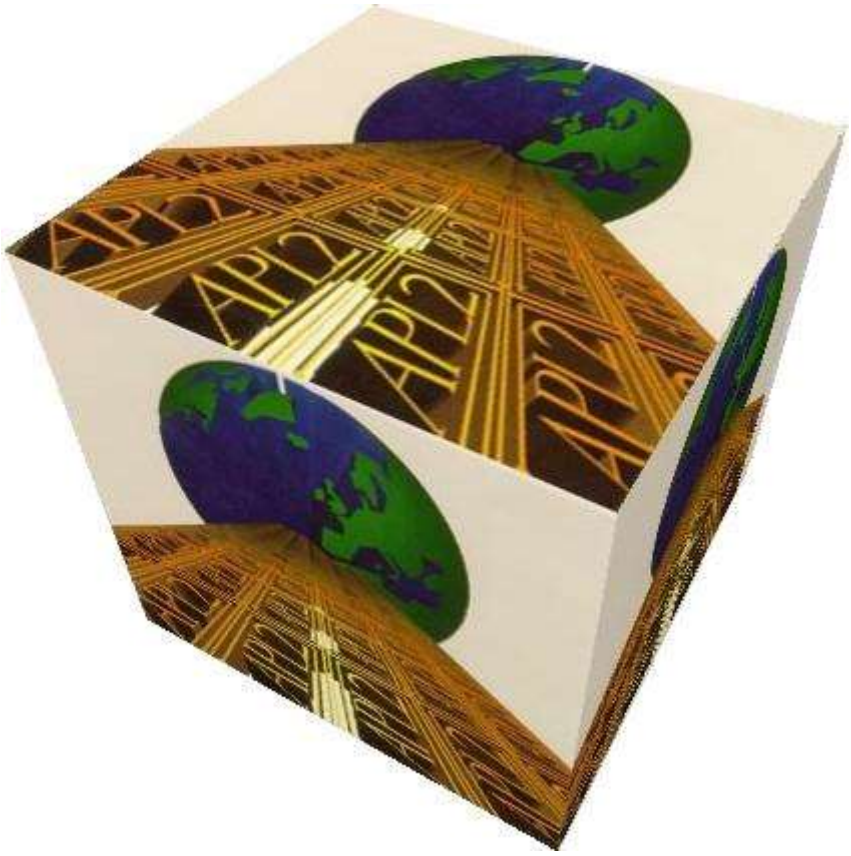
APL2 User's Guide

SC18-7021-23

APL Products and Services
IBM Silicon Valley Laboratory
555 Bailey Avenue
San Jose, California 95141
APL2@vnet.ibm.com



Copyrights

© Copyright IBM Corporation 1994, 2017 All Rights Reserved.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corporation

Contents

- Notices
- We Would Like to Hear from You
- About This Book

Part 1: Getting Started

- Overview
- Installing and Customizing APL2
  - Installing APL2 on Unix Systems
  - Installing APL2 on Windows Systems
- Invoking APL2
  - Invoking APL2 on Unix Systems
  - Invoking APL2 on Windows Systems
- Migrating from Version 1 to Version 2

Part 2: Using APL2

- General Information
- Workspaces and Libraries
- Using the Session Manager
  - The APL2 Session Manager on Unix Systems
  - The APL2 Session Manager on Windows
- Editors
  - Editing APL Objects
  - Editing Text Files
- The APL2 Library Manager
- Using APL2 Across Systems
  - Cooperative Processing
  - Shared Variable Interpreter Interface
  - Transferring Workspaces and Files

Part 3: Application Guide

- Associated Processors
  - Processor 10 - Communicating with Rexx
  - Processor 11 - External Routines and Namespaces
  - Processor 12 - Files as Arrays
  - Processor 14 - Calls to Java
  - Processor 15 - Access Structured Storage
- Supplied External Routines
  - APL2CFG - Configure APL2
  - APL2LM - APL2 Library Manager
  - APL2PIA - APL2 Programming Interface for APL2
  - ATR - Array to Record
  - ATS - Array to SCAR

© Copyright IBM Corporation 1994, 2017

- BEEP - Sound a Beep
- CHECK_ERROR - Get System Error Text
- CNS - Create Namespace
- COM - Component Object Model Interface
- COMBROWSE - COM Browser
- COPY - Copy Workspace Objects
- CPUCOUNT - Get Number of Processors
- CTK - Character to Kanji
- CTN - Character to Numeric
- CTUTF - Character to UTF
- DATATYPE - Array Data Type
- DISPLAY, DISPLAYC, and DISPLAYG - APL2 Array Structures
- EDITOR_2 - APL2 Editor 2
- EXP - Execute in Previous Namescope
- FDELETE - File Delete
- FILE - File Read or Write
- FILE Namespace
- FSTAT - File Status
- GETENV - Get Environment Variable
- GETLIB - Get Path for Library Number
- GMTSTAMP - Get or Set Timestamp
- GRAPHPAK Namespace
- GUITOOLS Namespace
- GUIVARS Namespace
- Host System Utilities
- IDIOMS and IDIOMSG - APL2 Idiom Library
- KTC - Kanji to Character
- LEXP - Lexical Parse
- LIB - List Library Contents
- LIBS - Get APL2 Library Definitions
- LTM - Tcl List to APL2 Matrix
- MATHFNS Namespace
- MD5 - Encode Data to MD5
- MTL - APL2 Matrix to Tcl List
- OPTION - Query or Set Session Options
- PCOPY - Protected COPY
- PEACHP and PEACHT - Parallel Each Operators
- PFA - Pattern From Array
- PRINTWSG - Print Workspace with GUI interface
- QNS - Query Namescope
- REPLACEX - Replace Strings
- ROUNDC - Commercial Rounding
- RF - RowFind
- RTA - Record to Array
- SCANDIR - List Files in a Directory
- SI - State Indicator
- SIS - State Indicator with Statements
- SIZEOF - Size of Array
- SQL Namespace

© Copyright IBM Corporation 1994, 2017

    - STA - SCAR to Array
    - TCL - Tool Command Language Interface
    - TIME - Application Performance Analysis
    - UTFTC - UTF to Character
    - WSCOMP - Workspace Compare
    - ZIP, UNZIP, ZIPWS, and UNZIPWS - Compression Utilities
- Auxiliary Processors
    - AP 100 - Host Command Processor
    - AP 101 - Alternate Input (Stack) Processor
    - AP 119 - Socket Interface Processor
    - AP 120 - Session Manager Interface
    - AP 124 - Text Display Processor
    - AP 127 - DB2 Processor
    - AP 144 - X Window Services Processor
    - AP 145 - GUI Services Processor
    - AP 200 - Calls to APL2
    - AP 207 - Universal Graphics Processor
    - AP 210 - File Processor
    - AP 211 - APL Object Library Processor
    - AP 227 - ODBC Processor
    - AP 488 - GPIB Support Processor
- Supplied Workspaces
    - AIX - AIX Operating System Functions
    - AP124 - Text Display Application Aids
    - AP144 - X Window Application Aids
    - AP488 - GPIB Application Aids
    - DDESHARE - High-Level DDE Access
    - DEMO124 - Text Display Demonstrations
    - DEMO144 - X Window Demonstrations
    - DEMO145 - GUI Demonstrations
    - DEMO207 - Graphics Demonstrations
    - DEMOJAVA - Calls to Java Demonstrations
    - DISPLAY - APL2 Array Structures
    - EDIT - Compatibility Editors
    - EXAMPLES - APL2 Language Examples
    - FILE - File I/O Routines
    - GRAPHPAK - Graphics Library
    - GUITOOLS - Building GUI Applications
    - GUIVARS - GUI Application Variables
    - IDIOMS - APL2 Idiom Library
    - LINUX - Linux Operating System Functions
    - MATHFNS - Mathematical Routines
    - MIGRATE - APL2 Migration Aids
    - NETTOOLS - Writing Network Applications
    - PRINT - Unix Printer Interface
    - PRINTWS - Workspace Printing Utility
    - SOLARIS - Solaris Operating System Functions
    - SQL - Structured Query Language Tools
    - TCL - Tool Command Language Demonstrations

© Copyright IBM Corporation 1994, 2017

    - TIME - Application Performance Analysis
    - UTILITY - General-Purpose APL2 Tools
    - WINDOWS - Windows Operating System Functions
    - WSCOMP - Workspace Compare
- APL2 Programming Interface (Calls to APL2)
    - Calling APL2 from APL2
    - Calling APL2 from C
    - Calling APL2 from Java
    - Calling APL2 from Visual Basic

Part 4: Advanced Topics

- Writing Your Own External Routines
    - Using Prebuilt DLLs (Runtime Libraries)
    - Creating SYSTEM Linkage Routines
    - Creating FUNCTION Linkage Routines
- Writing Your Own Auxiliary Processors
- The SVP Monitor Facility
- The APL2 Runtime Library
- Using the X Window System Interface

Appendices

- Windows Character Set Support
- Double Byte Character Set Support
- Implementation Limits
- Deviations from *APL2 Programming: Language Reference*
- Bibliography

© Copyright IBM Corporation 1994, 2017

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 on 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:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

© Copyright IBM Corporation 1994, 2017

Programming Interface Information

This user's guide is intended to help programmers write applications in APL2. It 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.

