the variables to close the file.

```
    [File Allocation—DDNAME=AP210FIL LRECL=40]
    .
    .
    .
       REC210←'AP210FIL (FMT U EBCD'  ⍝ initialize variables
       CTL210←'AP210FIL  (CTL'        ⍝   with ddname
       210 ⎕SVO¨ 'REC210' 'CTL210'    ⍝ offer (REC210 first)
2 2                          ←──────────Degree of coupling OK
       (⊂0 0 1 1) ⎕SVC ¨ 'REC210' 'CTL210'  ⍝ set access control
 0 0 1 1  0 0 1 1
       REC210                         ⍝ check return codes
0
       CTL210
0
                              ⍝ To format a file:
       CTL210←2000                    ⍝ specify number of records
       REC210←40↑''                   ⍝ specify typical file record
       CTL210                         ⍝ check return code
0
                              ⍝ Sequential processing example:
       REC210←40↑'FIRST'              ⍝ write first record
       CTL210                         ⍝ check return code
0
       REC210←40↑'SECOND'             ⍝ write second record
       REC210←40↑'THIRD'              ⍝ write third record
       CTL210←1                       ⍝ position record pointer
       REC210                         ⍝ read, from beginning
FIRST
                              ⍝ Direct processing example:
       CTL210←3                       ⍝ position record pointer
       REC210                         ⍝ read that record
THIRD
       REC210←40↑'NEW THIRD'          ⍝ replace that record
       REC210←40↑'FOURTH'             ⍝ write next sequential record
       CTL210                         ⍝ check return code
0
       ⎕SVR¨ 'REC210' 'CTL210'   ⍝ retract to close the file
2 2
```

Figure 82. Sample APL2 Session to Communicate with AP 210

Calculating the Length of APL2 Variables: When a BDAM file is written using the CDR conversion option, a variable is written that contains header information describing the data in the variable. To calculate the space required for a variable written in CDR, use the attributes (⎕*AT*) system function. Enter the left argument as the integer 4. ⎕*AT* returns a two-item integer vector:

```
      REC210←2 2⍴'VOLUME' 1021 'CHANGE' ¯23
      4 ⎕AT 'REC210'
80 20
```

The first item is the total number of bytes required by the variable. The second item is the number of bytes required by the data portion. The difference between the two items is the number of bytes required by the CDR header.

For a further description of ⎕*AT*, see *APL2 Programming: Language Reference*.

Cautions

The following restrictions apply to the use of AP 210:

- After the file has been formatted, records can neither be added to the end of the file nor be physically deleted from the file using AP 210.
- Existing records cannot be changed in size except with the VAR or CDR conversion option.

Return Codes

The BDAM auxiliary processor reports return codes as a one-item numeric vector through the control variable. Figure 83 contains the return codes returned from the processor.

Figure 83 (Page 1 of 2). Return Codes from AP 210

Code	Description
-nn	Negative number: During formatting, the file was too small to contain the number of records requested. The value returned represents the number of records *not* formatted.
0	Successful completion of requested function. If returned for the initial offer, it means that the *syntax* of the value was correct. It does not mean that the specified ddname is correctly allocated to a file.
1	Invalid initial value. **Action:** Retract the variable, specify a valid syntax for the initial value, and reoffer the variable.
12	End of file on a sequential read request. The value in the data variable is empty. **Action:** To continue reading the file, reposition the record pointer using the control variable. To close the file, retract the record variable.
15	Wrong-length record on fixed-length output. All conversion options other than *VAR* or *CDR* require that all records on the file be exactly equal to the LRECL.
17	Record too large for file. With conversion option *CDR* or *VAR*, you specified the record variable with a value greater than the LRECL of the file.
30	Record not found. This error occurs when you try to read or write a record that is beyond the range of existing record numbers in the file.
440	Error in open for output. This error occurs under the following conditions: • You did not properly allocate the file. • You allocated a file that cannot be processed by BDAM. • A system error occurred in an attempt to find the allocation information to open the file. • Your installation prohibited your access to the file. • You requested formatting, but did not specify a valid number of records to format. • You specified read-only access (the **R** access option), and you specified a value in the record variable, indicating output processing. **Action:** Verify your allocation, authorization, and initial values.

Figure 83 (Page 2 of 2). Return Codes from AP 210

Code	Description
441	Error in open for input. This error occurs under the following conditions: • You did not properly allocate the file. • You allocated a file that cannot be processed by BDAM. • A system error occurred in an attempt to find the allocation information to open the file. • Your installation prohibited your access to the file. • You specified write-only access (the **W** access option), and you referenced the record variable, requesting a read operation. **Action:** Verify your allocation, authorization, and initial values.
443	Error in writing a record. The value assigned to the shared variable exceeds 32760 bytes.
444	Invalid data specified. It is the wrong size, shape, or data type.
445	Insufficient shared storage (SHRSIZE) for input data on a read request. **Action:** Reinvoke APL2 and specify a larger value for the SHRSIZE invocation option. **Note:** If you are using conversion option *VAR* or *CDR* and you think your SHRSIZE value is properly set, verify that the record you are trying to read was written by AP 210 using the *VAR* or *CDR* option. Try reading the record, using the BYTE conversion option.
901	I/O error in an attempt to read data. This return code is accompanied by detailed TSO messages that describe the error. It is issued in place of the system ABEND code 001 that would have been issued if the AP had not isolated and detected the error. Normally, this error means that you are trying to process the file with DCB attributes that do not match the physical data in the file. **Action:** Verify your allocation, initial values, and authorization.

Data Management Error Codes

For I/O errors not covered in Figure 83 on page 253, a decimal value is returned, representing internal error flags set by the operating system. The decimal value can be converted to its 4-byte hexadecimal representation in 0-origin by:

```
'0123456789ABCDEF'[⎕IO+2 4⍴(8⍴16)⊤CTL⍳10]
```

If the top row of the resulting matrix contains all zeros, the bottom row indicates a system ABEND code.

Otherwise, the first two bytes represent the exception code for BDAM, and the last two are the status bytes for the SYNAD routine. For exception codes and status byte information, see *MVS/DFP: Using Data Sets*, or the status information in the macro instruction manual appropriate to your system:

- *OS/VS Data Management Macro Instructions*.
- *MVS/XA Data Administration: Macro Instruction Reference*
- *MVS/ESA Data Administration: Macro Instruction Reference*

Chapter 24. AP 211—The APL2 Object File Processor

AP 211 provides a facility for storing APL2 arrays by name in an object file.

The AP 211 in APL2 Version 2 allows compatibility in APL2 object storage between versions of APL2 on different platforms. Its syntax is compatible with AP 211 found on all the APL2 platforms. In addition, it writes and reads a special file format that can be used directly by any AP 211 that reads and writes the same file format. Currently, this format is used in the VM, TSO, OS/2, Sun Solaris, and RISC System/6000* environments. For information about transferring files between platforms, see “Cooperative Processing” on page 86.

Shared Variable Overview

Figure 84. Shared Variable Overview for AP 211

SV Protocol	AP 211 Conventions
General	One variable. The variable is assigned the operation code and any parameters required by that operation code.
Maximum Number of Shared Variables	Limited only by SVMAX and SHRSIZE.
Names	Any valid APL2 variable name up to 255 characters.
Initial Values	Ignored.
Subsequent Values	Depend on the requested AP 211 operations. Reference a result that contains a return code, data, or both.
Access Control	1 0 0 1 Note that for effective use, the user or the workspace function should convert this to 1 0 1 1. This prevents a reference before the auxiliary processor has specified a result.

Commands Accepted by AP 211

The following description of the commands accepted by AP 211 assumes that a variable called *SHR*211 has been shared with AP 211. For example:

```
      211 ⎕SVO 'SHR211'
1
      ⎕SVO 'SHR211'
2
```

CREATE

```
SHR211←'CREATE' filename[rec_size]
return_code←SHR211
```

This command creates an AP 211 object file.

filename **(under TSO)**
The name of an MVS data set. Note that the TSO convention is followed, so that if the name is not enclosed in quotation marks, the current `TSO PROFILE PREFIX` is added to it.

Note: Do not confuse the quotation marks that delimit the filename itself with the quotation marks used for a fully-qualified data set name. If a fully qualified name is a literal, it must be entered as:

```
'''fully.qualified.dataset.name'''
```

filename **(under CMS)**
The name of a CMS file, in this format:

```
'filename filetype [filemode]'
```

The *filemode* defaults to `A1` on CREATE. On USE and DROP, standard CMS search order is used.

rec_size The record size used to store APL2 objects in the file.

APL2 objects stored in the file use one or more records depending on size. Objects smaller than the record size still use a full record and any excess space is unused. Large objects that span several records cause more file input/output. Therefore, you should carefully select a record size that optimizes file space utilization and I/O count.

AP 211 uses a default record size of 1024 if none is specified. The record size must be at least 128, not greater than 32704, and a multiple of 64. If the user specifies a record size within the limits, but not a multiple of 64, it is rounded to the next multiple of 64.

return_code
An integer return code.

On CMS, the CREATE command is required to set up an AP 211 file. Once created, the file can grow to a size limited only by space available on the minidisk.

On TSO, the CREATE command is optional. If the CREATE command is issued, AP 211 allocates a file that starts at about 70K bytes, but under ideal conditions is expandable to nearly 10 megabytes. If the user wishes to have a file with different size parameters, the CREATE can be bypassed by preallocating a data set meeting the requirements for AP 211. The data set must be DSORG(PS) or DSORG(PSU), RECFM(FB), with a record length between 128 and 32704 that is a multiple of 64. Any block size that is a multiple of the record length and is less than or equal to 32704 is accepted, but larger block sizes in general give better performance.

DROP

```
SHR211←'DROP' filename
return_code←SHR211
```

This command deletes an entire APL2 object file from disk.

USE

```
SHR211←'USE' filename [user_id] [access]
(return_code rec_size)←SHR211
```

Opens an AP 211 file. It is then in a state where it can be managed by the SET, GET, ERASE, and LIST commands.

user_id A scalar integer that can be used to implement an audit trail of updates, particularly when a file is shared among users. The default value is ↑⎕AI.

access Specifies the type of access desired. 'READ' provides read-only access. 'UPDATE' and 'PRIVATE' both provide read/write access.

In APL2/6000 the terms 'UPDATE' and 'PRIVATE' have different meanings. 'UPDATE' provides shared read/write access, and 'PRIVATE' provides exclusive read/write access.

In the mainframe AP 211, 'UPDATE' and 'PRIVATE' are both accepted for compatibility, but have the same effect. File sharing capabilities are not built in to AP 211, but are governed by the operating system.

In CMS, the mode in which the disk is linked determines what sharing is possible. Note, however, that write access by multiple simultaneous users can result in corruption of the file and is not advised.

In TSO, AP 211 uses DISP=OLD when read/write access is requested to provide exclusive read/write access.

If no access mode is specified, read/write access is provided.

rec_size The record size of the file.

RELEASE

```
SHR211←'RELEASE'
return_code←SHR211
```

Releases the object file that was associated with a shared variable. This command is issued implicitly when retracting or expunging the shared variable, or if a subsequent USE, CREATE, or DROP are specified to the same shared variable.

SET

```
SHR211 ←'SET' name APL2_object
return_code←SHR211
```

Stores an APL2 array in the object file

name A character string identifier to be assigned to the object in the file.

The maximum permitted length of *name* is 31 characters.

If the name is already in use, the new definition is added to the object file, and the old definition is deleted. The space taken by the old definition is freed for later use.

APL2_object

Any APL2 array. Either a variable name or literal data can be given. Because the *SET* command requires a three-element vector to be specified to the shared variable, parentheses might be needed to form the array if it is entered as literal data.

GET

```
SHR211←'GET' name
(return_code APL2_object)←SHR211
```

This command returns the array (if any) associated with a given name.

RENAME

```
SHR211←'RENAME' oldname newname
```

Renames an object stored in an AP 211 file.

ERASE

```
SHR211←'ERASE' name
return_code←SHR211
```

This command allows you to remove an APL2 array from an object file, and makes its storage available for other updates. Note, however, that the overall size of the file remains unchanged.

LIST

```
      SHR211←'LIST' 'NAMES'
      ρ⎕←SHR211

Object1
Object2
Object3
3 31

      SHR211←'LIST' 'ATTS'
      ρ⎕←SHR211
1 1001 1993 1 2 12 30 14 12
2 1001 1993 1 2 12 30 14 12
1 1001 1993 1 2 12 30 14 12
3 9
```

This command allows you to list the names or attributes of all objects in the file.

For the attributes form, the information returned is:

1. Number of records used for this object
2. User ID number (as specified on the *USE* command) of the user who last updated this object
3. The date and time the object was updated (in ⎕*TS* format, GMT)

Each row in this list corresponds to the equivalent row in the list of object names.

Return Codes

The following figure lists the AP 211 return codes.

The negative return codes from AP 211 are compatible across all the APL2 environments. The positive return codes, however, can be different, since these are file system services return codes. Applications written to run on different machines should not depend on specific positive return codes, instead having more general error handling routines for those cases.

Figure 85 (Page 1 of 2). Return Codes from AP 211

Code	Description
0	Success
2	File, data set or DDname not found
4	Attempt to write when open for read-only access
7	Unsupported RECFM
8	Insufficient storage for file services
10	General File I/O Error
14	Unsupported file type
15	File Space Exceeded
¯2	Rank error
¯3	Length error

Figure 85 (Page 2 of 2). Return Codes from AP 211

Code	Description
¯4	Type error
¯7	Invalid command
¯8	Invalid block size
¯9	Not an AP 211 file
¯10	No file accessed
¯11	Name has no value
¯12	Invalid object name
¯13	Error encountered during set operation
¯14	Invalid file name
¯15	Invalid access mode
¯16	Invalid user ID
¯17	Filename already exists or is in use
¯18	Insufficient user authority
¯19	Name already exists
¯20	Object CDR type not recognized
¯21	Temporary interlock
¯22	Unexpected SVP return code
¯24	Media full
¯25	Data out of range
¯26	Insufficient storage for AP 211 work areas
¯27	Abend. AP 211 has restarted. Variables are still shared, but all files have been released.

Chapter 25. APL2 Shared Variable Interpreter Interface

APL2's Shared Variable Interpreter Interface provides a set of protocols whereby an APL2 interpreter can be controlled through a shared variable rather than through a terminal or file input. The normal session input and output are replaced with a single shared variable through which communication occurs. This shared variable, and hence the interpreter, can then be managed by a user or program running under another user ID.

The shared variable interpreter interface is started by use of the APL2 invocation keyword SMAPL. If the SMAPL parameter is numeric, the interpreter uses it as the processor ID with which it should share a variable called *APL2*. This variable is then used for all input and output to the interpreter. The variable is shared within the interpreter and is not available to nor does it conflict with variables and programs being run by the interpreter.

The Remote-Session Manager, *RAPL2* uses this interface to give a user the ability to conveniently control a remote interpreter. For information, consult the discussion of the *RAPL2* function in *APL2/370 Programming: Using the Supplied Routines*.

Using the shared variable interface to an interpreter has some impact on the use of system resources. For example, *WS FULL*, can happen on any output as the resulting array is prepared for a shared variable assignment when in a directly controlled session, no space would be required.

Once an interpreter is running using the shared variable interface, it operates normally except that its input and output is through the shared variable. It is the responsibility of the interpreter's shared variable partner to manage the variable. The interpreter processes requests until instructed to shutdown either by a shutdown control signal or an *)OFF* or *)CONTINUE* command.

The rest of this chapter describes the protocols that are used when communicating with an interpreter through the shared variable interface.

Shared Variable Interpreter Interface Protocols

The Shared Variable Interpreter Interface is designed to allow an APL2 function or user running under another user ID to control the interpreter. When the shared variable interpreter's shared variable offer is matched, the interpreter sends its current system variable values and any output produced during invocation. When the shared variable interpreter exhausts the input provided during invocation, the interpreter sends a control code indicating it is ready to receive input.

The interpreter executes any expressions or commands sent to it and sends back any messages and arrays generated through the shared variable. When the expression or command has completed, the interpreter again sends a control code indicating it is ready to receive input.

Two types of data can be sent to the shared variable interpreter:

- Character vectors are sent to represent terminal input to the interpreter, in response to a □ or ⍞ request in an APL statement, or in response to an imme-

diate execution or editor 1 prompt from the interpreter itself. Character vectors sent at any other times are likely to be ignored.

- Simple two element integer vectors may be sent as control signals to the APL2 session where the shared variable interpreter is running. They include execution control signals that can be sent at any time, and output control signals that can only be sent when input has been requested. These codes are defined in "Interpreter Input Data."

Communication from the shared variable interpreter is always in the form of 3-item nested arrays, as described in "Interpreter Output Data" on page 263. The first two items indicate the type of output in the third item. Array output (implicit display at the end of an APL statement, and output produced by ∇-tracing) is sent as an arbitrarily structured array in the third item. Other output is sent as character vectors in the third item.

Some messages generated during executing commands cannot be sent in a single specification of the shared variable. In these cases, the interpreter sends these messages in pieces. A control code is defined that indicates that a message is incomplete. This code is typically used if the expression being executed uses ⍞ output.

When an)*OFF* or)*CONTINUE* system command or a shutdown request is sent to the shared variable interpreter, the interpreter sends termination messages and retracts the shared variable. If the interpreter's partner retracts the shared variable before the interpreter retracts it, the interpreter does not shut down; the interpreter simply continues to execute the expression it is processing or wait for the next input.

Shared Variable Overview

Figure 86 provides an overview for sharing variables with an interpreter.

Figure 86. Shared Variable Overview for Interpreter

SV Protocol	Interpreter Conventions
General	One variable for passing input to and receiving output from interpreter.
Maximum number of shared variables	1 per interpreter
Name	*APL2*
Initial Values	None. Initial values are ignored by the interpreter.
Subsequent values	Input to the interpreter is either character vectors or pairs of scalar integer control codes. Output from the interpreter is 3-element vectors: return code, type code, and data.
Access Control	Interpreter sets `1 1 1 1`.

Interpreter Input Data

Input data for the interpreter may be either control signal codes or character vectors for execution. Control signal codes are pairs of scalar integers. Execution Control codes can be sent at any time. Character vectors for execution and other types of control codes should only be sent when the interpreter has indicated it is ready for input. When the interpreter has not indicated it is ready for input, a shared variable interlock may occur if data is sent. If the interpreter has not indicated it is ready for

input, it may ignore any data it receives except for defined execution control signal codes.

It is possible for the interpreter's partner to not be able to reference the shared variable (a *WS FULL* may occur during the reference.) In these cases, the partner should specify a no operation control signal. This frees the interlock condition and allows the interpreter to again reference and specify the variable.

Figure 87. Interpreter Control Signal Codes

Execution Control Signal Codes	
0 0	No operation
0 1	Attention
0 2	Interrupt
0 3	Suppress output
0 4	Shutdown
Output Control Signal Codes	
1 0	Set output to array mode (default)
1 1	Set output to line mode - Formats array output and sends data line by line.

Interpreter Output Data

The shared variable interpreter sends messages and arrays in 3 item arrays. The first item is an integer scalar return code. If the return code is not negative, then it and the rest of the array is defined as *⎕EC* output. If the return code is negative, then it indicates whether the rest of the array provides message, system variable, stacked input, or array output data. The second item is a 2 element integer vector. The item is always 0 0 for data other than messages and input requests. For messages, the first element of the second item indicates what type of message is sent or is an indication what type of prompt is requested. The second element indicates whether the message is complete. If the second element is 0, then the message is complete. If the second element is 1, then more information is sent to complete the message. 1 is used for ⍞ output from an APL statement, and also for input requests when prompt data is provided.

Figure 88. Interpreter Output

Messages and Input Requests

In the following definitions, *n* may be 0 indicating the message is complete or 1 indicating the message is incomplete.

¯1	0 *n*	System message
¯1	1 *n*	APL (immediate execution) prompt - The interpreter is ready for normal input. The interpreter provides the normal 6 blank prompt.
¯1	2 *n*	Quad prompt - The interpreter is ready for ⍞ input. The interpreter does not provide the ⍞:
¯1	3 *n*	Function definition prompt - The interpreter is ready for function definition input. The interpreter provides the line number prompt.
¯1	4 *n*	Function line display
¯1	5 *n*	Function time stamp
¯1	7 *n*	Function name line number (stop or trace)
¯1	8 *n*	Error message
¯1	9 *n*	System command result
¯1	10 *n*	Traced result
¯1	11 *n*	Quad output
¯1	12 *n*	Quad-Prime output
¯1	13 *n*	Debug output
¯1	14 *n*	Quad-Prime prompt - The interpreter is ready for ⍠ input. The interpreter provides ⍠ prompt.

System Variable Values.

This value is sent whenever one or more of the system variables important to a session are changed. The value is sent just before an output is sent. The third item contains the values of the listed items that currently exist in the shared variable interpreter. They can be used to modify the local value so that displays and prompts appear correctly.

¯2	0 0	⎕*PW* ⎕*PP PBS* ⎕*PR* ⎕*FC*

Stacked Input

This value is sent whenever the interpreter requests input and the input stack is not empty.

¯3	0 0	The character vector that was stacked.

Part 3. Associated Processors

Chapter 26. External Names and Associated Processors 268
Applications of External Names 268
Managing External Names from APL 270
Creating and Destroying an Association 270
Invoking an External Name 271
Querying an Associated Name 271
Checking the Association Information 271
Checking for Active Associations 272
Avoiding Name Conflicts 272
Environmental Considerations 273

Chapter 27. Processor 10—Communication with REXX 274
Overview 274
Detailed Description 275
Using REXX Functions 275
Constructing the Argument 277
Handling Results and Errors 278
Accessing REXX Variables and Constants 279
Associating Names 279
Values 280
Built-in Functions 280
Reading and Writing CMS Files 281
Reading and Writing MVS Sequential Data Sets 282
Querying a CMS File Status 284
Querying an MVS Data Set Status 285
Executing APL Arrays as REXX Programs 285
Unexpected Errors and Other Considerations 286
Failure when Associating a Name 286
APL Errors 287
Non-APL Error Messages 287
REXX Return Code 20040 287
“Missing” Argument Strings 288
Truncated Data Returned under TSO/E 289
Other Considerations 289
Environment 289

Chapter 28. Processor 11—Calling Compiled Programs 291
⎕*NA* Syntax for Non-APL Programs 292
Processor 11 Overview 292
Introduction 293
Processor 11 Glossary 293
Usage Overview 295
Routine Descriptions 295
Building Routine Descriptions 296
Building NAMES Files 297
Self-Describing Routines and NAMES File: Pros and Cons 298
Routine Lists 298
BUILDRL and Interface Management Routines 301
BUILDRL Interface Management and Self-Describing Routines 302
C/370 Scalar Integer Results 303

Environments . . . 303
Interface Details . . . 307
Routine Description Tags . . . 307
Argument Patterns . . . 310
Updated Arguments and Results . . . 313
Result Patterns . . . 314
C/370 Results . . . 316
:LINK.FUNCTION Arguments . . . 317
Explicit Results, Function Valence, and Operator Valence . . . 317
Explicit Results . . . 317
Function Valence . . . 318
Operator Valence . . . 318
NAMES Files . . . 318
Processor 11 Non-APL Routine Description Tag Rules . . . 319
System Usage Guidelines . . . 321
Linkage Conventions . . . 321
OBJECT and FORTRAN Linkage . . . 321
FUNCTION linkage . . . 322
Unexpected Errors . . . 322
Processor 11 Routine Search Order Guidelines . . . 325
External Function Names . . . 325
CMS Search Order Guidelines . . . 325
Using Routines Defined as Nucleus Extensions . . . 326
Using Routines in TEXT Decks or TXTLIBs . . . 326
TSO Search Order Guidelines . . . 326
Using Routines in TSO Load Libraries . . . 326
Using Routines in the Standard TSO Search Order . . . 327
Link-Editing External Routines . . . 327
Link-Edit Tools . . . 328
Using AP2MP11L and AP2MP11M . . . 328
Link-Editing External Routines on CMS . . . 329
Link-Editing External Routines on TSO . . . 330
Installation of External Routines . . . 331
Extended Addressing Considerations . . . 331
Preloading and Sharing External Routines . . . 331
Execution Time Libraries . . . 332
VS FORTRAN Execution Time Libraries . . . 332
Other Processor 11 Considerations . . . 332
Using Self-Describing Routines from Non-APL Programs . . . 332
Using Modules with Routine Lists from Non-APL Programs. . . . 332
FORTRAN Considerations . . . 332
APL2 versus FORTRAN Array Ordering . . . 332
FORTRAN External Names . . . 333
FORTRAN Linkage Convention . . . 333
FORTRAN Common . . . 333
FORTRAN Functions . . . 333

Chapter 29. Processor 11—Access to Namespaces . . . 334
Overview . . . 334
Detailed Description . . . 336
Creating Namespaces . . . 338
Workspace Names . . . 340
Accessing Objects in Namespaces . . . 340
NAMES Files . . . 342

Using Namespaces 343
Namescopes 344
Combining Several Namespaces in a Member 347
CMS Namespace Routine List Example 348
TSO Namespace Routine List Example 349
Link-Editing Namespaces 349
Unexpected Errors and Other Considerations 349

Chapter 30. Processor 12—Files as Arrays 352
□*NA* Syntax for Processor 12 352
Supported Primitive Operations 354
APL Files as External Variables 355
Record-oriented Files as External Variables 356
Format Descriptors for External Variables 358
Processor 12 Errors 359

Chapter 26. External Names and Associated Processors

Names are used in APL expressions to identify variables, defined functions and operators, and constants (such as labels). When APL encounters names during the execution of expressions, it passes control to the appropriate routines in the interpreter for evaluation.

Through the use of $\square NA$, and by associating a name with a specific processor, an APL application program can cause that name to be processed by routines in the specified associated processor instead of the APL interpreter. Associated processors provide an alternate mechanism for handling the evaluation of APL names.

Three processors are provided with APL2:

- Processor 10 provides facilities through which programs written in REXX may be executed. It also provides facilities to reference or specify REXX variables and to create or manipulate REXX EXEC's.
- Processor 11 provides facilities through which programs written in languages other than APL may be called. The processor provides services so that you can specify how to map APL data to and from the data structures that can be required by these programs.

  Processor 11 also provides facilities that allow access to APL objects in "namespaces." Because each namespace has its own namescope, an application placed in a namespace can avoid name conflicts with other applications.
- Processor 12 provides facilities through which APL data files and operating system files can be accessed using normal APL2 syntax. The processor makes the file appear to exist as an object in the workspace although it may actually be larger than the available workspace size.

Applications of External Names

A name associated with a processor is called an external name.

External names have a variety of uses in building production applications. By giving you additional options in the ways in which you process information from APL, external names help improve productivity. Some of the reasons you might use external names are:

- Reuse of Existing Programs

  A principal objective of Processor 11 is to permit you to reuse many existing programs without any need to modify them. Processor 11 provides mechanisms by which you describe where the programs exist, what data structures they require in arguments, and what execution environment, if any, is needed. This information is provided to Processor 11 either in a file or link-edited with the program. Once the information has been provided, you can then use Processor 11 to access these programs from your workspace just as if they were APL functions.

- Synchronous Access to System Information

  Sometimes an application needs easy access to information about the host system or from another application subsystem. Since information, like account codes, varies widely among installations, it is impractical for APL to provide it directly. You can use external names, however, to temporarily “escape” APL, access the information, and bring back the results to the workspace for use by the application.

- Improve Performance

  It is common for an application to have a uniquely tailored data structure or algorithm that is used widely by the application's own functions. This application-specific feature often assumes a fundamental, what APL might term “primitive,” nature and frequently becomes the bottleneck that limits either the capacity or performance of the application. External objects can be used to overcome such problems by permitting you to enclose the definition in compiled code. Because external objects are syntactically equivalent to the APL object from which they were derived, you need only replace the APL definition in the workspace with the external name association; much like copying an object. The remainder of the application is unmodified.

- Maintain Shared Code

  Shared code is also important to installations because only a single copy need exist in the system, no matter how many users are accessing it. This can significantly reduce input/output and real storage requirements.

  Sometimes an application is built on gate functions that control access to critical resources like files. These functions mask the application from the internal structure, location, or other attributes of the resource so that these may be changed in a transparent manner. If a gate function is modified, it must be updated in all the saved workspaces where it exists. This can be a burden in practice. However, since only the information that characterizes the association is known about an external function in a saved workspace, an external function can be replaced and be available to all subsequently activated application workspaces.

- Avoid Name Conflicts

  Users who attempt to combine APL applications or parts of applications together often encounter situations where the same name exists in more than one of the applications. Since names in the active workspace must be unique, the applications must be modified to have unique names if they are to be combined into a single workspace. Namespaces, supported through Processor 11 provide an alternative solution to this problem since each namespace contains its own name space. Names need only be unique to the namespace in which they reside.

- Increase Effective Workspace Size

  Processor 11 permits the use of large applications, and Processor 12 permits the use of large files, as if they resided within your own workspace. In fact, the storage requirements within the workspace may be a small fraction of the size of the file or application.

Managing External Names from APL

The system function, ⎕*NA*, is used to associate a name with a processor or to query the association of an existing name. A formal and detailed description of ⎕*NA* is included in *APL2 Programming: Language Reference*.

Creating and Destroying an Association

Briefly, ⎕*NA* in its dyadic form is used to associate a name with a processor. The right argument lists the name or names to be associated with the processor and then activated. The left argument of ⎕*NA* identifies the processor and provides information that is passed to the processor when the name is activated. For example:

```
      0 11 ⎕NA 'OPTION'
1
```

causes the name OPTION to be associated with Processor 11. The result, `1`, indicates that the name has been accepted by Processor 11 and the association is active; a result of `0` would indicate the processor was unable to activate the association due to an error, or perhaps due to a lack of resources. When a name is successfully associated with a processor and activated, the processor specifies the name class and valence (`1 ⎕AT`) for the name. The association, name class, and valence remain in effect until the object is deleted from the workspace. The association, name class and valence remain in effect even after using the commands)*SAVE*,)*LOAD*, or)*COPY*. The information necessary to produce an association is produced by `2 ⎕TF` and)*OUT* for use by `2 ⎕TF` or)*IN*.

A name may be disassociated from a processor by deleting it from the workspace with ⎕*EX*,)*ERASE*, or by completing execution of a function that localized the name. A name is also disassociated when)*IN* or)*COPY* replaces an associated object with another object of the same name.

When a name is disassociated from a processor, or when the active workspace is replaced with)*CLEAR*,)*LOAD*,)*OFF*, or)*CONTINUE*, the processor is contacted to allow it to free resources associated with that name. At this point the name is said to be inactive, even though it may still be associated with the processor in a workspace that was previously saved. If the workspace is subsequently reloaded, the processor is be contacted to reactivate the name when the name is first encountered in the execution of an APL expression or in the right argument of dyadic ⎕*NA*. If the processor is unable to reactivate a name encountered during the execution of an APL expression, a *VALENCE ERROR* or *VALUE ERROR* generated.

Invoking an External Name

When a name that has been associated with a processor is encountered during execution of an APL expression, control is passed, along with any arguments associated with the operation, to the processor. The processor then manages the execution of the requested routine and returns results or an error condition to APL. For example:

```
      3 11 ⎕NA 'MEAN'       ⍝ AN AVERAGE FUNCTION
1
      MEAN 1 2 3 4
2.5
      ⎕←RESULT←2+MEAN 1 2 3 4
4.5
      MEAN 'ABCD'
DOMAIN ERROR
      MEAN 'ABCD'
      ^
```

Names defined and associated with processors through the use of dyadic ⎕*NA* appear and act like any other names in the APL workspace. They are reported by)*NMS*,)*FNS*,)*VARS*,)*OPS*, and ⎕*NL*, and they may be used in APL expressions. They are saved as part of a saved workspace and retain their name class and association when subsequently loaded or copied. When such a name is erased or otherwise deleted (as the result of localization,)*COPY*,)*ERASE*, etc.), it is no longer associated with any processor. Since the name is then undefined, it is then available to be defined as an APL function, operator or variable, or associated with another processor.

Querying an Associated Name

Checking the Association Information

Monadic ⎕*NA* is used to query the name class and associated processor for one or a list of names. For example:

```
      3 11 ⎕NA 'GEORGE'
1
      ⎕NA 'GEORGE'
3 11
```

The following expression lists all names that are associated with processors other than APL:

```
      (0≠,0 1↓⎕NA ⎕NL 1 2 3)⌿⎕NL 1 2 3
GEORGE
MEAN
```

Checking for Active Associations

Dyadic ⎕*NA* can be used to query a previously associated name to find out if it is currently active. A result of 1 indicates the association is active, while a result of 0 indicates that it is inactive. An inactive association is most likely to result after loading a saved workspace. Usually, the processor can no longer find the file or program requested by the association. An attempt to use a function whose association is currently inactive results in *VALENCE ERROR*. For example:

```
      ⎕NA¨'GEORGE' 'SALLY'
 3 11  3 11
      ⍝ GEORGE AND SALLY ARE
      ⍝ 'ASSOCIATED' WITH PROCESSOR 11
      (⊂3 11) ⎕NA¨'GEORGE' 'SALLY'
1 0
      ⍝ SALLY IS 'ASSOCIATED' BUT NOT 'ACTIVE'

      SALLY 'GO ROUND THE ROSES'
      VALENCE ERROR
      SALLY 'GO ROUND THE ROSES'
      ^
```

Objects in the APL workspace that are not associated with an external processor are associated with the APL interpreter, which can also be thought of as a processor. Monadic ⎕*NA* returns a 0 for the processor number for such names to indicate that they are handled by the APL interpreter. For example:

```
      VAR←1 2 3
      ⎕FX 'Z←FN A' 'Z←1÷A'
FN
      ⎕NA 2 3⍴'VARFN '
2 0
3 0
```

Specifying processor 0 in dyadic ⎕*NA*, while valid, has no effect for an undefined name:

```
      2 0 ⎕NA 'MARY'
0
      ⎕NA 'MARY'
0 0
      ⎕NC 'MARY'
0
```

Avoiding Name Conflicts

A second, or surrogate, name may be used with the name of any object in monadic or dyadic ⎕*NA*:

```
      3 11 ⎕NA 'NANCY LIL'
1
```

In such cases, the first name, *NANCY*, is used to refer to the function in APL expressions in the workspace. The second or surrogate name, on the other hand, is used to identify the object, *LIL*, to the processor. Surrogate names are particularly useful when a processor requires a specific name that would cause a name conflict with other names in the application.

Use of a wrong or conflicting surrogate name in dyadic ⎕*NA* causes ⎕*NA* to return a 0 result. The ⎕*TF* function can be used to determine the correct surrogate name:

```
      3 11 ⎕NA 'NANCY MCGILL'
0
      2 ⎕TF 'NANCY'
3 11 ⎕NA 'NANCY LIL'
```

Environmental Considerations

Associated Processors and any programs they may call execute as direct extensions of the APL language. The programs themselves are presumed to be well-behaved production programs. As such, they are expected to preserve the APL execution environment and not compromise the integrity of the APL workspace. If an application requires isolation from the APL environment rather than the synchronous behavior of external names, you should consider a solution based on shared variables and Auxiliary Processors as described in Part 2.

While you are testing and debugging external functions, it is recommended that you invoke APL2 with the DEBUG(1) invocation option. This causes any)*MORE* messages generated by an associated processor to be displayed automatically as soon they are generated.

Chapter 27. Processor 10—Communication with REXX

REXX is a computer language available, like APL2, under CMS and TSO/E.

Processor 10 provides:

- Use of REXX functions as APL functions
- Access to REXX variables
- Manipulation of REXX programs as APL arrays
- Access to CMS files and MVS sequential data sets

This chapter provides a very brief introduction to REXX, and some examples of using the REXX language, but does attempt to provide a tutorial on REXX. Detailed information on REXX can be found in manuals supplied with CMS or TSO/E.

Overview

The REXX language provides functions and variables, objects that are familiar to users of APL. REXX functions are similar to APL monadic functions and REXX variables are like APL character vectors. REXX communicates exclusively with character strings so that not only REXX variables, but also the arguments and results of REXX functions, are characters.

To use a REXX function from APL, you must first establish an association with dyadic $\square NA$ (See “Using REXX Functions” on page 275). The function thus established is monadic, and its argument is either a character vector or vector of character vectors (See “Constructing the Argument” on page 277). The function may be built-in to REXX, part of a REXX function package, a REXX program in an EXEC file, or a module. The result of invoking a REXX function successfully is always a character vector. If REXX detects an error while interpreting the function, the result is a numeric scalar giving the REXX return code. (See “Handling Results and Errors” on page 278).

As an example, consider the use of the REXX built-in functions DELWORD, FIND, and WORDS that operate on blank-delimited substrings of vectors, called “words.”

DELWORD (*string,offset,length***)**
deletes *length* words from *string* beginning at *offset*

FIND (*string,target***)**
returns the offset in *string* of the *target*

WORDS (*string***)**
returns the number of words in the *string*

```
      ⍝----- START OFF SIMPLY -----
      3 10 ⎕NA 'DELWORD'
1
      DELWORD 'Now is the time' '2' '2'
Now time
      DELWORD 'Now is the time' '3'
Now is
      DELWORD 'Now is the time' '5'
Now is the time
      DELWORD 'Now is the time'
20040
      ⍝  DELWORD requires two arguments
      ⍝           See DMSREX475E Error 40 message
      ⍝----- NOW FOR A MORE PERENNIAL EXAMPLE -----
      (⊂3 10) ⎕NA¨ 'FIND' 'WORDS'
1 1
      HERBS←'Parsley, sage, rosemary and thyme'
      SUBHERBS←'sage, rosemary'
      FIND HERBS SUBHERBS
2
      WORDS SUBHERBS
2
      ⍝  NOTE THAT THE RESULT IS CHARACTER
      2=WORDS SUBHERBS
0
      2=⍎WORDS SUBHERBS
1
      ⍝  NOTE THAT CHARACTER RESULT CAN BE USEFUL
      DELWORD HERBS (FIND HERBS SUBHERBS) (WORDS SUBHERBS)
Parsley, and thyme
      ⍝----- COMBINE APL and REXX -----
      ∇Z←PHRASE REMOVE_FROM STRING;POSITION
[1]   Z←STRING                                ⍝ ASSUME NOT FOUND
[2]   →('0'= POSITION←FIND STRING PHRASE)/0   ⍝  - DONE IF NOT FOUND
[3]   Z←DELWORD STRING (POSITION) (WORDS PHRASE) ⍝ DELETE 'PHRASE'
[4]   ∇
      SUBHERBS REMOVE_FROM HERBS
Parsley, and thyme
```

The example illustrates the basic technique for building APL applications that use REXX functions. Although the example limited itself to built-in REXX functions, you may also access REXX external functions. These external functions may be written as modules or as REXX programs. By writing your own functions in REXX, you can enhance the power of APL with the string handling and system access of REXX.

Detailed Description

This section provides a detailed description of how you can use processor 10.

Using REXX Functions

Before you can invoke a REXX function, you must first use dyadic ⎕*NA* to associate the name of the function with Processor 10. Processor 10 nearly always accepts a name association from ⎕*NA* provided there is virtual storage available to build any required internal control blocks. Since there is no way for Processor 10 to validate the existence of any REXX function, it assumes that you know what you're doing and returns a 1 to show that the association is active.

Processor 10 expects that the first item in each row of the left argument of dyadic ⎕*NA* be the name class of the name to be associated with Processor 10. Name classes of 1, 2 and 3 are valid.

```
      3 10 ⎕NA 'DELWORD'
1
```

requests association with the REXX function DELWORD.

```
      2 10 ⎕NA 'SYSTEM'
1
      1 10 ⎕NA 'VERSION'
1
```

request association of the variable SYSTEM and the constant VERSION. If 0 is specified as the first item of a row in the left argument of ⎕*NA* and if the name already exists, Processor 10 assumes the correct name class. If the name does not exist, it assumes the name class 3.

Because names (and surrogate names) used with ⎕*NA* must also be valid, ordinary APL names, you cannot directly access REXX functions whose names contain the characters '#.$@!&?'. If you need to use a function that has any of these characters, you can easily write a REXX function that has a valid APL name and invoke it to use the other function.

Processor 10 also provides a set of built-in functions for use in manipulating REXX programs and files. The names of these built-in functions are distinguished by beginning with the character '∆'. They are described in “Built-in Functions” on page 280.

External names persist in the workspace. Once you have associated a name with a REXX function, you can continue to use the function (as long as the REXX interpreter can find it) just as if it were a defined function in your workspace.

When the function is invoked, Processor 10 calls REXX to execute it. If the function is built-in to REXX, REXX executes it directly. Otherwise REXX searches for an external REXX function of that name.

Under CMS, REXX searches for an external function by first prefixing the name with 'RX' and searching for a module or function package. If the prefixed name is not found, REXX uses the name without a prefix to search for an EXEC or MODULE. If the function still cannot be found, then REXX returns a numeric 20043 (Routine Not Found) as the result.

Under TSO/E, REXX searches for an external function by looking for a partitioned data set member of the same name in the DDNAMES SYSEXEC and SYSPROC. The member must include a comment in the first line containing the word 'REXX'. If the function still cannot be found, then REXX returns a numeric 20043 (Routine Not Found) as the result.