© Copyright IBM Corporation 1994, 2017

Trademarks

IBM Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

AIX
APL2
CUA
DATABASE 2
DB2
IBM
OS/2
Presentation Manager
System/370
System/390
VisualAge
WebSphere

Other Trademarks

The following terms are trademarks of other companies:

ActiveX	Microsoft Corporation
APL+Win	APL2000, Inc.
Adobe	Adobe Systems, Inc.
Acrobat	Adobe Systems, Inc.
InstallShield	Acresso Software Inc. and/or InstallShield Co. Inc.
Java	Sun Microsystems, Inc.
JavaMail	Sun Microsystems, Inc.
Linux	Linus Torvalds
Motif	The Open Group
MSDN	Microsoft Corporation
ooRexx	Rexx Language Association
PostScript	Adobe Systems, Inc.
Solaris	Sun Microsystems, Inc.
Sun	Sun Microsystems, Inc.
TrueType	Apple Computer, Inc.
Visual Basic	Microsoft Corporation
Windows	Microsoft Corporation

© Copyright IBM Corporation 1994, 2017

We Would Like to Hear from You

APL2 User's Guide

Please let us know how you feel about this online documentation by placing a check mark in one of the columns following each question below:

To return this form, print it, write your comments, and mail it to:

> International Business Machines Corporation
> APL Products and Services - PGUA/E1
> 555 Bailey Avenue
> San Jose, California 95141
> USA

For postage-paid mailing, please give the form to your IBM representative.

You can also send us your comments by email. To send us this form, copy it to a file, write your comments using a file editor, and then send it to:

> apl2@vnet.ibm.com

Overall, how satisfied are you with the online documentation?

	Very Satisfied 1	2	3	Very Dissatisfied 4
Overall Satisfaction	___	___	___	___

Are you satisfied that the online documentation is:

Accurate	___	___	___	___
Complete	___	___	___	___
Easy to find	___	___	___	___
Easy to understand	___	___	___	___
Well organized	___	___	___	___
Applicable to your tasks	___	___	___	___

Please tell us how we can improve the online documentation:

```
_____________________________________________________________
_____________________________________________________________
_____________________________________________________________
```

Thank you! May we contact you to discuss your responses?

```
___Yes  ___No
Name:
```


© Copyright IBM Corporation 1994, 2017

```
_____________________________________________
Title:
_____________________________________________
Company or Organization:
_____________________________________________
Address:
_____________________________________________
_____________________________________________
_____________________________________________
Phone:
(___)_____________________________________
E-mail:
_____________________________________________
```

Please do not use this form to request IBM publications. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office servicing your locality.

© Copyright IBM Corporation 1994, 2017

About This Book

This book describes IBM APL2 for use with AIX, Linux, Sun Solaris and Windows.

This User' Guide is intended to help you use APL2. It contains a collection of tables and descriptions giving full details of installation, operation, supplied workspaces, and external interfaces.

- APL2 Publications
- Conventions Used in This Book

© Copyright IBM Corporation 1994, 2017

APL2 Publications

Along with this *APL2 User's Guide*, the following additional on-line manuals are included with Workstation APL2:

- *APL2 Language Summary*
- *APL2 Programming: Language Reference*
- *APL2 Programming: Developing GUI Applications* (for Windows only)
- *APL2 Programming: Using APL2 with WebSphere*
- *APL2 Programming: Using SQL*
- *APL2 GRAPHPAK: User's Guide and Reference*

For the titles and order numbers of the complete APL2 library, and for information on other related publications, see Bibliography.

© Copyright IBM Corporation 1994, 2017

Conventions Used in This Book

The notations used in this user's guide and their meanings are:

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.

`←→`
: Used to mean "is equivalent to." It does not denote an APL2 operation.

The following APL2 names are used throughout this user's guide:

`Z`	Result name
`F`	Function name
`L`	Left argument name
`R`	Right argument name
`MOP`	Monadic operator name
`DOP`	Dyadic operator name
`LO`	Left operand name
`RO`	Right operand name

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

Throughout this book, the following abbreviated product names may be used:

Product Name	**Platform**
APL2/370	CMS or TSO
APL2/PC	DOS

© Copyright IBM Corporation 1994, 2017

Overview

These sections contain an overview of APL2, including:

- Why Should You Learn APL2?
- What is IBM APL2?
- Required Hardware
- Required Software

© Copyright IBM Corporation 1994, 2017

Why Should You Learn APL2?

Learning a programming language requires an investment of your most precious resources - time and effort. Why should you invest your time in APL2 in preference to any of the others? There are programming languages you can learn faster than APL2. But a tool is worth what you invest in it. Roman numerals were easier to use than Arabic numerals for most computations of Roman times. They were, however, impossible to extend for use in more sophisticated mathematics. Arabic numerals are harder to learn but are clearly a superior notation for mathematics.

On the surface, some other programming languages are easier to learn because APL2 contains so much advanced function. However, you do not need to learn all of APL2 in order to use it effectively. You can learn a subset of APL2 first, leaving more advanced function for later.

APL2 is extremely concise. A significant computation can fit in one page of code or even a few lines. One reason APL2 programs are so compact is that operations are represented by symbols, not English words. This has the advantage of being international, but may look intimidating. If you have paged through this manual, you have probably noticed that some of the symbols used come from the Greek alphabet. This leads some to say that APL2 looks like Greek. Just remember that Greek looks like Greek yet there are children who can read and write it fluently. It is a question not of difficulty, but of familiarity. Just as you can read the symbols used for international road signs once you know what they mean, you quickly become comfortable with APL2 symbols and appreciate the conciseness of the notation. Think of APL2 as the international road signs of programming.

© Copyright IBM Corporation 1994, 2017

What is IBM APL2?

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

When implemented as a computing system, the user types statements that specify the work to be done, and the computer responds by displaying the result of the work at a device such as a video display or a printer. In addition to work purely at the keyboard and its associated display, entries can also specify the use of printers, disk files, or other remote devices.

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

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

The letters "APL" originated with the initials of a book written by K. E. Iverson, *A Programming Language* (New York: Wiley, 1962). Dr. Iverson first worked on the language at Harvard University, and then continued its development at IBM with the collaboration of Adin Falkoff and others at the IBM T. J. Watson Research Center. The term **APL** now refers to the language that is an outgrowth of that work. **APL** is the language, and **APL2** is the brand name of IBM's extended version of APL.

This implementation of APL2 has the following features:

- The language processor is highly compatible with the other members of the APL2 family. It uses the same *APL2 Programming: Language Reference*. APL2 allows the transporting of workspaces, in transfer file format, between APL2 platforms. EBCDIC-ASCII translation is performed automatically where required, and is transparent to the user.
- The session manager is common user access (CUA) conforming and has a similar look and feel to the session managers of the other products in the APL2 family, with the added benefit of GUI (graphical user interface) features such as window movement, window resizing, scroll bars, font specification, and the use of icons. It provides scrolling capability, permanent session logs, stacked input, color control, function key support, keyboard redefinition, and national language support.
- Many of the auxiliary processors available with the APL2/370 or other workstation products are provided:
    - AP 100 (host system command processor)
    - AP 101 (alternative input (stack) processor)
    - AP 119 (socket interface processor)
    - AP 124 (text display processor)

© Copyright IBM Corporation 1994, 2017

    - AP 127 (SQL relational database processor for DB2)
    - AP 207 (universal graphics processor)
    - AP 210 (file I/O processor)
    - AP 211 (APL2 common object library processor)
- AP 144 (Unix systems), which provides a programmable interface to X Window services.
- AP 145 (Windows), which provides a programmable interface to GUI services.
- AP 227, which allows access to databases and programs which support the ODBC protocol, using the SQL language.
- Many of the common distributed workspaces and functions that are currently available on the System/370 or other workstation products.
- National language support. External command and message translation tables are provided that can be modified to support other countries, or individual user preferences.

© Copyright IBM Corporation 1994, 2017

Required Hardware

The following hardware is the minimum required for APL2:

- A computer running one of the operating systems listed below.
- At least 25 MB of unused hard disk space.
- At least 12 MB of memory above that required by the operating system.