Constructing the Argument

The argument to a REXX function contains from 1 to 20 items. Each item may be either:

- A string—that is, a character vector or scalar

or

- Omitted—indicated by either:
  1. (⍳0) - an empty numeric vector
  2. (0⍴⊂'') - an empty enclosed character vector. The vector is enclosed because empty character vectors are valid REXX strings.

For example, the function USERID expects no strings at all and would be invoked in a REXX program with USERID(). To invoke it from Processor 10, you must provide an argument that contains one omitted string. An omitted string is not the same as an empty one.

```
      3 10 ⎕NA 'USERID'
1
      USERID ⍳0
WHEELS
      USERID ''
20040
```

The argument is always a character vector in form, even though the intent of the argument may be numeric. For example, the REXX function SUBWORD expects at least two strings: a list of words, and the index of the first word to be returned.

```
      3 10 ⎕NA 'SUBWORD'
1
      X←2
      SUBWORD 'Now is the time' X
DOMAIN ERROR
      SUBWORD 'Now is the time' X
      ^
      ⍝  But it is easy to use the FORMAT primitive
      SUBWORD 'Now is the time' (⍕X)
is the time
```

All of the items of the argument to a function must be character vectors or scalars.

```
      SUBWORD 'Now is the time' (1 2⍴' 2')
RANK ERROR
      SUBWORD 'Now is the time' (1 2⍴' 2')
      ^
```

If you attempt to invoke a REXX function with an argument consisting of more than 20 strings, Processor 10 signals an APL error.

```
      SUBWORD 21⍴⊂'FRED'
LENGTH ERROR
      SUBWORD 21⍴⊂'FRED'
      ^
```

If you provide either too few or too many strings for a specific function, REXX generates Error 40, which is returned to APL as a numeric return code.

```
      SUBWORD 1⍴⊂'FRED'
20040
```

This can sometimes happen inadvertently if you forget that when simple scalars are juxtaposed, you get a simple vector, not a nested vector.

```
      DELWORD 'A B' '2'
A
      DELWORD 'A ' '2'
A
      DELWORD 'A' '2'
20040
     'A' '2'
A2
      ⍝
      ⍝ USE OF 'RAVEL EACH' WILL OVERCOME THIS
      ⍝
      DELWORD ,¨ 'A' '2'
A
```

Handling Results and Errors

The result of a successful REXX function is always a character vector. If the result is needed in a numeric context, you can use one of the APL EXECUTE functions (⍎,⎕*EA*,⎕*EC*) or the *CTN* function of Processor 11 to convert from character to numeric.

Some REXX functions return a character vector that is really several strings joined together by a separator character; the New-Line character (x'15'), for example. An example of this under CMS is the DIAG function, which can return the results of a CP command. You can use the *PARTITION* function (⊂) to convert such a partitioned string into an APL vector of vectors.

Other REXX functions, such as STORAGE under CMS, are capable of returning a vector that might represent a structure of data containing both numbers and characters. You can often use the *RTA* function of Processor 11 to remap such data into an APL2 general array.

There are two classes of errors that can occur using Processor 10: either an APL error in the workspace, or some kind of REXX error outside of APL.

The first situation usually is caused by an attempt to invoke the function with something other than a character vector. The error may be reported as *DOMAIN*, *RANK*, or *LENGTH* (see “Constructing the Argument” on page 277).

REXX errors are typically some sort of syntax error in a REXX EXEC producing a value in the range 20001 through 20100. These are always returned as a numeric scalar result, so if you check for such a result you can always positively determine if an error occurred.

Accessing REXX Variables and Constants

REXX has a direct interface to its current variable pool through the EXECCOMM subcommand interface under CMS, or IRXEXCOM under TSO/E. The interface permits you to inspect or change the value of REXX variables in the pool. In addition, there are constants in the pool that contain status information such as the level of REXX, the source of the active EXEC, etc. Since REXX limits access only to variables of the most current program, this facility is principally of value in an application where an EXEC invokes APL and wants to communicate parameters or get return codes.

The EXECCOMM and IRXEXCOM interfaces are available only when REXX is active. Thus REXX variables and constants can only be accessed when APL2 has been invoked from a REXX exec.

Associating Names

Before you can use a REXX variable, you must first use dyadic ⎕*NA* to associate the name of the variable with Processor 10. Processor 10 accepts a name association from ⎕*NA* as long as there is virtual storage available to build the required internal control block and as long as the EXECCOMM subcommand or IRXEXCOM has been made available by REXX. If the EXECCOMM or IRXEXCOM environment is not available when a request is received to associate a name with a variable or constant, Processor 10 queues the following messages and returns 0 to ⎕*NA*.

Under CMS:

```
AP2VN01011 EXECCOMM UNAVAILABLE
```

Under TSO/E:

```
AP2TN01011 IRXEXCOM UNAVAILABLE
```

This message displays in response to)*MORE*.

Name Length: REXX limits name length to 250 characters. Processor 10, however, only enforces the APL limit of 255 characters. Use of a name that is between 250 and 255 characters in length results in *SYSTEM ERROR* with ⎕*ET* set to `1 2`.

Character Set: Because names (and surrogate names) used with ⎕*NA* must also be valid ordinary APL names, you cannot access REXX variables whose names contain the characters '#.$@!&?'.

Name Class: Processor 10 uses the name class provided by you to decide how to reference the variable. Name class 1 (constant) is used to refer to REXX Private Information while name class 2 (variable) denotes other data. You must specify one of these classes if want to access data through EXECCOMM and IRXEXCOM. If Processor 10 is given name class 0 with a name, it assumes that you want a function and return a 3 as the name class to APL.

```
      1 10 ⎕NA 'VERSION'
1
      VERSION
REXX370 3.46 31 May 1988

      2 10 ⎕NA 'FRED'
1
      FRED
VALUE ERROR+
      FRED
      ^
      FRED←'ABC'
      FRED
ABC
```

Stemmed Variables: REXX stemmed variables are accessed by coding '∆' instead of '.' in the name. For example, `2 10 ⎕NA 'FRED∆2'` accesses the REXX variable FRED.2 in the current pool.

Uninitialized Variables: Processor 10 permits you to have an association with an uninitialized variable, but causes a *VALUE ERROR* if you attempt to reference it before setting a value.

Values

REXX deals exclusively in character strings. An attempt to assign anything other than a character vector or scalar to a REXX variable results in *SYSTEM LIMIT* with ⎕*ET* set to `1 12` to indicate an unrepresentable value.

If an uninitialized REXX variable is referenced, then Processor 10 signals *VALUE ERROR* with ⎕*ET* set to `3 1`.

When APL ends the association with the name, it remains defined to REXX. This is important for applications that need to return results to a REXX program, but it means that there is no way to use Processor 10 to drop a REXX variable.

Built-in Functions

Processor 10 provides a limited set of built-in functions to aid the manipulation of REXX files by applications. Functions are provided to read and write CMS files or MVS sequential data sets, query file or data set status, and execute REXX programs that are contained in APL2 arrays. All built-in functions begin with the character '∆'.

Reading and Writing CMS Files

The built-in functions 'ΔFM' and 'ΔFV' read and write CMS files directly to and from APL arrays.

- Reading Files

  ΔFM ω Reads a file 'ω' and returns a matrix. For files with variable length records, records are padded on the right with blanks to the length of the longest record.

  ΔFV ω Reads a file 'ω' and returns a vector of vectors. Trailing blanks in any record are deleted.

- Writing Files

  α ΔFM ω Writes the data from the matrix or vector of vectors 'α' to the file 'ω'. A file with fixed-length records is created by padding the records with blanks as necessary.

  α ΔFV ω Writes the data from the matrix or vector of vectors 'α' to the file 'ω'. A file with variable length records is created. Trailing blanks in each record are deleted.

  When a file is written, if a file of the name 'ω' already exists the original file is erased and a new file is created. Thus ΔFM always writes a file with fixed length records, and ΔFV always writes a file with variable length records.

In the above, the argument 'ω' is a character string containing the file name, file type, and file mode separated by blanks. File type and file mode default to '*' for reading. File mode defaults to 'A' for writing.

ΔFM and ΔFV return a character matrix or vector of vectors if a file is successfully read, or a numeric return code if a file is written or if the operation results in a CMS file system error. A complete description of the CMS file system return codes can be found in the description of the file system macros in the VM manuals. The most common return codes are:

0 Successful operation

12 Attempt to write on read-only disk

13 Disk is full

17 Record length exceeds 65,535 characters

20 Invalid character in fileid

28 File not found

36 Disk not accessed

In the following example, a function is created to return CMS disk statistics.

```
      X←'TRACE ''O'' ' 'Q DISK ARG(1) ''(LIFO''' 'PULL R' 'PULL' 'RETURN R'
      3 10 ⎕NA 'ΔFV'
1
      X ΔFV 'DISKQ EXEC A'
0
      3 10 ⎕NA 'DISKQ'
1
      DISKQ 'A'
JAG191 191 A   R/W  10 3380  1024     177     3528-76     1121    465
      ⍝
      ⍝ MODIFY FUNCTION TO REMOVE DUPLICATE BLANKS
      ⍝
      ⊃ΔFV 'DISKQ'
TRACE 'O'
Q DISK ARG(1) '(LIFO'
PULL R
PULL
RETURN R

      Y←ΔFV 'DISKQ'
      ¯1↑Y
 RETURN R
      (¯1↑Y)←⊂'RETURN SPACE(R)'
      Y ΔFV 'DISKQ EXEC A'
0
      DISKQ 'A'
JAG191 191 A R/W 10 3380 1024 177 3530-76 1118 4650
```

Reading and Writing MVS Sequential Data Sets

The built-in functions '*ΔFM*' and '*ΔFV*' read and write MVS sequential data sets (or members of a partitioned data sets) to and from APL arrays.

- Reading Data Sets

  ΔFM ω Reads a sequential data set '*ω*' and returns a matrix. For data sets with variable length records, records are padded on the right with blanks to the length of the longest record.

  ΔFV ω Reads a sequential data set '*ω*' and returns a vector of vectors. Trailing blanks in any record are deleted.

- Writing Files

  α ΔFM ω Writes the data from the matrix or vector of vectors '*α*' to the sequential data set '*ω*'. If the data set does not already exist, a sequential data set with fixed-length records is created.

  α ΔFV ω Writes the data from the matrix or vector of vectors '*α*' to the sequential data set '*ω*'. If the data set does not already exist, a sequential data set with variable length records is created.

  When writing to a data set that already exists, the data set is not reallocated. The record format and other data set attributes are not changed.

  When writing to fixed length data sets, records are padded with blanks as necessary.

  When writing to variable length data sets, trailing blanks are dropped before records are written.

In the above, the argument 'ω' is a character string containing the data set name. The character string should include beginning and ending quotes when the name is fully qualified. The profile prefix is added if the name does not start with a quote character. A member name must be included in parentheses if a PDS is to be accessed.

ΔFM and ΔFV return a character matrix or vector of vectors if a data set is successfully read, or a numeric return code if a data set is written or if the operation results in an error.

The possible return codes from ΔFM and ΔFV under TSO/E are:

0 Successful operation

12 Data set not available

13 No space available

16 Data set cannot be processed through REXX

17 Lrecl exceeds 32760 or is 0

20 Invalid data set name

28 Data set not found

New Data Set Allocation Considerations Under TSO

ΔFM If a data set is to be written that does not already exist, one is allocated on a default DASD device using record format FB (other record formats are supported for existing data sets.)

The LRECL is set to the number of columns in the matrix being written.

The BLKSIZE is set to the LRECL if the LRECL is larger than 7250. If the LRECL is less than 7250, the BLKSIZE is set to the largest integer multiple of the LRECL that is less than or equal to 7250.

If the data set is a simple sequential (not partitioned) data set, the primary SPACE is set to the number of bytes in the matrix, plus 25%. If the data set is a partitioned data set, the primary SPACE is set to 16 times that size.

The secondary SPACE is set to half the primary SPACE.

If the data set is a partitioned data set, the DIR is set to 60.

ΔFV If a data set is to be written that does not already exist, one is allocated on a default DASD device using record format VB (other record formats are supported for existing data sets.)

The LRECL is set to 4 plus the length of the longest vector being written if that number is greater than 259; otherwise the LRECL is set to 259.

The BLKSIZE is set to 4 plus the LRECL if that number is larger than 7250; otherwise the BLKSIZE is set to 7250.

If the data set is a simple sequential (not partitioned) data set, the primary SPACE is set to the number of bytes in the vectors, plus 25%. If the data set is a partitioned data set, the primary SPACE is set to 16 times that size.

The secondary SPACE is set to half the primary SPACE.

If the data set is a partitioned data set, the DIR is set to 60.

Querying a CMS File Status

The function Δ*F* *ω* returns file status information obtained from CMS FSSTATE macro (in the FST and ADT data areas). Its argument '*ω*' is a character string containing the file name, file type, and file mode separated by blanks. File type and file mode default to '*'. The result of Δ*F* is a nine-element nested vector with the following contents:

1 - File Identification.
20 element character vector of file name, file type, and file mode

2 - Record Format.
Character scalar from FSTRECFM ('F' or 'V')

3 - Record Length.
Integer scalar from FSTLRECL

4 - Number of Records.
Integer scalar from FSTAIC

5 - Number of Data Blocks.
Integer scalar from FSTADBC

6 - When Last Written.
Seven element APL time stamp from FSTATADI

7 - Disk Label.
Six element character vector from ADTID

8 - Disk Mode (and parent).
Two element character vector:

- A 'W' if writable disk, an 'R' if read-only, or an 'E' if a read-only extension.

and

- A ' ' if writable disk, otherwise the mode letter of the parent disk.

9 - Disk Block Size.
Integer scalar from ADTDBSIZ

```
      3 10 ⎕NA 'ΔF'
1
      9 1⍴ΔF 'LIBTAB APL2'
 LIBTAB   APL2      D1
                     V
                    61
                    46
                     2
 1984 1 26 14 37 29 0
               CMS192
                    W
                  1024
```

If a CMS file system error occurs during the execution of Δ*F*, a numeric return code is returned instead of the character result. See “Reading and Writing CMS Files” on page 281 for additional information.

Querying an MVS Data Set Status

The function ΔF ω returns status information about an MVS sequential data set, or a member of a PDS. Its argument 'ω' is a character string containing the data set name, constructed following the same rules as a data set name for ΔFM or ΔFV. The result of ΔF is a nine-element nested vector with the following contents:

1 - File Identification
56 element character vector containing the data set name

2 - Record Format.
2 element character vector: first element 'F' or 'V', second element ' ' or 'B'

3 - Record Length.
Integer scalar

4 - Number of Records.
Integer scalar

5 - Number of Data Blocks (unknown).
one element character vector: '?'

6 - When Last Written (unknown).
one element character vector: '?'

7 - Disk Label (unknown).
one element character vector: '?'

8 - Disk Mode (not applicable).
one element character vector: '?'

9 - Block Size.
Integer scalar

If an error occurs during the execution of ΔF, a numeric return code is returned instead of the character result. See “Reading and Writing MVS Sequential Data Sets” on page 282 for additional information.

Executing APL Arrays as REXX Programs

α ΔEXEC ω Executes the REXX program contained in 'α' with the items of 'ω' as the argument strings. 'α' is a character vector, matrix, or vector of vectors containing the REXX program. 'ω' is a vector of 1 to 10 character vectors that are the strings passed as arguments to the REXX program.

The right argument of ΔEXEC does not include the 'name' of the EXEC. In effect, the temporary program contained in the left argument array is invoked in the same way as described in “Using REXX Functions” on page 275.

The 'ΔF' and 'ΔEXEC' built-in functions provide the basis to build tools for maintaining and testing REXX EXECs.

Note: You can use APL operators like EACH (¨) with ΔEXEC to test a REXX function with many combinations of arguments or use operators with ΔFM to compare the results of different EXECs on the same arguments.

The following APL function demonstrates the use of ΔEXEC in an application that issues a CMS command but suppresses any output.

```
      ∇QUIET_CMD[□]∇
    ∇
[0]   Z←QUIET_CMD CMD;ΔEXEC;X
[1]   □ES(1≠3 10 □NA 'ΔEXEC')/'ΔEXEC UNAVAILABLE'
[2]   ⍝
[3]   ⍝   SEE VM REXX MANUALS FOR INFO ON HT AND RT
[4]   ⍝
[5]   ⍝   ISSUE COMMAND BUT SUPPRESS OUTPUT
[6]   ⍝
[7]   X←⊂'TRACE ''O'''
[8]   X←X,⊂'TF=CMSFLAG(CMSTYPE)'
[9]   X←X,⊂'IF TF THEN ''SET CMSTYPE HT'' '
[10]  X←X,⊂''''''ARG(1)'
[11]  X←X,⊂'R=RC'
[12]  X←X,⊂'IF TF THEN ''SET CMSTYPE RT'' '
[13]  X←X,⊂'RETURN R'
[14]  Z←⍎X ΔEXEC CMD
    ∇ 1984-05-30  8.51.14 (GMT-4)

      QUIET_CMD 'STATEW * * A'
0
      QUIET_CMD 'STATEW * * B'
36
      QUIET_CMD 'STATEW * * S'
28
```

Unexpected Errors and Other Considerations

This section discusses unexpected errors and other programming considerations.

Failure when Associating a Name

You cannot do a `2 10 □NA` association with an already existing variable (even if it is a character string). The following example fails:

```
      FILEID←'TEST SCRIPT A'
      2 10 □NA 'FILEID'
0
```

but this example works:

```
      2 10 □NA 'FILEID'
1
      FILEID←'TEST SCRIPT A'
```

For variables, there is an additional requirement that the EXECCOMM subcommand or IRXEXCOM be declared by REXX as described in “Associating Names” on page 279.

Under CMS, be careful to ensure that EXECCOMM was established by REXX and not by EXEC2. Processor 10 operates only with REXX.

APL Errors

- *DOMAIN ERROR*, *RANK ERROR*, *LENGTH ERROR*

  These errors generally indicate a problem with the argument. REXX communicates exclusively with character strings. The arguments to any function associated with Processor 10 must be APL character vectors or vectors of vectors. (See "Constructing the Argument" on page 277 and "Handling Results and Errors" on page 278).

- *VALENCE ERROR*

  This error is generated when APL invokes a function and either:

  1. The association is not active

     An association is inactive whenever 0 is the result of applying dyadic ⎕*NA* to the name. It means that Processor 10 cannot accept the name. This could happen after loading a saved workspace and invoking a function associated with Processor 10 for the first time.

  or

  2. The wrong number of APL arguments was given.

     Processor 10 defines all REXX functions as monadic APL functions so APL would cause this error if the invocation is dyadic. A common cause of this when you compose a line in REXX-style using blanks to concatenate strings. APL employs the comma for catenation.

Non-APL Error Messages

REXX, CMS, CP, or TSO may issue messages while executing your request. Without APL2's session manager, such messages are intermixed with your APL output. With APL2's session manager, these messages cause a 3270 terminal to leave APL2's session manager mode to display the message. In addition, this message does not become part of APL2's session manager Log.

The following is an example of a common REXX error as it would appear with session manager off:

```
      3 10 ⎕NA 'INDEX'
1
      INDEX 'FRED' 'R'
2
      INDEX 'FRED'
20040
```

REXX Return Code 20040

This can be a cryptic REXX return code. It normally indicates that the function has been invoked with the wrong number of arguments. If that does not seem to be the case, you may want to note the following three situations.

1. Simple scalars

   A list of APL simple scalars is a simple vector. If all the items forming the argument to a REXX function happen to be simple scalars, then the argument is not a vector of vectors, but a simple vector. This can be remedied by using ,⍤1 that always creates a vector of vectors from a simple vector.

2. Commas and Blanks

   It is easy to confuse the use of commas and blanks when trying to simultaneously program in REXX and APL. See ""Missing" Argument Strings."

3. Old CMS commands

   Most CMS commands were not designed to be invoked as REXX functions. If by accident you invoke a command that is not designed to be a REXX function, it is normally unable to successfully analyze its operands and returns an error as well as print a message on the terminal. REXX interprets the error code to mean the program was invoked improperly and itself returns a numeric 20040 (Incorrect Call to Routine).

```
      3 10 ⎕NA 'STATE'
1
      STATE 'OF UNION'
20040
```

   You can, however, use the built-in function '∆EXEC' to execute such a command.

```
      3 10 ⎕NA '∆EXEC'
1
      'STATE OF UNION' 'RETURN RC' ∆EXEC ⍳0
28
```

   Note: The '28' is two characters and not a number. Had the clause 'RETURN RC' been omitted, the result would have been numeric 0, indicating that REXX had successfully completed execution.

"Missing" Argument Strings

Comma and blank mean decidedly different things to REXX than they do to APL. It is reasonably accurate to say that REXX's use of a comma is like APL's use of a blank and vice versa. The two facilities are catenation and string separation. APL uses a comma for catenation and a blank for string separation. REXX uses a blank for catenation and a comma to separate strings when invoking functions.

```
      ⍝ 3 COMMAS IN REXX REQUIRES 3 STRINGS FROM APL
      ⍝

      'PARSE ARG X , Y , Z' 'RETURN Y' ⎕EXEC 'AB' 'CD' 'EF'
 CD

      ⍝ REMOVING COMMAS IMPLIES 1 STRING TO REXX
      ⍝    WITHOUT THEM 'CD' AND 'EF' SEEM INACCESSIBLE
      ⍝

      'PARSE ARG X   Y   Z' 'RETURN Y' ⎕EXEC 'AB' 'CD' 'EF'

      'PARSE ARG X   Y   Z' 'RETURN X' ⎕EXEC 'AB' 'CD' 'EF'
 AB

      ⍝
      ⍝ IF YOU REMOVE COMMAS FROM REXX,
      ⍝    USE A SIMPLE APL VECTOR

      'PARSE ARG X   Y   Z' 'RETURN Y' ⎕EXEC 'AB   CD   EF'
 CD

      ⍝ THE OTHER MISTAKE IS JUST AS EASY TO MAKE
      ⍝ 3 COMMAS IN REXX REQUIRES 3 STRINGS FROM APL
      ⍝   SO 1 STRING FROM APL BECOMES 'X'

      'PARSE ARG X , Y , Z' 'RETURN X' ⎕EXEC 'AB   CD   EF'
AB    CD    EF
```

Truncated Data Returned under TSO/E

APL2 under TSO/E allocates a buffer of 2048 bytes for data returned from REXX functions. If REXX is already active when processor 10 is used, and a function generates more bytes of data than fit in the buffer, then REXX for TSO/E does not allow APL2 to dynamically allocate a larger buffer for the data. If this occurs, Processor 10 produces a *SYSTEM ERROR* and no result is returned.

Your installation may change the size of the buffer APL2 allocates to something other than 2048 bytes by using the P10EVSZ parameter of the AP2TITOP macro in the AP2TIOPT or AP2TIOPX options modules.

Other Considerations

Processor 10 uses standard REXX interfaces from an APL environment. REXX programs that make assumptions about the environment in which they run may behave differently.

Environment

REXX programs that alter the environment when invoked and do not restore it when finished may cause unpredictable failures in APL. In particular, the following areas must be carefully considered when using REXX applications.

- Error Recovery

  REXX provides no error recovery for program checks and ends abnormally when one occurs. If that happens, APL2 error exits gain control. System resources, like storage acquired by the REXX program, cannot be recovered by APL2. In most cases, the APL2 session can continue unaffected, but there is no way APL can guarantee the integrity of the CMS or TSO/E system.

- Terminal Handling

  REXX programs may read from and write to the terminal, but they should not disturb the terminal handling environment. If a REXX program needs to control attentions and immediate commands, you should use it through the Host System Command Processor (AP 100) rather than through Processor 10.

  Programs that operate in full-screen mode may need to be modified to refresh the screen when invoked from APL with the Session Manager.

- Compiled REXX

  REXX Execs can be compiled into either EXECs or MODULEs on CMS and used with no change.

  REXX Execs can be compiled into either EXECs or MODULEs on TSO and used with no change also. However, on TSO, the object file produced by the compiler must be link-edited with a stub module. Four stub modules are provided with the REXX/370 compiler. Different stubs are used depending upon what sort of environment is going to be used to call the routine. Only two stubs can be used if the routine is going to be called from APL2 through P10.

  The names and uses of the stub modules are:

Name	Use	Useable with P10
EAGSTCAL	CALL Command	Yes
EAGSTCPP	CPPL	No
EAGSTEFP	EFPL	Yes
EAGSTMVS	MVS	No

Figure 89. REXX Interface Stubs

For further information about compiling REXX programs, consult the REXX compiler manuals.

Chapter 28. Processor 11—Calling Compiled Programs

Processor 11 provides facilities that allow access to objects outside the active workspace that are either objects in APL2 namespaces or are routines written in languages other than APL2. Non-APL routines may be functions or operators and may be any valence just like APL2 defined functions and operators. Once access to a routine is established through Processor 11, the routine is treated like a locked APL function or operator.

This chapter describes access to routines written in languages other than APL2. The beginning of the chapter describes how to access non-APL routines from APL2, and gives an overview of the services provided by Processor 11. The middle portion of the chapter explains how the different facilities are used to design and prepare non-APL routines for use from APL2. The rest of the chapter gives detailed descriptions of the interfaces available to non-APL routines, and is intended for readers who are writing such routines. Access to APL objects in namespaces is described in Chapter 29, “Processor 11—Access to Namespaces” on page 334.

There are several reasons why it is desirable to be able to call non-APL programs from APL2:

- Access operating system services that are unavailable from APL2
- Use of preferred language
- Reuse of existing non-APL program libraries
- APL2 provides an interactive environment for calling programs.
- Performance bottlenecks in APL applications can be recoded in compiled language for improved overall application performance.

No changes to most existing C/370, PL/I, Fortran, or Assembler Language routines are required to access them from the APL2 environment under either VM/CMS or MVS/TSO. APL2 and Processor 11 manage the necessary housekeeping and argument and result conversion based upon descriptive information provided for each routine by the user.

Routines can be written that receive arguments and return results using standard operating system, Fortran, or some C/370 conventions. Routines can also be written to use APL2 conventions. The APL2 conventions provide additional argument information such as data type, shape, and length. All non-APL routines are given access to services that can be used to access the APL2 workspace and operating system services.

APL2 also includes a programming interface called APL2PI that can be used to call APL2 from programs written in other high-level languages. APL2PI provides the ability to start APL2, make calls to execute expressions, retrieve results, switch namescopes, and terminate APL2. APL2PI is discussed in *APL2/370 Programming: Processor Interface Reference*.

⎕NA Syntax for Non-APL Programs

The APL2 system function ⎕NA is used to establish an association between a workspace name and an external routine that Processor 11 manages. The syntax of the ⎕NA function can take four forms.

The first two forms are used only with self-describing non-APL routines and objects in APL2 namespaces.

- A load library name and a member name may be specified in the left argument of ⎕NA.

  ```
  'lib.memb' 11 ⎕NA 'qnaname'
  ```

  This directs Processor 11 to load member *memb* from *lib* LOADLIB * on CMS or from the load library currently allocated to ddname *lib* on TSO.

- Just a member name may be specified in the left argument of ⎕NA.

  ```
  'memb' 11 ⎕NA 'qnaname'
  ```

  This directs Processor 11 to load member *memb* using the default search order. Consult “Processor 11 Routine Search Order Guidelines” on page 325 for information about Processor 11's search order rules.

The second two forms always cause Processor 11 to read a NAMES file. They may be used with self-describing routines.

- The CMS filename or TSO ddname of a private NAMES file may be specified between parentheses in the left argument of ⎕NA.

  ```
  '(private)' 11 ⎕NA 'qnaname'
  ```

  This directs Processor 11 to search *private* NAMES * on CMS or the NAMES file library allocated to ddname *private* on TSO for a names file entry for the external routine *qnaname*.

- The name class of the object may be specified in the left argument of ⎕NA.

  ```
  3 11 ⎕NA 'qnaname'
  ```

  This directs Processor 11 to search the default NAMES files for a names file entry for the external routine *qnaname*.

Surrogate names may be used to avoid name conflicts in the current namescope. For example:

```
3 11 ⎕NA 'OBJNAME EXTNAME'
```

could be coded to associate the name *OBJNAME* with the external routine *EXTNAME*.

Processor 11 Overview

Processor 11 manages the interface between the APL2 active workspace and non-APL routines outside the workspace. Processor 11 has the responsibility to load the non-APL routine, translate arguments from the APL2 workspace to a form the non-APL routine can work with, call the non-APL routine, and translate the routine's results to a form the APL2 interpreter can work with.

Conceptually, non-APL routines are merely called by Processor 11. Every time the user executes an expression that mentions the non-APL program, Processor 11 is

called by the APL2 interpreter and it in turn calls the routine. Processor 11 requires several types of information to perform this task.

Processor 11 needs to know about the arguments and results for each routine it calls. Since Processor 11 translates arguments from APL2 workspace format to the non-APL routine's format, it needs to know what format the routine is expecting. Likewise, when the routine returns results, Processor 11 needs to understand their format to be able to translate them to a workspace format.

Before Processor 11 can call a routine, it must locate the routine. Routines typically reside in modules and modules can reside in load libraries. Modules can contain multiple routines and libraries can contain multiple modules. For each routine Processor 11 calls, it needs to be informed what library and module contain the routine and where the routine is located within the module.

Processor 11 also needs to know about any special environments required by the routines it calls. Some language compilers produce programs that start and depend on environments. Processor 11 can be instructed to run a program to start an environment on behalf of a routine and leave it running until it is no longer needed.

Finally, Processor 11 can call programs using a variety of linkage conventions. These conventions provide different formats for arguments and results and provide different levels of access to the APL2 interpreter. Processor 11 needs to know what convention to use when calling a routine.

This chapter describes each of these topics in detail:

- Associating APL workspace names with external routines
- Argument and result descriptions
- Routine location
- Environments
- Linkage conventions

The information is introduced using a series of examples that demonstrate the uses of each of Processor 11's facilities.

Introduction

Processor 11 provides a wide variety of facilities for preparing and using non-APL routines. These facilities and the routines they support can be quite complicated. To facilitate accurate descriptions of the facilities, here is a short glossary of terms that are used in APL2 publications for referring to Processor 11 concepts.

Processor 11 Glossary

Non-APL Language Concepts

program. A set of computer instructions. Programs start any environment that they may require. The requirement of an environment is usually predicated by the language in which the program is written.

subroutine. A set of computer instructions that requires a environment to be active. An appropriate program must have been run prior to use of a subroutine.

routine. A generic term used to refer to both subroutines and programs.

module or member. A member of a TSO partitioned data set or CMS loadlib file, or a CMS module file. A module can contain one or more routines. The term member is used interchangeably with module in many cases.

environment. An environment established by a program that persists across multiple calls to one or more subroutines. Environments usually provide services to subroutines such as virtual storage allocation, file access, and error recovery.

environment program. A program that initializes anr environment.

Processor 11 Concepts

routine description. Information that informs Processor 11 how a routine should be found and used. Each routine must have a corresponding routine description.

NAMES file. A human readable file containing one or more routine descriptions.

self-describing. An adjective applied to routines that contain their own routine descriptions.

routine list. An object file used for informing Processor 11 of the names and locations of the routines within a module.

linkage convention. The convention specified for a routine that Processor 11 uses to call the routine. Depending on the linkage convention, processor 11 passes arguments, receive results, and provides access to APL2 services using different techniques. Three conventions are provided: OBJECT, FORTRAN, and FUNCTION. They are used for programs using standard, Fortran, and APL2 entry and exit linkage conventions respectively.

interface management routines. Routines that assist Processor 11 in managing the interface between APL2 and non-APL routines.

APL2 Tools

BUILDRD. An APL2 function that builds a routine description in object file form that is suitable for link-editing with a routine to make the routine self-describing.

BUILDRL. An APL2 function that builds a routine list in object file form that is suitable for link-editing several routines together into a single module.

AP2XCMAP. An interface management routine supplied with APL2 to help call C/370 programs.

AP2TNL and AP2VNL. Programs supplied with APL2 that are used by environment programs to pass control back to APL2 without terminating. AP2TNL is used on TSO; AP2VNL is used on VM.

AP2MP11L and AP2MP11M. EXECs that can be used to link-edit one or more routines, their corresponding descriptions, and a routine list into a useable member of a load library or a CMS module.

Many of these terms refer to information or programs that you must supply to use Processor 11. However, defaults are provided for many of them. Depending on

your external routine, you may need to concern yourself with only a few of these definitions.

Usage Overview

Routine Descriptions

Our first example of a non-APL routine demonstrates Routine Descriptions. Each non-APL routine to be called using Processor 11 must have a routine description. Processor 11 requires specific pieces of information be supplied for each routine. For each of these pieces of information, a different tag is used in the routine description to supply the data. Tags have a form that is familiar to Script users. For example:

```
:DESC.Sample Tag
```

In this example, the :DESC. portion is the tag name. Each tag starts with a colon and ends with a period. The tag name can written in upper-, lower-, or mixed-case format. The *Sample Tag* portion is the data supplied with the tag. The :DESC. tag is used for routine description comments. As with most of the tags, the :DESC. tag is optional. If an optional tag is omitted, Processor 11 provides default values based on other information.

Here is an Assembler routine to add three numbers.

```
* Routine to add three integers
ASMADD   CSECT
         STM   14,12,12(13)     Save registers
         L     2,0(,1)          Retrieve address of first argument
         L     3,0(,2)          Load first argument
         L     4,4(,1)          Retrieve address of second argument
         L     4,0(,4)          Load second argument
         L     5,8(,1)          Retrieve address of third argument
         L     5,0(,5)          Load second argument
         AR    3,4              Add first two arguments
         AR    3,5              Add third argument to result
         ST    3,0(,2)          Replace first argument with result
         LM    14,12,12(13)     Restore registers
         BR    14               And return to caller
         END
```

Figure 90. Assembler routine to add 3 integers

Note: Don't worry if you aren't familiar with assembler language. This example merely demonstrates building a routine description. There are more examples in other languages and there are complete examples in C/370, PL/I, and VS FORTRAN at the back of the book in Appendix G, “Sample Non-APL Programs to be Called through Processor 11” on page 385.

This routine takes three integer arguments, adds them, and updates the storage used to pass the first argument with the result. Consider what would be needed to use this routine with processor 11. We would need to tell Processor 11 what argu-

ments and results the routine expects. Here is what the routine description would look like:

```
:LINK.OBJECT :RARG.(G0 1 3)(1 <I4 *)(1 I4 *)(1 I4 *)
```

The :LINK. tag specifies that the routine should be called using standard OS linkage conventions. This topic is discussed in more detail later.

External functions accessed through Processor 11 are viewed from from APL as taking zero, one, or two arguments, like all APL functions. Our Assembler routine expects three arguments and in fact, most non-APL routines are designed to accept argument lists that can contain many arguments. This apparent incompatibility is resolved by calling normal non-APL routines as if they were APL functions expecting only a simple or nested right argument. Processor 11 creates a parameter list acceptable to the external routine from each of the items of the APL argument.

Programs written specifically to be called by APL2 can also use the :LARG. tag to describe the left argument. The :RSLT. tag is used to describe a non-APL routine's results.

Argument tags are used to specify the general structure and data types required of the arguments. The sample :RARG. tag's value states that the routine expects an argument containing a rank 1 length 3 array (`G0 1 3`), each of whose elements contains 1 integer, (`1 I4 *`). The array is explicitly described as a list of three separate items so that the first item can be identified with the less-than sign, <.

There are no facilities in languages like Fortran or Assembler to differentiate between argument and result data. Routines in those languages often update argument data passed to them. APL functions, on the other hand, take arguments that are never updated and produce explicit results that are not arguments to the function. Processor 11 provides facilities to resolve this apparent conflict. Specified argument items can be updated or made the explicit result of the external function. More detailed information on this topic can be found in section “Updated Arguments and Results” on page 313. Our Assembler function returns its result by updating the storage that contained the first argument. This fact is indicated by the < in the tag value.

There are two ways to provide the routine description to processor 11. The first is to place the routine description in the module containing the non-APL routine. This is done by using the external function *BUILDRD* to build an object file containing the description and link-editing this object file with the routine. This makes the routine self- describing. The second way to provide the routine description is to place it in a separate file that Processor 11 has access to. This file is called a NAMES file.

Building Routine Descriptions

If the *BUILDRD* technique is used, the resulting module can be placed in the normal search order and used directly with processor 11. Here is how we would use *BUILDRD* to build an object file.

```
      3 11 ⎕NA 'BUILDRD'
1
      RD←':LINK.OBJECT '
      RD←RD,':RARG.(G0 1 3)(1 <I4 *)(1 I4 *)(1 I4 *)'
      'ADDRD TEXT' BUILDRD 'ADDRD' 'ASMADD' RD
0
```

Figure 91. Build routine description for ASMADD function. On TSO, BUILDRD's left argument would be a data set name.

ADDRD is the name we want assigned to the routine description; it must be different than the routine name. In addition, this is the name we specify in the right argument to ⎕*NA* when we associate a name with the routine. Processor 11 uses it when searching for the routine. *ASMADD* is the name of the routine. *RD* is simply our routine description.

Once we have assembled our routine and built our routine description, we must link-edit them together into a module. Link-edit techniques differ between operating systems, languages, and even sites. Discussions and examples of several link-editing techniques for different languages are provided in “Link-Editing External Routines on CMS” on page 329, “Link-Editing External Routines on TSO” on page 330, and Appendix G, “Sample Non-APL Programs to be Called through Processor 11” on page 385.

Once we have link-edited the object files produced by assembling our routines and the object files produced by *BUILDRD*, we can discard the object files. They were only required as intermediate steps in the process.

A sample of the CSECT that *BUILDRD* produces can be found in *APL2/370 Programming: Processor Interface Reference*.

Building NAMES Files

If the names file technique is used on CMS we add a :NICK. tag to our description. The :NICK. tag provides the association between the last name specified in the right argument to ⎕*NA* and our routine description in the NAMES file entry.

On CMS, many routine descriptions can be combined in a single NAMES file. In order to separate and distinguish the descriptions, each description is preceded with a :NICK. tag. When Processor 11 is looking for a routine description in a CMS NAMES file, it searches for a :NICK. tag with the same name specified in the right argument to ⎕*NA*. The routine description immediately following the tag is then used.

If the names file technique is used on TSO the routine description is placed in a member of a partitioned data set. When Processor 11 is looking for a routine description on TSO, it looks for a NAMES file member with the same name as the last word specified in the right argument to ⎕*NA*. The contents of the member are then used as the routine's description. The :NICK. tag is redundant in TSO names files and so is not required, although it is allowed.

When using NAMES files, the NAMES file must inform Processor 11 how to locate the member containing the routine. This is done using the :MEMB., and optionally

the :LOAD., tags. The :MEMB. tag is used to identify which module, or member of a load library, contains a routine. The :LOAD. tag is used to identify which load library contains the member that contains the routine.

With the addition of the :NICK., :LOAD., and :MEMB., tags, our routine description looks like this:

```
:NICK.ADD
:LOAD.ASMLOAD
:MEMB.ASMADD
:LINK.OBJECT
:RARG.(G0 1 3)(1 <I4 *)(1 I4 *)(1 I4 *)
```

Self-Describing Routines and NAMES File: Pros and Cons

On both CMS and TSO, there is a benefit to using NAMES files over self-describing routines. Because Processor 11 searches for a nickname (or member on TSO) with the same name as the last word in the argument of $\square NA$, the nickname does not have to be the same as our routine name. Therefore, we can refer to our routine ASMADD by another name, for example ADD. Notice that the :NICK. tag in our NAMES file entry specifies that we are referring to the routine using the name ADD.

The :NICK. tag can be used either in NAMES files or in self-describing routines. However, if a NAMES file is not used, the :NICK. tag is not used to locate the routine description. If it in the routine's self-description it must match the name specified in $\square NA$'s right argument.

NAMES files can point to self-describing routines. In this case, the NAMES file is used only to specify where to find the routine; the routine's argument descriptions and calling convention information are located with the routine. The NAMES file entry may contain only :NICK., :LOAD., :MEMB., :ENTRY., and :DESC. tags. Any other descriptive tags must be in the routine description link-edited with the routine.

Using routine descriptions allows you to not have to manage NAMES files. Self-describing routines can generally be installed more easily since NAMES files do not have to be provided. Also, slightly better performance during name association is possible since the overhead of reading NAMES files is not incurred.

Routine Lists

The next topic concerns building routine lists that are used when combining several routines into a module. This is desirable if they are used together or if they are interdependent and call one another. This is frequently done with groups of routines that are part of the same application or that are written in the same language. In fact, we shall see that it is required for routines written in C/370 and PL/I.

Processor 11 can only locate, by itself, one routine within a module. Any other routines that may reside within the module are not accessible to Processor 11 because it cannot locate them. This apparent dilemma is solved by the introduction of a routine list.

A routine list is a table of all the routines within a module. It is used by Processor 11 for locating routines within the module. When a group of routines' object files are link-edited together with a routine list, the name of the routine list is specified as the main routine within the module. This is done so that Processor 11 can find it. Once Processor 11 has found the routine list, the list can be used to locate the other routines within the module.

We have seen that the :MEMB. tag provides the name of the module. If several routines are combined into a single module, the :ENTRY. tag provides the name of the routine within the module. If the :ENTRY. tag is not provided, Processor 11 uses the name specified in □*NA*'s right argument as a default. If the :ENTRY. tag is provided, Processor 11 searches the routine list for the routine named in the :ENTRY. tag.