© Copyright IBM Corporation 1994, 2017

Required Software

One of the following operating systems is required to run APL2:

- Windows 7, Windows 8 or Windows 10
- AIX 5.0 (or later) with
  Motif 1.2 (or later) and
  X Window System X11R5 (or later)
- Solaris 7 (or later) with
  Motif 1.4 (or later) and
  X Window System X11R5 (or later)
- A Linux distribution for PC compatible machines with
  glibc 2.3.2 (or later) and
  Motif 2.2 (or later) and
  X Window System X11R6 (or later)

Some features of APL2 have additional requirements:

- Adobe Acrobat Reader is required to view the on-line manuals
- IBM DATABASE 2 for AP 127
- An ODBC Driver Manager which supports the ODBC 3.0 level of Application Programming Interfaces for AP 227
- TCP/IP for AP 119 and cross-system shared variables
- IBM Object Rexx or Open Object Rexx for calling Rexx routines with Processor 10
- Tcl and Tk 8.4 or later for the TCL external function
- Java 2 Version 1.4 or later for Processor 14 or to call APL2 from Java.

© Copyright IBM Corporation 1994, 2017

Installing and Customizing APL2

- Installing APL2 on Unix Systems
- Installing APL2 on Windows Systems

© Copyright IBM Corporation 1994, 2017

Installing APL2 on Unix Systems

- Installing APL2
- APL2 Directory Structure
- Displaying APL2 Characters
- APL2 Keyboard Support
- Customizing APL2

Installing APL2

AIX Systems:

APL2 for AIX is distributed on CD or by download as a compatible application program. It can be installed using the System Management Interface Tool `smit`, or using the AIX command `installp`. Installation is under directory `/usr/APL2/`.

1. (Download method only) Download the `apl2av20.installp` file in binary mode to a temporary directory on your AIX machine.
2. Switch to root authority (`su` command)
3. (Download method only) Generate the `.toc` for the directory where the installp image file has been placed.

   `inutoc` *`directory_path`*

4. Run the installation.

   Using the `smit` utility:

   Follow the menus for installation and maintenance for your system configuration. For detailed information on using the `smit` command, see your AIX documentation.

   Using the `installp` command:

   `installp -ac -X -d` *`directory_path`*`/apl2av20.installp APL2.obj`

5. Exit from root authority

Linux Systems:

APL2 for Linux is distributed on CD or by download in `gzip`'d `tar` file format. The default installation is under directory `/usr/APL2/`.

1. (Download method only) Download the `apl2lv20.tgz` and `apl2lv20` files in binary mode to a temporary directory on your Linux machine.
2. Change to the directory where the `apl2lv20.tgz` and `apl2lv20` files reside.
3. (Download method only) Make the `apl2lv20` file into an executable file with the command:

© Copyright IBM Corporation 1994, 2017

```
chmod 755 apl2lv20
```

4. Switch to root authority (`su` command)
5. Run the installation script and follow the prompts:

   ```
   ./apl2lv20
   ```

6. Exit from root authority

Downloaded files can be removed after the installation. A copy of the install script is saved into the `APL2/install` directory for your future reference.

After the installation is complete, you will need to modify the `.bash_profile` (or appropriate shell profile) on all userids that will be running APL2 to add access to the APL2 fonts:

```
xset fp+ /usr/APL2/fonts/X11
xset fp rehash
```

Solaris Systems:

APL2 for Sun Solaris is distributed on CD or by download in compressed `tar` file format. The default installation is under directory `/usr/APL2/`.

1. (Download method only) Download the `apl2sv20.tarz` and `apl2sv20` files in binary mode to a temporary directory on your Linux machine.
2. Change to the directory where the `apl2sv20.tarz` and `apl2sv20` files reside.
3. (Download method only) Make the `apl2sv20` file into an executable file with the command:

   ```
   chmod 755 apl2sv20
   ```

4. Switch to root authority (`su` command)
5. Run the installation script and follow the prompts:

   ```
   ./apl2sv20
   ```

6. Exit from root authority

Downloaded files can be removed after the installation. A copy of the install script is saved into the `APL2/install` directory for your future reference.

The installation process will attempt to install the APL2 fonts into the X Window System so they will be available automatically. However, the font installation process may fail if the X Window System is not active or if the font compilation tools are not present on the system.

If the font installation is not successful, you may correct the problem that caused it to fail and re-run the font installation program (`/usr/APL2/bin/InstallFonts`), or you may provide access to the fonts manually by modifying the `.profile` (or appropriate shell profile) on all userids that will be running APL2 to add access to the APL2 fonts:

© Copyright IBM Corporation 1994, 2017

```
xset fp+ /usr/APL2/fonts/X11
xset fp rehash
```

National Language Selection

When APL2 is started, the `apl2` shell script attempts to select from the available national languages for APL2 based on the value of the system environment variable `LANG`. It contains logic to map some of the common names for these languages to the names used by APL2 (`En_US`, `En_UK`, `Fr_FR`, `Gr_GR`, and `It_IT`).

If `LANG` is not set, or is not one of the values recognized by the shell script, English will be used for system messages and the X resource files used will be those that contain mapping for the USA standard keyboard. Local modification of the `apl2` shell script may be necessary if another national language setting is desired for APL2.

AIX Only: On AIX, location of resource files is based strictly on the value of `LANG`. The mapping the `apl2` shell script does is effective only in selecting message text. If `LANG` does not match one of the names used for the directories in `/usr/APL2/nls`, you will need to create a directory with a name matching `LANG`. In the new directory, create links to the files in the directory for the language you wish to use. For example, if `LANG` is `en_US` rather than `En_US`,

```
cd  /usr/APL2/nls
su
mkdir  en_US
cd  en_US
ln  -s  /usr/APL2/nls/En_US/ap120  ap120
ln  -s  /usr/APL2/nls/En_US/ap124  ap124
ln  -s  /usr/APL2/nls/En_US/APL2win  APL2win
ln  -s  /usr/APL2/nls/En_US/cmd_msg.txt  cmd_msg.txt
exit
```

APL2 Directory Structure

When the product is installed, subdirectory `/APL2/` is created under `/usr` and is the main APL2 installation directory. It contains installation control files and a `README` file that has important information about the product. All other directories in the table below are created under this main installation directory.

Subdirectory	Description
`bin`	Binary executable files
`defaults`	Language-independent files
`doc`	On-line documentation (in Adobe .pdf format)
`examples`	Sample programs categorized by subdirectory
`fonts/X11`	APL2 fonts for X Window System
`fonts/printer`	APL2 printer fonts
`fonts/vector`	Vector fonts for AP 207
`include`	C Include files for user-written auxiliary processors, external functions and Calls to APL2
`lib`	C libraries for user-written auxiliary processors and Calls to APL2

© Copyright IBM Corporation 1994, 2017

Subdirectory	Description
`lib1`	Distributed workspaces)LIB 1
`lib2`	Distributed workspaces)LIB 2
`nls`	Language-dependent files. There is a subdirectory for each supported locale under this directory.

Displaying APL2 Characters

Installation of the APL2 character fonts for X Window System takes place during the product installation described in Installing APL2. If X Window System was active during the installation, the newly-installed APL2 fonts are not available until X Window System rereads the font directories. This is accomplished by restarting the X server or issuing the command `xset fp rehash`.

AIX and Sun Solaris Only:

Font installation is an automatic part of the installation process, and if successful, fonts will be available to all users. However, after APL2 has been installed, it is possible that the fonts may inadvertently be uninstalled during X Window System-related maintenance. If this should occur, the APL2 fonts can be reinstalled by doing the following:

1. Log in as root user (`login root`)
2. Type:

   ```
   $(APL2)/bin/InstallFonts
   ```

   where `$(APL2)` is the APL2 installation directory.

APL Characters on Remote Workstations

APL2 can be run on a local workstation, or accessed remotely by other workstations using a LAN connection.

These remote workstations (X servers) require the installation of the APL2 fonts in order to display APL characters. To install the fonts on the remote workstation complete the following steps:

1. Enter the command `xset q` and look in the first directory for the font path to determine which font format is needed. If the files end in `.pcf`, then you are running X11R5. If they end in `.snf`, then you most likely have X11R4. If the format is not `snf` or `pcf`, then you need to build the necessary font from the `bdf` files that are in the `tar` file, `pc910bdf.tar`.
2. Log in as root.
3. `cd` to the 1st directory in the font path.
4. If you are running X11R4, copy the `*.snf` files from the APL2 `/fonts/X11`. If you are running X11R5, copy the `*.pcf` files.
5. `mkfontdir`
6. `xset fp rehash`

© Copyright IBM Corporation 1994, 2017

These are roman-style fonts conforming to IBM code page 910 (see *APL2 Programming: Language Reference*). In most cases they can be used successfully for normal terminal operation as well as for APL2 operation. In addition, IBM code page 910 supports many of the international characters found in code page 850. These include characters required for use in France, Germany, Italy, the United Kingdom, and the USA.

APL2 Keyboard Support

This section describes the keyboard support provided for APL2.

- APL2 Keyboard Layouts
- Keyboard Decals
- Keyboards

APL2 Keyboard Layouts

Several standard APL2 keyboard layouts are provided with this product.

You can choose one of these, or design your own. For more information about customizing your keyboard, see Customizing the Keyboard.