The *BUILDRL* function is provided to build routine lists. Assuming we had two routines we wanted to combine into a module, we could use *BUILDRL* like this:

```
      3 11 □NA 'BUILDRL'
1
      'ASMRL TEXT' BUILDRL 'ASMRL' 'ASMADD' 'ASMSUB'
0
```

Figure 92. Build routine list for two functions

ASMRL is the name we want assigned to the routine list. Like with *BUILDRD*, the name must be different from any of the names specified in the rest of the argument. *ASMADD* and *ASMSUB* are the names of the routines that are link-edited with the routine list.

A sample of the CSECT that *BUILDRL* produces can be found in *APL2/370 Programming: Processor Interface Reference*.

Once we have built the routine list and link-edited it with our routines, you can then build this NAMES file containing the routine descriptions:

```
:NICK.ADD
:MEMB.ASMRL
:ENTRY.ASMADD
:LINK.OBJECT
:RARG.(G0 1 3)(1 <I4 *)(1 I4 *)(1 I4 *)

:NICK.SUB
:MEMB.ASMRL
:ENTRY.ASMSUB
:LINK.OBJECT
:RARG.(G0 1 2)(1 <I4 *)(1 I4 *)
```

Figure 93. Descriptions for Assembler Routines

Notice that both routines have :MEMB. tags pointing to the same member and that its name is the same as the routine list's name. Each of the routine descriptions contains an :ENTRY. tag whose value is the name of the actual routine.

If we want to include the routine descriptions in the module rather than a names file, our use of *BUILDRL* is a little more complicated. In the example below we have used *BUILDRD* differently to demonstrate the difference in *BUILDRL*.

```
      RD←':LINK.OBJECT '
      RD←RD,':RARG.(G0 1 3)(1 <I4 *)(1 I4 *)(1 I4 *)'
      'ADDRD TEXT' BUILDRD 'ADDRD' 'ASMADD' RD
0
      RD←':LINK.OBJECT '
      RD←RD,':RARG.(G0 1 2)(1 <I4 *)(1 I4 *)'
      'SUBRD TEXT' BUILDRD 'SUBRD' 'ASMSUB' RD
0
      R1←'ADD ADDRD'
      R2←'SUB SUBRD'
      'ASMRL TEXT' BUILDRL 'ASMRL' R1 R2
0
```

Figure 94. Build routine list for two self-describing functions

ASMADD and *ASMSUB* are the names of the actual routines for which we build descriptions. Recall that when using self-describing routines, Processor 11 does not know the name of the actual routine; it only knows the name of the routine description. Hence, the names of the routine descriptions built using *BUILDRD* are specified in *BUILDRL*'s argument.

ASMRL is still the name we want assigned to the routine list. The rest of *BUILDRL*'s argument has changed though.

A routine list actually contains two lists of names, the list of names Processor 11 searches through to find the name mentioned in the right argument of ⎕*NA*, and the list of names of the non-APL routines (or their descriptions.) If you code only the name of the routine in a *BUILDRL* argument, like in Figure 92 on page 299, *BUILDRL* places the name in both lists. This works well for non-self-describing routines when the name of the routine is usually convenient to use. However, since routine description names must be different than the names of routines they describe, it would be convenient to be able to specify another name to be used in the right argument of ⎕*NA*

ADD and *SUB* are the names that Processor 11 searches through when trying to locate a routine in the routine list. These names are specified in the right argument of ⎕*NA*, or are the value of the :ENTRY. tag in NAMES files,

Since the names of our routines are not very convenient to use, we could have use convenient names for our routine descriptions and provided them directly to *BUILDRL*. Figure 95 on page 301 demonstrates this simpler technique of using the same name for the routine descriptions and ⎕*NA* arguments.

```
      RD←':LINK.OBJECT '
      RD←RD,':RARG.(G0 1 3)(1 <I4 *)(1 I4 *)(1 I4 *)'
      'ADD TEXT' BUILDRD 'ADD' 'ASMADD' RD
0
      RD←':LINK.OBJECT '
      RD←RD,':RARG.(G0 1 2)(1 <I4 *)(1 I4 *)'
      'SUB TEXT' BUILDRD 'SUB' 'ASMSUB' RD
0
      'ASMRL TEXT' BUILDRL 'ASMRL' 'ADD' 'SUB'
0
```

Figure 95. Build simplified routine list for two self-describing functions

A module can contain one or more routines. If the module contains more than one routine, the module must contain a routine list. (It may contain a routine list even if it contains only one routine.)

A module can be composed of both self-describing and non-self-describing routines. Processor 11 identifies that a routine is self-describing when it finds the routine. Note that routines can be self-describing but that modules are never self-describing. Modules are not executable; only routines within them are executable.

BUILDRL and Interface Management Routines

This section covers a still more complicated use of the *BUILDRL* function. Many first time readers may want to skip this section, but it is required for some C/370 and PL/I routines.

Conceptually at least we can now use most non-APL programs from APL2 with Processor 11. Or can we?

Suppose there is a program that expects its arguments to be passed in a particular style for which Processor 11 has no provisions. Further, suppose it returns its results in a form that processor 11 also has no provisions for. In such a case, you might want to write an intermediate program that could be called by Processor 11, manipulate the data passed by Processor 11 into a form acceptable to the routine and call the routine. It could perform a similar function with the routine's results. Processor 11 and *BUILDRL* support just this type of intermediate program. They are called Interface Management Routines. Here's how our calls to *BUILDRL* would look if we had an intermediate program called *INTMAN*.

```
      R1←'ADD INTMAN ASMADD'
      R2←'SUB INTMAN ASMSUB'
      'ASMRL TEXT' BUILDRL 'ASMRL' R1 R2
0
```

Figure 96. Build routine list for two self-describing functions

Once again, *ASMRL* is still the name we want assigned to the routine list. However, the rest of the argument has changed even more.

ADD and *SUB* are still the names that Processor 11 uses when searching the routine list.

INTMAN is now the name of the routine that Processor 11 calls. Processor 11 is calling the second name in each entry. In this case, this is the name of the interface manager program. It is the name of a routine that is included when the group of routines is link-edited together. *ASMADD* and *ASMSUB* are the names of the routines that the interface manager calls. For further details on the interfaces used by Processor 11 for interface manager programs, consult *APL2/370 Programming: Processor Interface Reference*.

BUILDRL Interface Management and Self-Describing Routines

There is still one more complication in the possible uses of *BUILDRL*. Suppose you wanted to make a routine requiring an interface manager self-describing. In this case, the calls to *BUILDRD* change again.

```
      RD←':LINK.OBJECT '
      RD←RD,':RARG.(G0 1 3)(1 <I4 *)(1 I4 *)(1 I4 *)'
      'ADDRD TEXT' BUILDRD 'ADDRD' 'INTMAN' RD
0
      RD←':LINK.OBJECT '
      RD←RD,':RARG.(G0 1 2)(1 <I4 *)(1 I4 *)'
      'SUBRD TEXT' BUILDRD 'SUBRD' 'INTMAN' RD
0
      R1←'ADD ADDRD ASMADD'
      R2←'SUB SUBRD ASMSUB'
      'ASMRL TEXT' BUILDRL 'ASMRL' R1 R2
0
```

Figure 97. Building routine lists. Builds a routine list object file for two self-describing functions called through Interface Manager routines.

Processor 11 recognizes that a routine is self-describing by examining the routine that it calls. Processor 11 calls the routine that is listed second in the name list entries passed to *BUILDRL*; that routine, which in this case is an interface manager, calls the routine that does work. Therefore, you make the interface manager self-describing.

In Figure 97 the uses of *BUILDRD* are creating routine descriptions for an interface manager routine. The argument of *BUILDRL* then specifies that Processor 11 is calling these self-describing instances of the interface manager and that it should in turn call the Assembler routines.

If you use languages like C/370 or PL/I, the Interface Managers can become quite important. Both C/370 and PL/I store arrays, pass arguments, and return results using their own non-standard conventions. You need an interface manager routine to call many C/370 and PL/I programs. “C/370 Scalar Integer Results” on page 303 provides further information on this topic.

C/370 Scalar Integer Results

When a :RSLT. tag is coded for a :LINK.OBJECT routine specifying that a scalar integer result is returned, Processor 11 expects that result is returned in register 0. C/370 routines return scalar integer results in register 15. To resolve this dilemma, APL2 includes an interface manager routine named AP2XCMAP.

A routine list is needed to use AP2XCMAP. For example:

```
      RD←':LINK.OBJECT '
      RD←RD,':RARG.(G0 1 2)(1 I4 *)(1 I4 *) '
      RD←RD,':RSLT.(I4 0) '
      RD←RD,':INIT.CENVRD'
      'CADDRD TEXT' BUILDRD 'CADDRD' 'AP2XCMAP' RD
0
      RL←'CRL' 'ADD CADDRD CADD' 'CENVRD'
      'CRL TEXT' BUILDRL RL
0
```

Figure 98. Using AP2XCMAP

In this case, we have a routine called CADD. CADD returns a scalar integer result. Because C/370 returns scalar integer results in a different register than Processor 11 expects, we use the interface manager program AP2XCMAP. We build a routine description for the interface manager. We state in the *BUILDRD* argument that the interface manager's argument is two integers and that it returns a scalar integer. We name the Routine Description CADDRD.

We then build a Routine List called CRL. It lists ADD as the name that is associated with *⎕NA*, CADDRD as the name that Processor 11 calls, and finally CADD, the name of the routine that the interface manager calls.

In use, AP2XCMAP is called by Processor 11, and pass the arguments on to CADD. When CADD completes, AP2XCMAP copies the contents of register 15 (CADD's result) to register 0 (where Processor 11 expects to find it.)

The AP2XCMAP program is provided because it is common to call a routine that returns a scalar integer. C/370 also returns non-integer scalar results using its own conventions. These are described in “C/370 Results” on page 316.

You may notice that the *BUILDRD* and *BUILDRL* arguments include a routine description called CENVRD. This describes a program that starts the C/370 environment and is discussed later.

Environments

Certain applications and languages such as Fortran, C/370 and PL/I require an environment to be established prior to calling routines that make use of environmental services such as input/output, error processing, or interrupt facilities. Normally, the necessary environment is established by specifying that the routine is a main program, since main programs typically establish the environment as a matter of course while subroutines do not.

When calling a non-APL routine from APL2, it is undesirable to start the environment on each call to the routine. It is preferable to start the environment once and leave it running between calls to subroutines. This is accomplished by separating the program into two (or more) parts: an environment program (sometimes referred to as the main program), and one or more subroutines. When the main program is called, the environment is started and the program then returns control to APL2 before terminating. APL2 can then make calls to subroutines that require the environment. The main program must not terminate before returning control to APL2, since the main program termination process typically involves shutting down the environment.

APL2 includes a program through which a environment program can return control to processor 11 without terminating. It is called AP2TNL on TSO and AP2VNL on CMS. When the main program is called, it should call AP2TNL (or AP2VNL on CMS). AP2TNL returns control to Processor 11 in such a way that the environment established by the main program is left available to subroutines. APL2 can then call the subroutines without the overhead of reestablishing the environment.

Processor 11 also ensures that such a program is allowed to terminate normally before it is deleted. When all the name associations to the subroutines are deleted from the workspace (either through $\square EX$, $)CLEAR$, $)OFF$, or $)CONTINUE$) Processor 11 returns control to the main program through AP2TNL. The main program can then end and the environment terminates. This process ensures that the necessary termination tasks, such as closing of data sets, can be accomplished in an orderly fashion.

A subroutine can be identified as requiring an environment through the use of the :INIT. tag in its Routine Description. In this case the name of the program that starts the required environment is coded as the value of the :INIT. tag.

The :INIT. tag is also used to identify that a program starts an environment. In this case a value of INITIAL is coded as the value of the :INIT. tag. Processor 11 ensures that the program has been successfully called before calls to the subroutine are made. The word `INITIAL` is prefixed with either an #, @, +, or - character to indicate what type of environment program the routine is.

In the following figure, the external function COMPUTE is identified as requiring the environment FORTENV that, in turn, is tagged as an environment program.

```
:NICK.FORTENV
     :LOAD.FORTLOAD
     :MEMB.FORTENV
     :LINK.OBJECT
     :INIT.#INITIAL
     :RARG.(C1 1 *)
     :DESC.FORTRAN environment program

:NICK.COMPUTE
     :LOAD.FORTLOAD
     :MEMB.CALC
     :LINK.FORTRAN
     :INIT.FORTENV
     :RARG.(G0 1 3)(1 I4 *)(E4 1 *)(<E8 1 *)
     :DESC.Executable subroutine requiring an environment
```

Figure 99. NAMES file Routine Descriptions for FORTRAN routines

In the description of FORTENV in the example, note the following:

- An environment program is described in the routine description in the same way any other routine would be, except that :INIT. has a value of INITIAL(preceded with a @, #, +, or - character.)
- :LINK. is specified as OBJECT rather than FORTRAN. This is because FORTRAN main routines use OS linkage conventions.
- :RARG. is specified for the environment program. Processor 11 may automatically invoked environment program in which case no argument is passed to the FORTENV program. FORTENV can, however, be invoked specifically before calling COMPUTE, and can be passed initialization parameters by treating it as a normal external function. For example:

```
        0 11 ⎕NA 'FORTENV'
1
        FORTENV 'NOXUFLOW'
```

Any attempt to specifically invoke an environment that is already active results in a *SYSTEM LIMIT* error (⎕*ET*=1 6) and a)*MORE* message queued, indicating that the environment program is already active.

When an routine is invoked through Processor 11, Processor 11 first checks to see if any environment is necessary and whether it has been established. If it has not and if the environment is identified :INIT.#INITIAL, it is automatically invoked, otherwise a *VALENCE ERROR* is signalled and a)*MORE* message queued indicating that the environment routine is unavailable.

More than one subroutine can be associated with a single environment program by coding the same name in the :INIT. tags for each of the subroutines.

It is good practice to use the same environment program for all executable subroutines written in a given language, since this ensures proper handling of common resources such as ddnames.

On TSO, processor 11 creates a task for the environment program; it switches to that task before calling the program or any subroutines dependent on the environment. Before calling subroutines associated with an environment, Processor 11

also reestablishes any abend and interrupt trapping exits established during the call to the environment program.

On TSO, if the :TASKLIB. tag is specified in the environment program's routine description, Processor 11 also creates a tasklib that it activates during calls to the main program or subroutines associated with an environment. This makes it possible to build an application with a unique search order.

Certain languages have an additional requirement concerning environment programs and subroutines. The Fortran language allows its environment programs and subroutines to each be in separate modules. C/370 and PL/I however, unless specially coded, each require that the environment program and all the subroutines be link-edited into a single load module. Because Processor 11 can only locate the main routine within a module this requires that a Routine List must be link-edited into modules containing C/370 and PL/I routines. The C/370 and PL/I languages also specify the names that may be used for the environment program names. Unless specially coded, the main entry point of C/370 environment programs are automatically named CEESTART, and PL/I environment programs are named PLISTART. Sample C/370 and PL/I routines can be found in Appendix G, “Sample Non-APL Programs to be Called through Processor 11” on page 385.

When an environment program is called by Processor 11, it is expected to establish the environment and return control to Processor 11 by calling AP2VNL in CMS/VM or AP2TNL in MVS/TSO (AP2VNL and AP2TNL are provided with APL2 and may be link-edited with user implemented environment programs). If the program returns control to Processor 11, at this point, other than by calling AP2VNL or AP2TNL, Processor 11 judges that the environment initialization has failed. The environment program and all routines that depend on its environment are marked unusable. Attempts to use them result in a *VALENCE ERROR* with a *NAME UNAVAILABLE* message queued.

When all the routines that use an environment program have been expunged (possibly because of an *)OFF*, *)CLEAR*, *)LOAD*, etc.), processor 11 returns control to the environment program as a return from AP2VNL or AP2TNL. When the program then returns control to Processor 11, it too is expunged.

The following program can be used as an environment for FORTRAN subroutines if compiled and link-edited with the AP2VNL (CMS) or AP2TNL (TSO). Notice that this program has no real function except to call AP2VNL. There is no explicit code to start the FORTRAN environment; the fact that the routine is a program rather than a subroutine causes FORTRAN to start its environment when the routine is called.

```
Program FORTENV
"
"  FORTRAN Environment Program
"
CALL AP2VNL    (for CMS, or CALL AP2TNL for TSO)
END
```

Figure 100. FORTRAN Environment Program

Interface Details

Routine Description Tags

Every non-APL external routine to be accessed through Processor 11 must be described by a routine description. The routine description informs Processor 11 about the arguments and results of the routine, what linkage convention should be used in calling the routine, what environment is associated with the routine, and how to locate the routine. This information is provided to Processor 11 through the arguments to ⎕*NA* and tags.

The format of routine description tags is as follows:

:TAG.value

where *:TAG.* is chosen from the following set of keywords and identifies the meaning of *value*. Tags and their values can be coded in either upper-, lower-, or mixed-case letters.

:NICK.*name* — specifies the name of the external routine. The same name must be specified in the right argument of ⎕*NA*. This tag is used to associate the routine description following the tag with the specified name. It is normally only used in NAMES files. If the :NICK. tag is specified in a routine description built by *BUILDRD*, it must match the name specified in the right argument of ⎕*NA*. In TSO NAMES files, this tag is optional since the partitioned data set member name provides the same function. In CMS NAMES files, this tag is required and must immediately precede the rest of the routine description tags. In both CMS and TSO, the name is restricted to uppercase letters and numerics. The maximum length of a name is 249 characters in CMS and 8 characters in TSO.

:LOAD.*library* — the name (CMS) or ddname (TSO) of the load library into which the routine has been link-edited. In TSO, the data set must have been previously allocated using the specified ddname. In CMS, the library name is of the form *FN FT FM*. *FT* and *FM* default to LOADLIB * if not specified. Consult section “Processor 11 Routine Search Order Guidelines” on page 325 for information about processor 11's search order if the :LOAD. tag is not specified.

:MEMB.*name* — the member name of the routine in the specified load library. If :LOAD. is not specified, it is the name of a previously loaded module in MVS/TSO or the name of a CMS nucleus extension, module, or TEXT file in CMS/VM. The :MEMB. tag is optional in a routine description built using *BUILDRD*.

:ENTRY.*name* — the routine name for the external routine. *name* is converted to uppercase for non-APL routines. Load modules with multiple routines must have the entry points of the routines listed in a Routine List located at the beginning of the load module. The *BUILDRL* function can be used to build an appropriate Routine List. The :ENTRY. tag should not be used unless a routine list in the format required by Processor 11 has been created and specifically included in the module. See “Routine Lists” on page 298 for details.

:LINK.OBJECT|FORTRAN|FUNCTION — specifies the type of program linkage to be used when calling the routine.

`OBJECT` — specifies the routine uses standard operating system entry and exit linkage conventions.

`FORTRAN` — specifies the routine uses FORTRAN's entry and exit linkage conventions.

`FUNCTION` — specifies the routine uses APL2's entry and exit linkage conventions. This linkage convention supports routines that have been specifically designed to operate with APL2 and Processor 11 and to resemble APL functions and operators. Additional information and facilities are provided to routines using this linkage convention. Consult “Linkage Conventions” on page 321 for further details. See 'Linkage Conventions' in *APL2/370 Programming: Processor Interface Reference* for detailed information.

:INIT. This tag may be specified with any of the following formats:

The following forms of the :INIT. tag identify routines that are environment programs.

:INIT.`#INITIAL` *or* :INIT.`+INITIAL` -identifies an environment program that Processor 11 automatically calls before invoking routines that utilize the environment. Unless the :PARM. tag is coded, no argument is passed to automatically invoked environment programs. In addition, they are assumed to have no explicit result. Any explicit result they produce is discarded. `+INITIAL` is provided because the symbol # is not available on all terminals.

:INIT.`@INITIAL` *or* :INIT.`-INITIAL` - identifies an environment program that must be explicitly called by the user before invoking routines that utilize the environment. `-INITIAL` is provided because the symbol @ is not available on all terminals.

The following forms of the :INIT. tag are alternative ways to inform Processor 11 of the name of a required environment program. They are used when the program is self-describing or is described in a NAMES file other than the current NAMES file. See “Environments” on page 303 for details.

:INIT.*name* - specifies the name of the environment program. If this form of the tag is specified by a self-describing routine in a module containing a routine list, the specified *name* is searched for in the same module's routine list. Otherwise, the specified *name* is searched for in the current NAMES file.

:INIT.*(private).name* - specifies the nick name of the environment program. The specified *name* is searched for in the specified *private* NAMES file.

:INIT.*member.name* - specifies the *member* in which the environment program *name* should be found. A search equivalent to `'member' 11 ⎕NA 'name'` is used to find the environment program. With this form of the tag, the environment program must be a self-describing routine.

:INIT.*library.member.name* - specifies the *library* and *member* in which the environment program *name* should be found. A search equivalent to `'library.member' 11 ⎕NA 'name'` is used to find the environment routine. With this form of the tag, the environment program must be a self-describing routine. Further details can be found in “⎕NA Syntax for Non-APL Programs” on page 292.

:TASKLIB.*ddname* — The ddname of a tasklib to be used in conjunction with this routine. This tag should only be used for environment programs. Use of the :TASKLIB. tag causes Processor 11 to set up a tasklib with the specified ddname. This tasklib is activated for any calls to the environment program and any subroutines that have specified that they depend on the environment. It is optional. The :TASKLIB. tag is ignored on CMS and for all routines other than environment programs on TSO.

:TIME.*yyyy mm dd hh mm ss* — specifies the GMT fix time for the external routine. It becomes `2 ⎕AT`. This tag is optional. If it is not specified, fix time is defaulted to the time at which the name was associated with Processor 11.

:DESC.*description* — allows inclusion of descriptive text in a routine description. Where the descriptive text exceeds the record length of a NAMES file, multiple records of text may be included, but each must be prefixed by a :DESC. tag. Comments may also be included in the NAMES file by placing an asterisk in column 1 of comment records.

:LARG.*pattern* — specifies a pattern for the left argument of the external routine. The tag is optional. It may be omitted, specified as null (for example, :LARG.) or specified with a pattern. This tag may not be used with :LINK.OBJECT or :LINK.FORTRAN routines. See “Argument Patterns” on page 310 for details.

:RARG.*pattern* — specifies a pattern for the right argument of the external routine. The tag is optional. It may be omitted, or specified with or without a patter. See “Argument Patterns” on page 310 for details.

:RSLT.*pattern* — specifies a pattern for the result of the external routine. This tag is optional. It may only be specified for routines with :LINK.OBJECT or :LINK.FORTRAN to specify the explicit result of the routine. See “Result Patterns” on page 314 for details.

Note: This tag cannot be specified for environment and :LINK.FUNCTION programs.

:PARM.*parameter* — provides a parameter string for automatically called environment programs. The data for the :PARM. tag is a quoted character string (double apostrophes are supported in the string.) If the environment program is automatically started the character string, prefixed with a 2 byte length field, is provided to the program using OS linkage conventions:

```
R1 => A(argument+X'80000000') => H'length',C'string'
```

For programs that are started with an explicit call from APL2, an argument can be passed if :RARG. is specified in the program's routine description.

For example:

```
0 11 ⎕NA 'ENVPGM'
ARGS←'string'
ENVPGM ('I2 1 1' RTA ⍴ARGS),ARGS
```

:VALENCE.*er fv ov* — specifies the valences of the routine. It becomes `1 ⎕AT` for the routine.

er specifies whether the routine has an explicit result. If *er* is 0, the routine does not have an explicit result. Neither the :RSLT. tag may be used nor may either < or > be used in the :RARG. tag for a routine for which the :VALENCE. tag specifies that no explicit result is produced. Any result returned through the ECV is discarded. If *er* is 1, the routine has an explicit result.

fv is the number of arguments the routine expects. If *fv* is 0, the routine is niladic. If *fv* is 1, the routine is monadic. If *fv* is 2, the routine is ambivalent. A *fv* value of 2 may not be coded for :LINK.OBJECT or :LINK.FORTRAN routines.

ov is the number of operands the routine expects. If *ov* is 0, the routine is a function. If *ov* is 1, the routine is an operator with one operand. If *ov* is 2, the routine is an operator with two operands. Only a *fv* value of 0 may be coded for :LINK.OBJECT or :LINK.FORTRAN routines.

This tag is optional. If it is not specified, the valences default as follows:

- The routine is assumed to have an explicit result (*er*=1).
- The routine is assumed to be monadic if :LINK.OBJECT or :LINK.FORTRAN (*fv*=1) and ambivalent if :LINK.FUNCTION (*fv*=2).
- The routine is assumed to be a function.

:LINK.OBJECT and :LINK.FORTRAN routines cannot be ambivalent.

Like operators written in APL2, non-APL operators may not be niladic.

The :VALENCE. tag is not allowed with :LINK.APL objects. The valences of objects in namespaces are determined from the objects' definitions.

Argument Patterns

The :LARG., :RARG., and :RSLT. tags are used to specify the expected arguments and results for an external routine. They can be specified with argument patterns that provide descriptions of the expected arguments and results for the routine.

When an external name is encountered during the execution of an APL expression, APL compares the actual arguments against the patterns provided in the routine's description. If possible, APL converts the actual arguments to match the patterns so the external routine receives its argument data in an expected and predictable form. If it is not possible to accomplish the conversion, or if an inconsistency is found between the actual arguments and the patterns, APL issues an appropriate error message.

When the external routine completes, if the :RSLT. tag has been used, APL uses the pattern specified in the :RSLT. tag to convert the data returned by the routine to an APL2 array.

Simple Arrays

[*count*] [>|<] *type rank* [shape]

Where:

count is the number of elements in the array. This number may be omitted if fully specified by *shape* or may be specified as * if a variable number of elements is permitted.

> or < are optional output or update indicators. If neither > or < is specified, the item is an input argument only and may not be updated. If < is specified, the item may be updated by the routine and is made an item of the explicit result. If > is specified, the argument must be specified by name and may be updated directly by the routine. For additional information, see "Updated Arguments and Results" on page 313. Unpredictable results may occur if an external routine updates an argument that is not marked > or <.

type provides information on the representation type and representation length of the data. The following *type* combinations are supported:

Figure 101. Representation Types and Lengths

Representation	Type and Length
B1	unsigned integer, 1 bit/element (Boolean).
B4	unsigned integer, 4 bits/element (hex).
B8	unsigned integer, 8 bits/element.
I2	integer halfword.
I4	integer fullword.
E4	single precision floating point.
E8	double precision floating point.
E16	extended precision floating point.
J8	single precision complex.
J16	double precision complex.
J32	extended precision complex.
C1	character byte.
C4	character fullword.
P*n*	variable length packed decimal. The value for *n* may be 1 to 16 bytes.
Z*n*	variable length zoned decimal. The value for n may be 1 to 16 bytes.
A8	fullword integer progression (first value, increment).
G0	general object, (Note: See nested array below.)

rank is the rank of the array. If varying ranks are accepted, it may be specified as *, and *shape* should not be specified.

shape is the shape of the array. One integer must be specified for each dimension of the array in row major order. If varying shapes are accepted, one or more elements of *shape* may be specified as *.

Numeric elements of the pattern must be separated from one another by one or more blanks. Parentheses are treated as blanks and may be used freely.

Simple Array Pattern Examples

:RARG.I4 2 2 3 A matrix, `2 3⍴6?6`

:RARG.I4 2 * * Fullword integer matrix of any shape

:RARG.1 I4 * A single fullword integer of any rank

:RARG.100 E8 2 * * Floating point matrix of any shape containing 100 numbers

Nested or mixed arrays: The pattern for a nested array or for a mixed array is a recursive structure where the first subpattern has *type* = G0 (general object) and describes the overall array. Subsequent subpatterns describe each item of the array. For example, the 2-element vector of vectors (100 200 300) ('*ABCD*') may be described:

```
(2 G0 1 2) (3 I4 1 3) (4 C1 1 4)
```

A general object pattern is needed when calling a :LINK.OBJECT or :LINK.FORTRAN routine that requires more than one argument because the routine is called monadically from APL with a nested argument. For example, to call the following FORTRAN subroutine:

```
      SUBROUTINE AUTO(A,N,R)
C     ARGUMENTS A AND R ARE VECTORS OF LENGTH N
      REAL*8 A(1),R(1)
      INTEGER*4 N
```

Produces a result in R, the following argument pattern would be required:

```
(G0 1 3) (E8 1 *) (1 I4 *) (<E8 1 *)
```

and the routine could be called with the following APL statements:

```
        '(FORTNMS)' 11 ⎕NA 'AUTO'
1
        ARGUMENT←100?1000
        RESULT←AUTO (ARGUMENT 100 (100⍴0))
```

The record length and the total length of any individual tag in the NAMES file is limited to 255 characters. Certain arguments, however, require patterns of more than 255 characters. To accommodate this requirement, the argument may be described in successive records in a NAMES file, each of which is prefixed with an appropriate argument tag. For example:

```
:RARG.(G0 1 4) (1 E8 *)
:RARG.(I4 2 * 10)
:RARG.(<I4 2 10 *)
```

If a :RARG. tag is coded without a pattern for a :LINK.OBJECT or :LINK.FORTRAN routine, Processor 11 builds a parameter list that contains pointers to the data in each simple element of the argument array. No length information is provided.

Nested and Mixed Array Pattern Examples

:RARG.G0 1 2 (I4 2 2 3)(C1 1 4)
A nested array, (2 3⍴6 6) '*ABCD*'

:RARG.G0 1 3 (C1 1 *)(C1 1 *)(C1 1 *)
Vector of 3 words of any length

:RARG.
Any array, no data conversion

:RARG.G0 1 3 (1 I4 *)(C1 1 *)(1 P5 *)
General array

- 1st element integer scalar

- 2nd element character vector any length
- 3rd element 5 byte packed decimal number

Updated Arguments and Results

Routines in languages like FORTRAN or Assembler typically do not distinguish between input arguments and results. They are passed a list of pointers to the values that may represent input arguments, values to be updated, or preallocated space for results. APL functions, on the other hand, take arguments that are never updated and produce explicit results that were not previously passed as arguments to the function. For example:

```
      ∇RESULT←COMPUTE ARG
[1]   RESULT←2×ARG
[2]   ∇
      COMPUTE 10
20
```

APL requires that functions that update argument data in place be called with the names of the arguments rather than their values. For example:

```
      ∇UPDATE ARG
[1]     ⍎ARG,'←2×',ARG
[2]   ∇
      NUMBER←10
      UPDATE 'NUMBER'
      NUMBER
20
```

Both approaches are supported by Processor 11. Argument items that are to be updated in place are indicated with the symbol > preceding the representation type in the argument pattern. For example, a FORTRAN routine that expects two integer vectors as arguments, the second of which is to be updated in place, would be described:

```
:RARG.(G0 1 2) (I4 1 3) (>I4 1 3)
```

As with APL functions, such routines must be called with the names of the arguments to be updated rather than their values:

```
      RESULT←13⍴0
      COMPUTE (1 2 3) 'RESULT'
```

APL checks to ensure that arguments updated with the > symbol are names and not values. If names are not found when the function call occurs, an error results.

Arguments that are updated and made items of the external function's explicit result are indicated with the symbol < preceding representation type in the argument pattern. For example, a monadic routine that takes a vector of integers as input and produces a vector of real numbers as a result might be described as follows:

```
:RARG.(G0 1 2) (I4 1 3) (<E8 1 3)
```

and later called in an APL expression such as:

```
      OUTPUT←COMPUTE (⍳3) (3⍴0)
```

A second item must be passed as an argument to the function because subroutines in languages like FORTRAN require that space for results be preallocated by the

caller. It is also possible to pass input values to a routine using an argument that is marked as a result item:

```
OUTPUT←COMPUTE (⍳3) (1.23 100.2 .456)
```

Routines that describe more than one argument item as a result item (with the symbol <) produces a nested vector as their result with successive elements corresponding to the indicated argument items. A < may only be coded on a section of the pattern that describes a simple array; it may not be coded preceding a G0 representation type.

Including updated arguments in the explicit result of an external function (with the symbol <) leads to a coding style that is less cumbersome and more familiar to the APL user. Users should be aware, however, that this technique may require temporary allocation of two copies of the value— one for the argument and one for the result. Marking argument items able to be updated in place (with the symbol >) is more cumbersome since the function must be called with the name of the data, but requires only a single copy of the value in most situations.

:LINK.FUNCTION routines may use < and > in their :LARG. and :RARG. tags. If they then return no result through their ECV, Processor 11 returns all the specified items from the tags as a nested vector of arrays.

If an external routine updates an argument item that is not marked as able to be updated with either < or >, unpredictable results may occur. While the external function may appear to operate correctly and produce the desired result, other values in the workspace may have been destroyed and workspace damage may have occurred. Furthermore, if an external routine misbehaves, for example by writing beyond the end of an argument that is able to be updated, workspace damage and possibly *SYSTEM ERROR* may occur. It is essential that valid argument patterns be constructed before calling an external function. For additional information on this topic, see the section “Unexpected Errors” on page 322.

Result Patterns

The :RSLT. tag can be specified for routines with :LINK.OBJECT or :LINK.FORTRAN to specify the explicit result of the routine. The :RSLT. tag cannot be coded if < output indicators are coded in the argument tags.

:RSLT. can be specified with or without a pattern. If specified with a pattern, the pattern takes the same form as argument patterns described above, with the following exceptions:

- * cannot be used in the pattern
- the update indicator, >, cannot be used in the pattern
- the output indicator, <, can be used only immediately before the representation type in the pattern.

If :RSLT. is specified without a pattern, the external routine is expected to return in register 0 the address of a self-describing data structure called a CDR. For a definition of the CDR, see *APL2/370 Programming: Processor Interface Reference*.

If :RSLT. is specified with a pattern in which the first representation type is preceded by an output indicator, for example:

```
(<G0 1 3) (I4 1 3) (1 E8 *)
```

the external routine is expected to return in register 0 the address of a fullword containing the length of the result data immediately followed by the result data.

If :RSLT. is specified with a pattern in which the first representation type is not preceded by an output indicator, for example:

```
( 1  I4  * )
```

the pattern must describe a simple scalar with binary, integer, real or complex representation type. The following table lists the acceptable simple scalar patterns and where the external routine is expected to provide the result:

Figure 102. Acceptable Simple Scalar Patterns

Pattern	Result Provided In:
B0 0	the low-order bit of register 0
B4 0	the 4 low-order bits of register 0
B8 0	the 8 low-order bits of register 0
I2 0	the low-order half word of register 0
I4 0	register 0
E4 0	floating point register 0
E8 0	floating point register 0
E16 0	floating point registers 0 and 2
J8 0	floating point registers 0 and 2. The real portion in floating point register 0, imaginary portion in floating point register 2.
J16 0	floating point registers 0 and 2. The real portion in floating point register 0, imaginary portion in floating point register 2.
J32 0	floating point registers 0, 2, 4, and 6. The real portion in floating point register 0 and 2, imaginary portion in floating point register 4 and 6.

Note: Simple scalar patterns (without the preceding output indicator) of types C1, C4, Pn, Zn, A8 and G0 are not supported.

In summary:

:RSLT. the external function is expected to return the address of a CDR in register 0.

:RSLT.<*pattern* the external function is expected to return the address of a fullword containing the length of the result data followed by the data.

:RSLT.*pattern* the pattern must represent a simple scalar of type B1, B4, B8, I2, I4, E4, E8, E16 J8, J16, or J32. The external function is expected to return the result in register 0 or floating-point register 0 or floating-point registers 0, 2, 4, and 6 as described in Figure 102.

C/370 Results

Processor 11 expects that non-APL routines return scalar integer results in register 0. C/370 returns scalar integer results in register 15. An interface management program called AP2XCMAP is supplied with APL2 that can be used to map C/370 scalar integer results to register 0. A discussion of using AP2XCMAP can be found in “C/370 Scalar Integer Results” on page 303.

Processor 11 expects that results other than scalar integers are in a register. Register 0 contains the address of a CDR, or register 0 contains the address of the length of the result followed by the result. C/370 does not use any of these techniques for returning results.

C/370 subroutines that return results (as opposed to updating arguments) expect that the first element of the parameter list passed to them is the address of storage into which they should place their results. This element of the parameter list is not explicitly mentioned either in the C/370 subroutine nor its caller. It is simply a C/370 convention. It is the responsibility of the calling program to provide a large enough storage area to hold the subroutine's result.

Although Processor 11 does not provide support for this C/370 convention, it is possible to call these routines and receive their results. An extra parameter should be coded on the :RARG. tag to receive the result. If the APL2 caller of the C/370 subroutine passes an array for the parameter, Processor 11 allocates the appropriate amount of storage and adds the address to the parameter list. For example, if this were our C/370 routine:

```
#pragma linkage(pitimes,OS)
double pitimes(int i) {
   return 3.14159 * i;
   }
```

we could use this routine description:

```
:LINK.OBJECT
:RARG.(G0 1 2)(<E8 0)(I4 0)
```

and call the subroutine like this:

```
      PItimes 0 3
9.42477
```

Finally, note that this entire discussion concerns C/370 routines that are coded without a `#pragma` statement that causes them to accept arguments and return results using standard operating system conventions.

:LINK.FUNCTION Arguments

When used with :LINK.FUNCTION routines, the :LARG. and :RARG. tags instruct Processor 11 what technique should be used to pass arguments to the external routine.

Processor 11 has two techniques for passing arguments to a :LINK.FUNCTION routine. The first is to pass a token representing the argument. The routine uses APL2 services to retrieve the data represented by the token. The second is to pass the actual data in a form called a CDR. CDRs are APL2's method of passing actual data to programs outside the workspace. Processor 11 retrieves the data from the workspace and converts it to the CDR format.

The existence of the :LARG. and :RARG. tags controls whether or not Processor 11 builds and passes CDRs for the arguments or whether only tokens are passed.

If an argument tag is not coded, then Processor 11 does not build a CDR for the argument, it only passes the token.

If an argument tag is coded with a pattern then Processor 11 builds a CDR. During the process, Processor 11 may need to convert the representation of the argument to fulfill the requirements specified in the pattern. If Processor 11 is unable to convert the argument to the form specified by the pattern, a *DOMAIN ERROR* is issued.

If an argument tag is coded without a pattern (for example, :RARG.), then processor 11 builds a CDR, but performs no conversion and therefore imposes no conformability restrictions on the argument.

Notice that the mere existence of the :LARG. and :RARG. tags does **not** affect the valence of the routine. The default valence of a routine is implied by the :LINK. tag. The valence of a routine can be modified by the :VALENCE. tag. The :VALENCE. tag is also used to specify the number of operands required by non-APL operators.

For further information about CDRs, tokens, and :LINK.FUNCTION interface protocols, consult *APL2/370 Programming: Processor Interface Reference*.

Explicit Results, Function Valence, and Operator Valence

This section discusses:

- Explicit results
- Function valence
- Operator valence

Explicit Results

The first value of the :VALENCE. tag is used to specify whether the routine returns an explicit result. This value does not imply that the routine is required to return a result. Rather, this value is used to specify what `1 ⎕AT` should return when it is used to query the valence of the routine. :LINK.OBJECT and :LINK.FORTRAN routines whose :VALENCE. tags specify that they have no explicit result must not use :RSLT. tags or have < characters in the :RARG. tag. :LINK.OBJECT, :LINK.FORTRAN, and :LINK.FUNCTION routines whose :VALENCE. tags specify that no explicit result is produced, have any result returned through their ECV discarded. :LINK.FUNCTION

routines whose :VALENCE. tags specify that an explicit result is produced can choose not to return a result.

Function Valence

Function valence is controlled by the second value coded in the :VALENCE. tag and is implied by the value of the :LINK. tag.

:LINK.OBJECT and :LINK.FORTRAN routines are monadic by default. The :VALENCE. tag can specify that such routines are niladic. A :LARG. tag should not be coded for these routines.

Note: Unless the :VALENCE. tag indicates that a :LINK.OBJECT or :LINK.FORTRAN routine is niladic, then a :RARG. tag must be coded.

:LINK.FUNCTION routines are ambivalent by default. The :VALENCE. tag can specify that such routines are monadic or niladic. The omission or inclusion of the :LARG. and :RARG. tags determines whether Processor 11 builds a CDR when calling the routines.

Operator Valence

:LINK.FUNCTION routines can be operators. The third value of the :VALENCE. tag is used to specify whether the routine expects one or two operands. Processor 11 passes tokens for the operands whether they are functions or arrays. It is the external operator's responsibility to determine the name class of the operands and respond accordingly.

NAMES Files

Every non-self-describing external routine to be accessed through Processor 11 must be described in a NAMES file available to the processor.

In the CMS/VM environment, a NAMES file is a sequential file with file type NAMES on an accessible CMS minidisk that contains descriptive information for one or more external routines. Each description must begin with a :NICK. tag and continues until the next :NICK. tag or the end of the file. In the MVS/TSO environment, a NAMES file library is a partitioned data set in which each member describes an individual external routine. It may contain fixed or variable length records with a maximum record length of 255 bytes. Any line that starts with an * is a comment. Tag data may not be continued onto new records. Argument pattern tags can be repeated if the pattern does not fit on one record. For example:

```
:DESC.Right argument is 10 vectors of 6 integers each.
:RARG.(G0 1 10)(I4 1 6)(I4 1 6)(I4 1 6)(I4 1 6)(I4 1 6)
:RARG.(I4 1 6)(I4 1 6)(I4 1 6)(I4 1 6)(I4 1 6)
```

In MVS/TSO, APL2's default NAMES file library is allocated to ddname AP2TN011. Concatenated allocation is supported and can be used to support multiple libraries and specify search order. Entries provided with APL2 are distributed in a library with the name 'APL2.AP2TN011.NAMES'. Installation and user NAMES file entries can be provided in libraries concatenated with it.

In VM/CMS, APL2's default NAMES files are P011 NAMES * and AP2VN011 NAMES * and are searched in that order. Entries provided with APL2 are distrib-

uted in AP2VN011 NAMES. P011 NAMES may be defined to include descriptions of external functions not distributed with APL2.

Note: For performance reasons, when a partitioned data set containing NAMES file entries (private or the default) is used by Processor 11 on TSO, it is left open until another NAMES file is used or the APL2 session ends. Once the data set is opened, changes made to the data set are not detected until Processor 11 closes and reopens the data set. When making changes to a NAMES file, either exit APL2 or instruct Processor 11 to open a different NAMES file by specifying a different NAMES file in a $\square NA$ expression before trying to use your changes.

Note: When a tag occurs multiple times in a NAMES file, the first occurrence of the tag is used on CMS, the last occurrence is used on TSO.

Processor 11 Non-APL Routine Description Tag Rules

For each external routine, Processor 11 must be informed how to locate the routine. The location information (search order, member name, and entry) can reside either in the left argument of $\square NA$ or in a NAMES file.

For each external routine accessed through Processor 11, a routine description must also be provided. The routine may either be self-describing or the routine description may reside in the NAMES file. If the routine is self-describing, it may also have a NAMES file entry. However, if the routine is self-describing, the NAMES file entry may only contain tags that instruct Processor 11 how to locate the routine.

If you use NAMES files, we recommend that you use a private NAMES file and specify its name in the left argument of $\square NA$ rather than using the default NAMES files. This helps to avoid possible conflict with routine names in the APL2 default NAMES files.

If you use self-describing routines, IBM recommends that you do not use NAMES files and rather specify the load library and member names in the left argument of $\square NA$. This also helps to avoid possible conflict with routine names in the APL2 default NAMES files and additionally avoid the overhead of NAMES file searches.

Figure 103 on page 320 illustrates how Processor 11 routine description tags can be used in NAMES files and self-describing routines. Notice that whether or not a NAMES file exists, the tags allowed in the routine's self-description do not change.

Figure 103. Processor 11 Non-APL Routine Description Tag Rules

Tag	In NAMES file if routine is not self-describing	In NAMES file if routine is self-describing	In routine's self-description
Routine location tags			
:NICK.	Required [1]	Required [1]	Tolerated [2]
:LOAD.	Optional [3]	Optional [3] [4]	Ignored [4]
:MEMB.	Required	Required [4]	Ignored [4]
:ENTRY. [5]	Optional	Optional	Ignored [4]
Routine calling procedure tags			
:LINK.	Required	Not allowed [6]	Required
:INIT.	Optional	Not allowed	Optional
:TASKLIB.	Optional	Not allowed	Optional
Routine parameter tags			
:LARG.	Optional	Not allowed	Optional
:RARG.	Optional	Not allowed	Optional
:RSLT.	Optional	Not allowed	Optional
:PARM.	Optional	Not allowed	Optional
:VALENCE.	Optional	Not allowed	Optional
Miscellaneous tags			
:TIME.	Optional	Not allowed	Optional
:DESC. [7]	Ignored	Ignored	Ignored

Notes:

1. :NICK. tags are required in NAMES files on CMS. However, since NAMES files entries on TSO are located by member name, the :NICK. tag is optional on TSO. For portability, we suggest you always include :NICK. tags in NAMES files.
2. The :NICK. tag is allowed within a self-describing routine, but its value must match name specified in the right argument of $\square NA$.
3. If the :LOAD. tag is omitted from a NAMES file, then a standard operating system search order is used to find the member.
4. Although the :LOAD., :MEMB., and :ENTRY. tags are all allowed in self-describing routines, they are not needed and their values are ignored. If there is no NAMES file, the member name, and optionally the load library name, must be provided in the left argument of $\square NA$.
5. The :ENTRY. tag defaults to the name specified in the right argument of $\square NA$.
6. APL2 namespaces are actually self-describing also and the :LINK. tag can be used in the NAMES file entries. The restriction applies only to non-APL routines.
7. The :DESC. tag is provided merely for documentation purposes. Processor 11 never uses the tag's description.

System Usage Guidelines

This section discusses the system usage guidelines.

Linkage Conventions

The :LINK. tag is used to inform Processor 11 what convention to use when calling a routine.

The OBJECT and FORTRAN linkage conventions provide the ability to call non-APL programs using operating system and FORTRAN conventions. The programs are passed arguments and expected to return results using standard parameter lists and structures. These programs are passed information that can be used to exploit APL2 service routines.

The FUNCTION linkage convention is intended for programs that are specifically designed to be called by APL2. These programs are passed arguments and expected to return results using APL2's conventions. These programs are also passed information that can be used to exploit APL2 service routines. In addition, their interface provides for complete mimicry of APL programs; their valence, results, and errors can all be controlled like APL programs.

OBJECT and FORTRAN Linkage

Standard OS linkage is used for calling external routines. Routines with :LINK.`OBJECT` are entered with register 1 pointing to a list of argument addresses, terminated with a 1 in the high-order bit of the last address. For routines that take a single argument, this is the address of the data described by the simple array pattern. For routines that take more than one argument, each address corresponds to an item of the general object described.

Routines with :LINK.FORTRAN are entered using the VS FORTRAN (Version 1 Release 4 or Version 2 Release 1 or later) linkage convention as described in the *VS FORTRAN Version 2 Programming Guide*. With this linkage convention, register 1 points to a parameter list with the following format:

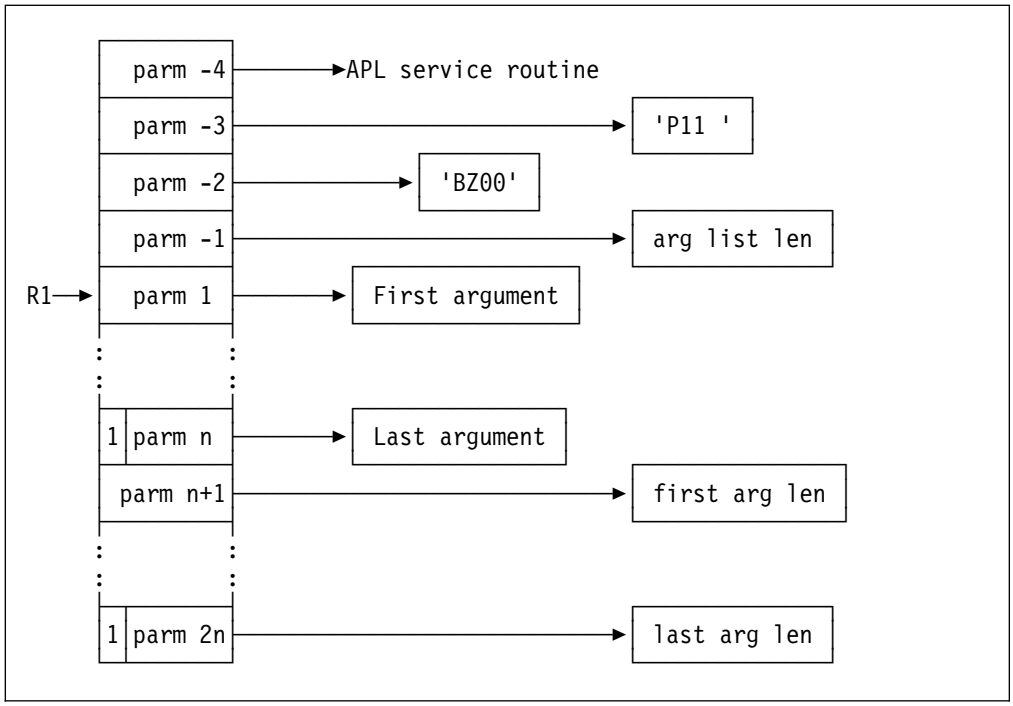


Figure 104. :LINK.FORTRAN Routine Invocation Parameter List

The entry protocols for :link.OBJECT and :link.FORTRAN are well suited to calling FORTRAN or Assembler routines. On entry, the routine is given a set of pointers to the data arguments it expects. These arguments have been checked and converted by APL to conform to the argument patterns provided in the routine's description and, because of this, no additional special processing is required.

FUNCTION linkage

Routines with :LINK.FUNCTION can mimic APL functions; their valence can be niladic, monadic, or ambivalent, and they can produce an explicit result or not. They can also queue messages and signal errors. These routines are designed to be written specifically for the APL2 environment, and are given access to data and storage within the workspace. Detailed information on the interfaces to these routines is provided in *APL2/370 Programming: Processor Interface Reference*.

Unexpected Errors

While testing and debugging external routines, it is recommended that the user invoke APL2 with the DEBUG(1) invocation option. This causes all)*MORE* messages to be displayed automatically as they occur.

Name Association Failures: There are a number of reasons why dyadic ⎕*NA* fails to activate a name and return 0:

- Incorrect arguments. Malformed APL names in the right argument of dyadic ⎕*NA* cause the ⎕*NA* to fail and return 0. If monadic ⎕*NA* on the same right argument returns ¯1, the object name is malformed.

An invalid left argument of dyadic ⎕*NA* also causes the request to fail and return 0. Specification of the incorrect or a non-existent processor number causes a failure. More likely, however, the first item in a row of the left argument is malformed or incorrect. In these cases a)*MORE* message indicating a *PARAMETER ERROR* is queued. See "NAMES Files" on page 318 for additional information on valid left arguments for dyadic ⎕*NA*.

Finally, if dyadic ⎕*NA* is issued for a name that already exists in the APL workspace, its left argument must match the left argument of the dyadic ⎕*NA* originally used to establish the name. Monadic ⎕*NA* returns the original left argument of dyadic ⎕*NA* used to establish an existing name.

- Error in the Routine Description. An invalid parameter in the routine description causes dyadic ⎕*NA* to return a 0 and a)*MORE* message to be queued indicating the problem.
- Routine cannot be located. If the external routine name cannot be located either in the default search order or in the NAMES file, or if the routine specified in the NAMES file cannot be found or loaded, dyadic ⎕*NA* returns a 0 and a)*MORE* message is queued indicating that the external routine is unavailable.
- Environment damaged. A *VALENCE ERROR* resulting from an attempt to execute an external routine with an environment can indicate that the environment has terminated. If this is the case,)*MORE* messages are queued indicating that both the external routine and its external environment are unavailable, and a subsequent dyadic ⎕*NA* for the external routine returns a `0`. If this situation occurs, the external routine name must be explicitly erased from the workspace and its execution stack before a successful dyadic ⎕*NA* can occur. Other names associated with external routines requiring the same environment may also have to be erased.
- Insufficient freespace for proper operation of Processor 11 can cause dyadic ⎕*NA* to return a 0. A)*MORE* message is queued in this situation. The user should invoke APL2 with more freespace.
- Processor 11 or NAMES file unavailable causes dyadic ⎕*NA* to return 0. Proper operation of Processor 11 and availability of the NAMES file supplied with APL2 can be verified by running the APL2 installation verification procedure. Details can be found in the *APL2/370 Programming: Using the Supplied Routines*.

APL Errors during External Routine Execution: Various errors can occur during the execution of an external routine. For routines with :LINK.`OBJECT` or :LINK.`FORTRAN`, error reports are generated either by Processor 11 or by APL2. For routines with :LINK.`FUNCTION`, errors can also be generated by the routine itself.

- Errors such as *RANK ERROR*, *LENGTH ERROR*, or *DOMAIN ERROR*, often indicate a mismatch between the external routine arguments and the argument patterns in the routine description. Argument patterns should be checked carefully against the actual arguments provided when this situation occurs. Common errors include using a scalar when a vector is expected (or vice versa) and providing a value rather than the name of an object when an object is to be updated. The external function *PFA*, distributed with APL2, may be of assistance in building argument patterns or in understanding the structure of an argument.
- *VALENCE ERROR* indicates that the external routine is called with an incorrect number of arguments or that the external routine is unavailable. An

external routine is marked unavailable if Processor 11 encounters an error in locating or loading the routine, an error in the NAMES file, an invalid or inconsistent argument descriptor, or if the environment routine associated with it terminates. When an external routine is marked unavailable, a)*MORE* message is queued and a subsequent dyadic ⎕*NA* for the external routine name returns a `0`.

The environment associated with an external routine can terminate as the result of an unexpected error in any of the routines that share the environment. A)*MORE* message is queued along with the *VALENCE ERROR* when this situation occurs to indicate that the external routine is no longer available.

- A *SYSTEM ERROR* that does not lead to a *CLEAR* workspace during the execution of an external routine generally indicates that the external routine terminated abnormally. For most external routines, a)*MORE* message is queued when this situation occurs to indicate that an internal error occurred during execution of the routine. See "ABENDS and Internal Errors in External Routines" for additional information.
- A *SYSTEM LIMIT* error with ⎕*ET*=`1 6` is issued when the user attempts to invoke an environment that is already active. A)*MORE* message explaining this error is also queued. An environment that was automatically started (:INIT.`#INITIAL`) can be deleted only by expunging all the names that require the environment.
- A *SYSTEM LIMIT* error with ⎕*ET*=`1 12` is issued when an external routine returns a result in an invalid format. No)*MORE* message is queued. This error rarely occurs for routines with :LINK.`FORTRAN` or :LINK.`OBJECT`, unless the routine malfunctions and destroys data in the workspace. (A *SYSTEM ERROR* is more likely in this situation.) This problem can be fairly common, however, for routines with :LINK.`FUNCTION` that have not been thoroughly debugged. The error generally indicates an invalid result CDR format or descriptor.
- Some routines expect that CMS cleans up the environment when they complete as control returns to CMS ready. These routines can fail unexpectedly when used from APL2 because APL2 does not perform all the same clean up operations after each call to a routine that CMS performs. In particular, you should issue a `FILEDEF` *ddname* `CLEAR` for any ddnames established during the session for external routines when the routines have finished processing. Failure to do so can cause problems after leaving APL2; storage acquired for each filedef is not automatically freed upon leaving APL2.

Suspended External Routines: Like any APL routine, an external routine whose execution is incomplete, either because it is suspended, is recorded on the execution stack. An external routine cannot be deleted by Processor 11 until all stacked references have been cleared and the external routine erased.

ABENDS and Internal Errors in External Routines: External routines that expect or wish to handle program checks or ABENDS should establish an environment that handles them. Processor 11 does not support subroutines that set their own ABEND recovery, except for subroutines with :LINK.`FUNCTION` that use the ABEND recovery service provided by APL.

FORTRAN and similar languages typically provide program check and ABEND recovery in their environment and not in individual subroutines. To provide access to those facilities, it is therefore necessary to establish an environment (see

"Environments" on page 303) for these external routines. Note that this may be necessary even for external routines that operate normally, since certain languages depend on capturing program checks to detect and correct conditions such as numeric overflow and underflow.

Program checks and ABENDS in external routines that are not handled by its environment (or by the APL ABEND service in the case of routines with :LINK.FUNCTION) are handled by Processor 11. This situation causes a *SYSTEM ERROR* to be reported and a *)MORE* message to be queued, indicating an internal error in the external routine.

Program checks, ABENDS, or other conditions that cause an environment to terminate during the execution of an external routine causes the environment to be shut down and all external routines that depend upon it to be marked unavailable. A *SYSTEM ERROR* is reported and *)MORE* messages are queued indicating that the environment and all external routines that depend on it are unavailable. Any subsequent attempt to execute one of these external routines results in a *VALENCE ERROR*. To clear this situation, each of these external routines must be erased from the workspace and cleared from the execution stack.

Caveat: External routines accessed through Processor 11 are expected to be tested, well behaved programs. In certain circumstances, if these routines access or destroy data beyond what is presented to them through their arguments, damage to the APL workspace or to APL's internal operation may occur. Care must also be taken to ensure that the information in the routine description is valid and accurate.

Processor 11 Routine Search Order Guidelines

This section discusses guidelines for the processor 11 routine search order.

External Function Names

In both CMS and TSO, external routine names must be valid APL names and are restricted to uppercase letters and numbers. The maximum length of an external name is 249 characters in CMS and 8 characters in TSO.

By using a surrogate name, the user can satisfy these restrictions and at the same time use any valid APL name for the external routine in his applications. Use of surrogate names also helps to avoid name conflicts in the APL workspace.

CMS Search Order Guidelines

Under CMS, Processor 11 supports routines that:

- Reside in members of load libraries
- Have been link-edited with APL2
- Reside in MODULE files created with GENMOD
- Reside in TEXT decks
- Reside in TXTLIBs

Routines that reside in a member of a load library may be loaded by specifying the library and member names in the left argument of ⎕*NA* or in :LOAD. and :MEMB. tags in a NAMES file.

All other routines may be loaded by simply specifying a member name in the left argument of ⎕*NA* or in a :MEMB. tag in a NAMES file. For routines in saved seg-

ments and nucleus extensions, the name of the nucleus extension should be specified as the member name. For routines link-edited with APL2, the name of the original object file should be specified as the member name. In all these cases, Processor 11 uses the following search order:

1. Nucleus Extensions
2. MODULE files
3. TEXT files
4. TXTLIBs

While the convenience of object files or TXTLIB's may seem attractive, it is suggested that the user avoid the use of these facilities since they may be damaged by the use of other CMS facilities normally available to the APL2 user. The CMS loader that processes object files or TXTLIB members places the code and associated control blocks in vulnerable storage. This storage may be destroyed inadvertently by a variety of CMS commands and services.

A discussion of the advantages of placing routines in saved segments can be found in “Preloading and Sharing External Routines” on page 331. Instructions on how to install routines in saved segments can be found in *APL2/370 Installation and Customization under CMS*.

Using Routines Defined as Nucleus Extensions

To use a routine that is defined as a nucleus extension simply supply the nucleus extension name either as a member name in the left argument of $\square NA$ if the routine is self-describing or use the :MEMB. tag in a NAMES file. To prevent Processor 11 from trying to load the routine from a LOADLIB, do not code a load library name. Further information about defining routines as nucleus extensions can be found in “Preloading and Sharing External Routines” on page 331.

Using Routines in TEXT Decks or TXTLIBs

To use an external routine that is defined as a text deck simply supply the TEXT deck name in the :MEMB. tag in a NAMES file. To prevent Processor 11 from trying to load the routine from a LOADLIB, do not code a load library name. External routines in TEXT decks must be described using a NAMES file since they cannot be self-describing. Before using a routine in a TXTLIB, a `GLOBAL TXTLIB` *name* command must be issued, where *name* is the file name of the TXTLIB file.

TSO Search Order Guidelines

Under MVS/TSO, Processor 11 supports routines that have been link-edited and:

- Placed in a load library, or
- Placed in the MVS Link Pack Area (LPA)

Using Routines in TSO Load Libraries

A routine can be accessed from a load library either by placing the load library in the standard operating system search order or by allocating the library to a ddname and specifying the ddname in the left argument of $\square NA$ or the :LOAD. tag. The member name is coded in the left argument of $\square NA$ or the :MEMB. tag of a NAMES file entry.

Using Routines in the Standard TSO Search Order

If a routine is to be loaded using the standard TSO search order, only the member name should be coded in the left argument of ⎕*NA* or the :MEMB. tag of a NAMES file entry. No ddname should be provided. The TSO search order includes any data sets allocated to ddname LOADLIB either prior to APL2 invocation or during APL2 invocation by use of the LOADLIB invocation option. The search then proceeds to the normal TSO search order, including the current tasklib (if one has been established), STEPLIB, JOBLIB, and the LPA.

Link-Editing External Routines

Language compilers, and the APL2 *BUILDRD*, *BUILDRL*, and *PACKAGE* functions, produce object files that are link-edited before execution. Applications may be composed of a single routine in a single object file or may be composed of multiple object files and language routines. The linkage editor resolves references to external routines and combines the application's files with any necessary language routines.

Many languages provide libraries of routines that are combined with user programs during link-editing. These routines provide common services or are stub routines that dynamically load common service routines from language execution routine libraries. These libraries need to be made available to the linkage editor so that the referenced routines can be loaded during link-edit. Languages routines to be accessed during link-editing are usually provided in object files (TXTLIBs) on CMS or load libraries (partitioned data sets) on TSO.

APL2 provides three routines used with Processor 11 that may be loaded during link-editing. They are AP2VNL (for CMS), AP2TNL (for TSO), and AP2XCMAP. They are shipped to CMS in TEXT files (not a TXTLIB) and to TSO in a load library.

The linkage editors acquire input from three sources:

1. The main input object file
2. Object files specifically included by linkage editor INCLUDE control cards (statements.) An INCLUDE card can specify that a routine should be loaded from a library defined with a specific ddname.
3. Automatic inclusion of routines indicated by external references in routines included by the first two methods. The linkage editor searches the library(s) defined to the ddname SYSLIB.

The linkage editors assume that routines are only going to use 24 bit addresses. If a routine is going to use 31 bit addresses, or be called in 31 bit addressing mode by other programs, the routine should be link-edited with the AMODE(31) linkage editor parameter.

The linkage editors also assume that routines are designed to be loaded within the 24 bit address range (below the 16MB line.) If a routine can be loaded and run above the 16MB line, the routine should be link-edited with the linkage editor parameter RMODE(ANY) option. If a routine must be loaded and run above the 16MB line, the routine should be link-edited with the linkage editor parameter RMODE(31) option. Further information about addressing and residency modes can be found in “Extended Addressing Considerations” on page 331.

When link-editing a routine list with a set of object files, the routine list should be designated as the main entry point of the output file. When link-editing a routine description with a routine (and no routine list is included), the routine description should be designated as the main entry point of the output file.

Link-Edit Tools

Two execs, **AP2MP11L** and **AP2MP11M**, are provided to programmers for link-editing their routines.

AP2MP11L Link-edits object files and produces a member of a load library. It can be used on either CMS or TSO.

AP2MP11M Generates a MODULE file from object files. It can be used only on CMS.

Both execs' arguments have the same general form:

```
AP2MP11L target sources libraries
```

Where:

target is the file into which the generated output should be placed.

sources are the object files to be link-edited together.

libraries are language object file libraries.

Target Files: On TSO, *target* is a fully-qualified unquoted data set name including a parenthesized member name. On CMS, *target* is the filename of the LOADLIB or MODULE file to be generated. The member name to be built within the LOADLIB can be specified in parentheses or defaults to the first name listed in *sources*.

Source Files: On TSO, *sources* is a fully-qualified unquoted data set name including a parenthesized list of one or more members to be included in the output file. On CMS, *sources* is a parenthesized list of filenames of TEXT files to be included in the output file. If only one filename is listed, the parentheses may be omitted.

Library Files: On TSO, *libraries* is a parenthesized list of the fully-qualified unquoted names of language routine library data sets that should be allocated to SYSLIB. On CMS, *libraries* is a parenthesized list of filenames of language routine library TXTLIB files to be made available to the link-editor's or loader's search order. If only one name is listed, the parentheses may be omitted.

If no language routine libraries are needed, *libraries* may be omitted.

Using AP2MP11L and AP2MP11M

These execs can be used to link-edit:

- One object file produced by *PACKAGE* or a compiler. For example on CMS:

```
AP2MP11L MYLIB ADD (EDCBASE IBMBASE)
```

- One routine description and one non-APL routine object file. The name of the routine description should be listed first in *sources*. For example on TSO:

```
AP2MP11L USERID.LIB(PGM) USERID.OBJ(RD PGM) SYS1.LINKLIB
```

- One routine list and one or more routine descriptions and object files produced *PACKAGE* or compilers. The name of the routine list should be listed first in *sources*. For example on CMS:

```
AP2MP11M CMODULE (RL ENV RTN AP2VNL) (EDCBASE IBMBASE)
```

Note: On CMS AP2VNL and AP2XCMAP may also need to be listed in *sources*.

Examples demonstrating link-editing routines written in several languages using AP2MP11L and AP2MP11M can be found in Appendix G, “Sample Non-APL Programs to be Called through Processor 11” on page 385. For those readers who prefer not to use these execs or are interested in the mechanics of the process, the following sections provide further information about link-editing.

Link-Editing External Routines on CMS

The basic command for link-editing routines on CMS is:

```
LKED routine (LIBE library LIST MAP [AMODE(31) RMODE(ANY)]
```

The basic commands for creating a MODULE file on CMS are:

```
LOAD Module (RLDSAVE LIBE CLEAR NODUP AUTO RESET Rlname
GENMOD Module (FROM Rlname [AMODE(31) RMODE(ANY)]
```

The AP2MP11L and AP2MP11M execs issue the necessary GLOBAL and FILEDEF commands prior to performing the link-edit or module generation.

When multiple TEXT deck files are to be link-edited or loaded in preparation for generating a module, two basic techniques are available for causing the linkage editor or loader to find all the files.

1. They can all be copied into the same file that is then used as the main input to the linkage editor.
2. They can be copied into members of a TXTLIB file that is then added to the linkage editor's search order with the FILEDEF command.

When language routines to be automatically included reside in multiple object file libraries (TXTLIB files), two techniques are available for causing the linkage editor to be able to access all the libraries.

1. Copy all the members of all the libraries into a single library.

2. FILEDEF all the libraries to the ddname SYSLIB. Unfortunately, the CMS linkage editor has a restriction that it only supports one library associated with SYSLIB unless the following extra step is also performed:

   MACLIB files with the same filenames as each of the TXTLIB files must be available on an accessed disk and a GLOBAL MACLIB command must be issued listing all the names of the TXTLIB files.

When a MODULE file is to be generated and the language routines reside in multiple object file libraries (TXTLIB files), a GLOBAL MACLIB command must be issued listing all the names of the TXTLIB files.

Unlike the language routines provided with most compilers, APL2's routines are not shipped in TXTLIB files. AP2VNL and AP2XCMAP are shipped in separate object files. Therefore, since the linkage editor and loader only automatically loads from TXTLIB files filedefed to SYSLIB, they must be explicitly included

The second argument of the AP2MP11L and AP2MP11M execs lists the TEXT files to be explicitly included. The execs include TEXT files. The filetype of the AP2VNL and AP2XCMAP files is TXT210 in the initial shipment of APL2 Version 2. If maintenance is applied to these files, the filetype changes to TEXT. Before attempting to include either of these files, you should locate the most recent copy. If the filetype is TEXT, simply have the disk accessed, If the filetype is TXT210, make a copy of it with a filetype of TEXT so that the execs can find it.

Some languages require that libraries of dynamically loaded execution environment routines are available. For further information about this topic, consult “Execution Time Libraries” on page 332.

Link-Editing External Routines on TSO

The basic command for link-editing routines on TSO is:

```
LINK input LOAD(outlib(member)) LIB(langlib) LIST MAP [AMODE(31) RMODE(ANY)]
```

When multiple object files are to be link-edited, two basic techniques are available for causing the linkage editor to find all the files.

1. They can all be copied into the same file that is then used as the main input to the linkage editor.
2. They can be specifically INCLUDEd from object file libraries either allocated to a specific ddname or from SYSLIB (the default DDNAME.)

Libraries containing language routines that are automatically included should be allocated to ddname SYSLIB. Multiple libraries can be concatenated to SYSLIB.

Because AP2TNL and AP2XCMAP are shipped in a load library, the library can be concatenated to SYSLIB along with other language routine libraries. The linkage editor automatically includes them (if there are external references to them) and they do not need to be explicitly included.

Some languages require that libraries of dynamically loaded execution environment routines are available. For further information about this topic, consult “Execution Time Libraries” on page 332.

Installation of External Routines

This section discusses the installation of external routines.

Extended Addressing Considerations

On systems supporting extended addressing, unless APL2 is invoked with XA(24), arguments passed to external routines are located above the 16-megabyte line. External routines are entered in the addressing mode specified when they were link-edited.

To execute routines with 24-bit addressing dependencies, the routines must be link-edited with the linkage editor parameters AMODE(24) RMODE(24) and APL2 must be invoked with the invocation option XA(24). This causes the external routine and the APL workspace to be located below the 16-megabyte line, the external routine to be entered in 24-bit mode, and arguments passed to it located below the 16-megabyte line.

Invoking APL2 with option XA(24) restricts the size of the workspace and local shared memory.

Preloading and Sharing External Routines

It may be desirable to preload frequently used routines or make them available on a shared basis so every user does not incur the overhead of loading an individual copy.

This can be done in the MVS/TSO environment by placing the routines, with the help of your system administrator, in MVS Link Pack Area (LPA). The manual *APL2/370 Installation and Customization under TSO* includes more information on this topic.

In the VM/CMS environment, external routines can be placed in a saved segment or be link-edited and installed with APL2 itself. When routines are placed in a saved segment, they are shared between users. If they are link-edited with APL2, they are loaded with APL2 and shared between users if APL2 is installed on a shared basis. The manual *icms* includes more information on this topic.

Routines that are shared between users, either by placing them in the MVS Link Pack Area or by link-editing them as part of APL2 itself, must be reentrant and refreshable. Routines written in languages other than APL2 may or may not be reentrant and refreshable depending upon how they were written. APL2 namespaces are reentrant and refreshable.

In the VM/CMS environment, it is possible to preload an external routine on a non-shared basis by loading it as a nucleus extension. This can be done by issuing a CMS NUCXLOAD command for the routine or package of routines before attempting to associate the name. More information on the NUCXLOAD command can be found in the *VM/SP CMS Command and Macro Reference*.

Execution Time Libraries

FORTRAN and similar languages often require access to execution time libraries to access processing, service, and error handling routines that are not compiled or link-edited with the program. Typically, access to these routines is provided through the environment and not directly from an individual subroutine. If you are not sure if your external routine requires access to execution time library routines, it is safer to establish an environment and make the execution time library available to it.

VS FORTRAN Execution Time Libraries

The VS FORTRAN execution time library in the CMS/VM environment must be on an accessible minidisk and made available to the APL session with a GLOBAL LOADLIB CMS command.

In the MVS/TSO environment, the libraries should be allocated to a ddname and the ddname must specified in the :TASKLIB. tag in the environment program's routine description.

If the execution time library is not available when it is required during the execution of an external function, a *SYSTEM ERROR* message typically occurs along with termination of the environment.

Other Processor 11 Considerations

This section discusses processor 11 considerations.

Using Self-Describing Routines from Non-APL Programs

The addition of a routine description to a non-APL program does not affect the ability to use the program from outside APL2. As long as the program is called using standard operating system entry and exit linkage conventions, the self-description does not interfere.

Using Modules with Routine Lists from Non-APL Programs.

The addition of a routine list to a module makes the main entry point of the module unusable from non-APL programs. It is possible to set up aliases for routines within the module using the linkage editor. This would enable those routines to be called by non-APL programs.

FORTRAN Considerations

This section discusses the considerations for FORTRAN.

APL2 versus FORTRAN Array Ordering

In APL, arrays are stored and referred to in row major order. In FORTRAN they are in column major order. Processor 11 does not modify data ordering. Among other things, this means that the APL statement:

```
ARRAY[I;J]
```

selects the item at the row I and column J of ARRAY. In FORTRAN, the statement:

```
ARRAY(I,J)
```

selects the item at the column I and row J of ARRAY. Applications that share data between APL and FORTRAN routines must be aware of this difference.

FORTRAN External Names

VS FORTRAN supports external names up to 7 characters long.

FORTRAN Linkage Convention

:LINK.FORTRAN should be coded for VS FORTRAN subroutines. Some VS FORTRAN subroutines may work even if a :LINK.OBJECT tag is coded. However, results are unpredictable. A general guideline is that :LINK.FORTRAN must be used if any of the arguments of the subroutine is a character vector.

FORTRAN Common

FORTRAN routines that share data using static named or blank common must be link-edited together if they are to be used with Processor 11. Under MVS, link-edited load modules containing routines that share common storage should be reusable. See “Routine Lists” on page 298 for additional information. FORTRAN routines that share data using dynamic common require the FORTRAN environment (see “Environments” on page 303), but may be link-edited to separate modules.

FORTRAN Functions

FORTRAN functions can be called through processor 11 and their explicit results returned. For FORTRAN functions of type other than CHARACTER, the :RSLT. tag must be coded to describe the explicit result (see “Result Patterns” for details). FORTRAN functions of type other than CHARACTER return only scalar results and pass those results back to the caller in register 0 or in floating point registers. Thus, the :RSLT. pattern should not include the output indicator (<).

FORTRAN functions of type character return their results in a dummy extra argument in the caller's parameter list. Thus to call the following FORTRAN function:

```
CHARACTER*10 FUNCTION SUFFIX(STR)
CHARACTER*7 STR
SUFFIX = STR // 'SUF'
END
```

The following :RARG. pattern should be coded in the routine's description:

```
:RARG.(G0 1 2) (C1 1 7) (<C1 1 10)
```

and the function called from APL with an extra dummy argument:

```
        '(MYNAMES)' 11 ⎕NA 'SUFFIX'
1
        SUFFIX (7↑WORD) (10⍴' ')
```

Note: For FORTRAN functions of type CHARACTER the :RSLT. tag should not be coded in Routine Descriptions.

Chapter 29. Processor 11—Access to Namespaces

Processor 11 provides facilities that allow access to APL objects in namespaces and to routines written in languages other than APL. This chapter describes access to APL objects in namespaces. Access to FORTRAN and Assembler Language routines is described in Chapter 28, “Processor 11—Calling Compiled Programs” on page 291.

Overview

Facilities are provided with APL2 that allow a saved workspace to be encapsulated or “packaged” and converted to an object file. Such object files can be processed by a system provided linkage editor and subsequently accessed through Processor 11. Access to the APL objects (arrays, functions and operators) in a namespace is provided through the use of ⎕*NA*.

To convert a saved workspace to a namespace, the saved workspace must be processed by the external function *PACKAGE*. In most cases, it must then be link-edited into a load library.

The objects (arrays, functions or operators) in the namespace that are to be accessed from the active workspace or from other namespaces may be described in a NAMES file available to Processor 11. Alternatively, information used to locate the namespace can be provided in the left argument of dyadic ⎕*NA*.

APL applications access objects in the namespace through the use of the system function ⎕*NA*. Once external objects are declared and activated through the use of dyadic ⎕*NA*, they can be treated as normal arrays, functions or operators in the user's workspace.

The following example illustrates this process:

1. The user develops an APL application and saves it in his private library as workspace REPORT with the command:

   ```
   )SAVE REPORT
   ```

   Assume that the workspace contains two functions functions *SETUP* and *RUN* that are designed to be called directly and a number of subsidiary functions, operators and arrays that are used by the *SETUP* and *RUN* functions.

For purposes of illustration, assume the following simplistic definitions for *SETUP* and *RUN*:

```
      ∇Z←SETUP A
[1]   INITIALIZE A    ⍝ CALL INITIALIZATION FUNCTION
[2]   Z←'SETUP COMPLETE'
[3]   ∇

      ∇Z←RUN A
[1]   PROCESS A       ⍝ CALL PROCESS FUNCTION
[2]   Z←'RUN COMPLETE'
[3]   ∇
```

2. In the MVS/TSO environment, a data set must be allocated to ddname SYSPUNCH before the next step is performed. The output of the *PACKAGE* function in the next step is placed into this data set. Assuming that the PDS REPLIB.OBJ exists and has been cataloged, and that a new member REPORT is created in step 3, this allocation can be performed with the TSO command:

   ```
   ALLOCATE FILE(SYSPUNCH) DA(TEMP.OBJ) SP(10,20) BL(8000) LR(80) REC(F,B)
   ```

3. The saved workspace is converted to a namespace using the external function *PACKAGE* that is provided with APL2:

   ```
         3 11 ⎕NA 'PACKAGE'
   1
         'SETUP' 'RUN' PACKAGE 'REPORT'
   ```

   The external function *PACKAGE* converts the workspace REPORT to an object file. Under CMS/VM, the object file is placed in a CMS file named 'REPORT TEXT'. Under MVS/TSO, the object file is saved in the data set allocated with ddname SYSPUNCH (in our example, as member REPORT in data set REPLIB.OBJ).

   Note: Under MVS/TSO, SYSPUNCH must be allocated prior to running the external function *PACKAGE*.

4. The object deck produced in step 3 is then link-edited into a load library. Under VM/CMS, this can be accomplished with the CMS commands:

   ```
   FILEDEF SYSLMOD DISK REPLIB LOADLIB A (RECFM U
   LKED REPORT (NAME REPORT
   ```

   Under MVS/TSO, assuming that the PDS REPLIB.LOAD has been previously created and catalogued, the link-edit can be accomplished with the TSO command:

   ```
   LINK (TEMP.OBJ) LOAD(REPLIB(REPORT)) RMODE(ANY)
   ```

   In non-MVS/XA environments, the RMODE(ANY) parameter shown on the LINK command above may not be supported and should not be specified.

5. Under the MVS/TSO, the load library in which the link-edited module resides must be allocated using the ddname specified in the :load. tag in the NAMES file in the left argument of ⎕*NA*. This can be accomplished with the TSO command.

   ```
   ALLOCATE FILE(REPLIB) DSN(REPLIB.LOAD) SHR
   ```

6. The *SETUP* and *RUN* functions may then be accessed through as normal APL functions through the use of ⎕*NA*:

```
      'REPLIB.REPORT' 11 ⎕NA 'SETUP'
1
      SETUP 'INITIAL TEST'
SETUP COMPLETE

      'REPLIB.REPORT' 11 ⎕NA 'RUN'
1
      RUN 'INITIAL TEST'
RUN COMPLETE
```

Detailed Description

Namespaces can be created from APL saved workspaces. Objects within them can be accessed dynamically from the user's active workspace or from other namespaces.

Namespaces are created from saved workspaces using the external function *PACKAGE* that is described below. The output of the *PACKAGE* function is an object file that is then usually processed by a linkage editor to create a load module containing the namespace. That load module can be placed in a load library. Under MVS, a system administrator can arrange to have the load module placed in the MVS link pack area (LPA), thus making it able to be shared by more than one APL user. Under VM, a system administrator can arrange to have the load module link-edited with APL2 or placed in a saved segment, thus making it able to be shared by more than one APL user.

Entries in a NAMES file associated with Processor 11 define the names in the namespace that can be accessed through the use of ⎕*NA* and provide information on where the namespace can be found by Processor 11. Alternatively, the information necessary to locate the namespace can be provided in the left argument of ⎕*NA*.

A namespace that is not in the LPA or a saved segment is loaded by Processor 11 into the free space in the APL user's address space or virtual machine when names within the namespace are activated through the use of ⎕*NA*.

Once a name has been declared, through the use of ⎕*NA*, to be in a namespace, that name remains associated with the namespace until it is explicitly deleted (with)*ERASE*, ⎕*EX*, etc.). The association is retained in the workspace where the ⎕*NA* was issued even after that workspace is)*SAVE*d, and later)*LOAD*ed, or)*COPY*ed. The information required to form an association is produced by)*OUT* for use by)*IN*.

If a saved workspace that contains names associated with one or more namespaces is loaded, the namespaces are not loaded into the user's address space or virtual machine until the associated names are first encountered during

the execution of APL expressions, or until they are specifically reactivated through the use of ⎕*NA*.

If the namespace is no longer available when the workspace is reloaded, attempts to access objects that were previously declared through the use of ⎕*NA* fail.

Each namespace contains its own namescope. That is to say, it contains a set of APL arrays, functions and operators that are known within that package. The functions and operators in a namespace can call other functions and operators or access arrays within the same namescope.

Each namespace contains its own copy of the system variables, except ⎕*NLT*, ⎕*PW*, and ⎕*TZ* that are session variables and have only a single value in the user's session. System variables such as ⎕*IO*, ⎕*PP*, ⎕*RL*, etc. can have a different values in a namespace than they have in the user's active workspace.

To access a name in another namescope, that name must be declared and linked to the alternate namescope through the use of ⎕*NA*. ⎕*NA* allows an APL object (array, function or operator) that exists in a namespace to be known and used from the user's active workspace or from another namespace. Once a name in a namespace has been activated in the user's active workspace or in another namespace through the use of ⎕*NA*, it appears and can be used as a normal APL function, operator or array.

Functions or defined operators in the namespace do not consume space in the user's workspace unless their definitions are modified during execution in the package's namescope. If a function or defined operator's definition is so modified or if new functions or operators are dynamically created in the package's namescope, those new definitions consume space in the user's workspace. Any variables in a namespace that are referenced or specified during execution in the package's namescope consume space in the user's workspace.

In addition, on the first ⎕*NA* to a package, a copy of the name table in the package is made in the active workspace. Any changes to objects in the package are recorded in this copy of the name table and such changes are local to the user who issued the ⎕*NA*.

Variables in a namespace that have been referenced or specified during execution in the package's namescope, or defined functions and operators whose definition has been modified or created during execution in the package's namescope, are saved along with the user's workspace when a)*SAVE* command is issued. When a)*SAVE* command is issued, sufficient information to re-access namespaces referenced by the workspace is also saved along with the user's active workspace. The namespace itself is not saved.

If, after a)*LOAD* command, it is found that the namespace has been modified (that is, recreated from a workspace with a different SAVED date) since its last use, a warning message is produced and the user is given access to the objects in the new namespace. Data or objects in the package that were created or modified through previous use of the package, and saved when the user's workspace was saved, are lost.