Keyboard Decals

Decals with the APL2 character set for IBM keyboards are included with this product. Additional sets of decals are available from IBM Publications as *APL2 Keyboard Decals*, SC33-0604.

Carefully peel each decal away from the backing sheet and place it on the appropriate key. Note that some decals (those for Alternate-shift characters) go on the fronts of the keys, and that not all the decals are needed for some keyboards. For a French keyboard layout, which has the numeric keys in the shifted position, you need to cut the decals for these keys to move the APL2 characters to the lowercase position.

Keyboards

APL keyboards for many machines can be purchased from Unicomp Corporation:

web: www.pckeyboard.com
phone: 1-800-777-4886

Customizing APL2

This section describes how you can customize your APL2 session during installation.

- The `apl2` Shell Script
- The `apl2.ini` Configuration File
- The SVP Parameter File
- The SVP Profile

The `apl2` Shell Script

© Copyright IBM Corporation 1994, 2017

Upon completion of the installation of APL2, a symbolic link is defined for `/usr/bin/apl2`, pointing to the default installation directory for the APL2 executable file `apl2`. The `apl2` shell script is the recommended way to start the APL2 interpreter (`apl2exe`). It establishes the default environment variables and opens the appropriate windows to run the APL2 system.

The `apl2` shell script is designed to be flexible enough to support the most common default overrides, through command line optional parameters, the `apl2.ini` configuration file, or preassignment of environment variables (see Invoking APL2 on Unix Systems). For user-level customization, environment variable definitions are typically placed in `$HOME/.profile`, or a private `apl2.ini` file maintained in `$HOME`. For multiuser, system-level customization, the `apl2` shell script can be tailored to local convention, or a system `apl2.ini` can be maintained in `/usr/APL2/bin`.

The `apl2.ini` Configuration File

APL2 invocation options, library directory definitions and several Shared Variable Processor settings can be stored in the `apl2.ini` configuration file.

To find the `apl2.ini` file, APL2 will search first in the current working directory (the directory from which APL2 was invoked) and then in `$APL2/bin`. The first `apl2.ini` file that is found in that search order will be used, and any others will be ignored.

APL2 searches `apl2.ini` *after* command line parameters and *before* environment variables.

The file uses the following syntax:

```
[Section Name]
keyword=value
keyword=value
...
[Section Name]
keyword=value
keyword=value
...
...
```

Leading and trailing blanks, and any lines that do not contain a valid section name or keyword definition, are ignored. Searches for section names and keywords are case-insensitive.

The following section names are supported:

`[Invocation Options]`
: APL2 interpreter invocation options as listed in Invocation Parameters, except for `hostwin` and `svptrace`. On Unix systems, those two options are managed entirely within the shell script and settings in `apl2.ini` will be ignored. For example:

```
[Invocation Options]
WS=50m
P11=/home/myuserid/mynames.nam:/usr/APL2/defaults/aplnm011.nam
```

`[Libraries]`
: APL2 library definitions, in the format described in Library Specification. For example:

© Copyright IBM Corporation 1994, 2017

```
      [Libraries]
      APL01001=/home/myuserid/wss
      APL00019=/home/myuserid/AGSS
[Shared Variable Processor]
```

Shared variable processor settings listed in APL2 Environment Variables. Remove the `APL2` prefix from the environment variable name to form the keyword. For example:

```
[Shared Variable Processor]
SVPPARMS=/home/myuserid/apl2svp.prm
SMKEY=MYOWNKEY
```

A sample `apl2.ini` file is shipped with APL2 in directory `/usr/APL2/defaults`.

The SVP Parameter File

The APL2 Shared Variable Processor (SVP) uses a parameter file to set some initialization options. The APL2SVPPARMS environment variable, or the SVPPARMS keyword in the [Shared Variable Processor] section of `apl2.ini`, can be used to specify the name of this file. The file name can include path information, or just the name of the file to search for in the current directory. If the file name is not established using either of those methods, the default file name `apl2svp.prm` is searched for in the current directory. However, note that the `apl2` shell script sets the APL2SVPPARMS environment variable, if undefined, to the distributed system default parameter file.

The file uses the following syntax:

```
keyword=value [value...]
```

Blank lines or lines with an asterisk in the first column are ignored. In the case of keywords that accept more than one value, the values can be separated by a comma or spaces.

The keywords that can be specified are:

`PROCS` Maximum number of processors that can be signed on at one time.
Default: 100 Minimum: 32

`VARS` Maximum number of variables that can be shared at one time.
Default: 400 Minimum: 128

`MAXSM` Maximum amount of memory, in bytes, for all shared variable data.
Default: 16 Megabytes Minimum: 2 Megabytes

`MAXSV` Maximum amount of memory, in bytes, for a single shared variable.
Default: 12 Megabytes Minimum: 1 Megabytes

`NOTRACE` List of processor IDs for which events are not to be traced by the SVP monitor facility. An asterisk may be specified to signify that no processors are to be traced.
Default: 1 101 120

The default file `apl2svp.prm` is provided as follows:

```
* Sample svp parms file
*
* Maximum number of processors signed on
PROCS=100
```


© Copyright IBM Corporation 1994, 2017

```
* Maximum number of variables shared
VARS=400
* Maximum amount of shared memory for all variable data
MAXSM=16000000
* Maximum amount of shared memory for any one shared variable
MAXSV=12000000
* processor numbers to omit from the SVP trace
NOTRACE=1 101 120
*
```

Although there is no SVP-enforced maximum allowed value for MAXSM, this is limited by the amount of paging space available on the system. If the SVP parms file is not found, defaults for PROCS, VARS, MAXSM, and MAXSV are set as noted in the sample file above, and all processors are traced.

The SVP Profile

For information about customizing the SVP profile, see Processor Profile Structure.

© Copyright IBM Corporation 1994, 2017

Installing APL2 on Windows Systems

- Installing APL2
- Customizing APL2
- APL2 Keyboard Support

Installing APL2

APL2 uses the *InstallShield* product for installation. Installation is supported from CD-ROM or a Local Area Network (LAN) server.

This section contains information that you need for:

- Installing APL2 from a CD-ROM
- Installing from a LAN Server
- APL2 Directory Structure
- Installing the APL2 Fonts
- Accessing the Online Documentation
- Establishing the APL2 Environment

Installing APL2 from a CD-ROM

To install APL2 from a CD-ROM:

Insert the CD-ROM into the drive. If the installation program does not start automatically,

1. Click the **Start** button, point to **Settings**, and then click **Control Panel**.
2. Double-click **Add/Remove Programs**.
3. Click **Install**.
4. Click **Next**.
5. The correct installation program is `x:\windows\setup.exe`, where `x` represents the letter of the CD-ROM drive. Click **Finish**.

Follow the instructions on your screen.

Note: On some versions of Windows, if the installation process is performed from a userid that is a member of the administrator group, you may be prompted to choose whether the installation will be for the current user or all users.

Installing from a LAN Server

If you have purchased multiple licenses of APL2, you might want to consider placing the installation files on a LAN server for easier access. To do this, perform the following steps:

1. Create a directory on the LAN server to store the APL2 installation files.
2. Copy the contents of the CD-ROM Windows installation directory into this directory.
3. Give all licensed users access to the LAN drive you copied the install files to.

© Copyright IBM Corporation 1994, 2017

After performing these steps, users with access to the LAN drive can access the alias drive\directory and install APL2 by running the `SETUP` program.

Your LAN users are now ready to use APL2.

Note: On some versions of Windows, if the installation process is performed from a userid that is a member of the administrator group, you may be prompted to choose whether the installation will be for the current user or all users.

APL2 Directory Structure

When the APL2 product is installed, a directory is created as the top level installation directory. The default directory name is `C:\Program Files\ibmapl2w` (on 64-bit systems, `C:\Program Files (x86)\ibmapl2w`), although this can be changed during installation if desired. It contains the installation control files and a `readme.txt` file that has important information about the product. All other directories in the table below are created as subdirectories of the main installation directory.

Subdirectory	Description
`bin`	Binary executable files, dynamic link libraries, and online help
`doc`	Online documentation
`fonts`	APL2 fonts (session manager, AP 124, and AP 207)
`include`	C Include files for user-written auxiliary processors, external functions and Calls to APL2
`lib`	C libraries for user-written auxiliary processors and Calls to APL2
`runtime`	Installation file for the APL2 Runtime Library
`samples`	Sample programs and profiles
`wslib1`	Distributed workspaces)LIB 1
`wslib2`	Distributed workspaces)LIB 2

Installing the APL2 Fonts

APL2 includes the following TrueType fonts:

- APL2 Italic (APL2ITAL.TTF)
- APL2 Unicode Italic (APL2UI__.TTF)
- Courier APL2 (COURAPL.TTF)
- Courier APL2 Bold (COURAPLB.TTF)
- Courier APL2 Unicode (COURAU__.TTF)

the following bitmap font:

- APL2 Image (APL2IMAG.FON)

and a variety of vector fonts for use by AP 207.