Creating Namespaces

Namespaces are created from saved workspaces using the external function *PACKAGE* that is provided with APL2. Saved workspaces to be converted to namespaces must have been saved using the current release of APL2. Workspaces saved under prior releases of APL2 must be reloaded and saved again under the current release of APL2 before they can be successfully converted. Under MVS/TSO, saved workspaces to be converted must exist in a SAM library (see “Sequential Access Method (SAM) Library System” on page 70); workspaces saved in VSAM libraries must be saved again in a SAM library before they can be successfully processed.

It is recommended that workspaces be)*COPY*ed and re-)*SAVE*d before converting them to namespaces. This technique causes the workspace to be compacted and the state indicator to be cleared before the workspace is packaged.

The function, *PACKAGE*, can be accessed through the use of ⎕*NA*:

```
      3 11 ⎕NA 'PACKAGE'
1
```

This function is ambivalent and expects the following syntax:

```
RESULT←NAME_LIST PACKAGE WS_NAME
```

where

WS_NAME is the data set name of the saved workspace to be converted. See “Workspace Names” below for additional information. Under VM/CMS, the data set must exist on an accessible minidisk. Under MVS/TSO, it must be a cataloged sequential data set.

NAME_LIST is a list of names of APL objects in the resulting namespace that is accessible (through the use of ⎕*NA*) from the user's active workspace or from other namespaces. APL objects in the namespace that are not specified in this name list cannot be accessed from outside the namespace. If the *NAME_LIST* argument to *PACKAGE* is not specified, all APL objects in the resulting namespace are accessible through the use of ⎕*NA*. *NAME_LIST* can be a simple character scalar or vector representing one name, or can be a matrix or vector or vectors representing a list of names.

RESULT is the name of the data set containing the resulting namespace. In CMS/VM, this is a file with name 'fn TEXT A' where 'fn' is the file name of the workspace data set name provided in the right argument to *PACKAGE*. In MVS/TSO, it is the name of the data set allocated to ddname SYSPUNCH. If SYSPUNCH is not allocated when *PACKAGE* is executed, an error message is produced and the function terminated.

If the *PACKAGE* function is not successful in converting the saved workspace, *RESULT* is returned as an empty vector.

Once a saved workspace has been converted to a namespace, the resulting data set should be link-edited into a load library. Under VM/CMS, this can be accomplished with the CMS commands:

```
FILEDEF SYSLMOD CLEAR
FILEDEF SYSLMOD DISK fn2 LOADLIB A (RECFM U
LKED fn1 (NAME fn1
```

where 'fn1' is the file name of the namespace returned by the *PACKAGE* function and 'fn2' is the file name of the load library into which the link-edited namespace is to be placed.

Under MVS/TSO, the link-edit can be accomplished with the TSO command:

```
LINK dsn1 LOAD(dsn2(memb)) RMODE(any)
```

where 'dsn1' is the name of the data set returned by the *PACKAGE* function. 'dsn2(memb)' is the data set name and member name of the load library into which the link-edited namespace is to be placed. In non-MVS/XA environments, the RMODE(ANY) parameter on the LINK command shown above may not be supported and should not be specified.

The following publications provide additional information and reference material on the linkage editors:

- *OS/VS Linkage Editor and Loader*, GC26-3813
- *MVS/Extended Architecture Linkage Editor and User's Guide*, GC26-4011
- *Virtual Machine/System Product: CMS Command and Macro Reference*, SC19-6209

A link-edited namespace can be accessed by Processor 11 from the load library in which it exists. Alternatively, in the MVS environment, it can be placed in the link pack area and thus shared between all users. Contact your system administrator for assistance in placing the namespace load module in the MVS link pack area or extended link pack area.

In the CMS/VM environment, a namespace can be made able to be shared by installing it in a saved segment. Contact your system administrator for assistance in this area. The procedure is described in *APL2/370 Installation and Customization under CMS*.

Alternatively, in the VM/CMS environment, a namespace can be explicitly loaded as a CMS nucleus extension with the CMS commands:

```
FILEDEF SYSLIB CLEAR
FILEDEF SYSLIB DISK fn LOADLIB *
NUCXLOAD memb memb SYSLIB
```

where 'fn' is the file name of the CMS load library in which the link-edited namespace resides and 'memb' is its member name in that load library.

Loading a namespace as a CMS nucleus extension in the VM/CMS environment allows it to be preloaded and remain loaded across)*LOAD*s and)*CLEAR*s. The namespace must be explicitly deleted by the user in this situation. This can be accomplished with the CMS NUCXDROP command.

Workspace Names

The external function *PACKAGE* requires as a right argument the data set name of the saved workspace that is to be converted. Under VM/CMS, the data set names of saved workspaces take the form 'FN FT FM' where:

FN is the name of the saved workspace, used in a)*LOAD* or)*SAVE* command

FT is APLWSV2 for private workspaces and V*nnnnnnn* for public workspaces where *nnnnnnn* is the library number of the public workspace.

FM is the CMS filemode of the minidisk on which the saved workspace resides.

If FT or FM are not specified, they are defaulted to 'APLWSV2' and '*' respectively.

Under MVS/TSO, workspaces can be saved in SAM or VSAM libraries. (See "APL2 Libraries, Workspaces, and Data Files Under TSO" on page 67.) The *PACKAGE* function supports only workspaces saved in SAM libraries. Workspaces saved in VSAM libraries must be saved again in a SAM library before they can be processed by *PACKAGE*.

Under MVS/TSO, the data set name for a saved workspace can be obtained using AP 100 by issuing the command:

)*HOST APL WSNAME* *wsid*

(See the APL WSNAME command on page 125 for additional details.) Note that the *PACKAGE* function requires the workspace data set name as its right argument. On TSO, if an unqualified workspace data set name is used in the right argument of the *PACKAGE* function, the *PACKAGE* function issues the equivalent of a)*HOST APL WSNAME* command to determine its qualified name.

Accessing Objects in Namespaces

APL objects (arrays, defined functions and operators) in a namespace are accessed from the user's active workspace or from other namespaces through the use of dyadic ⎕*NA*. The right argument of ⎕*NA* specifies the name of the object or the name and surrogate name of the object.

The left argument of ⎕*NA* can be a 2 element vector or a two column array (with each row corresponding to a row in the right argument). The first item of the left argument (or of each row of the left argument) is used to specify the name and location of the namespace or to direct Processor 11 to a NAMES file in which the name and location of the namespace may be found. The following options are valid:

'*LIB.MEMBER*' 11 ⎕*NA* '*ROUTINE*'
: Specifies that the object ROUTINE is located in the namespace that is stored as member *MEMBER* in load library LIB. In MVS/TSO, LIB is the ddname allocated to the data set in which MEMBER resides. In CMS/VM, the load library is named LIB LOADLIB *.

'*MEMBER*' 11 ⎕*NA* '*ROUTINE*'
: Specifies that the object *ROUTINE* is located in the namespace named *MEMBER*. In MVS/TSO, *MEMBER* is found using standard OS search

order (that is, LPA, JPA, STEPLIB, etc.). In CMS/VM, a CMS nucleus extension name MEMBER is used if it exists, otherwise, Processor 11 attempts to load a TEXT deck named MEMBER TEXT * or a module named MEMBER MODULE *.

'(NAMEFILE)' 11 ⎕NA 'ROUTINE'
Specifies that the object *ROUTINE* is to be located by searching the NAMES file allocated to ddname NAMEFILE in MVS/TSO or with name NAMEFILE NAMES * in VM/CMS. See "NAMES Files" on page 342 for additional information.

NAME_CLASS 11 ⎕NA 'ROUTINE'
Where *NAME_CLASS* is an integer scalar from the set 1, 2, 3, or 4. In this case, Processor 11 searches its default names files for information to locate the namespace in which the object ROUTINE resides and ensures that the name class of ROUTINE matches that specified.

0 11 ⎕NA 'ROUTINE'
Specifies that the object *ROUTINE* is to be located by searching the default Processor 11 NAMES files. See "NAMES Files" on page 342 for additional information.

'' 11 ⎕NA 'ROUTINE'
Specifies that the object *ROUTINE* is to be located in the user's active workspace and not in a namespace. This variant of *⎕NA* can only be successfully issued from a namespace. Attempts to issue it from the user's active workspace fail and return 0.

If the first element of the left argument of *⎕NA* is invalid, the request to associate the name is rejected and 0 is returned as the result of *⎕NA*. A *)MORE* message is also queued indicating PARAMETER ERROR.

If the first element of the left argument of *⎕NA* specifies a name class that does not match the name class of the specified object in the namespace, the request is rejected and 0 is returned as the result of *⎕NA*. A *)MORE* message is queued indicating that the requested name is NOT AVAILABLE.

If an object with the same name as that specified in the right argument of *⎕NA* already exists in the workspace when *⎕NA* is issued, (perhaps due to an earlier *⎕NA*), the left argument of *⎕NA* must match the result of monadic *⎕NA* for the same name or *⎕NA* fails.

Once an object in a namespace has been successfully activated through the use of *⎕NA*, it can be used in APL expressions, just like any other names in the APL workspace.

```
      3 11 ⎕NA 'FUNCTION'
1
      2 11 ⎕NA 'VARIABLE'
1
      RESULT←VARIABLE+FUNCTION ⍳4
```

NAMES Files

The location of the namespace in which an object resides can be specified in the left argument of ⎕*NA* as described above. Alternatively, that information can be placed in a NAMES file available to Processor 11. If the file name (VM/CMS) or ddname (MVS/TSO) of the NAMES file is not specified in the left argument of ⎕*NA*, default NAMES files are used. To avoid conflicts with names defined in the default NAMES files, it is recommended that private NAMES files or alternate forms of ⎕*NA* should be used in applications that include namespaces.

In the CMS/VM environment, one NAMES file called AP2VN011 NAMES is provided with APL2 and contains descriptive information on namespaces and external functions provided with APL2. A second NAMES file called 'P011 NAMES' may be defined to contain descriptions of objects in namespaces and external functions. When searching for a specific name in this file, if the name is not found there, AP2VN011 NAMES is searched.

In the MVS/TSO environment, the NAMES file may contain fixed or variable length records with a maximum record length of 255 bytes. To be available to Processor 11, the data set must be allocated with the ddname specified in the left argument of ⎕*NA* or with ddname AP2TN011 if the default NAMES file library is to be used. Concatenated allocation can be used to specify search order if more than one NAMES file is desired. One NAMES file, distributed with the name APL2.NAMES.AP2TN011, is provided with APL2 and contains descriptions of external functions provided with APL2.

The format of data lines in the NAMES file is as follows:

```
:TAG.value
```

where ':TAG.' is chosen from a set of keywords and identifies the meaning of '*value*'. Tags and their values can be coded in either upper, lower, or mixed case letters. The following tags are valid in the description of an APL object in a namespace:

:NICK.*name*

Specifies the name of the APL object as specified in the right argument of ⎕*NA*. If a surrogate name is specified in the right argument of ⎕*NA*, it is that surrogate name that must match the :nick.*name*.

This tag is used to create a link between the name (or surrogate name) specified with ⎕*NA* and the descriptive information that follows. In MVS/TSO, this tag is optional since the partitioned data set member name provides the same function. In VM/CMS, this tag is required and must immediately precede the other tags that describe the object. In both MVS/TSO and VM/CMS, the name is restricted to uppercase letters and numerics. The maximum length of the name is 249 characters in VM/CMS and 8 characters in MVS/TSO.

:LOAD.*library*

The name (VM/CMS) or ddname (MVS/TSO) of the load library into which the namespace has been link-edited. In CMS/VM, the library name is of the form FN FT FM. FT and FM default to LOADLIB *. On TSO, the load library data set must have been previously allocated using the specified ddname. If the :LOAD. tag is not specified in MVS/TSO, standard OS search conventions are used to attempt to locate the namespace. If

the :LOAD. tag is not specified in VM/CMS, Processor 11 attempts to first locate a CMS nucleus extension, and failing that, a TEXT deck or module file whose name matches the :MEMB. tag.

:MEMB.*name* The member name of the namespace. If the :LOAD. tag is not specified, it is the name of a previously loaded module in MVS/TSO or the name of a CMS nucleus extension, TEXT deck, or module file in CMS/VM.

:ENTRY.*name* The name of the object to be accessed in the namespace or, if the member contains a routine list, the name Processor 11 should search for in the routine list.

Load modules containing more than one namespace and non-APL routine must have the accessible object, namespace, and non-APL routine names listed in a routine list. The *BUILDRL* function can be used to build an appropriate routine list. See "Combining Several Namespaces in a Member" on page 347 for details.

:LINK.APL Specifies to Processor 11 that this set of descriptive information describes an APL object in a namespace.

:DESC.*description* Allows inclusion of descriptive text in the NAMES file. Where the descriptive text exceeds the NAMES file record length, multiple records of text may be included, but each must be prefixed by a :DESC. tag. Comments may also be included in the NAMES file by placing an asterisk in column 1 of the records.

No other tags may be coded in a NAMES file entry for an object in a namespace.

Using Namespaces

APL applications often consist of one or a small number of functions that are accessed or called by the user and a large number of sub-functions, sub-operators, and variables used by them. If such an application is converted to a namespace the following benefits may be achieved:

- The complexity of the overall structure of the application can be hidden. Only those functions that are designed to be used directly by the user need be made visible (through the use of □*NA*).
- Applications can be more easily combined together. Since only the name of APL objects declared with □*NA* are known in the user's active workspace, the probability of encountering name conflicts when more than one application is combined is significantly reduced. Further, since a surrogate name can be assigned with □*NA*, or through the use of the :ENTRY. tag, name conflicts are usually simply resolved.
- APL application code can be shared between users. Under MVS/TSO, namespaces can be placed in the MVS link pack area. Under CMS/VM, they can be placed in a saved segment. In either case, the APL application code in the namespace is shared between multiple simultaneous APL users who access it.

Objects in namespaces can be accessed from the user's active workspace and/or from other namespaces. Objects are accessed through the use of dyadic ⎕*NA*. Dyadic ⎕*NA* cannot be used from within a package to access objects in the same package. If objects in a namespace are accessed simultaneously from the user's active workspace and another namespace, or from two or more namespaces, only one copy of the namespace is loaded.

Namescopes

The active APL workspace and every namespace contains its own namescope; they may each contain a set of APL arrays, functions and operators that are known and may be referenced within that namescope.

While executing APL expressions, functions or operators, one namescope is active and is used in locating the definitions or values for APL names. In a)*CLEAR* workspace or a)*LOAD*ed workspace without suspended functions or operators, the primary namescope (that of the active workspace, rather than any namespace) is activated. In that state, references to local and global names are resolved in the primary namescope.

You can declare a name in a namespace, and thus in another namescope, with ⎕*NA*. When a name, declared with ⎕*NA*, is encountered during the execution of an APL expression, the system switches to the namescope in which the name's definition exists so that its value or definition and names (local or global) that it references are taken from the correct namescope. This change of namescope is said to be an explicit change, because the name was explicitly declared to be in another namescope through the use of ⎕*NA*.

If a defined function or operator is declared, through the use of ⎕*NA*, to exist in another namescope, the system switches to that namescope when the function or operator is executed and remains in that namescope until the function or operator completes or until it causes another namescope switch. While executing in its namescope, local and global names referenced by the function or operator are resolved from the same namescope. If the function or operator suspends during its execution, the user is left in that namescope and commands such as)*FNS*,)*VARS*, etc. report names in that namescope. The user can return to the primary namescope by abandoning execution with the command)*RESET*.

The external function *QNS*, provided with APL2, can be used to query the current namescope. It returns the left argument to ⎕*NA* of the function or operator used to enter the current namescope. For the primary namescope (the user's active workspace), it returns `'' 11`.

An implicit change of namescopes occurs when an operand to an external operator is referenced. For example, if an external operator *MOP* is declared through the use of ⎕*NA*, and then called with a defined function operand, *FN*, when and if the operator references its operand, an implicit switch of namescopes occurs. This change occurs to allow the function, *FN*, to operate correctly and refer to names in the namescope in which it is defined.

Certain applications, when implemented as namespaces, need to reach back into the namescope from which they were entered to retrieve or set values, or to execute system functions or defined functions or operators. An application in a namespace, for example, might wish to obtain the value for ⎕*PP* from the caller's namescope, or might want to use ⎕*CR* to obtain the canonical representation of a function in the caller's namescope. Such a “reach back” facility can be implemented in 3 ways:

1. The application in the namespace can be designed to be entered through a defined operator. This approach allows the caller to provide a functional operand that, when used, operates in the caller's namescope. For example, the following defined operator in a namespace returns the canonical representation of the *FUNCTION* from the caller's namescope.

```
      ∇RESULT←(FN OPERATOR) ARGUMENT
[1]   RESULT←FN ARGUMENT
[2]   ∇

      4 11 ⎕NA 'OPERATOR'
1
      ⎕CR OPERATOR 'FUNCTION'
```

   If called with:

```
      ⍎OPERATOR 'VARIABLE'
```

   it returns the value of *VARIABLE* from the caller's namescope.

2. The application in the namespace can be designed to be entered through a defined function or operator with an argument that identifies the caller's namescope. That argument can subsequently be used with ⎕*NA* to access names in the caller's namescope. For example, before calling the packaged application, the caller could issue:

```
      3 11 ⎕NA 'QNS'
1
      CURRENT←QNS 0
```

   to determine the left argument for ⎕*NA* that allows re-entry to the current namescope. Then that value can be passed to the packaged application:

```
      'PKGLOAD.PKGMEMB' 11 ⎕NA 'FUNCTION'
1
      FUNCTION CURRENT
```

   The packaged application then could use the value passed to it as a left argument to ⎕*NA* to access names in the previous namescope:

```
      CURRENT ⎕NA 'NL ⎕NL'
1
      NL 2 3 4
```

3. An external function, *EXP*, is provided with APL2 to allow packaged applications to access names in the previous namescope. The syntax for its use is as follows:

```
      3 11 ⎕NA 'EXP'
1
      RESULT←EXP ⊂'NAME'
```

   references a variable '*NAME*' or executes a niladic function '*NAME*' in the previous namescope.

```
      RESULT←EXP 'NAME' VALUE
```

   executes the monadic function '*NAME*' in the previous namescope with argument *VALUE* from the current namescope.

```
      RESULT←EXP VALUE1 'NAME' VALUE2
```

   executes the dyadic function '*NAME*' in the previous namescope with arguments *VALUE*1 and *VALUE*2 from the current namescope.

```
      RESULT←EXP 'NAME' '←' VALUE
```

   specifies the variable '*NAME*' in the previous namescope with the value *VALUE* from the current namescope.

   '*NAME*' can be the name of a defined function or operator, the name of a variable, or the name of a system function or variable. It cannot be the symbol for a primitive function or operator.

   The external function *EXP* causes an implicit switch of namescopes to the namescope that caused explicit entry into the current namescope. If the current namescope was entered implicitly (as a result of executing the operand to an external operator, or as the result of another *EXP* function), this namescope switch accesses the namescope that last issued an explicit call to the current namescope.

Combining Several Namespaces in a Member

It is possible to place more than one namespace in a single member of a load library (or a MODULE file on CMS.) To do this, you must create a routine list containing the names of the each of the objects to be accessed in each of the namespaces in the member. The routine list can be created using the *BUILDRL* function.

```
file BUILDRL rlname 'OBJa NSPx' 'OBJb NSPx' ...
```

Figure 105. Building a routine list for several namespaces

Where:

file is the name of the file into which the routine list should be written.

rlname is the name of the routine list to be generated.

OBJa, *OBJb*, **...**
are the names of the objects to be accessed in the namespaces *NSPx*, *NSPy*, ...

NSPx, *NSPy*, **...**
are the names of the namespace object files to be included in the member.

Link-edit the routine list object file with the object files produced by the *PACKAGE* function. A discussion of the link-edit process can be found in section “Link-Editing Namespaces” on page 349. Once the object files have been link-edited, they can be discarded.

CMS Namespace Routine List Example

Figure 106 demonstrates placing several namespaces into a CMS MODULE file and using them from APL2.

```
      3 11 ⎕NA 'PACKAGE'
1
      PACKAGE 'MATHFNS V0000001'
MATHFNS TEXT A
      PACKAGE 'UTILITY V0000001'
UTILITY TEXT A
      RL←⊂'APL2RL'
      RL←RL,'EIGEN MATHFNS' 'POLYZ MATHFNS'
      RL←RL,'LINEFOLD UTILITY' 'NAMES UTILITY'
      'APL2RL TEXT A' BUILDRL RL
0
      )HOST AP2MP11M APL2WS (APL2RL MATHFNS UTILITY)
CMS(0)
      'APL2WS' 11 ⎕NA ⊃'EIGEN' 'POLYZ' 'LINEFOLD' 'NAMES'
1 1 1 1
      EIGEN 2 2⍴1 0 0 2
1 2
1 0
0 1
      POLYZ 1 ¯6 11 ¯6
1 2 3
      15 LINEFOLD 'THIS IS SOME SAMPLE TEXT'
THIS IS SOME
SAMPLE TEXT
      NAMES '⎕FRED∘.×⎕JANE←⍳9'
FRED
JANE
```

Figure 106. Build a module containing several namespaces.

TSO Namespace Routine List Example

Figure 107 demonstrates placing several namespaces into a partitioned data set member and using them from APL2.

```
      100 ⎕SVO 'CTL100'
2
      DA←'DA(''USERID.PWS.OBJ(MATHFNS)'')'
      CTL100←'ALLOC FI(SYSPUNCH) SHR REUSE ',DA
      PACKAGE '''APL2.V0000001.MATHFNS'''
USERID.PWS.OBJ
      DA←'DA(''USERID.PWS.OBJ(UTILITY)'')'
      CTL100←'ALLOC FI(SYSPUNCH) SHR REUSE ',DA
      PACKAGE '''APL2.V0000001.UTILITY'''
USERID.PWS.OBJ
      RL←⊂'APL2RL'
      RL←RL,'EIGEN MATHFNS' 'POLYZ MATHFNS'
      RL←RL,'LINEFOLD UTILITY' 'NAMES UTILITY'
      '''USERID.PWS.OBJ(APL2RL)''' BUILDRL RL
0
      TARGET←'L460581.PWS.LOAD(APL2WS)'
      SOURCE←'L460581.PWS.OBJ(APL2RL MATHFNS UTILITY)'
      CTL100←⍕'AP2MP11L' TARGET SOURCE
      )HOST ALLOC FI(PWS) SHR REU DA('USERID.PWS.LOAD')
TSO(0)
      'PWS.APL2WS' 11 ⎕NA⊃'EIGEN' 'POLYZ' 'LINEFOLD' 'NAMES'
1 1 1 1
      EIGEN 2 2⍴1 0 0 2
1 2
1 0
0 1
      POLYZ 1 ¯6 11 ¯6
1 2 3
      15 LINEFOLD 'THIS IS SOME SAMPLE TEXT'
THIS IS SOME
SAMPLE TEXT
      NAMES 'FRED∘.×JANE←⍳9'
FRED
JANE
```

Figure 107. Build a member containing several namespaces.

Link-Editing Namespaces

The APL2 *PACKAGE* function produces object files that are usually link-edited before use from APL2. In addition, multiple namespace object files may be link-edited together into a single file with a routine list built by *BUILDRL*. The object files produced by *PACKAGE* and *BUILDRL* can be link-edited with the same procedures as non-APL external routines discussed in “Link-Editing External Routines” on page 327.

Unexpected Errors and Other Considerations

- Name Association Failures: There are a number of reasons why dyadic ⎕*NA* fails to activate a name and return 0:
  - Incorrect arguments. Malformed APL names in the right argument or incorrect name class or processor number in the left argument cause a failure of

dyadic ⎕*NA*. If monadic ⎕*NA* on the same right argument returns a ¯1 the object name is malformed.

If the first item in the left argument (or a row of the left argument) is invalid, dyadic ⎕*NA* fails and returns 0. In this situation, a)*MORE* message is queued specifying *PARAMETER ERROR*.

If dyadic ⎕*NA* is issued for a name that already exists in the workspace, the left argument of ⎕*NA* must match the left argument used when the original ⎕*NA* was issued. That original left argument can be determined by issuing monadic ⎕*NA*.

– The object cannot be located. If the specified name cannot be located in the namespace, or if the namespace itself cannot be located, dyadic ⎕*NA* returns a 0 and)*MORE* messages are queued to indicate that the object is not available. In this situation a message of the form:

```
name PARAMETER ERROR :LOAD
name PARAMETER ERROR :MEMB
name PARAMETER ERROR :ENTRY
```

may be queued to indicate, respectively, that the specified load library cannot be accessed, the specified member cannot be found or loaded, or that the namespace cannot be found in the member, or the desired object does not exist, or does not exist with the correct name class in the namespace.

– Name not specified as an entry point. When the namespace is created from a saved workspace using the external function *PACKAGE*, the creator can specify as a left argument to *PACKAGE*, a list of names of objects in the namespace that can be accessed through the use of ⎕*NA*. If the user attempts to access any other object, not specified in that list, dyadic ⎕*NA* returns a 0 and)*MORE* messages are queued:

```
name PARAMETER ERROR :ENTRY
name NOT AVAILABLE
```

– Error in the NAMES file. An invalid parameter in the NAMES file causes dyadic ⎕*NA* to return a 0 and a)*MORE* message to be queued indicating the problem.

– Insufficient freespace for proper execution of Processor 11 or for loading of the namespace may cause dyadic ⎕*NA* to return a 0 and a)*MORE* message to be queued. The user should invoke APL2 with more freespace.

– Processor 11 or the NAMES file unavailable causes ⎕*NA* to return a 0. Proper operation of Processor 11 and availability of the names file supplied with APL2 can be verified by running the APL2 installation verification procedure.

- *VALENCE ERROR* or *VALUE ERROR*: If a *VALENCE ERROR* occurs when execution of an external function or operator is attempted, or a *VALUE ERROR* occurs when an external variable is referenced, it may indicate that the linkage to the namespace cannot be established for any of the reasons mentioned above. In this situation a)*MORE* message is queued along with the *VALENCE ERROR* or *VALUE ERROR* and a subsequent dyadic ⎕*NA* for the object returns a 0.
- *VALENCE ERROR*, *VALUE ERROR*, or *SYNTAX ERROR* occurs if after a name is successfully activated with ⎕*NA*, the named object is deleted or its

class or valence changed as a result of execution in the namespace's namescope.

- Suspension in a Namespace: If the user's session becomes suspended during execution in a namespace, the user is left in the namescope of the namespace. That is to say, names in the namespace are visible and accessible, but names in the user's active workspace are not.)*FNS*,)*VARS*,)*OPS*,)*NMS*, and ⎕*NL* report names from the namespace. To return to the namescope of the active workspace, the packaged application should be resumed or abandoned.)*RESET* always returns the user to the namescope of his active workspace. The external function *QNS* can be used to determine which namescope is currently active.
- Operand to operators. Operands to defined operators declared through the use of ⎕*NA* are defined and execute in the namescope in which the defined operator is called. For example, if the following operator was defined as part of a namespace:

```
        ∇Z←(FN OPER) NAME
[1]     Z←FN NAME
[2]     ∇
```

and called from the user's active workspace as follows,

```
        ⎕IO←497
        4 11 ⎕NA 'OPER'
1
        ⍎ OPER '⎕IO'
497
```

the (invalid) value of 497 for ⎕*IO* would be obtained from the user's active workspace.

- Loss of data in a namespace. Some applications in namespaces, create or modify objects (typically variables) in the namescope of the namespace when the application is executed. The state of the application (including these modifications) is saved along with the user's workspace when a)*SAVE* command is issued. If the namespace is found to have been modified (that is, recreated from a workspace with a different saved date) when the user's workspace is reloaded, an error message is produced when objects in the package are accessed, and access to the new version of the namespace is provided. Objects in the package that were created or modified by the application before the user's workspace was saved are lost.

  Developers of packaged applications and personnel who maintain them should be aware of this behavior. When providing maintenance or enhancements to a namespace, it may be wise to keep the old version around, so that users who have a dependence on it do not lose data.

Chapter 30. Processor 12—Files as Arrays

Associated Processor 12 provides access to a variety of types of files by maintaining an image of the file as an array that appears to reside in the active workspace. This is analogous to the behavior of Processor 11 for functions. That processor can create an image of a program (written in any of a variety of languages) as a function that appears to reside in the active workspace. Neither the program (for Processor 11) nor the file (for Processor 12) is actually within the workspace. This has the following implications for Processor 12 files:

- Very large files can be accessed, files that can be many times larger than the active workspace. And yet the access can be done using normal APL constructs such as (to show only a few examples):

Compression	*bool*/*file*
Each	*process*¨*file*
selective assignment	(*recno*⊃*file*)←*value*
catenation	*file*←*file*,⊂*record*

- Associations can be retained across)*SAVE* and)*LOAD* but the data is preserved in the file, and may be updated by other programs between uses.

  Note: In particular this should be contrasted with the Processor 11 definition for association with variables in namespaces. The general rule used by Processor 11 is that any time a variable is modified the new version is a private one known only to the workspace that was active at the time of modification.

  It should also be noted that files, even files newly created by Processor 12, have an existence independent of the workspace. Assigning a value to a Processor 12 variable causes (at least conceptually) an immediate and permanent change to the file. This is not affected by later expunging the variable, and is independent of whether the workspace containing it is later saved.

Processor 12 variables are also quite different from variables shared with file auxiliary processors.

- Processor 12 variables contain only the data, and (at least conceptually) all of the data at once. Shared variables contain both data and control information, and only relatively small pieces of the file data at a time.
- Processor 12 variables are really a path between the workspace and the actual file. Shared variables are a path between two programs, one of which in turn is capable of accessing files.
- Processor 12 associations can be retained across)*SAVE* and)*LOAD*. Shared variable associations must be reestablished explicitly.

⎕*NA* Syntax for Processor 12

The general syntax for name association through Processor 12 is:

```
('type' 'locator' 'format') 12 ⎕NA 'name'
```

name A name to be used within the APL workspace to refer to the file. The particular name used has no significance to Processor 12, and bears no required relationship to the name of the file with which it is associated. Surrogate names are permitted, but have no functional significance.

type A vector of two or more characters, the first specifying what class of file support is desired, and the others indicating how the file is to be accessed.

The classes supported are "*A*" for APL files and "*F*" for operating system sequential files.

The types of access are:

W The file may be read and/or written. Depending on the type of system and class of file, this may also imply that exclusive control of the file is to be gotten and held for so long as the association is active. If write access (and exclusive control where appropriate) cannot be obtained, the ⎕*NA* returns `0` even if the file is otherwise available.

R The file can be read, but not written. Any attempt to modify the associated variable causes an interruption of the responsible APL statement. The ⎕*ET* error code is `2 4`, for which the message and)*MORE* message are *SYNTAX ERROR*+ and *Invalid operation in context*.

C The file is to be created and then written. If both *C* and *W* are specified, the file is created if it does not exist. If either *C* or *W* are specified alone, the file must exist (*W*) or not exist (*C*) before the operation.

This access code can be followed by one or two numbers in parentheses. (The square brackets indicate an optional field. They are not to be coded as part of the access information.):

C(*record_length* [*file_size*])

D The file is to be deleted on completion of processing. This deletion occurs when the file connection is erased, whether by ⎕*EX*, exit from a function where it is localized,)*ERASE*,)*COPY*,)*CLEAR*,)*LOAD*,)*OFF*, or)*CONTINUE*. *D* can be used only along with other access codes. That is, you can only delete a file if you have first opened it successfully for Read, Write, or Create. In some environments, the combination of *R* and *D* (without *C* or *W*) requires the same access authority initially as if file output had been requested.

Note: The above access codes may be given in any order, and may be separated by blanks as desired. If the parenthetic expression associated with *C* is used it must follow the *C* access code. No access code may be given more than once.

Examples of type items:

`'FW'` Gain read/write access to an existing system file.

`'ACW'` Create an APL file, or gain read/write acess to it if it already exists.

`'FRD'` Read a system file and then delete it.

`'A C (0 500000)'`
Create an APL file, rejecting the association if one by the same name already exists, and set a file size limit of 500,000 bytes. (Usage of the numbers is described later for each class of file support.)

`'FC(80)DRW'`
Create a file with a record length of 80 bytes, or accept an existing one. Allow read/write access to the file (the *R* is redundant, but allowed) and delete it when the connection is erased.

locator A character vector indicating where the file is located. The format and content of this field vary depending on the type of file being accessed and the operating system in control.

format A character vector that defines the format in which the data is to be viewed by the application. This vector must be empty if the file support class is *A* (APL files), and it must be nonempty for class *F* (flat files). See "Format Descriptors for External Variables" on page 358 for details.

The explicit result of ⎕*NA* is `1` if the association was successful, or `0` if it failed. When `0` is returned, explanatory messages are usually queued. These may be seen by entering)*MORE* at the first terminal input opportunity or by running with *DEBUG*`(1)`.

Supported Primitive Operations

Regardless of the file system in use, the following primitive operations are defined for external variables supported by Processor 12:

Each	*function*¨*file* *var function*¨*file* *file function*¨*var* *file*1 *function*¨*file*2
Outer product	*var*∘.*function file* *file*∘.*function var* *file*1∘.*function file*2
Pick	*i*⊃*file*
Indexing	*file*[*i*] *i*⌷*file*
Indexed assignment	*file*[*i*]←⊂*array*
Selective Spec	(*i*⊃*file*)←*array* (↑*i*↓*file*)←⊂*array* etc.
Catenate	*file*←*file*1,⊂*array*
Shape	⍴*file*
Compress/Replicate	*i*/*file*
First	↑*file*
Take	*i*↑*file*
Drop	*i*↓*file*

Notes:

1. Operations other than those defined here either attempt to bring the entire file into the workspace or give *DOMAIN ERROR*.
2. The functions referred to in Each and Outer product can be arbitrary primitive, defined, or derived functions. Since they are invoked repeatedly with one item of the array at a time, there is no immediate requirement that the entire array truly reside in the workspace. But if the invoked function produces a result, the full accumulated result returned by the derived function is a normal variable stored in the workspace.
3. When applied to an empty file, First returns a prototype, which for the vector of vectors form is a blank vector whose length is the record length of the file. For the vector of matrixes form, a zero-row matrix is returned with the second dimension being the record length.

APL Files as External Variables

This facility supports certain APL format files created and written by the APL2 file system services. This includes files created using either AP 121 or the FC service defined in *APL2/370 Programming: Processor Interface Reference*. Only APL2-format records in direct files are supported; that is, files created using

'C fileid D'
'SWC fileid length'

where *length* is often, but need not be, zero. Note that if records are later written with *'SW'* or *'DU'* (rather than *'SWC'* or *'DUC'*), those records cannot be handled by Processor 12.

Conversely, files created by this facility can be processed by AP 121 or the file services defined in *APL2/370 Programming: Processor Interface Reference*.

APL files are always viewed by the APL application as a vector of arbitrary arrays, with each item of the vector representing one object in the file. Each item may be of any depth or shape.

The left argument syntax for name association with APL files is:

```
('A {C[(len size)]|R|W|D}...' '[libno ]filename' '') 12
```

where { | | }**...** and [ ] indicate choices, repetition, and options, but these symbols are not coded in the argument

Note: The last sub-item of the first item in the left argument must be empty for APL files. In general this is the format descriptor, but an APL file is always self-describing.

If *'AC'* or *'AW'* is specified (perhaps in combination with other access codes) this is treated as a writable file. The library containing the file (for TSO), or the disk containing the file (for CMS), is accessed in write mode for so long as the association is active. This normally guarantees exclusive write access.

len This value can be provided, but is currently ignored. It can be used in the future to support direct files with a limited length on individual objects.

size If provided, this value limits the maximum size of the file in bytes. This is equivalent to using the AP 121 '*FS*' service request.

libno APL file system library number. This field can be omitted for a private library, and defaults to ⍎⎕*AI*.

filename The simple APL data file name. This must be one to eight characters, with the first alphabetic (uppercase *A* to *Z* only), and subsequent characters uppercase letters or numbers.

Record-oriented Files as External Variables

This facility provides read, write, and update access to data portions of operating system sequential files stored on direct access devices. These files are viewed by the APL application as a vector of arrays in which the subarrays are always character vectors or character matrixes. Each character vector, or each row of a character matrix, represents one record in the file.

for CMS All CMS files are supported except for those in the Shared File System. Files on OS-formatted disks are not supported.

for TSO Most types of sequential DASD files with unkeyed records are supported. Specifically:

DSORG Must be PS or PSU. Partitioned data sets are not supported, even if the member name is included as a part of the data set name.

KEYLEN Only unkeyed records, KEYLEN(0), are supported.

OPTCD(J) The Table Reference Character (used for 3800 font selection) is not supported.

RECFM(A or M) Printer carriage control characters are not supported.

RECFM(V) Blocksize and record length are not passed as part of the data. Spanned records are not supported.

RECFM(U) Not supported.

Note: Concatenated data sets are not supported.

Note: When allocation is by ddname, the application can override record format or options. This can result in control characters and length fields being handled as data.

The left argument syntax for record-oriented file association is:

```
('F{C[(len size)]|R|W|D}...' 'filename|=ddname' 'format') 12
```

where { | | }**...** and [ ] indicate choices, repetition, and options, but these symbols are not coded in the argument.

If '*FC*' or '*FW*' is specified (perhaps in combination with other access codes) this is treated as a writable file. On MVS, exclusive control of the data set is held

for so long as the association is active. On CMS the file must exist on a disk to which you already have write access, and this normally prevents other CMS users from writing to it. On both systems it may be possible to concurrently access the file through other paths within the same APL session. It is an application responsibility to prevent this, or to deal with possibly confusing consequences. These can include erroneous indications of current file size, and file appends that actually replace (or attempt to replace) existing records.

len This value may be used as the logical record length of a new file. If omitted, the rightmost dimension item in the format descriptor is used. At least one of these values must be given explicitly or the create request fails. Note that for variable-length record files on CMS, the length given during create does not become a permanent attribute of the file. It does limit the length of records added to the file during the association when the file is created, but has no effect on subsequent associations (either explicit or automatic).

The new file has variable record format (VB for MVS) if the last shape in the format descriptor is given as `*`, or fixed format (FB for MVS) if it is given explicitly. For MVS the block size is selected by APL2, and is not affected by the format descriptor. MVS users can override the record format and block size by preallocating the data set.

size This value is currently ignored, but may be used in the future to control the size of a newly-allocated MVS data set.

=ddname

A one to eight character name previously defined by a TSO `ALLOCATE` command, or by an MVS JCL `DD` statement. The "`=`" is required to distinguish this case.

This form is not supported for CMS.

Note: Use of a separate `ALLOCATE` command for a new data set allows more control over allocation quantities and location, but the `RELEASE` parameter should be avoided. APL2 may close and reopen the data set during processing, which could reduce the size allocated before all data is written if `RELEASE` is used. This caution applies equally to the `DD` statement `RELSE` option.

filename

Under TSO:

tso.dataset.name
'fully.qualified.dataset.name'

The name of an existing cataloged MVS data set. Note that the TSO convention is followed, so that if the name is not enclosed in single quotation marks, then the current `TSO PROFILE PREFIX` is added to it.

Note: Do not confuse the quotation marks that delimit the locator item with the quotation marks used for a fully-qualified data set name. If a fully-qualified name is a literal, it must be entered as:

'''*fully.qualified.dataset.name*'''

Under CMS:

filename filetype [*filemode*]
: The name of a new or existing CMS file. The *filemode* defaults to the first matching file found in standard CMS search order, or to A1 if no matching file is found.

format See the following section for a discussion of this topic.

Format Descriptors for External Variables

The syntax of the format descriptor for an external variable is similar to that used by Processor 11. Its focus here is on the view of the data as seen by the application, rather than the format of the data as it exists externally.

The fields of a format descriptor must be separated by blanks except where parentheses are used. A separator is required only when one field ends with a numeric digit and the next one begins with a numeric digit. Parentheses may be used as separators wherever desired. Blanks may also be used as leading and trailing characters or adjacent to parentheses, and multiple blanks may be used wherever one is allowed. The total length of the descriptor is limited to 80 characters, excluding leading and trailing blanks.

The supported descriptors are:

Vector of character records
'*C*0 1 * *C*1 1 *length*'

length The length of each item in the array. This is typically defaulted to the length of each record by specifying *.

If specified as *, the length of each array item matches the actual length of the corresponding record. When replacing existing records in the file, the application must provide items of the correct length. Added items are inserted into the file as given, without any padding or truncation. A length error results if the record is longer than the system-imposed limit on records in the file (64K-1 for CMS; LRECL-4 for TSO).

A specific value can be given for *length* only for fixed-length record files, and if used with existing files must match the existing record length.

Vector of character matrixes of records
'*G*0 1 * *C*1 2 *pack length*'

pack The number of records to include in each matrix of the array. The larger this number is, the more efficiently APL can process the records in the file, but larger numbers also require more workspace storage. The value must be provided explicitly.

This value is used for the first dimension of each matrix within the array, except the last item, which may contain a short matrix. Note that applications are not permitted to change the number of rows in any matrix, except that they can append to the file by increasing the rows in the last matrix up to *pack*. Indeed, the application is not permitted to add a new item to the array unless the current last item has the full number of rows.

length The number of columns in the matrix. This is typically the length of each record.

If specified as *, the width of each submatrix is padded with blanks to the length of the longest record represented in it.

- When replacing existing records in a variable length record file, APL2 processes each row of a modified array item by deleting trailing blanks as necessary to make it the correct length for the file.
- When adding new records to a variable length record file, all trailing blanks are first stripped from the records.
- For fixed length files, the matrix width must match the existing record length.

Processor 12 Errors

Processor 12 uses the following rules for handling errors.

- If □*NA* cannot be completed successfully its explicit result is 0. An error message can be queued to explain the failure, but no message is displayed, and no error is signaled.
- If a problem occurs while referencing or setting an associated variable, a □*ET* error is signaled. This is treated as any other error that occurs while processing an APL statement. The specific errors that may be signaled are listed below.
- Actual output to a file is usually performed asynchronously, and errors in this process may not be known for some time after the variable assignment that caused the records to be written. These may include things like out-of-space conditions as well as actual I/O errors on the disk media. If detected during a subsequent access, errors of this sort are normally signaled as follows:

  I/O errors
  □*ET*=1 5 (*SYSTEM LIMIT*+, *Interface unavailable*)

Out of space
⎕ET=1 7 (*SYSTEM LIMIT*+, *Interface capacity*)

Note: If a Processor 12 variable is passed as an argument to a function, it is not actually referenced until used by that function. Thus, the above errors may be signaled on some line of the function that looks at its argument.

- In some cases errors may not be signaled until the association is deactivated. Processor 12 waits for all pending writes to complete at this time, and may also have already detected errors that it could not signal because no recent reference or assignment to the variable has occurred. There are several ways in which file associations may be deactivated:

  1. *⎕EX* terminates with an explicit result of 0 and leaves the file open.
  2.)*ERASE* responds with *NOT ERASED* and leaves the file open.
  3.)*COPY* or)*IN* of an object that would replace the Processor 12 variable responds with *NOT COPIED* and leaves the file open.
  4. *⎕TF* inverse operation, creating an object that would replace the Processor 12 variable, terminates with an empty result, and leaves the file open.
  5. Exit from a defined function or operator where the Processor 12 variable was localized suspends the program on line 0 and signals a *⎕ET* error (normally 1 5 or 1 7 as described above). The file remains open.
  6. → with no arguments closes the file and continues processing with no error indication except a queued message.
  7. Any of the following system commands close the file and continue processing with no error indication except a queued message that is usually indicated by a + at the end of the normal system response:)*CLEAR*,)*LOAD*,)*RESET*, or)*SIC*
  8.)*CONTINUE* and)*OFF* close the file and terminate the APL2 session.

  Note: For the cases above in which the file was left open, repeating the request usually causes the file to be closed, and the request to be processed normally.

The following errors may be signaled by Processor 12:

1 3 *WS FULL*
Required portion of the array does not fit in the workspace.

1 5 *SYSTEM LIMIT+ Interface unavailable*
File I/O error or file in use.

1 7 *SYSTEM LIMIT+ Interface capacity*
File full, or unable to get required working storage outside of the workspace.

1 12 *SYSTEM LIMIT+ Interface representation*
The named file is not a valid APL data file.

2 4 *SYNTAX ERROR+ Invalid operation in context*
A Selective Specification operation was used that is not supported for external variables. This error is also signaled if an attempt is made to modify a variable associated with a read-only file.

5 5 *INDEX ERROR*
The format descriptor has defined a variable larger than the file, and an item has been referenced that is beyond the end of the file.

The following errors are reported only for flat (operating system) files.

5 2 *RANK ERROR*
An item assigned to the variable is of the wrong rank.

5 3 *LENGTH ERROR*
A new item is added to the variable when the last matrix is not full, or an item assigned to the variable is longer than permitted by the format descriptor, or a record being replaced is a different length from its replacement and no padding or truncation is permitted.

5 4 *DOMAIN ERROR*
An item assigned to the variable does not contain character data, or contains extended characters whose □*AF* value is greater than 255, or is itself a nested array.

Appendixes

Appendix A. Implementation Limits . . . 365

Appendix B. Deviations from APL2 Programming: Language Reference . 366
System Functions and Variables . . . 366
Full-Screen Editor—Editor 2 . . . 366
System Commands . . . 367

Appendix C. National Languages Supported by APL2 . . . 368

Appendix D. Auxiliary Processor Conversion Options . . . 370
APL . . . 370
370 or BCD . . . 371
BIT . . . 371
BYTE . . . 371
CDR . . . 372
COD1 . . . 372
DBCS[(nnn)] . . . 372
Reading DBCS Data . . . 373
Writing DBCS Data . . . 374
EBCD or 192 . . . 374
TN . . . 374
VAR . . . 374

Appendix E. Conversion of Atomic Vector Characters . . . 376

Appendix F. APL2 Files and Data Sets . . . 381
CMS Files . . . 381
CMS Filedef (DD) Names . . . 382
TSO DD Names . . . 382
TSO Data Set Names . . . 383

Appendix G. Sample Non-APL Programs to be Called through Processor 11 . . . 385
C/370 Examples . . . 386
Updating Arguments with C/370 . . . 386
Source Code . . . 386
Routine Descriptions, Routine List, and Link-Editing . . . 386
NAMES Files Entries . . . 387
Routine List and Link-Editing Non-Self-Describing Routines . . . 387
PL/I Examples . . . 388
Updating Arguments with PL/I . . . 388
Source Code . . . 388
Routine Descriptions, Routine List, and Link-Editing . . . 388
NAMES Files Entries . . . 389
Routine List and Link-Editing Non-Self-Describing Routines . . . 389
VS FORTRAN Examples . . . 390
Updating Arguments with VS FORTRAN . . . 390
Source Code . . . 390
Routine Descriptions, Routine List, and Link-Editing . . . 390
NAMES Files Entries . . . 391

Routine List and Link-Editing Non-Self-Describing Routines 391
Link-Editing Examples 392
Link-Editing on TSO using a CLIST 392
Link-Editing on TSO using JCL 392
Link-Editing on CMS 393
Generating a MODULE on CMS 393

Appendix H. Summary of Terminal Information for APL2 Tasks 394
IBM 2741 Communication Terminal 394
IBM 3270 Information Display System 396

Appendix I. Printer Fonts Supplied with APL2 399

Appendix A. Implementation Limits

The APL2 interpreter has the following implementation limits.

Figure 108. Limitations by System

Limitation	Workstations (excluding APL2/PC)	APL2/370
Largest and smallest representable numbers in an array	1.7976931348623158*E*+308 and ¯1.7976931348623158*E*+308	7.2370055773322621*E*75 and ¯7.2370055773322621*E*75
Most infinitesimal (near `0`) representable numbers in an array	2.2250738585072014*E*¯308 and ¯2.2250738585072014*E*¯308	5.397605346934027891*E*¯79 and ¯5.397605346934027891*E*¯79
Maximum rank of an array	63	64
Maximum length of any axis in an array	`¯1+2*31` (2147483647)	`¯1+2*31` (2147483647)
Maximum product of all dimensions in an array	`¯1+2*31` (2147483647)	`¯1+2*31` (2147483647)
Maximum depth of an array applied with the primitive functions depth (≡*R*) and match (*L*≡*R*)	181	181
Maximum depth of a shared variable	181	181
Maximum depth of a copied variable	181	181
Maximum number of characters in the name of a shared variable	255	255
Maximum number of characters in a comment (minus leading blanks)	4090	32764
Maximum length of line	8190	N/A
Maximum number of lines in a defined function or operator	`¯1+2*15` (32767)	`¯1+2*31` (2147483647)
Maximum number of labels in a defined function or operator	Limited by number of lines	32767
Maximum number of local names (excluding labels) in a defined function or operator	Limited by lengths of lines and names	32767
Maximum number of slots in the internal symbol table. A slot is required for each unique name, each unique constant, and each ill-formed constant in the workspace.	N/A	32767
Maximum value of □*PW*	254	390
Maximum value of □*PP*	16	18
Maximum number of users with whom a user can share cross systems variables	N/A	59

Appendix B. Deviations from APL2 Programming: Language Reference

This appendix describes the areas in which the implementation of APL2/370 differs from the APL2 language as defined in *APL2 Programming: Language Reference*.

The deviations are classified as follows:

(D) Deviation from *APL2 Programming: Language Reference*: The feature is implemented in APL2/370, but the implementation is different from that defined in *APL2 Programming: Language Reference*.

(R) Implementation restriction: The feature is not fully implemented in APL2/370.

(S) System dependency: The feature is not required in APL2/370 or is implemented differently due to system requirements.

System Functions and Variables

(D) When running under a CMS batch machine, a reference of $\square SVE$ is satisfied immediately, even when no shared variable event has occurred. For example:

```
      □SVE←180
      □SVE
179.991822
```

Use $\square DL$ between references to $\square SVE$ to avoid excessive processor time.

(D) $\square UCS$ is correctly defined only for characters included within $\square AV$. For all other values, it behaves like $\square AF$. It does not map Kanji characters between the ISO 10646 and Codepage 300 encodings.

Full-Screen Editor—Editor 2

(D) Using Editor 2 in Nondisplay Mode.

If GDDM is not available, GDDM does not support your terminal, or if a GDDM error occurs during your use of Editor 2, Editor 2 enters nondisplay mode. The following message is displayed after a GDDM error:

```
AP2P2ED558 PROCESSOR 126 ERROR nnn (HIGHEST) xxx yyy (FIRST)
AP2P2ED559 NON-DISPLAY EDIT MODE...∇name.type.
```

Thereafter, the message is displayed, showing the name and type of the object being edited. The cursor indents two spaces to indicate that you are in edit mode.

If nondisplay mode occurs while you are editing, it is recommended that you enter ∇ or `[→]` once for each open segment to terminate editing.

System Commands

(D) The)*LOAD* command produces a *SAVED* message that indicates the size of the active workspace after the)*LOAD* and when it was last saved. The two sizes may differ because of differences in the previous and current settings of:

- Size of the active workspace
- Invocation WSSIZE parameter
- *size* parameter of the)*LOAD* command

The size information can be useful in diagnosing space problems.

Unless otherwise specified with the *size* parameter, the size of the active workspace is the maximum size available.

(S) The following library commands support the entering of a read or write password established for the library or workspace through the facilities of the host system:

)*COPY*
)*DROP*
)*LIB*
)*LOAD*
)*MCOPY*
)*PCOPY*
)*SAVE*
)*WSID*

A colon (:) separates the password from the workspace name. The colon must be the first parameter after the workspace name, followed by the password or a blank.

If a blank follows the colon or if you do not specify a password, the system may prompt you to enter a password when one is required.

For example:

```
      )COPY 1010 TOOLBOX:SECRET LOCKOUT SUMCOL MODIFY
SAVED 1993-10-20 14.02.54 (GMT-7)
```

(S) If)*RESET*,)*SIC*, or)*SAVE* is issued, special workspace cleanup is performed. The cleanup includes:

- Removing namespace associations used within the system itself. Parts of the system, such as default display of a nested array and the)*OUT* command, are written as APL namespace code.
- Discarding any namespaces that are no longer in use.
- Making each symbol table (active workspace and namespaces) smaller, if possible.
- Returning all free storage from specialized storage pools to the general freespace pool.

)*RESET* 0 or)*SIC* 0 can be issued to cause this cleanup without saving the workspace or resetting the execution stack.

(S) The)*MSG*,)*OPR*, and)*TIME* system commands are provided for compatiblity with older mainframe APL implementations.

Appendix C. National Languages Supported by APL2

The languages shown in Figure 109 are provided with the APL2 product. Installations can choose to make a subset of these available on their systems, or may define additional languages.

Figure 109. Values Provided for the National Language Translation

Value of ⎕NLT	Abbreviated Value	Language
'DANSK'	'DAN'	Danish
'DEUTSCH'	'DEU'	German
''	'ENP'	American English, uppercase
'DEFAULT'	'ENU'	American English, lowercase
'ESPANOL'	'ESP'	Spanish
'FRANCAIS'	'FRA'	French
'FRC'	'FRC'	Canadian French
'HEBREW'	'HEB'	Hebrew
'ITALIANO'	'ITA'	Italian
'JAPANESE'	'JPN'	Japanese, double byte
'KATAKANA'		Japanese, single byte
'NORSK'	'NOR'	Norwegian
'PORTUGUES'	'PTG'	Portuguese
'SUOMI'	'FIN'	Finnish
'SVENSKA'	'SVE'	Swedish

Note: Appropriately equipped terminals are required for Kanji, Katakana, and Hebrew, as well as some other languages; additional languages may be supported in certain countries.

APL2 recognizes 25 other three-character language codes which are not distributed with the product. Figure 110 on page 368 lists these codes along with the language name into which APL2 attempts to convert each.

Figure 110. Other Language Codes Recognized by APL2

Abbreviated Value	Language
'*AFR*'	AFRIKAANS
'*ARA*'	ARABI
'*BGR*'	BULGARSKI
'*CAT*'	CATALA
'*CHT*'	ZHONGWEN
'*CSY*'	CESKY
'*DES*'	SCHWEIZER
'*ELL*'	ELLINIKA
'*ENP*'	(empty)
'*FRS*'	SUISSE
'*ISL*'	ISLENSKA
'*ITS*'	SVIZZERO
'*KOR*'	CHOSON
'*NLD*'	NEDERLANDS
'*NON*'	NYNORSK
'*PLK*'	POLSKI
'*RMS*'	ROMONTSCH
'*ROM*'	ROMANA
'*RUS*'	RUSSKIJ
'*SHC*'	SRPSKO
'*SKY*'	SLOVENSKY
'*SQI*'	SHQIP
'*THA*'	THAI
'*TRK*'	TURKCE
'*URD*'	URDU

Note:

1. "(empty)" listed above for *ENP* means that □*NLT* is set to an empty vector. This provides uppercase American English using text that is linked with the APL2 product.
2. "DEFAULT" listed above for *ENU* is a nonstandard name that APL2 uses to refer to mixed-case American English. This is distributed with the product for the first time in this release, and is the installation default unless locally modified.
3. IBM has defined short names for several other languages for which it has no approved spelled out name. The short names for those languages are retained unchanged in □*NLT* if defined by a corresponding APL2LANG file. Canadian French is a language in this category that is distributed with the product, and is represented as *FRC*.

Appendix D. Auxiliary Processor Conversion Options

The conversion options described in this appendix are used with auxiliary processors. When a system facility is encoded in a coding format different from that of APL2, you may specify a conversion option in the initial value of the shared variable. The format of the specification depends on the requirements of each auxiliary processor.

Figure 111 is a cross-reference table indicating the conversion options that are available with the auxiliary processors. Each option is described following the figure. The following codes are used in the figure:

C = Available under CMS
T = Available under TSO
X = Valid option under either CMS or TSO
D = Default value
ONLY = Cannot be overridden

Figure 111. Conversion Options Available with Auxiliary Processors

Option	AP 100	AP 101	AP 102	AP 110	AP 111	AP 120	AP 121	AP 123	AP 126	AP 127	AP 210
APL				C	X						
370 (BCD)	X C-D			C	X						T
BIT				C	X						T
BYTE				C	X			X-D [1]			T
COD1				C	X			X [1]	X [2]		T
DBCS				C	X						T
EBCD (192)	X T-D	ONLY		C	X	ONLY		X [1]	X-D [2]	ONLY	T
TN					T						T
The CDR and VAR conversion options are for use when reading and writing APL2 and VS APL variables, respectively.											
CDR				C	X		X				T
VAR				C-D	X-D		X				T-D

Notes:

1. Equivalent function is provided but not through the listed options. See T, T1, T2 request.
2. Equivalent function is provided but not through the listed options. See –4 request.

APL

The APL conversion option is provided as a migration aid for users who have existing data in this format. This option has special support for the compound characters of VS APL, but the support has not been extended to the additional APL2 characters.

IBM recommends that you plan to convert to the EBCD option, because the APL option may not be supported in future possible releases of APL2.

This option provides a one- to three-character translation of APL2 compound characters. Compound characters of VS APL are expanded to their constituent parts. For example, 'A backspace _' (or '_ backspace A') entered as an input character is converted to $\underline{A}$. On output, $\underline{A}$ is converted to 'A backspace _'. Compound characters of APL2 that were not supported in VS APL are not expanded. Invalid or unsupported characters on input are converted to the first item of $\Box AV$, X'00'.

The characters that are generated by this option for each item of $\Box AV$ are shown in Appendix E, “Conversion of Atomic Vector Characters” on page 376.

370 or BCD

The 370 option provides limited graphic character translation from APL to EBCDIC. It is provided as a migration aid for users of previous versions of APL. IBM recommends you plan to convert to the EBCD option, because the 370 option may not be supported in future possible releases of APL2.

The character that is generated for each item of $\Box AV$ is shown in Appendix E, “Conversion of Atomic Vector Characters” on page 376.

BIT

The BIT option translates character or numeric 0's and 1's to binary 0's and 1's (or from binary 0's and 1's to numeric 0's and 1's).

When sharing a variable with an AP, the value must be a character or numeric vector or scalar. Every eight items of the value are combined into one byte in the result. If the value does not contain a multiple of eight items, the last result byte is padded with 0's.

When an AP returns a value, it is returned as a numeric vector of 0's and 1's, one item per bit of the value being passed. The vector always has a multiple of eight items.

BYTE

The BYTE option transfers data byte-for-byte without translation, regardless of the character encoding being used by APL. Two examples of its use are:

- Packed decimal fields, which would be decoded by mapping against $\Box AV$
- Data being transferred from one file to another without inspection

Migration Consideration: If you have used this option to store internal VS APL character data in a file under the control of the operating system, no conversion is done when the file is read by APL2, and the encoding does not match the EBCDIC used by APL2. Change the option to COD1.

If you have used this option in conjunction with a translate table to convert data into VS APL encoding, the translate table or the conversion option must be changed to convert the data to APL2 encoding.

CDR

CDR is an acronym for "common data representation." With this option, any APL2 variable, including nested arrays, may be read or written.

When a variable is shared with an auxiliary processor using this option, the entire variable is written in CDR format. Its size, shape, and type are included.

The internal VS APL format is VAR. Files may contain a mixture of VAR and CDR formats, but:

- VS APL cannot process records in CDR format.
- Several APL2 data types cannot be represented in VAR format.

The CDR option can read records that are in either format. While somewhat faster than the VAR option, the CDR option does use slightly more space (approximately eight bytes per variable).

When referencing a shared variable, the entire APL data object is read. The file being read must contain valid APL2 or VS APL objects. Otherwise, a VALUE ERROR results when attempting to reference the value.

When the CDR option is specified, each specification of the record variable causes the array specified to be written as a single logical record to the file; each reference of the record variable causes a single logical record to be read from the file and returned as an APL array.

For AP 111 and AP 210, the logical record length (LRECL) of the file limits the size of an array that can be written to a file. The size (in bytes) of a specific APL array can be determined through the use of $4\ \square AT$. Attempts to write arrays whose size is greater than the logical record length of the file are rejected. For files with fixed length records (RECFM = F or FB), arrays that are shorter in size than the logical record length are padded transparently to match the required logical record length.

COD1

The COD1 option allows you to access existing files that were written using the BYTE option to store internal VS APL character data in files under the control of the operating system. If such data is read under APL2 using the BYTE option, it is returned without conversion, and its encoding does not match the EBCDIC used by APL2. COD1 should be used in place of BYTE to provide compatibility in such cases.

DBCS[(nnn)]

DBCS is an acronym for *double-byte character set*. This option allows access to files containing double-byte character data, such as Kanji.

In an APL workspace, DBCS characters are represented as extended characters and occupy 4 bytes of storage each. Each extended character contains a character set identifier (2 bytes) and the character itself (2 bytes). When DBCS characters are stored in extended character format in an APL workspace, the first 2 bytes of each character represents the character set identifier and the last 2 bytes repre-

sent the DBCS character. When non-DBCS characters are stored in extended character format in an APL workspace, the first 2 bytes of each character represent the character set identifier, the third byte is zero, and the last byte represents the character. APL characters (those mapped by $\square AV$), have the character set identifier zero, and thus the first 3 bytes of each such character is zero.

DBCS data may be stored in files in two ways:

- Homogeneous files, in which only each consecutive 2 bytes of data represents a character. If the first byte of a character in the file is nonzero, it is a DBCS character; if the first byte of a character in the file is zero, it is an APL character.
- Heterogeneous files, which can contain a mixture of DBCS data and *single-byte character set* (SBCS) data. In such a file, SBCS characters occupy 1 byte each and DBCS characters occupy 2 bytes each. DBCS characters are distinguished from SBCS characters by enclosing strings of 1 or more consecutive DBCS characters between *shift-out* (SO, X'0E') and *shift-in* (SI, X'0F') characters.

The DBCS option allows access to homogeneous files of DBCS data. Heterogeneous files of SBCS/DBCS data can be read or written using option 192, and converted to and from APL extended characters using the external functions CTK and KTC that are described in the *APL2/370 Programming: Using the Supplied Routines*.

The DBCS conversion option used for a particular file access interacts with the DBCS invocation option that is in effect for the APL2 session. See page 15 for the behavior of the invocation option. Note that invocation options DBCS(ON) or DBCS(TRY) were resolved during invocation, so that the only possibilities during execution are DBCS(OFF) or DBCS(*nnn*), where *nnn* is a session character set identifier. The DBCS invocation option may be modified during the session (see the OPTION external function in *APL2/370 Programming: Using the Supplied Routines*), but that is also resolved immediately in the same way.

The DBCS conversion option allows specification of a character set identifier (*nnn*) that is used to override the session character set identifier. *nnn*, if specified, must be an integer between 0 and 32767.

Reading DBCS Data

When data is read from a file with conversion option DBCS, the two-byte characters on the file are converted to extended characters before being assigned to the shared variable. A character set identifier is supplied as follows:

1. If the first of the two file bytes is zero, then the character set identifier is set to zero.
2. Else if the optional character set identifier has been specified on the DBCS conversion option, that value is used.
3. Else if a session character set identifier is in effect, as specified by the DBCS invocation option, that value is used.
4. Else the character set identifier on each extended character is set to zero.

Writing DBCS Data

When extended character data is written to a file with conversion option DBCS, the two low-order bytes of each extended character are written to the file. SBCS data is padded with a high-order file byte of zero for each character.

Before writing the data, some validation may occur:

1. If DBCS(OFF) is in effect for the session, no validation is done. Character set identifiers on extended data are simply discarded. The optional identifier on the DBCS conversion option has no effect.
2. Else if the optional character set identifier has been specified on the DBCS conversion option, that value is checked against the character set identifier of each extended character in the output data. If any mismatch is found, error 444 is returned, and no data is written to the file.
3. Else the session character set identifier is checked in the same way against extended characters in the output. If any mismatch is found, error 444 is returned, and no data is written to the file.

EBCD or 192

The EBCD option provides a one-for-one mapping of the elements of $\square AV$ to unique EBCDIC codes. It is a preferred synonym for the VS APL option, 192. Data is translated as necessary between EBCDIC and the internal encoding.

Because APL2 uses EBCDIC, no translation is performed with this option under Release 1 of APL2.

TN

The TN option is used only under TSO. This option transfers character vectors with translation of extended APL symbols to their equivalents found on the technical notation (TN) printer train. Its use is primarily for output data routed to high-speed system printers.

VAR

The VAR option is intended only for data that is read or written by VS APL applications.

When writing a variable, data is written in VS APL format, including size, shape, and type. Such data can be read by a VS APL processor.

When reading a variable, the VAR option is equivalent to the CDR option. The entire variable is read, including size, shape, and type. The file being read must contain valid APL2 or VS APL objects. If not, a VALUE ERROR results when attempting to use the value.

The VAR option does not support writing the following data types:

- Mixed arrays
- Nested arrays
- Complex numbers
- Extended (4-byte) characters

When the VAR option is specified, each specification of the record variable causes the array specified to be written as a single logical record to the file; each reference of the record variable causes a single logical record to be read from the file and returned as an APL array.

The logical record length (LRECL) of the file limits the size of an array that can be written to a file. Attempts to write arrays whose size (when converted to VSAPL format) is greater than the logical record length of the file are rejected. For files with fixed length records (RECFM = F or FB), arrays that are shorter in size than the logical record length are padded transparently to match the required logical record length.

Appendix E. Conversion of Atomic Vector Characters

Figure 112 shows how each item of the atomic vector ⎕*AV* is transformed by various character conversion options used with auxiliary processors. For APL2 encoding, both the hexadecimal code and the corresponding APL2 character are shown. For other encodings, hexadecimal and EBCDIC are shown.

The numbers in the first column of the table are indexes into ⎕*AV* with ⎕*IO* equal to 0.

Figure 112 (Page 1 of 5). AP Conversion of Characters

⎕*AV*	APL2 code & graphic	After APL conv.	After BYTE conv.	After COD1 conv.	After EBCD conv.	After BCD conv.
0	00	40	00	FF	00	40
1	01	40	01	00	01	40
2	02	40	02	01	02	40
3	03	40	03	02	03	40
4	04	17	04	03	04	17
5	05	05	05	04	05	05
6	06	40	06	05	06	40
7	07	40	07	06	07	40
8	08	40	08	07	08	40
9	09	40	09	CB	09	40
10	0A	40	0A	CC	0A	40
11	0B	40	0B	CD	0B	40
12	0C	40	0C	CE	0C	40
13	0D	40	0D	CF	0D	40
14	0E	40	0E	D0 }	0E	40
15	0F	40	0F	D1 J	0F	40
16	10	40	10	D2 K	10	40
17	11	40	11	D3 L	11	40
18	12	40	12	D4 M	12	40
19	13	40	13	D5 N	13	40
20	14	40	14	D6 O	14	40
21	15	15	15	CA	15	15
22	16	16	16	C8 H	16	16
23	17	40	17	D7 P	17	40
24	18	40	18	D8 Q	18	40
25	19	40	19	D9 R	19	40
26	1A	40	1A	DA	1A	40
27	1B	40	1B	DB	1B	40
28	1C	40	1C	DC	1C	40
29	1D	40	1D	DD	1D	40
30	1E	40	1E	DE	1E	40
31	1F	40	1F	DF	1F	40
32	20	40	20	E0	20	40
33	21	40	21	E1	21	40
34	22	40	22	E2 S	22	40
35	23	40	23	E3 T	23	40
36	24	40	24	E4 U	24	40
37	25	25	25	C9 I	25	25
38	26	40	26	E5 V	26	40
39	27	40	27	E6 W	27	40
40	28	40	28	E7 X	28	40
41	29	40	29	E8 Y	29	40
42	2A	40	2A	E9 Z	2A	40
43	2B	40	2B	EA	2B	40

Figure 112 (Page 2 of 5). AP Conversion of Characters

□*AV*	APL2 code & graphic	After APL conv.	After BYTE conv.	After COD1 conv.	After EBCD conv.	After BCD conv.
44	2C	40	2C	EB	2C	40
45	2D	40	2D	EC	2D	40
46	2E	40	2E	ED	2E	40
47	2F	40	2F	EE	2F	40
48	30	40	30	EF	30	40
49	31	40	31	F0 0	31	40
50	32	40	32	F1 1	32	40
51	33	40	33	F2 2	33	40
52	34	40	34	F3 3	34	40
53	35	40	35	F4 4	35	40
54	36	40	36	F5 5	36	40
55	37	40	37	F6 6	37	40
56	38	40	38	F7 7	38	40
57	39	40	39	F8 8	39	40
58	3A	40	3A	F9 9	3A	40
59	3B	40	3B	FA	3B	40
60	3C	40	3C	FB	3C	40
61	3D	40	3D	FC	3D	40
62	3E	40	3E	FD	3E	40
63	3F	40	3F	FE	3F	40
64	40	40	40	40	40	40
65	41 *A*	C1166D	41	5C *	41	81 a
66	42 *B*	C2166D	42	5D)	42	82 b
67	43 *C*	C3166D	43	5E ;	43	83 c
68	44 *D*	C4166D	44	5F ¬	44	84 d
69	45 *E*	C5166D	45	60 -	45	85 e
70	46 *F*	C6166D	46	61 /	46	86 f
71	47 *G*	C7166D	47	62	47	87 g
72	48 *H*	C8166D	48	63	48	88 h
73	49 *I*	C9166D	49	64	49	89 i
74	4A ¢	40	4A ¢	16	4A ¢	40
75	4B .	4B.	4B.	82 b	4B.	4B.
76	4C <	4C <	4C <	9E	4C <	4C <
77	4D (	4D (	4D (	C2 B	4D (	4D (
78	4E +	4E +	4E +	90	4E +	4E +
79	4F |	40	4F|	1E	4F|	40
80	50 &	40	50 &	09	50 &	40
81	51 *J*	D1166D	51	65	51	91 j
82	52 *K*	D2166D	52	66	52	92 k
83	53 *L*	D3166D	53	67	53	93 l
84	54 *M*	D4166D	54	68	54	94 m
85	55 *N*	D5166D	55	69	55	95 n
86	56 *O*	D6166D	56	6A]	56	96 o
87	57 *P*	D7166D	57	6B ,	57	97 p
88	58 *Q*	D8166D	58	6C %	58	98 q
89	59 *R*	D9166D	59	6D _	59	99 r
90	5A !	40	5A !	17	5A !	40
91	5B $	40	5B $	25	5B $	40
92	5C *	5C *	5C *	94 m	5C *	5C *
93	5D)	5D)	5D)	C3 C	5D)	5D)
94	5E ;	5E ;	5E ;	C4 D	5E ;	5E ;
95	5F ¬	40	5F ¬	1F	5F ¬	40
96	60 -	60 -	60 -	91 j	60 -	60 -
97	61 /	61 /	61 /	AE	61 /	61 /
98	62 *S*	E2166D	62	6E >	62	A2 s

Figure 112 (Page 3 of 5). AP Conversion of Characters

⎕AV	APL2 code & graphic	After APL conv.	After BYTE conv.	After COD1 conv.	After EBCD conv.	After BCD conv.
99	63 T	E3166D	63	6F ?	63	A3 t
100	64 U	E4166D	64	70	64	A4 u
101	65 V	E5166D	65	71	65	A5 v
102	66 W	E6166D	66	72	66	A6 w
103	67 X	E7166D	67	73	67	A7 x
104	68 Y	E8166D	68	74	68	A8 y
105	69 Z	E9166D	69	75	69	A9 z
106	6A ¦	40	6A ¦	1D	6A ¦	40
107	6B ,	6B ,	6B ,	B5	6B ,	6B ,
108	6C %	40	6C %	0A	6C %	40
109	6D _	6D _	6D _	85 e	6D _	6D _
110	6E >	6E >	6E >	A2 s	6E >	6E >
111	6F ?	6F ?	6F ?	B6	6F ?	6F ?
112	70 ◊	40	70	0E	70	40
113	71 ^	47	71	9A	71	50 &
114	72 ¨	41	72	86 f	72	7F "
115	73 ⍥	40	73	0F	73	40
116	74 ⍳	40	74	10	74	40
117	75 ∈	40	75	11	75	40
118	76 ⊢	40	76	12	76	40
119	77 ⊣	40	77	13	77	40
120	78 ∨	46	78	9B	78	40
121	79 `	40	79	22	79	40
122	7A :	7A :	7A :	C5 E	7A :	7A :
123	7B #	40	7B #	24	7B #	40
124	7C ⍝	40	7C ⍝	23	7C ⍝	40
125	7D '	7D '	7D '	C6 F	7D '	7D '
126	7E =	7E =	7E =	A0	7E =	7E =
127	7F "	40	7F "	0B	7F "	40
128	80 ~	54	80	A4 u	80	5F ¬
129	81 a	40	81 a	26	81 a	40
130	82 b	40	82 b	27	82 b	40
131	83 c	40	83 c	28	83 c	40
132	84 d	40	84 d	29	84 d	40
133	85 e	40	85 e	2A	85 e	40
134	86 f	40	86 f	2B	86 f	40
135	87 g	40	87 g	2C	87 g	40
136	88 h	40	88 h	2D	88 h	40
137	89 i	40	89 i	2E	89 i	40
138	8A ↑	55	8A	B7	8A	40
139	8B ↓	56	8B	B8	8B	40
140	8C ≤	43	8C	9F	8C	40
141	8D ⌈	64	8D	97 p	8D	40
142	8E ⌊	65	8E	98 q	8E	40
143	8F →	62	8F	B9	8F	40
144	90 ∆	69	90	BB	90	40
145	91 j	40	91 j	2F	91 j	40
146	92 k	40	92 k	30	92 k	40
147	93 l	40	93 l	31	93 l	40
148	94 m	40	94 m	32	94 m	40
149	95 n	40	95 n	33	95 n	40
150	96 o	40	96 o	34	96 o	40
151	97 p	40	97 p	35	97 p	40
152	98 q	40	98 q	36	98 q	40
153	99 r	40	99 r	37	99 r	40
154	9A ⊃	72	9A	8E	9A	40

Figure 112 (Page 4 of 5). AP Conversion of Characters

□*AV*	APL2 code & graphic	After APL conv.	After BYTE conv.	After COD1 conv.	After EBCD conv.	After BCD conv.
155	9B ⊂	71	9B	8D	9B	40
156	9C	40	9C	83 c	9C	40
157	9D ○	58	9D	95 n	9D	40
158	9E	40	9E	0C	9E	40
159	9F ←	59	9F	BA	9F	7E =
160	A0 ¯	42	A0	81 a	A0	60 -
161	A1 ~	40	A1~	21	A1 ~	40
162	A2 *s*	40	A2 s	38	A2 s	40
163	A3 *t*	40	A3 t	39	A3 t	40
164	A4 *u*	40	A4 u	3A	A4 u	40
165	A5 *v*	40	A5 v	3B	A5 v	40
166	A6 *w*	40	A6 w	3C	A6 w	40
167	A7 *x*	40	A7 x	3D	A7 x	40
168	A8 *y*	40	A8 y	3E	A8 y	40
169	A9 *z*	40	A9 z	3F	A9 z	40
170	AA ∩	73	AA	8B	AA	40
171	AB ∪	74	AB	8C	AB	40
172	AC ⊥	75	AC	A9 z	AC	40
173	AD [	6A]	AD	C0 {	AD	40
174	AE ≥	44	AE	A1 ~	AE	40
175	AF ∘	68	AF	BF	AF	40
176	B0 α	63	B0	89 i	B0	7C @
177	B1 ∈	52	B1	A8 y	B1	40
178	B2 ⍳	57	B2	A7 x	B2	40
179	B3 ρ	53	B3	A6 w	B3	40
180	B4 *ω*	51	B4	8A	B4	40
181	B5	40	B5	14	B5	40
182	B6 ×	49	B6	92 k	B6	40
183	B7 \	77	B7	B0	B7	E0 \
184	B8 ÷	48	B8	93 l	B8	6C %
185	B9	40	B9	0D	B9	40
186	BA ∇	66	BA	87 g	BA	40
187	BB Δ	67	BB	5B $	BB	7B #
188	BC ⍴	76	BC	AA	BC	40
189	BD]	70	BD	C1 A	BD	40
190	BE ≠	45	BE	A3 t	BE	5B $
191	BF |	4F |	BF	99 r	BF	4F |
192	C0 {	40	C0 {	18	C0 {	40
193	C1 *A*	C1 A	C1 A	41	C1 A	C1 A
194	C2 *B*	C2 B	C2 B	42	C2 B	C2 B
195	C3 *C*	C3 C	C3 C	43	C3 C	C3 C
196	C4 *D*	C4 D	C4 D	44	C4 D	C4 D
197	C5 *E*	C5 E	C5 E	45	C5 E	C5 E
198	C6 *F*	C6 F	C6 F	46	C6 F	C6 F
199	C7 *G*	C7 G	C7 G	47	C7 G	C7 G
200	C8 *H*	C8 H	C8 H	48	C8 H	C8 H
201	C9 *I*	C9 I	C9 I	49	C9 I	C9 I
202	CA ⍲	471654	CA	9C	CA	40
203	CB ⍱	461654	CB	9D	CB	40
204	CC □	40	CC	1C	CC	40
205	CD ⍎	58164F	CD	AB	CD	C0 {
206	CE *?*	40	CE	1B	CE	40
207	CF *?*	581677	CF	AC	CF	40
208	D0 }	40	D0 }	19	D0 }	40
209	D1 *J*	D1 J	D1 J	4A ¢	D1 J	D1 J

Figure 112 (Page 5 of 5). AP Conversion of Characters

□*AV*	APL2 code & graphic	After APL conv.	After BYTE conv.	After COD1 conv.	After EBCD conv.	After BCD conv.
210	D2 *K*	D2 K	D2 K	4B .	D2 K	D2 K
211	D3 *L*	D3 L	D3 L	4C <	D3 L	D3 L
212	D4 *M*	D4 M	D4 M	4D (	D4 M	D4 M
213	D5 *N*	D5 N	D5 N	4E +	D5 N	D5 N
214	D6 *O*	D6 O	D6 O	4F |	D6 O	D6 O
215	D7 *P*	D7 P	D7 P	50 &	D7 P	D7 P
216	D8 *Q*	D8 Q	D8 Q	51	D8 Q	D8 Q
217	D9 *R*	D9 R	D9 R	52	D9 R	D9 R
218	DA ⌶	751676	DA	8F	DA	40
219	DB ⍠	7D164B	DB	A5 v	DB	5A !
220	DC ⍒	66164F	DC	B4	DC	40
221	DD ⍋	67164F	DD	B3	DD	40
222	DE ⍐	69167D	DE	BC	DE	40
223	DF ⍝	731668	DF	C7 G	DF	40
224	E0 \	40	E0 \	20	E0 \	40
225	E1 ≡	40	E1	15	E1	40
226	E2 *S*	E2 S	E2 S	53	E2 S	E2 S
227	E3 *T*	E3 T	E3 T	54	E3 T	E3 T
228	E4 *U*	E4 U	E4 U	55	E4 U	E4 U
229	E5 *V*	E5 V	E5 V	56	E5 V	E5 V
230	E6 *W*	E6 W	E6 W	57	E6 W	E6 W
231	E7 *X*	E7 X	E7 X	58	E7 X	E7 X
232	E8 *Y*	E8 Y	E8 Y	59	E8 Y	E8 Y
233	E9 *Z*	E9 Z	E9 Z	5A !	E9 Z	E9 Z
234	EA ≠	611660	EA	AF	EA	40
235	EB ⍀	771660	EB	B1	EB	40
236	EC ⁖	40	EC	1A	EC	40
237	ED *?*	581660	ED	AD	ED	D0 }
238	EE *?*	691648	EE	B2	EE	40
239	EF ⍤	761668	EF	BE	EF	40
240	F0 0	F0 0	F0 0	77	F0 0	F0 0
241	F1 1	F1 1	F1 1	78	F1 1	F1 1
242	F2 2	F2 2	F2 2	79	F2 2	F2 2
243	F3 3	F3 3	F3 3	7A :	F3 3	F3 3
244	F4 4	F4 4	F4 4	7B #	F4 4	F4 4
245	F5 5	F5 5	F5 5	7C @	F5 5	F5 5
246	F6 6	F6 6	F6 6	7D '	F6 6	F6 6
247	F7 7	F7 7	F7 7	7E =	F7 7	F7 7
248	F8 8	F8 8	F8 8	7F "	F8 8	F8 8
249	F9 9	F9 9	F9 9	80	F9 9	F9 9
250	FA	40	FA	84 d	FA	40
251	FB ⍢	661654	FB	88 h	FB	40
252	FC ⍙	67166D	FC	76	FC	40
253	FD ⍟	5C1658	FD	96 o	FD	40
254	FE ⍤	751668	FE	BD	FE	40
255	FF	40	FF	08	FF	40

Appendix F. APL2 Files and Data Sets

This appendix lists and defines the default files and data sets that are used with the APL2 Licensed Program. In most cases, the default definitions can be modified by the installation or the individual user. Also included in this list are the files and data sets that are defined by the installation or the individual user for using or modifying APL2.

Pointers are provided to the more detailed descriptions of each of the files and data sets.

CMS Files

File name	File type	Description
file name	ADMPRINT	The default file type for print files from the the GDDM Environments Defaults Module for VM/SP. See discussion under "COPY" on page 46 in Chapter 3, "The APL2 Session Manager" on page 36.
file name	ADMSYMBL	Source of the APL2 program symbol set used by GDDM.
APLIBTAB	APLIBTAB	The library table used by the)*MCOPY* system command.
file name	APLTF	The default file type for transfer files written using the)*IN* and)*OUT* system commands. See discussion of transfer files in Chapter 4, "APL2 Libraries: Workspaces and Data Files" on page 63.
DUMP*nnn*	APLWSV2	The name of a workspace dump. For a description, see *APL2/370 Diagnosis Guide*.
wkspcname	APLWSV2	The default file type for APL2 private library workspaces. See discussion in Chapter 4, "APL2 Libraries: Workspaces and Data Files" on page 63.
LIBTAB	APL2	Used to associate a library number to a virtual disk. See discussion in Chapter 4, "APL2 Libraries: Workspaces and Data Files" on page 63.
language	APL2HELP	National language translation for help texts selected by ⎕*NLT* and *HELP* function.
language	APL2LANG	National language translation for messages and command keywords selected by ⎕*NLT*. The file name is the spelled out language name (not a three letter abbreviation), truncated to 8 characters if necessary.
AP2TCPIP	APL2PROF	The TCPIP profile file.
file name	APL2UT1	The temporary file type assigned to workspace files during)*SAVE* system command processing.
n	AP2EDCHR	File used when editing a simple character variable.
n	AP2EDEVL	File used when editing any other variable (for example, a vector of numbers, or a combination of numbers and characters).
n	AP2EDPGM	File used when editing a function or operator.
SINKWS	AP2UT2	Used by the)*COPY*,)*PCOPY*, and)*MCOPY* system commands.
AP2EXIT	EXEC	Used during APL2 invocation and termination. See discussion in Chapter 2, "APL2 Invocation and Termination" on page 8.
@LOG1 @LOG2	*file type*	Used to store the session manager log. The file name switches from @LOG1 to @LOG2 when you change the session log size. The file type and file mode depend on LIBTAB APL2. See discussion in Chapter 3, "The APL2 Session Manager" on page 36.
file names	NAMES	NAMES files are used by Processor 11 to provide descriptive information for external functions. The file AP2VN011 NAMES is provided with APL2 and contains descriptive information for external functions distributed with APL2. A second NAMES file, P011 NAMES, may be used as a default NAMES file by Processor 11. See "NAMES Files" in Chapter 28, "Processor 11—Calling Compiled Programs" on page 291 and Chapter 29, "Processor 11—Access to Namespaces" on page 334 for further information.