© Copyright IBM Corporation 1994, 2017

The APL2 installation program installs the APL2 fonts in the `fonts` subdirectory of the main APL2 installation directory. All the programs in the APL2 product can find them there.

If the userid performing the APL2 installation is a member of the administrator group, the APL2 installation program also installs the TrueType and bitmap fonts in the Windows Fonts directory, where non-APL2 programs can find them.

To install the APL2 fonts in the Windows Fonts directory manually, the procedure is as follows:

Windows 7, Windows 10:

1. Click the **Start** button, select **Computer**.
2. Navigate to the `fonts` subdirectory of the main APL2 installation directory.
3. For each font you wish to install, right-click on the file name and select **Install** from the menu.

Windows 8:

1. Open the **File Explorer** Windows App.
2. Navigate to the `fonts` subdirectory of the main APL2 installation directory.
3. For each font you wish to install, right-click on the file name and select **Install** from the menu.

Accessing the Online Documentation

Online Manuals

When you install APL2, the online manuals are automatically installed in the `doc` subdirectory of the main APL2 installation directory. You can access this information from the APL2 Session Manager window, from the Desktop or from the Adobe Acrobat Reader.

To access the information from the APL2 Session Manager window, click on **Help**, select **Books**, and then select a document from the list.

To access the information from the Desktop, click on **Start**, select **Programs**, select *IBM APL2*, and then double-click on a document object.

Online Help

The online help for the session manager, editors and SVP Monitor is installed in the `bin` subdirectory of the main APL2 installation directory. You can access the online help from the **Help** menu items in the windows for these components.

Establishing the APL2 Environment

By default, the APL2 installation program makes the following environment variable modifications:

- Add the following directory to PATH: `C:\Program Files\ibmapl2w\bin`
- Add the following file to CLASSPATH: `C:\Program Files\ibmapl2w\bin\apl2.jar`
- Set variable APL00001 to: `C:\Program Files\ibmapl2w\wslib1`

© Copyright IBM Corporation 1994, 2017

- Set variable APL00002 to: `C:\Program Files\ibmapl2w\wslib2`
- Set variable APL01001 to: `C:\Program Files\ibmapl2w\bin`
- Set variable APLP11 to: `C:\Program Files\ibmapl2w\bin\aplnm011.nam`
- Set variable APL207FL to: `C:\Program Files\ibmapl2w\fonts`

(The examples assume the main APL2 installation directory was `C:\Program Files\ibmapl2w\`)

If you choose not to have the installation program modify your environment variables, you need to configure your system manually for APL2 to run correctly.

Environment Variables

You must add the APL2 bin directory to the PATH environment variable to be able to invoke APL2 from any other directory.

You must add the apl2.jar file to the CLASSPATH environment variable for Associated Processor 14, the APL2-Java interface processor, to work properly.

You can set environment variables in the **System** notebook of the **Control Panel**.

Alternatively, you can set environment variables in a batch file and use the batch file to invoke APL2. See Batch Files for more information.

Workspace Libraries and P11 Names File

APL2 requires the location of public workspace libraries 1 and 2. The APL2 installation program sets the default locations shown above in the environment variables APL00001 and APL00002. You may also want to supply the location of personal workspace library 1001 and other libraries. For information about workspace library locations, see Library Specification.

APL2 also requires the location of the Processor 11 names file. The APL2 installation program sets the default location shown above in the environment variable APLP11. For information about Processor 11 names file locations, see the discussion of the -p11 invocation option in Invocation Parameters.

You may supply the locations of workspace libraries and the Processor 11 names file using any or a combination of the following techniques:

- The Configure APL2 Tool
- The `apl2.ini` Configuration File
- In environment variables in Batch Files
- In environment variables in the **Control Panel**
- Invocation option (P11 names file only)

Consult the next section, Customizing APL2, for more information.

Customizing APL2

This section describes how you can customize your APL2 sessions.

© Copyright IBM Corporation 1994, 2017

- The Configure APL2 Tool
- The `apl2.ini` Configuration File
- Batch Files
- APL2 and TCP/IP
- The SVP Parameter File
- The SVP Profile

The Configure APL2 Tool

Use the **Configure APL2** tool to set default invocation options, define APL2 libraries, add session manager toolbar buttons, and specify SVP profile file names.

The **Configure APL2** tool can run either as a stand-alone program or as an external function. When run as a stand-alone program, the **Configure APL2** tool runs independently of interactive APL2 sessions. When run as an external function, it runs as part of an interactive APL2 session.

To start the **Configure APL2** tool as a stand-alone program, use either of these methods:

- Click on the **Configure APL2** icon in the APL2 folder,
- Click on the **Configure APL2** button on the Session Manager toolbar,

To start the **Configure APL2** tool as an external function, associate and run the `APL2CFG` external function:

```
      3 11 ⎕NA 'APL2CFG'
1
      APL2CFG
```

For information about how to use the **Configure APL2** tool, consult its online help.

Note:

> The **Configure APL2** tool displays the contents of the apl2.ini file. For further information, consult The `apl2.ini` Configuration File.

The `apl2.ini` Configuration File

The `apl2.ini` configuration file contains invocation options, library directory definitions, user-defined Session Manager toolbar buttons and several Shared Variable Processor settings.

You can use The Configure APL2 Tool to edit the `apl2.ini` file.

To find the `apl2.ini` file, APL2 will search:

1. The directory from which the application loaded, usually `\ibmapl2w\bin`.
2. The current directory.
3. The system directory.
4. The 16-bit system directory.
5. The Windows directory.

© Copyright IBM Corporation 1994, 2017

6. The directories that are listed in the PATH environment variable.
7. The current user's Application Data directory.

The first `apl2.ini` file that is found in that search order will be used, and any others will be ignored.

APL2 searches `apl2.ini` *after* command line parameters and *before* environment variables.

The file uses the following syntax:

```
[Section Name]
keyword=value
keyword=value
...
[Section Name]
keyword=value
keyword=value
...
...
```

Leading and trailing blanks, and any lines that do not contain a valid section name or keyword definition, are ignored. Searches for section names and keywords are case-insensitive.

The following section names are supported:

`[Invocation Options]`
: APL2 interpreter invocation options as listed in Invocation Parameters. For example:

```
[Invocation Options]
WS=50m
P11=C:\myuser\mynames.nam;C:\Program Files\ibmapl2w\bin\aplnm011.nam
```

`[Libraries]`
: APL2 library definitions, in the format described in Library Specification. For example:

```
[Libraries]
APL01001=C:\myuser\wss
APL00019=C:\AGSS
```

`[Shared Variable Processor]`
: Shared variable processor settings listed in APL2 Environment Variables. Remove the `APL2` prefix from the environment variable name to form the keyword. For example:

```
[Shared Variable Processor]
SVPPARMS=C:\myuser\apl2svp.prm
```

`[Session Manager]`
: Definitions for toolbar buttons. To add buttons to the APL2 Session Manager, set the `Toolbar Butttons` keyword. The value of that keyword is a list of suffixes for button information sections. Each suffix is appended to `Toolbar button` to form a section name. Add the following keywords within each button section:

  `Command` The name of the file or program for Windows to open or an expression to be executed by APL when the button is pressed. The maximum length of the command is 32768 characters.

  `Text` The text to appear on the button

  `Bitmap` The picture to appear on the button, a bitmap filename.

  `APL` Whether the command should be opened by Windows or executed by APL. If TRUE, the expression should be executed by APL. The default value is FALSE.

© Copyright IBM Corporation 1994, 2017

Notes:

- The APL expression is executed in the context of the interpreter's current operation. If the interpreter is suspended, the expression is executed in the context of the suspension. If the interpreter is running, the expression is executed before the next input request, wait, or line of execution.
- System commands are not supported.
- All output is suppressed.
- If an error is encountered, the error message is reported in the status area.
- If input is requested, a `VALUE ERROR` is reported.

You must supply a value for the `Command` keyword. You must also supply at least one of the `Text` and `Bitmap` keywords. The `APL` keyword is optional. For example:

```
[Session Manager]
Toolbar buttons=APL2 IE Associate
[Toolbar button APL2]
Command=http://www.ibm.com/software/awdtools/apl
Bitmap=apl2button.bmp
[Toolbar button IE]
Command=iexplore.exe
Text=Internet Explorer
[Toolbar button Associate]
Command=⎕ES(~3 11 ⎕NA 'DISPLAY')/1 5
Text=Setup
APL=TRUE
```

A sample `apl2.ini` file is shipped with APL2 in directory `\ibmapl2w\samples`.

Batch Files

If all your APL2 sessions can share the same settings, you can supply the environment information APL2 requires using the **Configure APL2** tool or by setting environment variables in the **Control Panel**. If, however, you plan to start multiple APL2 sessions which each require a unique environment, you may want to start APL2 using batch files. A sample batch file, `apl2.bat` is shipped with APL2 in the `\ibmapl2w\samples` directory. You can copy this file to the `\ibmapl2w\bin` directory, customize it, and create a shortcut to it.

The following shows what the sample file contains, with a description added for each line:

`@echo off`	Suppresses echoing of lines
`SET APL00001=..\WSLIB1`	Sets up public workspace library 1
`SET APL00002=..\WSLIB2`	Sets up public workspace library 2
`SET APL01001=.\`	Sets default private library
`SET APL207FL=..\FONTS`	Sets location of AP 207 vector fonts
`SET APLP11=..\BIN\APLNM011.NAM`	Sets default processor 11 NAMES file
`SET CLASSPATH=%CLASSPATH%;..\bin\apl2.jar`	Appends the APL2 Java Archive file to CLASSPATH

© Copyright IBM Corporation 1994, 2017

`APL2WIN.EXE %1 %2 %3 %4 %5 %6 %7 %8`	Invokes APL2 with any arguments
`EXIT`	Exit this command file

To specify additional invocation options, add them to the line that invokes `APL2WIN.EXE`. Be sure to retain the `%1 %2 ...` at the end of the line to allow any extra parameters to be specified on the command line. See Invocation Parameters for a complete list of the invocation parameters accepted by APL2.

APL2 and TCP/IP

APL2 uses TCP/IP to provide cooperative processing facilities. The Shared Variable Processor (SVP) contains references to certain TCP/IP socket calls necessary for this function. The SVP routines are supplied in the form of a dynamic link library (DLL). Because this DLL has a load-time dependency on TCP/IP, two versions are shipped with APL2 and installed into the `C:\Program Files\ibmapl2w\bin` directory. The two versions are named `apl2svpt.dll` (TCP/IP enabled) and `apl2svpn.dll` (TCP/IP disabled).

The APL2 installation program installs the version of the DLL which is TCP/IP enabled (by copying `apl2svpt.dll` to `apl2svp.dll`.) If you do not wish to use cross-system shared variables, you can copy the TCP/IP disabled version of the DLL:

1. Change to the subdirectory where APL2 dynamic link libraries are located (by default, `C:\Program Files\ibmapl2w\bin`.)
2. Issue the following command:

```
copy apl2svpn.dll apl2svp.dll
```

This disables APL2's cooperative processing features (see Cooperative Processing).

To change back to the version with TCP/IP support:

```
copy apl2svpt.dll apl2svp.dll
```

The SVP Parameter File

The APL2 Shared Variable Processor (SVP) allows for the optional use of a parameter file to control some SVP settings. The APL2SVPPARMS environment variable, or the SVPPARMS keyword in the [Shared Variable Processor] section of `apl2.ini`, can be used to specify the name of this file. If the file name is not established using either of those methods, the default file name is `apl2svp.prm`. This file is searched for in the following sequence:

1. In the path supplied along with the file name in the environment variable or `apl2.ini`,
2. In the current directory used by the SVP. This is either the directory from which APL2 was invoked, or the directory from which the *SVP Monitor* window or any independent processor was invoked, whichever was started first.
3. In the list of directories specified in the PATH environment variable.

If the file is not found in any of these locations, SVP initialization proceeds using default values as indicated.

© Copyright IBM Corporation 1994, 2017

The file uses the following syntax:

```
keyword=value [value...]
```

Blank lines or lines with an asterisk in the first column are ignored. In the case of keywords that accept more than one value, the values can be separated by a comma, by spaces, or by both. If less than the minimum value is specified, the minimum will be used.

The keywords that can be specified are:

`PROCS` Maximum number of processors that can be signed on at one time.
Default: 100 Minimum: 32

`VARS` Maximum number of variables that can be shared at one time.
Default: 400 Minimum: 128

`MAXSM` Maximum amount of memory, in bytes, for all shared variable data.
Default: 16 Megabytes Minimum : 2 Megabytes

`MAXSV` Maximum amount of memory, in bytes, for a single shared variable.
Default: 12 Megabytes Minimum: 1 Megabyte

`NOTRACE` List of processor IDs for which events are not to be traced by the SVP monitor facility. An asterisk may be specified to signify that no processors are to be traced.
Default: 1 101 120

A sample parameter file `apl2svp.prm` is provided in `\ibmapl2w\samples`. To use the sample, copy it to `\ibmapl2w\bin` and edit as desired. Its contents as shipped are as follows:

```
* Sample svp parms file
*
* Maximum number of processors signed on
PROCS=100
* Maximum number of variables shared
VARS=400
* Maximum amount of shared memory for all variable data
MAXSM=16000000
* Maximum amount of shared memory for any one shared variable
MAXSV=12000000
* processor numbers to omit from the SVP trace
NOTRACE=1 101 120
*
```

Note: Although there is no enforced maximum allowed value for MAXSM or MAXSV, they can be limited by the size of memory, system limitations on shared memory allocation, or the space available for operating system swap files.

The SVP Profile

For information about customizing the SVP profile, see Processor Profile Structure.

APL2 Keyboard Support

This section describes the keyboard support provided for APL2.

© Copyright IBM Corporation 1994, 2017

- The APL2 Keyboard Handler
- APL2 Keyboard Layouts
- Keyboard Decals
- Keyboards

The APL2 Keyboard Handler

The APL2 keyboard handler enables you to type APL characters. It is automatically used by the APL2 session manager, editors, and auxiliary processors.

The APL2 keyboard handler can also enable you to type APL characters in other applications. To enable APL input in other applications, either open the *APL2 Keyboard Handler* icon in the IBM APL2 folder or run the `APL2KEY.EXE` program. Then, select an APL font in the application in which you want to type APL characters.

When the APL2 keyboard handler is started, an icon is added to the taskbar. Press Ctrl+Backspace to turn the APL2 keyboard on and off. When the icon is bright red, the APL2 keyboard is on. When the icon is dark red, the APL2 keyboard is off. To adjust the APL2 keyboard handler properties, use the right mouse button to click the icon and click Properties on the popup menu.

To stop the APL2 keyboard handler, use the right mouse button to click the icon and click Exit on the popup menu.

Some applications accept either single-byte or Unicode input. When the active window accepts Unicode input, a blue U appears in the APL taskbar icon. To use Unicode input, select a Unicode font containing APL characters.

Note: Not all applications accept APL input. For example, Microsoft Word and PowerPoint use a proprietary mechanism for handling keystrokes, and MS DOS Command Prompt windows use console input buffers. The APL2 keyboard handler does not work with these programs.

APL2 Keyboard Layouts

Several standard APL2 keyboard layouts are provided with this product. You can choose one of these or design your own. The keyboard layout can be changed during your APL2 session, and is retained across sessions. The default layout is dynamically configured based on your current input locale.

To see the key assignments, or to modify them, choose **Keyboard Properties** from **System Options** under the session manager **Options** menu. The layout displayed by default is your currently selected one, but all others are available in the **Layout name** drop-down combo box.

Union keyboard layouts provide standard ASCII characters in the unshifted and Shift keyboard states, and special APL characters on the Ctrl and Alt shifts. This permits intermixed keying of APL and non-APL characters.

Traditional APL keyboards include an APL on/off key which switches the keyboard between an APL mode and an ASCII mode. In APL mode, ASCII characters are available only if they are also used by APL, and they

© Copyright IBM Corporation 1994, 2017

appear (by default) on the keys traditionally used by APL keyboards, which are usually different from the normal key assignments.

APL2 uses Ctrl-Backspace as the APL on/off key. Ctrl-Backspace still switches APL on or off when using a union keyboard, but that basically indicates only whether all characters, or only the non-APL subset, should be accepted from the keyboard. European users should note, however, that in all layouts "dead" keys are disabled when the keyboard is in APL mode.

For reference information on the positions of characters in the APL2 character set, see *APL2 Programming: Language Reference*. For full details of keyboard layouts and key assignments, press **Help** when in the **Keyboard Properties** dialog.

Keyboard Decals

Decals with the APL2 character set for IBM keyboards are included with this product. Additional sets of decals are available from IBM Publications as *APL2 Keyboard Decals*, SC33-0604.

Carefully peel each decal away from the backing sheet and place it on the appropriate key. Note that some decals (those for Alternate-shift characters) go on the fronts of the keys, and that not all the decals are needed for some keyboards. For a French keyboard layout, which has the numeric keys in the shifted position, you need to cut the decals for these keys to move the APL2 characters to the lowercase position.

Keyboards

APL keyboards for many machines can be purchased from Unicomp Corporation:

web: www.pckeyboard.com
phone: 1-800-777-4886

© Copyright IBM Corporation 1994, 2017

Invoking APL2

- Invoking APL2 on Unix Systems
- Invoking APL2 on Windows Systems

© Copyright IBM Corporation 1994, 2017

Invoking APL2 on Unix Systems

This material describes the APL2 environment, techniques you can use to create that environment, parameters you can pass to the APL2 product, and environment variables that affect the way it behaves.