File name	File type	Description
file name	VMAPL*F	The default file type for AP 110 files. See the discussion in Chapter 14, “AP 110—CMS File Processor” on page 138.
file name	VSAPLFL VSAPL@L	The default file type for private-library AP 121 files; also used for session manager log files and copy work files. See the discussion in Chapter 18, “AP 121—APL2 Data File Processor” on page 182.
DEFAULT	VSAPLPR	The default APL2 session manager profile. See discussion of PROFILE in Chapter 3, “The APL2 Session Manager” on page 36.
file name	VSAPLPR	The default file type for the APL2 session manager profile. See discussion of PROFILE in Chapter 3, “The APL2 Session Manager” on page 36.
wkspcname	VSAPLWS	The default file type for VS APL private library workspaces that are used as input to the)*MCOPY* system command.
wkspcname	**V***nnnnnnn*	The default file type for APL2 public library workspaces. See the discussion in Chapter 4, “APL2 Libraries: Workspaces and Data Files” on page 63.
wkspcname	**W***nnnnnnn*	The default file type for VS APL project and public library workspaces that are used as input to the)*MCOPY* system command.

CMS Filedef (DD) Names

Name	Description
$APL2	Used to load nonshared copy of APL2. See the discussion in *APL2/370 Installation and Customization under CMS*.
APLDUMP	Used to request formatted dumps.
APLIN	Contains input statements acceptable to APL2.
APLPRINT	Contains a log of the APL session.
AP2LOAD	The ddname used to define auxiliary processor load modules. See the discussion of APNAMES in Chapter 2, “APL2 Invocation and Termination” on page 8.

TSO DD Names

Name	Description
ADMSYMBL	Used by GDDM.
APLDUMP	Used as a destination for any formatted dumps.
APLIN	Used in place of terminal input when TERMCODE(-1) or batch processing is in effect.
APLPRINT	Used in place of terminal output when TERMCODE(-1) or batch processing is in effect.
APLTF	Allocated automatically during)*IN* and)*OUT* processing.
APLTRACE	Destination for formatted trace output.
APL2EDIT	File used when a named editor has been specified with the)*EDITOR* system command.
APL2HELP	A partitioned data set or concatenation of partitioned data sets containing national language translations for help texts, selected by □*NLT*. The member name is the spelled out language name (not a three letter abbreviation), truncated to 8 characters if necessary.
APL2LANG	A partitioned data set or concatenation of partitioned data sets containing national language translations for messages and command keywords, selected by □*NLT*. The member name is the spelled out language name (not a three letter abbreviation), truncated to 8 characters if necessary.
APL2PROF	A partitioned data set or concatenation of partitioned data sets containing the TCP/IP profile file.

Name	Description
AP2TN011	Used to allocate default NAMES files used by Processor 11. See Chapter 28, “Processor 11—Calling Compiled Programs” on page 291 and Chapter 29, “Processor 11—Access to Namespaces” on page 334 for further information.
ATF	Allocated automatically during)*IN* and)*OUT* processing.
CPYSPILL	Used to store work files created by the)*COPY*,)*MCOPY*, and)*PCOPY* system commands. The file size should be as large as the workspace from which you are copying. For greater efficiency, the space allocation should be in units of cylinders or with the round option. If you have not allocated this file an attempt is made to use the F0 VSAM cluster, but performance suffers.
CPYSWAP	Used to store work files created by the)*COPY*,)*MCOPY*, and)*PCOPY* system commands. The file size is equal to the size of the active workspace. For greater efficiency, the space allocation should be in units of cylinders or with the round option. If you have not allocated this file an attempt is made to use the F0 VSAM cluster, but performance suffers.
F*nnnn* F0	A VSAM cluster that contains a library of AP 121 files. F0 is the private library, also used for)*COPY* files and session manager log files.
LOADLIB	One or more MVS private load libraries. See the discussion of LOADLIB in Chapter 2, “APL2 Invocation and Termination” on page 8.
W*nnnn* W0	A VSAM cluster that contains a library of workspaces. W0 is the private library, used when no library number is specified on system commands.

TSO Data Set Names

Note: Most of these names can be modified as an installation option.

Name	Description
APL2.SAP2HELP	National Language Help Files
APL2.SAP2LANG	National Language Message and Command Files
APL2.LIB0001.PUBWKSPS APL2.LIB0002.PUBWKSPS	Public library workspaces distributed with APL2, if stored in VSAM libraries.
APL2.SAP2NICK	NAMES files are used by Processor 11 to provide descriptive information for external functions. The file APL2.SAP2NICK is provided with APL2 and contains descriptive information for external functions distributed with APL2. See “NAMES Files” in Chapter 28, “Processor 11—Calling Compiled Programs” on page 291 and Chapter 29, “Processor 11—Access to Namespaces” on page 334 for further information.
APL2.SAP2PROF	Sample profile files (such as TCP/IP).
APL2.V*nnn.prefix*	SAM project library pointer. This must be cataloged as a data set, but no such data set physically exists.
APL2V2R02.DEFAULT.VSAPLPR	Default session manager profile supplied by IBM.
APL2V2R02.V*nnn.wsname*	SAM public workspace. See the discussion in Chapter 4, “APL2 Libraries: Workspaces and Data Files” on page 63.
prefix.APLTF.*xxxxxxxxx*	The sequential file created when a transfer file is written using the)*IN* and)*OUT* system commands. See the discussion of transfer files in Chapter 4, “APL2 Libraries: Workspaces and Data Files” on page 63.
prefix.APL2.EDIT	Data set used when a named system editor has been specified with the)*EDITOR* system command and nothing was preallocated to FILE(APL2EDIT).
prefix.xxx.VSAPLPR	The APL2 session manager profile, where VSAPLPR is the default third-level qualifier. See the discussion of PROFILE in Chapter 3, “The APL2 Session Manager” on page 36.
prefix.V.DUMP*nnnn*	The name of a workspace dump. For a description, see *APL2/370 Diagnosis Guide*.
prefix.V.*wsname*	SAM private workspace. See the discussion in Chapter 4, “APL2 Libraries: Workspaces and Data Files” on page 63.

Name	Description
prefix.V*nnnn.wsname*	SAM project workspace. See the discussion in Chapter 4, “APL2 Libraries: Workspaces and Data Files” on page 63.

Appendix G. Sample Non-APL Programs to be Called through Processor 11

This appendix contains sample programs written in several languages that can be called from APL2 using Processor 11. The samples illustrate the major items that must be be considered when using the languages. Techniques are demonstrated for building routine descriptions, routine lists, and link-editing them with the programs.

Several examples are provided for each language:

- Sample routines that update their arguments
- An APL2 session log showing how to build routine descriptions, a routine list, and link-edit the routines' object files.
- A set of NAMES file entries for use if the routines are not to be self-describing.
- An APL2 session log showing how to build a routine list and link-edit the non-self-describing versions of the routines.

In the set of samples for each language, the following notes are of particular interest.

(Note 1) Each environment program calls AP2TNL or AP2VNL to pass control back to APL2 before it returns to its caller.

(Note 2) The routines update their arguments by simply respecifying their values.

(Note 3) The argument patterns for the routines indicate which argument is updated with the < character.

Notice that although all the languages support having the source code for the routines in separate files, only VS FORTRAN supports link-editing them into separate modules. C/370 and PL/I both require that all the subroutines that are going to run in the environment must be link-edited with the main program.

The appendix concludes with samples that demonstrate the native operating system commands for link-editing object files. Remember that although the TSO linkage editor can automatically include AP2TNL, the CMS APL2 object files, AP2VNL and AP2XCMAP, need to be explicitly included during link-edit. This is because AP2VNL (and AP2XCMAP) are not shipped in TXTLIB files.

C/370 Examples

Updating Arguments with C/370

Source Code

```
#pragma linkage(add,os)
#pragma linkage(sub,os)
#pragma linkage(ap2vnl,os)

int main() {                        /* Main environment program    */
   ap2vnl():          (Note 1)      /* Return control to APL2      */
   return 0;}                       /* Integer result              */

void add(int arg1,int arg2)       { /* Integer argument            */
   arg1=arg1+arg2;}      (Note 2)   /* Update argument with result */

void sub(int arg1,int arg2)       { /* Integer argument            */
   arg1=arg1-arg2;}      (Note 2)   /* Update argument with result */
```

Routine Descriptions, Routine List, and Link-Editing

```
      3 11 ⎕NA⊃'BUILDRD' 'BUILDRL'
1 1
      RD←':LINK.OBJECT :INIT.+INITIAL :VALENCE.0 0 0'
      'ENVRD TEXT' BUILDRD 'ENVRD' 'CEESTART' RD
0                                                          (Note 3)
      RD←':LINK.OBJECT :INIT.CENV :RARG.G0 1 2(<I4 0)(I4 0)'
      'ADDRD TEXT' BUILDRD 'ADDRD' 'ADD' RD
0
      'SUBRD TEXT' BUILDRD 'SUBRD' 'ADD' RD
0
      RL←'CENV ENVRD' 'ADD ADDRD' 'SUB SUBRD'
      'CRL TEXT' BUILDRL (⊂'CRL'),RL
0
      100 ⎕SVO 'CTL100'
2
      SOURCES←'(CRL CUPD ENVRD ADDRD SUBRD AP2VNL)'
      LIBS←'(EDCBASE IBMLIB)'
      CTL100←⍎'AP2MP11L CLOAD(CRL)' SOURCES LIBS
      'CLOAD.CRL' 11 ⎕NA⊃'ADD' 'SUB'
1 1
      ADD 3 17
20
      SUB 15 6
9
```

NAMES Files Entries

```
:NICK.CENV
   :LOAD.CLOAD
   :MEMB.CRL
   :LINK.OBJECT
   :INIT.+INITIAL
   :VALENCE.0 0 0
:NICK.ADD
   :LOAD.CLOAD
   :MEMB.CRL
   :LINK.OBJECT
   :INIT.CENV
   :RARG.(G0 1 2)(<I4 0)(I4 0)     (Note 3)
:NICK.SUB
   :LOAD.CLOAD
   :MEMB.CRL
   :LINK.OBJECT
   :INIT.CENV
   :RARG.(G0 1 2)(<I4 0)(I4 0)     (Note 3)
```

Routine List and Link-Editing Non-Self-Describing Routines

```
      3 11 ⎕NA 'BUILDRL'
1
      RL←'CENV CEESTART' 'ADD' 'SUB'
      'CRL TEXT' BUILDRL (⊂'CRL'),RL
0
      100 ⎕SVO 'CTL100'
2
      SOURCES←'(CRL CUPD AP2VNL)'
      LIBS←'(EDCBASE IBMLIB)'
      CTL100←⍕'AP2MP11L CLOAD(CRL)' SOURCES LIBS
      '(CNAMES)' 11 ⎕NA⊃'ADD' 'SUB'
1 1
      ADD 3 17
20
      SUB 15 6
9
```

PL/I Examples

Updating Arguments with PL/I

Source Code

```
Updates: procedure options (Main) ;
dcl arg1 bin fixed(31) ;
dcl arg2 bin fixed(31) ;
dcl ap2vnl external entry ;
call ap2vnl ;     (Note 1)       /* Pass control to APL2 */
return;

Add: entry (arg1,arg2) ; /* Add two integers */
arg1 = arg1 + arg2 ;     (Note 2)
return;

Sub: entry (arg1,arg2) ; /* Subtract two integers */
arg1 = arg1 - arg2 ;     (Note 2)
return;

end Updates ;
```

Routine Descriptions, Routine List, and Link-Editing

```
      3 11 ⎕NA⊃'BUILDRD' 'BUILDRL'
1 1
      RD←':LINK.OBJECT :INIT.+INITIAL :VALENCE.0 0 0'
      'ENVRD TEXT' BUILDRD 'ENVRD' 'PLISTART' RD
0                                                          (Note 3)
      RD←':LINK.OBJECT :INIT.PLIENV :RARG.G0 1 2(<I4 0)(I4 0)'
      'ADDRD TEXT' BUILDRD 'ADDRD' 'ADD' RD
0
|     'SUBRD TEXT' BUILDRD 'SUBRD' 'SUB' RD
0
      RL←'PLIENV ENVRD' 'ADD ADDRD' 'SUB SUBRD'
      'PLIRL TEXT' BUILDRL (⊂'PLIRL'),RL
0
      100 ⎕SVO 'CTL100'
2
      SOURCES←'(PLIRL PLIUPD ENVRD ADDRD SUBRD AP2VNL)'
      LIBS←'(PLILIB IBMLIB)'
|     CTL100←⍕'AP2MP11L PLILOAD(PLIRL)' SOURCES LIBS
      'PLILOAD.PLIRL' 11 ⎕NA⊃'ADD' 'SUB'
1 1
      ADD 3 17
20
      SUB 15 6
9
```

NAMES Files Entries

```
:NICK.PLIENV
   :LOAD.PLILOAD
   :MEMB.PLIRL
   :LINK.OBJECT
   :INIT.+INITIAL
   :VALENCE.0 0 0
:NICK.ADD
   :LOAD.PLILOAD
   :MEMB.PLIRL
   :LINK.OBJECT
   :INIT.PLIENV
   :RARG.(G0 1 2)(<I4 0)(I4 0)     (Note 3)
:NICK.SUB
   :LOAD.PLILOAD
   :MEMB.PLIRL
   :LINK.OBJECT
   :INIT.PLIENV
   :RARG.(G0 1 2)(<I4 0)(I4 0)     (Note 3)
```

Routine List and Link-Editing Non-Self-Describing Routines

```
      3 11 ⎕NA 'BUILDRL'
1 1
      RL←'PLIENV PLISTART' 'ADD' 'SUB'
      'PLIRL TEXT' BUILDRL (⊂'PLIRL'),RL
0
      100 ⎕SVO 'CTL100'
2
      SOURCES←'(PLIRL PLIUPD AP2VNL)'
      LIBS←'(PLILIB IBMLIB)'
      CTL100←⍕'AP2MP11L PLILOAD(PLIRL)' SOURCES LIBS
      '(PLINAMES)' 11 ⎕NA⊃'ADD' 'SUB'
1 1
      ADD 3 17
20
      SUB 15 6
9
```

VS FORTRAN Examples

Updating Arguments with VS FORTRAN

Source Code

```
PROGRAM FORTENV
CALL AP2VNL     (Note 1)
END

SUBROUTINE ADD(num1,num2)
INTEGER*4 num1,num2
num1 = num1 + num2     (Note 2)
Return
END

SUBROUTINE SUB(num1,num2)
INTEGER*4 num1,num2
num1 = num1 - num2     (Note 2)
Return
END
```

Routine Descriptions, Routine List, and Link-Editing

```
      3 11 ⎕NA⊃'BUILDRD' 'BUILDRL'
1 1
      RD←':LINK.OBJECT :INIT.+INITIAL :VALENCE.0 0 0'
      'ENVRD TEXT' BUILDRD 'ENVRD' 'FORTENV' RD
0                                                          (Note 3)
      RD←':LINK.OBJECT :INIT.FORTENV :RARG.G0 1 2(<I4 0)(I4 0)'
      'ADDRD TEXT' BUILDRD 'ADDRD' 'ADD' RD
0
      'SUBRD TEXT' BUILDRD 'SUBRD' 'ADD' RD
0
      RL←'FORTENV ENVRD' 'ADD ADDRD' 'SUB SUBRD'
      'FORTRL TEXT' BUILDRL (⊂'FORTRL'),RL
0
      100 ⎕SVO 'CTL100'
2
      SOURCES←'(FORTRL FORTUPD ENVRD ADDRD SUBRD AP2VNL)'
|     LIBS←'(VSF2FORT)'
|     CTL100←⍎'AP2MP11L FORTLOAD(FORTRL)' SOURCES LIBS
      'FORTLOAD.FORTRL' 11 ⎕NA⊃'ADD' 'SUB'
1 1
      ADD 3 17
20
      SUB 15 6
9
```

NAMES Files Entries

```
:NICK.FORTENV
   :LOAD.FORTLOAD
   :MEMB.FORTRL
   :LINK.OBJECT
|  :INIT.+INITIAL
   :VALENCE.0 0 0
:NICK.ADD
   :LOAD.FORTLOAD
   :MEMB.FORTRL
   :LINK.OBJECT
   :INIT.FORTENV
   :RARG.(G0 1 2)(<I4 0)(I4 0)     (Note 3)
:NICK.SUB
   :LOAD.FORTLOAD
   :MEMB.FORTRL
   :LINK.OBJECT
   :INIT.FORTENV
   :RARG.(G0 1 2)(<I4 0)(I4 0)     (Note 3)
```

Routine List and Link-Editing Non-Self-Describing Routines

```
       3 11 ⎕NA 'BUILDRL'
1 1
       RL←'FORTENV' 'ADD' 'SUB'
       'FORTRL TEXT' BUILDRL (⊂'FORTRL'),RL
0
       100 ⎕SVO 'CTL100'
2
       SOURCES←'(FORTRL FORTUPD AP2VNL)'
|      LIBS←'(VSF2FORT)'
|      CTL100←⍕'AP2MP11L FORTLOAD(FORTRL)' SOURCES LIBS
       '(FORTNAMES)' 11 ⎕NA⊃'ADD' 'SUB'
1 1
       ADD 3 17
20
       SUB 15 6
9
```

Link-Editing Examples

Link-Editing on TSO using a CLIST

```
PROC 0
CONTROL MAIN PROMPT ASIS NOFLUSH
ALLOC FI(OBJDECKS) SHR REUSE DA('USERID.LIB.OBJ')
LINK *                      +
   LOAD('USERID.LIB.LOAD')  +
   PRINT(*)                 +
   LIB('lang.link.lib'      +
       'APL2.SAP2LMDS'      +
      )                     +
   AMODE(31)                +
   RMODE(ANY)
DATA PROMPT
    INCLUDE OBJDECKS(rlname)  Routine list
    INCLUDE OBJDECKS(name)    Compile object file name
    INCLUDE OBJDECKS(rdname)  Routine description
    ENTRY   rlname            Main entry point of member (frequently a routine list)
    NAME    member(R)         Name of member

ENDDATA
FREE FI(OBJDECKS)
EXIT CODE(&MAXCC)
```

Link-Editing on TSO using JCL

```
//LKED     EXEC PGM=IEWL,PARM='XREF,LIST,LET'
//OBJ      DD   DUMMY
//SYSLIB   DD   DISP=SHR,DSN=lang.link.lib
//         DD   DISP=SHR,DSN=APL2.SAP2LMDS
//SYSLIN   DD   *
    MODE AMODE(31),RMODE(ANY) APL2 expects this Amode and Rmode
    INCLUDE OBJDECKS(rlname)  Routine list
    INCLUDE OBJDECKS(name)    Compile object file name
    INCLUDE OBJDECKS(rdname)  Routine description
    ENTRY   rlname            Main entry point of member (frequently a routine list)
    NAME    member(R)         Name of member
//SYSLMOD  DD   DISP=OLD,DSN=USERID.LIB.LOAD
//SYSUT1   DD   UNIT=SYSDA,SPACE=(CYL,(2,1))
//SYSPRINT DD   SYSOUT=*
//OBJDECKS DD   DISP=SHR,DSN=USERID.LIB.OBJ
```

Link-Editing on CMS

```
FILEDEF SYSLIB CLEAR
FILEDEF SYSLIB DISK lib1 TXTLIB * (PERM CONCAT
FILEDEF SYSLIB DISK lib2 TXTLIB * (PERM CONCAT
  ...
FILEDEF SYSLIB DISK libn TXTLIB * (PERM CONCAT
GLOBAL  MACLIB lib1 lib2 ... libn
FILEDEF SYSLMOD DISK library LOADLIB A (RECFM U
LKED textname (NAME rlname LIBE library LIST MAP RENT XREF AMODE(31) RMODE(ANY)
```

Where:

lib1-libn are filenames of language TXTLIB files

library is the filename of the LOADLIB file where the output member should be placed.

textname is the filename of a TEXT file into which all the object files to be link-edited have been copied.

rlname is the name of the main entry point of the member (frequently the name of a routine list.)

Generating a MODULE on CMS

```
GLOBAL TXTLIB lib1 lib2 ... libn
LOAD textname (RLDSAVE RESET rlname
GENMOD module (AMODE 31 RMODE ANY FROM rlname
```

Where:

lib1-libn are filenames of language TXTLIB files

textname is the filename of a TEXT file into which all the object files from which the module is generated have been copied.

rlname is the name of the main entry point of the member (frequently the name of a routine list.)

module is the filename of the MODULE file to be generated.

Appendix H. Summary of Terminal Information for APL2 Tasks

This appendix explains how to accomplish key APL2 tasks on the following IBM terminals:

IBM 2741 Communication Terminal
IBM 3270 Information Display System

For additional information, refer to the operator's guide for your terminal.

IBM 2741 Communication Terminal

For TSO, the IBM 2741 Communication Terminal (Figure 113 and Figure 114) requires a connection through the Virtual Telecommunications Access Method (VTAM/NTO).

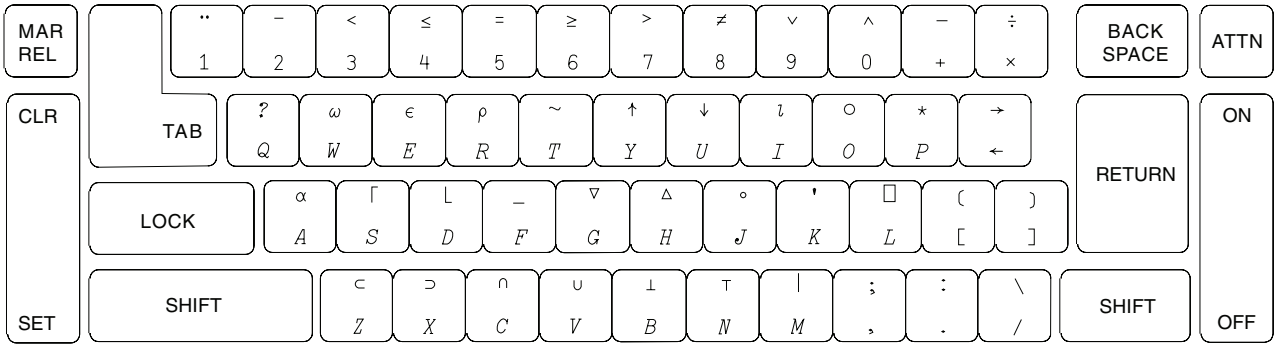


Figure 113. The IBM 2741 Communication Terminal Keyboard

Figure 114 (Page 1 of 2). Use of APL2 with the IBM 2741 Communication Terminal

To	Action
Switch to APL Mode and Produce APL Characters	On the IBM 2741 Communication Terminal, you can print the APL characters by placing on your terminal APL Selectric* typing element 987 (for correspondence IBM 2741s) or 988 (for EBCD or BCD IBM 2741s). On the IBM 2741, you can produce overstrike characters by typing one character, backspacing, and then typing the second character.
Produce the new APL2 Characters	Produce the new APL2 characters as you would any of the overstrike characters.
Enter data	Enter a line of input and press the RETURN key.

Figure 114 (Page 2 of 2). Use of APL2 with the IBM 2741 Communication Terminal

To	Action

Correct an Error	You can use the ATTN key to correct errors that you discover before pressing the RETURN key. Position the type element to the point on the line at which you want to begin retyping, and press the ATTN key. APL2 types a caret ∨ to mark the indicated position and places the typing element below the caret:
	`A←12 34 56 68`
	`∨`
	Type the remaining portion of the line, beginning with the character directly below the caret.
	`A←12 34 56 68`
	`∨`
	`3 55 68`
	Any character that you type to the left of the caret may create an overstrike character.
	You can use the ATTN key as often as necessary to correct a line. After you press the RETURN key, APL2 accepts the line of input.

Correct an error (continued)	Instead of using the ATTN key to correct characters to the right of the type element, you can intentionally create an *ENTRY ERROR*, which enables you to simply correct each erroneous character.
	To create an *ENTRY ERROR*, backspace then type over each character that you want to correct so that you form invalid characters. APL2 displays the message *ENTRY ERROR* and prompts you to replace the characters.
	For example, if you enter the _ symbol over the ÷ symbol in the expression `365×24÷60×60`, APL2 repeats the line that you typed but replaces the invalid character with a blank.
	`365×24÷60×60`
	ENTRY ERROR
	`365×24 60×60`
	`↑`
	Type element waits here
	for a new entry
	The type element is positioned at the leftmost invalid character. Retype a valid character into each blank position. Then, press the RETURN key.

Interrupt ☐ and ⍞ input	When APL2 gives you a ☐ or ⍞ prompt for input, type *O*, backspace, *U*, backspace, *T*, and press the RETURN key. APL2 treats this action as a strong interrupt.
Interrupt execution	**Attention:** Press the ATTN key once. APL2 interrupts execution after the current statement or function line has been executed. **Interrupt:** Press the ATTN key twice. (Wait for the type ball to stop wiggling before pressing ATTN the second time.) APL2 interrupts execution as soon as the current APL2 primitive operation has finished executing. **System Error under TSO:** If APL2 does not respond to **Interrupt**, pressing ATTN a third time may cause a message to be displayed that indicates the part of APL2 that is executing. Pressing ATTN again may cause that process to terminate. This action may result in a *SYSTEM ERROR* or a loss of an auxiliary processor. **Abrupt Termination of APL2 under TSO:** If you continue to press the ATTN key to interrupt execution, APL2 may temporarily abandon execution and place you in the TSO READY state. Depending on your method of terminal connection, you may see either the message READY or the characters `⍴∈⍺⌊↑`. If you **do not** want to abandon APL2 after you see the message, press only the RETURN key to give control to APL2. If you type any characters, you abruptly exit APL2 after the message is displayed. A *CONTINUE* workspace is saved if possible. **Abrupt Termination of APL2 under CMS:** If you continue to press the ATTN key, you may cause execution to be abandoned. If execution is abandoned, you are placed in CP mode (the letters CP READ appear in the bottom right-hand portion of the screen). To return to APL2, enter BEGIN or B.

IBM 3270 Information Display System

Figure 115, Figure 116, and Figure 117 illustrate typical 3270 family keyboards. Figure 118 describes common tasks and how to perform them using the IBM 3270 Information Display System.

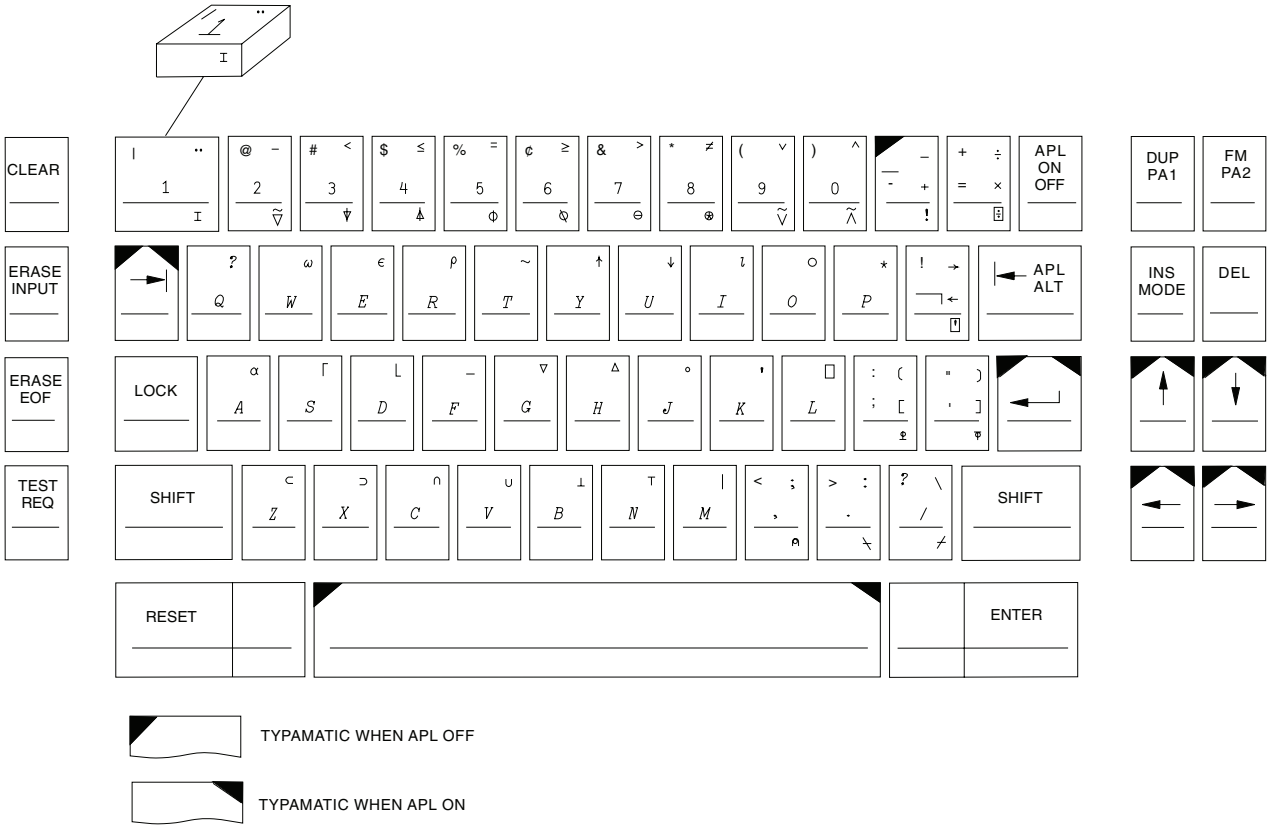


Figure 115. IBM 3277 APL Keyboard

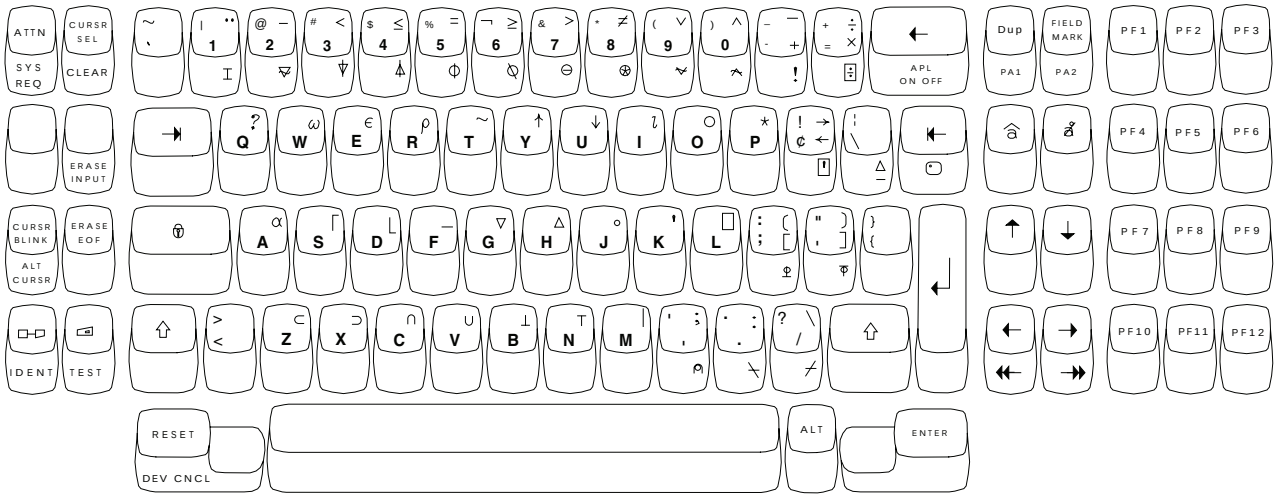


Figure 116. IBM 3278/3279 APL Keyboard

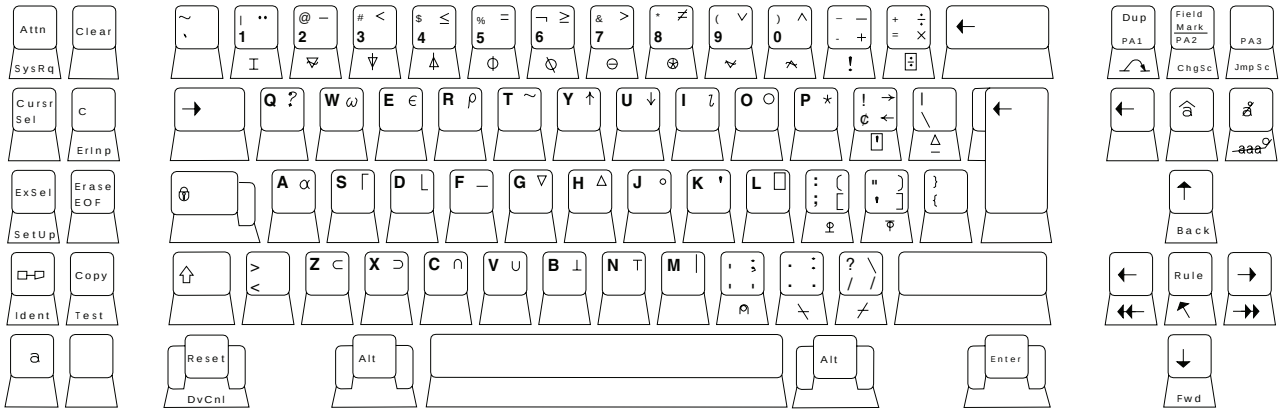


Figure 117. IBM 3290 APL Keyboard

Figure 118 (Page 1 of 2). Using APL2 with the IBM 3270 Information Display System

To	Action
Switch to APL Mode and Produce APL Characters	The keyboard contains APL keys if your terminal has the DATA-ANALYSIS feature (IBM 3277) or APL/TEXT feature (for example, IBM 3278, 3279, or 3290). To switch to APL mode on the 3277, press the APL ON/OFF key. To test if you are in APL mode, hold down the shift key while pressing the 6 key. If a ≥ appears, you are in APL mode. To switch to APL mode on a terminal with the APL/TEXT feature, hold down the ALT key and press the APL ON/OFF key. The letters APL appear at the bottom of the screen to let you know that you are in APL mode. **Note:** On some terminals (such as the 3277GA), display of the first 64 elements of $\square AV$ (X'00' through X'3F') gives unpredictable results.
Produce the new APL2 Characters	Some 3270s have keys to enter them directly. On others, you must use the APL2 $)PBS$ system command to enter them. For further information, see *APL2 Programming: Language Reference*.
Enter data	Type in your input and press the ENTER key.
Correct an error **before** it has been input	Move the cursor to the error by using the cursor-movement keys (↑ ↓ ← →). Replace the error with new characters. To insert or delete characters on a line, you may need to use the insert or the delete key. If a line is full, you cannot insert characters. If blanks follow the line, use the ERASE EOF key to delete those blanks before you insert characters.
Correct an error **after** it has been input	**With APL Session Manager:** Move the cursor to any input or output line. Replace the error with new characters and press the ENTER key. The old line remains unchanged and the resulting new line displays as the next line of input. **CMS Without APL Session Manager:** Use VM RETRIEVE facility, which can be assigned to a PA or function key. **TSO APL Without Session Manager:** Move the cursor to the line to be changed. Correct the characters and press the ENTER key. The line remains as you changed it. The changes you made do not cause a second line to be displayed.
Interrupt □ and ⍞ input	**CMS**:If you are using the session manager, press the PA2 key. If you are not using the session manager, press the PA2 key and then the ENTER key. **TSO**: If you are using the session manager, press the PA1 key. If you are not using the session manager, press the PA2 key.

Figure 118 (Page 2 of 2). Using APL2 with the IBM 3270 Information Display System

To	Action
Interrupt execution	**Attention:** An *attention* signals an interrupt after execution of the current statement or function line. Under CMS with the session manager, press the RESET key and then the PA2 key. Without the session manager under CMS, press the ENTER key. You can also press the PA1 key twice; the first PA1 displays the CP READ message; the second PA1 returns you to APL2. Under TSO, press the ATTN key or PA1 key once. **Interrupt:** An *interrupt* signals an interrupt as soon as possible. Under CMS with the session manager, press the RESET key and then the PA2 key. Repeat the procedure. Without the session manager, press the ENTER key, the RESET key, and then the ENTER key. Under TSO, press the ATTN key twice or the PA1 key twice. **Note:** The keys for interrupting execution work under most terminal configurations. However, certain configurations may reserve particular keys mentioned here for another use. In such cases, see your system administrator or experiment with the ATTN, PA1, and PA2 keys to discover which ones work for your particular configuration. **System Error under TSO:** If APL2 does not respond to **Interrupt**, pressing ATTN or PA1 a third time may cause a message to be displayed that indicates the part of APL2 that is executing. Pressing ATTN or PA1 again may cause that process to terminate. This action may result in a *SYSTEM ERROR* or a loss of an auxiliary processor. **Abrupt Termination of APL2 under TSO:** If you continue to press the ATTN key or PA1 key to interrupt execution, APL2 may temporarily abandon execution and place you in the TSO READY state. Depending on your method of terminal connection, you may see either the message READY or the characters ⍴∈⍺⌊↑.
Interrupt execution (continued)	If you **do not** want to abandon APL2 after you see the message, press only the RETURN key to give control to APL2. If you type any characters, you abruptly exit APL2 after the message is displayed. A *CONTINUE* workspace is saved if possible. **Abrupt Termination of APL2 under CMS:** If you continue to press the ATTN key or PA1 key, you may cause execution to be abandoned. If execution is abandoned, you are placed in CP mode (the letters CP READ appear in the bottom right-hand portion of the screen). To return to APL2, enter BEGIN or B.
To receive CP message	**Under CMS** with session manager on, or when using AP 126, CP messages may not be displayed immediately, but are displayed at the end of the session, or when you press the PA1 key.

Appendix I. Printer Fonts Supplied with APL2

APL2 includes an All-Points-Addressable (APA) font suitable for use on all 38xx APA printers (this excludes 3800 model 1). This is an italic font including all characters defined on the APL2 EBCDIC code page. It is available in point sizes 6 through 12, 14, 16, 18, 20, and 24. The typeface name is `'APL2 DOCUMENT FONT'`, and the codepage name is `T1200293`. The font is formally described as medium italic, normal width, although the alphabetic characters are actually quite light, to aid in distinguishing them from function symbols. This font can be used with Document Composition Facility (DCF), the GDDM family-4 printer support, or BookMaster*.

A number of older fonts are also supplied with the product, but may or may not have been made available at your installation. These fonts are for use on 3800 model 1 and model 3 printers only (not 3812, 3820, etc.) and are for high volume line oriented output. The font names given are to be used in the `CHARS` parameter of the CP `SPOOL` command, the TSO `ALLOCATE` command, `OUTPUT` JCL statements, or Script command options.

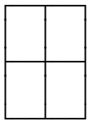
Font name	Uppercase Alphabet	Lowercase Alphabet	Underbar Alphabet	Box chars	Blot chars	Super-script
APL	Italic	Italic	Italic	yes	yes	yes
AD10	Serif	Serif	Italic	yes		
AD12	Serif	Serif	Italic	yes		
AG10	Sanserif	Sanserif	Sanserif	yes	yes	yes
AG12	Sanserif	Sanserif	Sanserif	yes	yes	yes
AG15	Sanserif	Sanserif	Sanserif	yes	yes	yes
AI10	Italic	none	none			
AI12	Italic	none	none			
AT10	Serif	Serif	Italic *	yes	yes	yes
AT12	Serif	Serif	Italic *	yes	yes	yes

Notes:

1. The “Serif” and “Sanserif” alphabetic characters are vertical (nonitalic) letters that respectively include or do not include small bars at the ends of the character strokes.
2. None of these fonts contain the three new characters ◊, ⊢, or ¬.
3. The character size for the font named APL is 10-pitch. The last two characters of the other font names indicate their pitch.
4. The code points that would normally display as underbarred produce instead uppercase italic letters.
5. The last three columns refer to code points below X'40'. These include:

   box chars Laid out as in this diagram, with hex code points on the left:

   ```
   1C 2D 2D 3B 2D 2D 1B
   1A        1A        1A
   3D 2D 2D 2C 2D 2D 3F
   1A        1A        1A
   1E 2D 2D 3E 2D 2D 1F
   ```

   Blot chars X'18' is a full cell blot; X'08' and X'28' are top half-cell and bottom half-cell respectively; X'17' and X'19' are left half-cell and right half-cell respectively.

Superscript X'20' through X'23' are ⁽⁾⁺⁻ while X'30' through X'39' are ⁰ through ⁹. Fonts containing superscript characters also contain § and ¶ at X'2B' and X'3A' respectively.