- The APL2 Environment
- Windows Opened by APL2
- Invocation Syntax
- Examples of APL2 Invocation
- Running APL2 in Batch Mode
- Additional Requirements for Remote Access

The APL2 Environment

APL2 is invoked by running a shell script called `apl2`. The shell script starts a new shell instance, sets certain environment variables required by APL2, and starts the APL2 interpreter. Environment variables set before running `apl2` remain in effect during the session, unless they are reset by the script. Invocation options can be set in any of the following ways:

1. Invoke `apl2` with command line parameters:

   ```
   apl2 -ws 50m
   ```

2. Store the options in an `apl2.ini` file:

   ```
   [Invocation Options]
   WS=50M
   ```

   See The `apl2.ini` Configuration File for more information.

3. Set an environment variable before invoking `apl2`:

   ```
   APLWS=50M
   apl2
   ```

4. Take the settings for environment variables given by the shell script. For example, the value for the APLSM variable is `on`.
5. Accept the APL2 system defaults.

If the option has been set in more than one way, the value set by a command line parameter takes precedence over a value set in an `apl2.ini` file or an environment variable, a value set in an `apl2.ini` file takes precedence over a value set in an environment variable, and all of these take precedence over a default setting.

Notes:

1. Except when using APL2 in batch mode, you must be in an X Window System environment before starting APL2.

© Copyright IBM Corporation 1994, 2017

2. The `hostwin` and `svptrace` options and the windows they control are managed by the `apl2` shell script. Values specified for these options in the `apl2.ini` file will be ignored.
3. The APL2 invocation parameters, the `apl2.ini` file and environment variables are also available to any dependent auxiliary processors. Dependent auxiliary processors are auto-started as children of the APL2 interpreter session when the first share offer is made to the auxiliary processor. Command line parameters are passed unmodified to the auxiliary processor, with the exception of the `-id` parameter, the value of which is adjusted to identify the processor number of the auxiliary processor, followed by the processor number of the parent interpreter session.

Windows Opened by APL2

When APL2 is started, two windows are typically created, though this can be affected by the way that you start the product.

1. The APL2 session manager. The session manager is a Motif program that manages the interaction between the user and the APL2 language interpreter. See Using the Session Manager for details of its usage.

   This window is not created if the default `-sm on` invocation option is overridden.

2. The interpreter console. The interpreter console window is used to display output from commands invoked through `)HOST` or AP 100, and for editing sessions through `)EDITOR nnnn`. Some informational and error messages may also be displayed in the console window.

   If APL2 is invoked with `-sm off`, but the input or output have not been redirected, then APL session input or output also goes through this window.

   Ctrl-C can be used from this window to interrupt the interpreter, and closing this window terminates the APL2 session. However, since signals and session termination are normally controlled from the session manager window, cleanup may not be complete if this technique is used.

   The interpreter window is normally opened as a separate X window, but can be suppressed by using the `-hostwin off` invocation option. In this case, output that would go to the interpreter window will go to the X window from which APL2 was started.

In addition, the *SVP Monitor* window is opened if the invocation option `-svptrace on` has been specified. This window can be used to monitor APL2 shared variable activity.

Additional windows can be opened by applications that are started automatically when APL2 is invoked.

Invocation Syntax

```
apl2 [optional parameters]
```

In addition to passing options as parameters to APL2, the `apl2.ini` file and environment variables may be used to control the APL2 session. For most session controls, any one of those methods may be used.

© Copyright IBM Corporation 1994, 2017

- Invocation Parameters
- APL2 Environment Variables
- System Environment Variables

Invocation Parameters

Parameter keywords can be entered in uppercase, lowercase, or mixed case, and must be marked with a leading hyphen, followed immediately by the keyword.

When placed in the `apl2.ini` file, the same parameter keyword is used (without the leading hyphen), followed immediately by = and the parameter value. See The `apl2.ini` Configuration File for more information.

Interpreter invocation options can also be set using environment variables. The environment variable names match the parameter keywords, but with a prefix of `APL`. For example, `APLID` is the environment variable name used for the `id` parameter. Environment variables must be set before invoking APL2.

Where parameters are specified they override the corresponding `apl2.ini` entries and environment variables. `apl2.ini` entries override their corresponding environment variables, except for `hostwin` and `svptrace`. Those options and the windows they control are managed by the `apl2` shell script. Values specified for them in `apl2.ini` are ignored.

The following is the list of valid interpreter invocation parameters for APL2 on Unix systems:

`-207fl` *`path`*
: Path to the AP 207 vector font files.
If not specified, or if a requested font is not found in the specified path, `/usr/APL2/fonts/vector/` is searched.

`-hostwin {on|off|`**`icon`**`}`
: Controls the visibility of the interpreter console window.
If this option is set to `icon`, the interpreter's console window will come up iconified.
If it is set to `off`, a separate console window will not be started. The Xterm window from which APL2 was invoked will serve as the console window.

`-id` *`procid`* `[,`*`parentid`* `[,`*`pparentid`*`]]`
: Sets user ID number, as reported by `⎕AI` and used when sharing variables with the session. Can also be used to specify a parent and grandparent ID for APL sessions that are to be subordinate to others.
If `-id` is not specified or *`procid`* is `0`, defaults to the first unused number greater than 1000. If *`procid`* is negative, requests the first unused number greater than or equal to the absolute value of *`procid`*.
The use of IDs of 1000 or less is not advised unless the session is being used as an auxiliary processor.

`-input " '`*`line1`*`' ['`*`line2`*`']...."`
: Defines initial lines of input to be used when APL2 is invoked.
Each line of input is enclosed in single quotation marks and the group of one or more lines of input is enclosed in double quotes. All displayable characters can be included within the quoted lines. Use two single quote marks for each one that is to be treated as part of the input.
Note: If an environment variable is used, the outer set of double quotes is not necessary.

`-isol {cs|`**`rr`**`|rs|ur}`
: Sets the default isolation level for AP 127 and AP 227

© Copyright IBM Corporation 1994, 2017

`-javaxmx` *`heap_size`*

Controls the maximum size, in bytes, of the memory heap used by the Java Virtual Machine started by Associated Processor 14. Append the letter k or K to indicate kilobytes, or m or M to indicate megabytes. For more information, consult the documentation of your Java implementation's `-Xmx` invocation option

`-lx {`**`on`**`|off}`

Controls execution of latent expression.

If `off` is specified, execution of the latent expression, `⎕LX`, will be suppressed when workspaces are loaded.

`-nlt` *`filename`*

Specifies which national language file to use for system commands and messages.

The files provided with APL2 are:

Language	**Filename**
US English	`/usr/APL2/nls/En_US/cmd_msg.txt`
UK English	`/usr/APL2/nls/En_UK/cmd_msg.txt`
German	`/usr/APL2/nls/Gr_GR/cmd_msg.txt`
French	`/usr/APL2/nls/Fr_FR/cmd_msg.txt`
Italian	`/usr/APL2/nls/It_IT/cmd_msg.txt`

The APL2 shell script chooses one of these files based on the value of system environment variable `$LANG`.

If the specified file is not found, file `/usr/APL2/nls/En_US/cmd_msg.txt` is used.

If `/usr/APL2/nls/En_US/cmd_msg.txt` is not found, built-in US English tables are used.

`-odbc` *`library`*

Specifies the name of the shared library containing ODBC routines for AP 227.

If not specified, defaults to `libodbc.so`.

`-p11` *`filespec`*

Processor 11 NAMES files. The value can be a single file name, or a series of file names separated by `:` (colon) characters. If multiple file names are provided, Processor 11 will search the files in order until a match is found.

If not specified, defaults to `/usr/APL2/defaults/aplnm011.nam`.

`-prtg` *`file`*

Name of print shell file required to print the AP 207 screen.

If not specified, defaults to `/usr/APL2/bin/graphprt`.

`-pw` *`printwidth`*

Specifies the number of characters in each line of interpreter output. This value will be the initial value of system variable `⎕PW`.

If *`printwidth`* is not a number between 30 and 254, the default value of 79 will be used.

`-quiet {on|`**`off`**`}`

Controls suppression of initial output.

If `on` is specified, interpreter output is suppressed until input is requested.

Input requests satisfied by the APL2 input stack do not reset `-quiet on`. Data is placed on the APL2 input stack either by the `-input` parameter or through AP 101.

`-rns function[:locator]`

Associates a name with and runs the niladic function `function` in the namespace identified by `locator`.

© Copyright IBM Corporation 1994, 2017

If `locator` is supplied, the following association will be performed:

```
'locator' 11 ⎕NA 'function'
```

If `locator` is omitted, the following association will be performed:

```
3 11 ⎕NA 'function'
```

`-sm {`**`on`**`|off|piped|`*`number`*`}`

Indicates the session manager to be used

`on` selects the default local session manager.

`off` indicates no session manager. This setting would be used for line-by-line I/O devices, or programs which emulate that behavior, such as typing directly into an Xterm Window.