Bibliography

APL2 Publications

- *APL2 Fact Sheet,* GH21-1090
- *APL2/370 Application Environment Licensed Program Specifications,* GH21-1063
- *APL2/370 Licensed Program Specifications,* GH21-1070
- *APL2 for AIX/6000 Licensed Program Specifications,* GC23-3058
- *APL2 for Sun Solaris Licensed Program Specifications,* GC26-3359
- *APL2/370 Installation and Customization under CMS,* SH21-1062
- *APL2/370 Installation and Customization under TSO,* SH21-1055
- *APL2 Migration Guide,* SH21-1069
- *APL2 Programming: Language Reference,* SH21-1061
- *APL2/370 Programming: Processor Interface Reference,* SH21-1058
- *APL2 Reference Summary,* SX26-3999
- *APL2 Programming: An Introduction to APL2,* SH21-1073
- *APL2 for AIX/6000: User's Guide,* SC23-3051
- *APL2 for OS/2: User's Guide,* SH21-1091
- *APL2 for Sun Solaris: User's Guide,* SH21-1092
- *APL2 for the IBM PC: User's Guide,* SC33-0600
- *APL2 GRAPHPAK: User's Guide and Reference,* SH21-1074
- *APL2 Programming: Using Structured Query Language,* SH21-1057
- *APL2/370 Programming: Using the Supplied Routines,* SH21-1056
- *APL2/370 Programming: System Services Reference,* SH21-1054
- *APL2/370 Diagnosis Guide,* LY27-9601
- *APL2/370 Messages and Codes,* SH21-1059

Other Books You Might Need

The following books might also be of use, and can be ordered from IBM.

DB2

- *IBM DATABASE 2 Application Programming and SQL Guide*, SC26-4377
- *IBM DATABASE 2 Command and Utility Reference*, SC26-4378
- *IBM DATABASE 2 Diagnosis Guide and Reference*, LY27-9536
- *IBM DATABASE 2 General Information*, GC26-4373
- *IBM DATABASE 2 Messages and Codes*, SC26-4379
- *IBM DATABASE 2 Reference Summary*, SX26-3771

GDDM

- *GDDM Application Programming Guide* SC33-0867
- *GDDM Base Application Programming Reference*, SC33-0868
- *GDDM General Information*, GC33-0866
- *GDDM Messages*, SC33-0869
- *GDDM System Customization and Administration*, SC33-0871
- *GDDM User's Guide*, SC33-0875
- *GDDM Diagnosis*, SC33-0870
- *GDDM Using the Image Symbol Editor*, SC33-0920
- *GDDM-PGF Programming Reference*, SC33-0333
- *GDDM-PGF Diagnosis Guide and Reference*, LC33-0104
- *GDDM-PGF Interactive Chart Utility*, SC33-0328
- *GDDM-PGF Vector Symbol Editor*, SC33-0330

MVS

- *Operator's Library: OS/VS2 MVS System Commands*, GC38-0229
- *OS/VS Message Library: VS2 System Codes*, GC38-1008
- *OS/VS Message Library: VS2 System Messages*, GC38-1002
- *OS/VS Virtual Storage Access Method (VSAM) Options for Advanced Applications*, GC26-3819
- *OS/VS Virtual Storage Access Method (VSAM) Programming Guide*, GC26-3838
- *OS/VS2 Access Method Services*, GC26-3841
- *OS/VS2 MVS Data Management Macro Instructions*, GC26-3873
- *OS/VS2 MVS Supervisor Services and Macro Instructions*, GC28-0683
- *OS/VS2 MVS System Programming Library: Initialization and Tuning Guide*, GC28-1029
- *OS/VS2 System Management Facilities (SMF)*, GC28-0706

- *OS/VS2 System Programming Library: Data Management*, GC26-3830
- *OS/VS2 System Programming Library: Debugging Handbook, Vol. 1-3*, GC28-1047
- *OS/VS2 System Programming Library: MVS Diagnostic Techniques*, GC28-0725
- *OS/VS2 System Programming Library: Service Aids*, GC28-0674

MVS/DFP*

- *Integrated Catalog Administration: Access Method Services for MVS/DFP*, SC26-4500
- *VSAM Catalog Administration: Access Method Services for MVS/DFP*, SC26-4501
- *Data Administration: Macro Instruction Reference for MVS/DFP*, SC26-4506
- *Linkage Editor and Loader for MVS/DFP*, SC26-4510
- *MVS/ESA VSAM Administration: Macro Instruction Reference for MVS/DFP*, SC26-4517
- *MVS/DFP: Access Method Services for the Integrated Catalog Facility*, SC26-4562
- *MVS/DFP: Linkage Editor and Loader*, SC26-4564
- *MVS/DFP: Macro Instructions for VSAM Data Sets*, SC26-4569
- *MVS/DFP: Access Method Services for VSAM Catalogs*, SC26-4570

MVS/ESA

- *MVS/ESA Initialization and Tuning Reference*, GC28-1635
- *MVS/ESA Appl Dev: Callable Services for High-Level Languages*, GC28-1639
- *MVS/ESA System Messages Volume 1*, GC28-1656
- *MVS/ESA System Messages Volume 2*, GC28-1657
- *MVS/ESA System Messages Volume 3*, GC28-1658
- *MVS/ESA System Codes*, GC28-1664
- *MVS/ESA Routing and Descriptor Codes*, GC28-1666
- *MVS/ESA Problem Determination Guide*, GC28-1667
- *MVS/ESA Service Aids*, GC28-1669
- *MVS/ESA JCL Reference*, GC28-1829

MVS/XA

- *MVS/Extended Architecture Access Method Services Reference for the Integrated Catalog Facility*, GC26-4019
- *MVS/Extended Architecture Access Method Services Reference for VSAM Catalogs*, GC26-4075
- *MVS/Extended Architecture Data Management Macro Instructions*, GC26-4014
- *MVS/Extended Architecture Debugging Handbook, Vols. 1-5*, LC28-1164
- *MVS/Extended Architecture Diagnostic Techniques*, LY28-1199
- *MVS/Extended Architecture Message Library: System Codes*, GC28-1157
- *MVS/Extended Architecture Message Library: System Messages*, GC28-1156
- *MVS/Extended Architecture Operations: System Commands*, GC28-1206
- *MVS/Extended Architecture Supervisor Services and Macro Instructions*, GC28-1154
- *MVS/Extended Architecture System Programming Library: Data Management*, GC26-4010
- *MVS/Extended Architecture System Programming Library: Initialization and Tuning*, GC28-1149
- *MVS/Extended Architecture System Programming Library: Service Aids*, GC28-1159
- *MVS/Extended Architecture System Programming Library: System Management Facilities*, GC28-1153
- *MVS/Extended Architecture System Programming Library: 31-Bit Addressing*, GC28-1158
- *MVS/Extended Architecture VSAM Reference*, GC26-4016

SMP/E

- *System Modification Program Extended (SMP/E): Messages and Codes* , GC28-1108
- *System Modification Program Extended (SMP/E): Reference*, SC28-1107

SQL/DS

- *SQL/Data System Application Programming*, SH09-8086
- *SQL/Data System Messages and Codes*, SH09-8079
- *SQL/Data System Diagnosis Guide and Reference*, SH09-8081
- *SQL/Data System Database Service Utility*, SH09-8088

TCP/IP

- *TCP/IP Version 2.0 for VM: Programmer's Reference*, SC31-6084
- *TCP/IP for VM: User's Guide*, SC31-6081
- *TCP/IP Version 2.0 for MVS: Programmer's Reference*, SC09-1261
- *TCP/IP for MVS: User's Guide*, SC09-1255
- *TCP/IP for MVS: Programmer's Reference Manual*, SC31-6087
- *TCP/IP for MVS Messages and Codes*, SC31-6142
- *TCP/IP for VM Messages and Codes*, SC31-6151

TSO

- *MVS/XA TSO/E Command Language Reference*, SC28-1134
- *MVS/XA TSO/E Guide to Writing a Terminal Monitor Program or a Command Processor*, SC28-1136
- *TSO/E Command Language Reference*, SC28-1307
- *TSO/E Programming Services*, SC28-1364
- *TSO/E Version 2 Programming Services*, SC28-1875
- *TSO/E Version 2 Command Reference*, SC28-1881

VM/ESA*

- *VM/ESA CMS User's Guide*, SC24-5460
- *VM/ESA Procedures Language VM/REXX Reference*, SC24-5466
- *VM/ESA CMS User's Guide*, SC24-5460
- *VM/ESA CMS Command Reference*, SC24-5461
- *VM/ESA CP General User Command Reference for 370*, SC24-5433
- *VM/ESA System Messages and Codes for 370*, SC24-5437
- *VM/ESA CP Command and Utility Reference*, SC24-5519
- *VM/ESA System Messages and Codes Reference*, SC24-5529

VM/SP

- *VM/SP Data Areas and Control Block Logic*, LY20-0891
- *VM/SP System Logic and Problem Determination Guide Volume 1 (CP)*, LY20-0892
- *VM/SP System Logic and Problem Determination Guide Volume 2 (CMS)*, LY20-0893
- *VM/SP Service Routines Program Logic*, LY20-0890
- *VM/SP System Programmer's Guide*, SC19-6203
- *VM/SP Messages and Codes*, SC19-6204
- *VM/SP CMS Command and Macro Reference*, SC19-6209
- *VM/SP CMS User's Guide*, SC19-6210

VM/XA*

- *VM/XA System Product CMS User's Guide*, SC23-0356
- *VM/XA System Product CP Command Reference*, SC23-0358
- *VM/XA System Product CMS Command Reference*, SC23-0354
- *VM/XA System Product System Product Interpreter Reference*, SC23-0374
- *VM/XA System Product System Messages and Codes Reference*, SC23-0376

Index

Special Characters

:DESC. 309
:ENTRY. 307
:INIT. 308
:LARG. 309
:LINK. 307, 321
:LOAD. 307
:MEMB. 307
:NICK. 307
:PARM. 309
:RARG. 309
:RSLT. 309
*:TAG.*value 307
:TASKLIB. 309
:TIME. 309
:VALENCE. 309
), use in specification of TSO AP 100 built-in commands 116
)*EDITOR* 74
)*HOST* system command
  CMS 107
  TSO 112
)*IN* system command 66, 73
)*MORE* system command 18
)*OUT* system command 66, 73
)*PBS* system command 37
)*QUOTA* system command
  maximum number of shared variables 28
  size of active workspace 32
  size of shared storage 26
Δ*EXEC* function 285
Δ*F ω* function 284
Δ*FM* function 281, 282
Δ*FV* function 281, 282
□*AI* account information, first item of 21, 70
□*AT* attributes, variables
  written in CDR 191, 252
□*AV* atomic vector
  transformation of items after conversion 376
□*NA*
  syntax for non-APL programs 292
□*NA* name association 268
□*PW* printing width
  printing with initial value for batch APL2 jobs (CMS) 79
  session manager log 38
□*SVO* shared variable offer 105

Numerics

2741 communication terminal, APL2 operations on 394
3270 display terminal 207
3270 series display terminal, APL2 operations on 396
3277 model 2 18
3278 model 2 18
3290 display terminal, APL2 operations on 397
370 conversion option
  with AP 100 108

A

ABEND code 152, 153, 243, 254
ABENDs in external routines 324
accessing namespaces
  accessing objects 340
  creating namespaces 338
  NAMES files 342
  unexpected errors 349
  using namespaces 343
  workspace names 340
accounting information, TSO 116
active associations 272
active workspace 3
  default size 31, 32
  querying the address of 135
  querying the identification of (TSO) 125
  querying the size of 136
  WSSIZE invocation option 14, 31
  XA invocation option 32
ADMK772A device-token 18
ADMK782A device-token 18
ADMPRINT filetype, GDDM print files (CMS) 48
□*AI* account information, first item of 21, 70
AI built-in command, AP 100 (TSO) 116
AISIZE invocation option 11
ALLOCATE command, TSO sample issue using AP 100 113
altering normal APL2 debugging actions (DEBUG invocation option) 17
AP 100, Host System Command Processor
  CMS 107
    associated workspaces 107
    cancelling output 109
    communication procedure 108
    conversion options 108
    initial value 108
    return codes 110
    shared variable requirements 107

AP 100, Host System Command Processor *(continued)*
TSO 112
associated workspaces 112
built-in commands 116
built-in commands summary table 116
cancelling output 113
communication procedure 113
return codes 114
shared variable requirements 113
using to invoke a CLIST 125
AP 101
commands 129
AP 101, Alternate Input (Stack) Processor
input stack
size of (TSO) 11
when exiting APL2 33, 34
invoking APL2 from, INPUT invocation option 22
AP 102, Main Storage Access Processor 134
cautions 136
communication procedure 135
converting integer result to character 136
converting integer result to hexadecimal 136
return codes 137
service requests 135
shared variable requirements 134
AP 110, CMS File Processor 138
associated workspaces 138
blocking factor 142
cautions 143
communication procedure 140
example 143
conversion options 140
initial values 139
return codes 144
shared variable requirements 138
AP 111, QSAM File Processor 146
associated workspaces 146
cautions 150
communication procedure 148
example 150
conversion options 147
converting QSAM decimal return codes 152
initial values 147
return codes 151
shared variable requirements 146
AP 119, Socket Interface Processor 154
AP commands 168
starting 171
TCP/IP commands 158
AP 120, APL2 Session Manager Processor 179
communication procedure 180
example 181
return codes 181
shared variable requirements 179
AP 120, Session Manager Command Processor 179
AP 121, APL2 Data File Processor 182
associated workspaces 182
cautions 191
changing the file size 190
CMS 63, 139
commands 185
communication procedure 185
example 189
deleting a file 190
direct access processing 187
files 183
CMS 63
explicit close caution 192
file identification 183
TSO 68
mismatched variables 105
return codes 192
sequential processing 187
shared variable requirements 182
space requirements for storing APL2 variables 191
AP 123, VSAM File Processor 195
associated workspaces 195
cautions 203
commands 197
communication procedure 198
example 198
open requests and possible processing options 199
return codes 204
shared variable requirements 195
translate options 203
VSAM files 195
closing 199, 203
entry-sequenced 196
identification of 196
key-sequenced 197
relative-record 196
AP 124, Text Display Auxiliary Processor 207
commands 210
return codes 221
shared variable overview 207
AP 126, GDDM Processor 222
associated workspaces 222
commands 228
communication procedure 226
example 226
data paths 237
error diagnosis 241
GDDM calls
calls you cannot issue 228
CHART call 234
retrieving list of (sample function) 230
handling attentions 239
obtain hard copy of screen image 237
program product requirements 223
return code vector
examples 225, 226, 241
format 224

AP 126, GDDM Processor *(continued)*
  return codes 242
  shared variable requirements 223
  sharing pages with APL2
    session manager 238
  translation considerations
    migrating from VS APL 231
AP 127, SQL Processor 244
  commands 246
  communication procedure 245
  return codes 247
  shared variable requirements 245
AP 210, BDAM File Processor 248
  associated workspaces 248
  cautions 253
  communication procedure 249
  conversion options 250
  converting MVS error codes 254
  file processing procedure 251
    example 252
  file requirements 249
  formatting a BDAM file 249
  formatting procedure 250
  initial values 249
  return codes 253
  shared variable requirements 248
AP 211
  transferring files 95
AP 211, Object File Processor 255
  commands 255, 256, 257, 258, 259
  return codes 259
AP2EXIT EXEC (CMS), ending your APL2 session 32
AP2MP11L 328
AP2MP11M 328
AP2TNL 306
AP2VNL 306
AP2XCMAP 303
APL arrays as REXX programs 285
APL files 355
APL, used in specification of AP 100 (TSO) built-in commands 116
APL2
  port server 169
  transfer of workspaces from TryAPL2 95
  using across systems 86
APL2 character set 37
APL2 command
  name of 5, 8
  overriding default invocation options 10
APL2 Data File Processor 182
  See also AP 121
APL2 data files 6, 63
APL2 editors 74
APL2 executor 3
APL2 files and data sets 381
APL2 interpreter 3
APL2 invocation options
  See invocation options
APL2 libraries
  See libraries, APL2
APL2 Licensed Program
  components of 3
  debugging (DEBUG invocation option) 17
  diagnosing problems
    (SYSDEBUG invocation option) 28
    (TRACE invocation option) 31
  discussion of 3
  national language in use, display of 40
  release level, display of 40
  support features 5
  system structure 3
APL2 port server 88
APL2 session manager
  See session manager
APL2/370
  deviations from APL2 366
APLDATA workspace
  with AP 121 182
APLIN file 382
APLPRINT file 382
APLWSV2, CMS default filetype for workspaces 65
APLX*nnnn* CMS file and TSO load module 47
APNAMES invocation option 12, 23
argument patterns for Processor 11 310
arguments, updating
  in place 313
  results 313
arguments, valence
  explicit results 317
  function valence 318
  operator valence 318
  valence 318
associated processors
  active associations 272
  description of 268
  environmental considerations 273
  external names 268
$\Box AT$ attributes, variables
  written in CDR 191, 252
ATTACH built-in command, AP 100 118
attention
  See terminal descriptions
ATTRIB command, TSO, sample issue using AP 100 113
authorization entries 90
auxiliary processors 4, 6
  accessing processors not automatically provided 12, 23
  availability of at your installation 6
  conversion options for 370
  diagnosing problems
    (SYSDEBUG invocation option) 28

auxiliary processors *(continued)*
diagnosing problems *(continued)*
(TRACE invocation option) 31
distributed with APL2 102
offering a shared variable 104
⎕*AV* atomic vector
transformation of items after conversion 376

B

batch processing of APL2 6, 78
CMS 78
input 78
output 79
passwords 79
TSO 80
input 80
output 80
without APL2 session manager 78
BDAM File Processor for TSO 248
See also AP 210, BDAM File Processor
BUILDRD 296
built-in commands, AP 100 (TSO only) 116
AI 116
ATTACH 118
CODE 118
DDI 118
DEGUG 120
DSI 119
EXEC 125
format of specifying 116
LIB 120
LIBS 121
NOMSG 121
QUIET 122
QUOTA 122
summary table of 116
TSO 126
USER 122
WSID 125
WSNAME 125
built-in functions 280

C

C/370 examples 386
C/370 results 316
C/370 scalar integer results 303
calculating length of APL2 variables 191, 252
calls, GDDM
See AP126, GDDM calls
cancelling output, AP 100
CP or CMS command 109
TSO command or CLIST 113
CASE 9, 14
CDR conversion option
calculating length of APL2 variables 191, 252
changes
summary of xvii
CLIST, TSO
ending APL2 33
executing from AP 100 125
invoking APL2 8
CMS EXEC, sample using AP 100 109
CMS File Processor 138
See also AP 110
CMS file status 284
CMS files
reading 281
writing 281
CMS system editors 75
CMS workspace
with AP 100 107
with AP 110 138
with AP 111 146
CMSBATCH facility 78
CODE built-in command, AP 100 (TSO) 118
CODE invocation option 15
COLUMN field, session manager control line 38
column major order 332
COLUMN session manager command 38, 44
command line, session manager screen 38
commands to host system
from AP 100
CMS 107
TSO 112
compiled programs 291
CONTINUE workspace
after forced termination 34
loading of 34
losing the contents or preventing loading 22
preventing loading, batch 78
continuing your terminal session after exiting APL2
CMS 32
TSO 34
control line, session manager screen 38
controlling invocation 82
conversion of atomic vector characters 376
conversion options
for auxiliary processors 370
transformation of ⎕*AV* 376
cooperative processing
discussion of 86
processor network identification 86
processor profile examples 91
processor profile structure 87
processor profile syntax 89
receiving a share offer 88
sending a share offer 88
using the port server 88

copy destination, with AP 126 237
COPY session manager command 46
  in session manager profile 49
copying lines from session log 46
cover functions, purpose of 6
CREATE 255
  commands
    CREATE 255
CTN external routine 278

D

data files 63
data paths, AP 126 237
data sets, TSO 383
date and time stamp display 15
DATEFORM invocation option 15
DB2 publications 401
DBCS conversion option 15, 84
DBCS invocation option 15
DDI built-in command, AP 100 (TSO) 118
ddname 307, 309
DEBUG built-in command, AP 100 (TSO) 120
DEBUG invocation option 17
debug options, querying and setting (TSO) 120
debugging APL2
  DEBUG invocation option 17
  SYSDEBUG invocation option 28
  TRACE invocation option 31
default library number (first item of ⎕*AI*) 21
default session manager profile 60
DEFINE command (Access Method Services), example
 to create VSAM library 69
defining libraries for CMS access of APL2-related
 files 63
:DESC. 309, 343
destination, COPY session manager command 47
  CMS 48
  TSO 49
  with AP 126 237
deviations
  in implementation 366
device-token, DSOPEN invocation option 18
DFP publications 402
diagnosing problems 28
disconnection of APL2
  CMS 33
  TSO 34
display of lines longer than screen width 38
display of output, preventing
  QUIET invocation option 24
  SUPPRESS session manager command 61
DISPLAY OFF session manager command, difference
 from SUPPRESS 62
display screen, session manager 37
DISPLAY session manager command 50
display, control of the screen, using AP 126 25
display, of APL2 character set 37
DOMAIN ERROR 287
DROP 256
  commands
    DROP 256
DSI built-in command, AP 100 (TSO) 119
DSOPEN invocation option 18

E

EBCD conversion option
  with AP 100 108
)*EDITOR* 74
EDITORs
  CMS 75
  TSO 75
ending your APL2 session
  CMS 32
  TSO 33
entering multiple lines of APL2 input 41
:ENTRY. 307, 343
entry-sequenced VSAM files 196
environment program 303
environment, obtaining information about (TSO) 122
environments 303, 323
ERASE 258
  commands
    ERASE 258
EXCLUDE invocation option 19
∆*EXEC* function 285
EXEC built-in command, AP 100 (TSO) 125
EXEC file 274
EXEC, CMS, sample execution using AP 100 109
EXECCOMM 279
executing a CLIST from within APL2, AP 100
 (TSO) 125
executing APL arrays as REXX programs 285
execution time libraries 332
executor, APL2 3
external function
  explicit result 313
  names 325
external names 7, 268
external routines
  ABENDs 324
  internal errors 324

F

∆*F* *ω* function 284
FILE filename, COPY session manager command
 (CMS) 48
FILEDEF CMS command, example issue using AP
 100 109, 150

files
APL2 data 6, 63, 183
BDAM 249
CMS 63, 138
MVS sequential 70
QSAM 146
VSAM 195
files as arrays, Processor 12 352
FIND session manager command 39, 53
example in user field 39
ΔFM function 281, 282
fonts
supplied with APL2 399
forced termination of APL2
CMS 33
TSO 34
formatting a BDAM file, AP 210 249, 250
FORTRAN
array ordering 332
linkage 321
FREESIZE invocation option 20, 191
FSC126 workspace
with AP 126 222
FSM workspace
with AP 126 222
full-screen interface, APL2 session manager 37
function keys
See program function keys
FUNCTION linkage 322
ΔFV function 281, 282

G

GDDM
ADMOPUT print utility (TSO) 49
ADMOPUV print utility (CMS) 48
ADMPRINT filetype (CMS) 48
alphanumerics defaults module 51
ASTYPE call 51
calls
See AP126, GDDM calls
controlling display of data 51
DSOPEN function 18
FREESIZE function 20
FSOPEN function 47
FSPCRT call, TYPE parameter 52
output from COPY session manager command 47
publications 401
requirements, use of session manager 36
GDDM Processor 222
See also AP 126, GDDM Processor
GDMX workspace 222
GET 258
commands
GET 258
GETLPORT 171
Graphical Data Display Manager
See GDDM
GRAPHPAK workspace
with AP 126 222

H

HELP session manager command 54
HEXDUMP function in *UTILITY* workspace 136
highlighting input/output 21
HILIGHT invocation option 21
HOLDING mode, session manager screen 40
)*HOST* system command
CMS 107
TSO 112
Host System Command Processor
See also AP 100
CMS 107
TSO 112

I

ID invocation option 21
identification entries 89
IMPCP setting (CMS) 109
IMPEX setting (CMS) 109
implementation limits 365
)*IN* system command 66, 73
in-storage log, session manager 55
:INIT. 308
input
to batch APL2
CMS 78
TSO 80
INPUT invocation option 22
INPUT 22
passing quoted strings from a CLIST 22
preventing loading of *CONTINUE* workspace 78
relationship to QUIET invocation option 24
INPUT mode, session manager screen 39
input/output area, session manager screen 40
interface management routines 301
interlock, shared variable
avoiding 105
returning to INPUT mode 40
internal errors in external routines 324
interpreter, APL2 3
interrupt
entering during RUNNING mode 40
using SUPPRESS command 61
interruption of APL2
CMS 33
TSO 34
invocation options 5
abbreviation of keywords 10

invocation options *(continued)*
AISIZE 11
APNAMES 12
CODE 15
DATEFORM 15
DBCS 15
DEBUG 17
DSOPEN 18
EXCLUDE 19
forced defaults 11
FREESIZE 20
HILIGHT 21
ID 21
LOADLIBS (TSO only) 23
NLT 24
online inquiry of 8
overriding default values 10
PROFILE 24
QUIET 24
SHRSIZE 26
SMAPL 27
summary table of 9
SVMAX 28
SYSDEBUG 28
TERMCODE 28
TRACE invocation option 31
WSSIZE 14, 31
XA 32
invoking a CLIST from within APL2, AP 100 (TSO) 125
invoking APL2 5, 8
APL2 command 8
controlling 82
requirements 5
various ways to 8
IRXEXCOM 279
ISPF 76, 122
IUCV paths and sockets 155

J

JCL, sample for batch APL2 80

K

key-sequenced VSAM files (AP 123) 197

L

:LARG. 309
LENGTH ERROR 287
LIB built-in command, AP 100 (TSO) 120
libraries
See workspaces
libraries, APL2
CMS 63
access 63, 65
libraries, APL2 *(continued)*
CMS *(continued)*
AP121 files 63
passwords 67
private 63, 65
public 65
session manager log 42, 63
transfer files 66
workspace names 65
contents of 6
creating 64, 274
number of 63
purpose of 3
TSO 67
access 69
AP 121 files 183
execute a TSO command as if from a CLIST with CONTROL NOMSG 121
private 68, 69, 70
project 68, 71
public 67, 68, 71
public and private, obtaining list of using AP 100 121
SAM library system 70
transfer files 73
VSAM library system 68
workspace names 71
library query 190
LIBS built-in command, AP 100 (TSO) 121
LIBTAB APL2 file, CMS
access 63, 65
contents of each record 63
creating your own 63
file identification 184
format of records 63
sample listing of 64
updating 64
limits, implementation 365
LINE field, session manager screen 38
LINE session manager command 38, 54
:LINK. 307, 321, 343
link-editing examples 392
link-editing external routines 327
link-editing namespaces 349
LIST 259
commands
LIST 259
LISTEN 172
listening port 170
getting 171
setting 171
:LOAD. 307, 342
loading session manager profile
at APL2 invocation 60
PROFILE LOAD command 59

LOADLIBS invocation option 23
and ATTACH built-in command 118
localizing shared variables 105
locating specified character strings in session log 53
LOG session manager command 55
log, session manager 41
logical terminal identifier, COPY session manager command (TSO) 49
losing the contents of *CONTINUE* workspace 22

M

Main Storage Access Processor 134
See also AP 102
main storage, querying contents 134
maximum limits 365
:MEMB. 307, 343
message number
See MORE
messages, session manager 62
mismatched variables 105
mode field, session manager screen 39
monitoring the APL2 systems environment 134
)*MORE* system command 18
MORE mode, session manager screen 40
MVS publications 401
MVS Sequential Data Sets
reading 282
writing 282
MVS/ESA publications 402
MVS/XA
main storage access processor 134
obtaining larger extended region size 31
publications 402

N

□*NA* name association 268
syntax for non-APL programs 292
name association failure 322
NAMES file 312, 318
NAMES files 342
namescope 337
namespaces 334
:NICK. 307, 342
NLT invocation option 24
NOMSG built-in command, AP 100 (TSO) 121

O

Object File Processor, AP 211 255
OBJECT linkage 321
OFF mode, session manager screen 40
offering a shared variable 104
)*OUT* system command 66
output
from batch APL2
CMS 79
TSO 80
preventing
QUIET invocation option 24
SUPPRESS session manager command 61
suppressed by Δ*EXEC* function 285
overriding normal APL2 debugging actions 17

P

PAGE session manager command 56
:PARM. 309
PARTITION function 278
password protection
CMS 67
TSO 69, 73
passwords, specifying
AP 121 files 184
AP 123 VSAM files 196
batch APL2 jobs 79
)*PBS* system command 37
PDSI 116, 121
permanent log, session manager 55
PFK session manager command 57
PL/I examples 388
port
APL2 port server 169
listening 170
port server
using for cross-system sharing 88
preventing display of output
QUIET invocation option 24
SUPPRESS session manager command 61
primary storage, querying contents of 135
printer fonts
supplied with APL2 399
private library 63, 65, 68, 69, 70
Processor 10
accessing REXX variables and constants 279
built-in functions 280
constructing argument to REXX function 277
EXECCOMM 279
executing APL arrays as REXX programs 285
IRXEXCOM 279
overview 274
PARTITION function 278
querying a CMS file status 284
querying an MVS data set status 285
reading and writing CMS files 281
reading and writing MVS sequential data sets 282
REXX return code 20040 287
unexpected errors 286
using REXX functions 275

Processor 11
  □*NA* syntax for non-APL programs 292
  argument patterns 310
  calling compiled programs 291
  environments 303
  external routines distributed with APL2 137
  interface management routines 301
  linkage conventions 321
  NAMES files 318
  non-APL routine description tag rules 320
  result patterns 314
  routine description tags 307
  routine descriptions 295
  routine lists 298
  search order 325
  unexpected errors 322
  updating arguments 313
Processor 12
  □*NA* syntax for 352
  APL files 355
  errors 359
  files as arrays 352
  primitive operations 354
  record-oriented files as external variables 356
processor network identification 86
processor profile
  authorization entries 90
  examples of 91
  identification entries 89
  structure 87
  syntax for 89
PROFILE invocation option 24, 36
PROFILE PREFIX 70, 73
PROFILE session manager command 58
profile-prefix 24
profile, session manager 36, 60
program function keys, PFK session manager command 57
Programmed Symbol Set (PSS) feature, and APL2 session manger 37
project library (TSO) 68, 71
protection, APL2 data
  CMS 67
  TSO 73
PSS feature
  See Programmed Symbol Set feature
PSSHUTD 170
public library 65, 67, 68, 71
□*PW* printing width
  printing with initial value for batch APL2 jobs (CMS) 79
  session manager log 38

Q

QSAM files
  accessing using AP 111 146
querying a CMS file status 284
querying an MVS data set status 285
QUIET built-in command, AP 100 (TSO) 122
QUIET invocation option 24
QUOTA built-in command, AP 100 (TSO) 122
)*QUOTA* system command
  maximum number of shared variables 28
  size of active workspace 32
  size of shared storage 26

R

RACF 71, 73
RANK ERROR 287
:RARG. 309
read pointer, AP 110 142
reading and writing CMS files 281
reading and writing MVS sequential data sets 282
reason codes, AP 126 225, 241
reconnecting APL2 after disconnection 33
record-oriented files as external variables 356
recovery, altering normal APL2 recovery actions 17
relative-record VSAM files (AP 123) 196
RELEASE 257
  commands
    RELEASE 257
RENAME 258
  commands
    RENAME 258
renumbering session log lines 56
repeat entry of session manager commands 39
representation length 311
representation type 311
result pattern 314
resuming an APL2 session
  CMS 33
  TSO 34
retracting a shared variable 105
return codes
  issued by AP 119 175
  issued by AP 120 181
  issued by AP 123 204
  issued by AP 124 221
  issued by AP 126 241
  issued by AP 127 247
  issued by AP 210 253
  issued by AP 211 259
reusing lines in the session log 41
REXX
  APL arrays as REXX programs 285
  CMS file status 284
  description of 274

REXX *(continued)*
return code 287
unexpected errors 286
REXX return code 20040 287
routine description tags 307
routine descriptions 295
routine lists 298
row major order 332
:RSLT.
See result pattern
RTA external routine 278
RUNNING mode, session manager screen 39
with AP 126 40
RX 276

S

SAM library system, TSO 70
querying the full-qualified name of a workspace 125
saving session manager profile 59
screen display control 25
scrolling session manager log 40, 54
searching for character strings in session manager log 53
security of APL2 data
CMS 67
TSO 73
sense bytes 152
server
APL2 port server 169
SERVPORT 171
session log 41
CMS 42, 63
in-storage log 55
permanent log 55
renumbering lines 56
saving 42
scrolling 42, 56
searching for strings 53
size of 42
TSO 43, 68
session manager 5, 36
commands
abbreviations 43
capabilities of 43
COLUMN 44
COPY 46
DISPLAY 50
entering 38
FIND 53
HELP 54
issuing using AP 120 179
LINE 54
LOG 55
PAGE 56
PFK 57
PROFILE 58
session manager *(continued)*
commands *(continued)*
repeat entry of 39
summary table of 44
SUPPRESS 61
copy destination and AP 126 237
excluding from batch APL2 78
features of 37
log 41
reusing lines 41
scrolling 42
messages 62
profile 36
CMS 24, 59
default 60
TSO 24, 59
requirements for using 27
screen 25
COLUMN field 38
command line 38
defining the upper left corner 52
illustration of 38
input/output area 40
LINE field 38
MODE field in control line 39
position of 37
size of 37, 52
suppressing display of 25
USER field 39
scrolling, LINE command 54
sharing pages with AP 126 238
specifying the use of (SMAPL invocation option) 36
with the line editor 37
Session Manager Command Processor, AP 120 179
SET 258
commands
SET 258
SETLPORT 171
share offer
receiving 88
sending 88
shared storage, querying size using AP 102, example 136
See also SHRSIZE invocation option
shared variable
offering to an auxiliary processor 104
shared variable interlock
See interlock, shared variable
shared variable interpreter interface 261
control signal codes 263
output 263
shared variable processor (SVP), purpose of 4
shared variables
concepts 279
localizing 105
maximum number of 28

shared variables *(continued)*
offer
purpose of 6
retraction 105
size of 26
system functions used with 280
SHRSIZE invocation option 26
shutting down the port server 170
size
of active workspace 14, 31, 32, 191
of shared variable storage 26
SMAPL invocation option 27, 78
SMP/E publications 402
socket application program interface (API) 154
sample session 172
Socket Interface Processor, AP 119 154
SQL Processor, AP 127 244
SQL workspace, description
with AP 127 244
SQL/DS publications 402
stack size (TSO), AP 101 11
starting AP 119
LISTEN 172
SERVPORT 171
TCPID 172
starting APL2
See invoking APL2
status bytes 152, 254
structure, of APL2 3
substasks, attaching to APL2 session (TSO) 118
summary of changes xvii
suppress messages from APL2 (TSO) 122
SUPPRESS session manager command 61
suppression of APL2 termination messages 24
surrogate names 325
SVMAX invocation option 28
SVP
See shared variable processor
syntax
processor profile 89
SYSDEBUG invocation option 28
SYSEXEC 276
SYSPROC 276
system editors
restrictions 74, 277
under CMS 75
under TSO 75
system error 289
system limits 365
system structure, APL2 3
illustration of 4

T

T translate option, AP 123 203
T1 translate option, AP 123 203
T2 translate option, AP 123 203
*:TAG.*value 307
:TASKLIB. 309
TASKLIB, specifying for modules named in ATTACH built-in command 23
TCP/IP commands 158
TCP/IP publications 402
TCPID 172
TERMCODE
invocation option 28
TSO 33
terminal device codes (TSO), table of 30
terminal type, identifying to APL2
DSOPEN invocation option 18, 28
TERMCODE or CODE invocation option 30
table of available codes 30
terminating an APL2 session
CMS 32
TSO 33
testing an APL2 APAR fix 23
3270 series display terminal 396
370 conversion option
with AP 100 108
:TIME. 309
time and date stamp (DATEFORM invocation option) 15
TRACE invocation option 31
transfer files
CMS 66
moving between systems 94
TSO 73
transferring files 95
transferring workspaces 94
translate options, AP 123 203
translation of output with COPY session manager command 47
TryAPL2
transfer of workspaces to workstations 95
TSO built-in command, AP 100 (TSO) 126
TSO system editors 75
TSO workspace
with AP 100 112
with AP 111 146
with AP 210 248
TSO/E publications 402
2741 communication terminal, APL2 operations on 394

U

USE 257
commands
USE 257
USER built-in command, AP 100 (TSO) 122

user field, session manager screen 39
user-to-user shared variable communication, ID invocation option 21
UTILITY workspace
  with AP 102 136
  with AP 123 195

V

:VALENCE. 309
valence error 287
valence of non-APL routines 317
VAPLFILE workspace
  with AP 121 182
variables, APL2, stored
  outside active workspace 182
variables, shared
  See shared variables
VM error messages 287
VM/ESA publications 403
VM/SP publications 403
VM/XA publications 403
VMAPLCF, default filetype for AP 110 files (CMS) 139, 140
VS FORTRAN examples 390
VSAM File Processor 195
VSAM files, AP 123 195
  entry-sequenced 196
  key-sequenced 197
  relative-record 196
VSAM libraries, TSO
  creating 68
  for APL2 data files 182, 183
VSAM library system, TSO 68, 125
  querying the full-qualified name of a library 125
VSAMDATA workspace
  with AP 123 195
VSAPLFL, filetype for AP 121
  files (CMS) 184
VSAPLPR
  CMS 59
  TSO 59

W

workspace names, TSO
  listing using AP 100 120
  querying full-qualified name 125
workspace, active
  See active workspace
workspaces
  See also libraries
  CMS 63, 65
  migration from TryAPL2 95
  transferring between APL2 systems 94
  TSO
    SAM library system 70
workspaces *(continued)*
  TSO *(continued)*
    VSAM library system 68
workspaces distributed with APL2
  contents of 7
  purpose of 6
WRAP specification, in session manager COLUMN field 38
WSID built-in command, AP 100 (TSO) 125
WSNAME built-in command, AP 100 (TSO) 125
WSSIZE invocation option 14, 31, 191

X

XA 9
XA invocation option 32
XEDIT 75

We'd Like to Hear from You

APL2 Programming:
System Services Reference
Version 2 Release 2

Publication No. SH21-1054-01

Please use one of the following ways to send us your comments about this book:

- Mail—Use the Readers' Comments form on the next page. If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Electronic mail—Use this Internet ID:
  - Internet: comments@us.ibm.com

  Be sure to include the following with your comments:
  - Title and publication number of this book
  - Your name, address, and telephone number if you would like a reply

Your comments should pertain only to the information in this book and the way the information is presented. To request additional publications, or to comment on other IBM information or the function of IBM products, please give your comments to your IBM representative or to your IBM authorized remarketer.

IBM may use or distribute your comments without obligation.

Readers' Comments

APL2 Programming:
System Services Reference
Version 2 Release 2

Publication No. SH21-1054-01

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐
Grammatically correct and consistent	☐	☐	☐	☐	☐
Graphically well designed	☐	☐	☐	☐	☐
Overall satisfaction	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

May we contact you to discuss your comments? ☐ Yes ☐ No

Name

Address

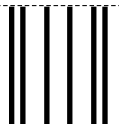
Company or Organization

Phone No.

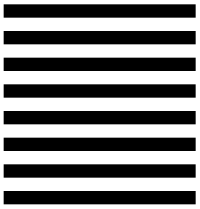
Readers' Comments
SH21-1054-01

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

APL Products and Services
IBM Silicon Valley Laboratory, H36/F40
555 Bailey Avenue
San Jose, CA
U.S.A. 95141-9989

Fold and Tape **Please do not staple** Fold and Tape

SH21-1054-01

Cut or Fold
Along Line

File Number: S370-40
Program Number: 5688-228

Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

The APL2 Library

GH21-1090 APL2 Family of Products (fact sheet)
SH21-1073 APL2 Programming: An Introduction to APL2
SH21-1061 APL2 Programming: Language Reference
SX26-3999 APL2 Reference Summary
SH21-1074 APL2 GRAPHPAK: User's Guide and Reference
SH21-1057 APL2 Programming: Using Structured Query Language
SH21-1069 APL2 Migration Guide
SC33-0600 APL2 for the IBM PC: User's Guide
SC33-0601 APL2 for the IBM PC: Reference Summary
SC33-0851 APL2 for the IBM PC: Reference Card
SH21-1091 APL2 for OS/2: User's Guide
GC23-3058 APL2 for AIX/6000 Licensed Program Specifications
SC23-3051 APL2 for AIX/6000: User's Guide
GC26-3359 APL2 for Sun Solaris Licensed Program Specifications
SH21-1092 APL2 for Sun Solaris: User's Guide
GH21-1063 APL2/370 Application Environment Licensed Program Specifications
GH21-1070 APL2/370 Licensed Program Specifications
SH21-1062 APL2/370 Installation and Customization under CMS
SH21-1055 APL2/370 Installation and Customization under TSO
SH21-1054 APL2/370 Programming: System Services Reference
SH21-1056 APL2/370 Programming: Using the Supplied Routines
SH21-1058 APL2/370 Programming: Processor Interface Reference
LY27-9601 APL2/370 Diagnosis Guide
SH21-1059 APL2/370 Messages and Codes

SH21-1054-01

Spine information:

APL2 Programming: System Services Reference *Version 2 Release 2*