`piped` is used for batch processing, whether the session is driven by a file or by another process. See Running APL2 in Batch Mode for more information on batch processing.

`number` indicates a remote session manager. See Processor Network Identification for information on how to define the association between *`number`* and a remote session manager. See Running a Remote Session Manager for information on how to set up the remote session manager.

`-svptrace {on|`**`off`**`|log}`

Enables display or logging of shared variable processor events.

`on` causes the *SVP Monitor* window to start and traces to be sent to that window.

`off` disables tracing.

`log` directs traces to the file identified by environment variable `APL2SVPLOG`.

`-tcl` *`tcl_library_name`*
`-tk` *`tk_library_name`*

Set the names of the Tcl and Tk libraries used by the `TCL` external function and the Tk_Init builtin Tcl command. For further information, see `TCL`.

`-ws` *`initial`*`[,`*`maximum`*`[,`*`increment`*`]]`

Controls workspace size.

Workspace size can be specified as a single value for a static size, or up to three values to request an expandable workspace. Each of the values provided are of the form:

`integer``[k|m]`

where `k` and `m` indicate kilobytes and megabytes, respectively.

If *`increment`*, or both *`maximum`* and *`increment`*, are omitted, they default to *`initial`*. The default for *`initial`* is 10M. The minimum value for *`initial`* is 32K. Values smaller than 32K will be rounded up to 32K.

The maximum allowed by APL2 for all three values is 2047M. However, the operating system may impose additional restrictions on the amount of storage it will allow APL2 to allocate, and sizes greater than the amount of available memory may cause performance degradation due to paging.

Here are some examples:

`-ws 5m`

The session is run with a 5 megabyte workspace. `WS FULL` is reported if this is not enough.

`-ws 2m,7m`

The session begins with a 2 megabyte workspace. Since the increment defaults to the initial value, this is expanded to 4 megabytes, 6 megabytes, and finally 7 megabytes as needed to avoid `WS FULL`.

© Copyright IBM Corporation 1994, 2017

`-ws 500k,1m,100k`
The session begins with a 500 kilobyte workspace. This is expanded as needed, 100 kilobytes at a time, but never exceeding 1 megabyte.

The following additional invocation parameters can be specified for the session manager (these options cannot be specified in `apl2.ini` or set in environment variables):

`-aplmode {`**`on`**`|off}`
Sets the initial keyboard mode to APL (`on`) or text (`off`) mode.

`-bg` *`color`*
Sets background color for the menu bar and the status components. Can be any valid color name.

`-copyfile` *`filename`*
Sets the name of the file for session copy. The default file name is `apl2Copy.txt`.

`-copy {on|`**`off`**`}`
Sets the initial copy mode

`-cr` *`color`*
Sets the color of the text cursor. Can be any valid color name.

`-display` *`name`*
Sets the `DISPLAY` environment variable to `name`. For example:

```
apl2 -display hostname:0.0
```

sets the output display device to display monitor 0 attached to the computer `hostname`.

`-fg` *`color`*
Sets the foreground color of the menu bar and status components. Can be any valid color name.

`-fn` *`fontname`*
Selects the session manager font. Possible choices for *`fontname`* are `apl6`, `apl10`, `apl12`, `apl14` and `apl22`. The default is `apl14`.

`-geometry [`*`widthxheight`*`] [{+|-}`*`xoff`*`{+|-}`*`yoff`*`]`
Sets the session window size and/or location.
Width and height are specified in character units. The x and y offsets are specified in pixels from the top left of the display to the top left of the window for positive values, or from the bottom right of the display to the bottom right of the window for negative values. The associated environment variable is `APLGEOM`. The default value is `80x25`.

`-log` *`filename`*
Selects the session manager log file.
The default is `./apl2ses.log`, which causes it to be stored in the current directory.

`-ms` *`color`*
Changes the color of the mouse pointer.

`-prof` *`filename`*
Sets the `XENVIRONMENT` variable to this file name. It can contain settings for colors, keyboard translations, widget labels, and other resources.

`-scroll {page|`**`line`**`}`
Sets the initial session manager scrolling mode.

`-xrm` *`string`*
Sets additional session manager resources. For more information, see Command Line Parameters.

© Copyright IBM Corporation 1994, 2017

APL2 Environment Variables

Environment variables can be set prior to invoking APL2 to control certain aspects of the session. Setting an environment variable differs by shell type. To set an environment variable, enter the following:

```
% setenv variable value              (C shell)
$ variable=value; export variable    (Bourne or Korn Shell)
```

Where *variable* is the name of the environment variable and *value* is the assigned content. Note that the variable name must also be exported. Variable names and their values are case sensitive. Any child processes spawned from the session, such as dependent auxiliary processors, inherit the environment variables set by the parent.

In addition to the variables listed here, the interpreter invocation parameters can also be specified as environment variables. The variable names match the parameter keywords, but with a prefix of `APL`. Where parameters are specified, they override the corresponding environment variable.

APL2
: Name of the directory in which APL2 is installed.

APL2SMKEY
: Name of the shared variable key. This value is used to build unique names for the resources used by the Shared Variable Processor. The default is `APL2SVP`.

APL2SVPLOG
: Name of the Shared Variable Processor trace file. This can be a simple file name or a fully qualified path and file name. The default is `apl2svp.trc`.

APL2SVPPARMS
: Name of the file that contains the Shared Variable Processor initialization parameters. This can be a simple file name or a fully qualified path and file name. The default is `apl2svp.prm`.

APL2SVPPRF
: Name of the user's Shared Variable Processor profile. This file contains entries that identify and authorize remote processors and independent local processors. This can be a simple file name or a fully qualified path and file name. The default is `apl2svp.prf`.

APL*nnnnn*
: Where *nnnnn* is a five-digit number such as 01234. Used to resolve library numbers with the system commands. A separate environment variable must be defined for each library accessed by number. Each of them is set to a directory path specification.
  For more information, see Library Specification and the `LIBS` function.

Note:
APL2SMKEY, APL2SVPPARMS and APL2SVPPRF can also be specified in the [Shared Variable Processor] section of `apl2.ini`.
APL*nnnnn* settings can be specified in the [Libraries] section of `apl2.ini`. See The `apl2.ini` Configuration File for more information.

System Environment Variables

The following list describes the operating system environment variables on which APL2 depends.

© Copyright IBM Corporation 1994, 2017

CLASSPATH
: Location of Java class (.class) and archive (.jar) files used by Processor 14 and APL2PI. For more information, see Installing the APL2-Java Interface Classes.

DISPLAY
: Specifies where to display a session in X environment. The format is `host:server_number.screen_number`.

HOME
: Specifies the full path name of the user's logon directory.

LANG
: Specifies the locale name currently in effect.

PATH
: Sequence of directories used to search for an executable file or command.

SHELL
: Specifies the full path name of the system command interpreter.

TZ
: Time zone in effect. Format is *sssoffsetddd[offset,start[/time],end[/time]*, where *sss* is the standard time identifier, *offset* is the offset from GMT, *ddd* is the daylight savings time name, *start* is the start day for daylight savings time, and *end* is the end day for daylight savings time. For example, `PST8PDT` is the setting for California in the USA, using default start and end dates for daylight savings time. See your operating system documentation or the man pages for function *tzset* for complete syntax information on TZ.

XENVIRONMENT
: Set to the file name specified with the -prof invocation option. The file is an X Window System resource file.

Examples of APL2 Invocation

`apl2`
: A common, simple form

`apl2 ?`
: For invocation help

`apl2 -ws 4m -input "')LOAD 1 DISPLAY' 'DISPLAY ''HI'''"`
: Invokes APL2 with a workspace size of 4 megabytes, and loads the public workspace `1 DISPLAY`. The `DISPLAY` function is then executed with an argument of `'HI'`.

`apl2 -fn apl22`
: APL2 is invoked with display font `apl22`

Running APL2 in Batch Mode

Batch mode enables you to redirect APL2 session input and output. In this mode, input is redirected from text files containing what would normally be the keyboard input during an interactive session. The resulting output can then be redirected to another file or process. For example, suppose that the file `session.in` contains the following input lines:

```
2+2
)OFF
```


© Copyright IBM Corporation 1994, 2017

Suppose APL2 is invoked with this as redirected input, and the output is redirected to a file called `session.out`, using:

```
apl2 -sm piped < session.in > session.out
```

The `session.out` file then contains the session manager output:

```
  ...
CLEAR WS
      2+2
4
```

If the input file ends before an `)OFF` or `)CONTINUE` command is encountered in the input file, the session is terminated with an `)OFF` command.

Additional Considerations for Remote Access

If the display output from APL2 is to be sent to a remote workstation (X Server), then these steps should be followed:

1. Make sure the APL2 fonts have been installed on the X Server. See Displaying APL2 Characters for more information.
2. Execute the `xhost` command on your local workstation so that the remote server where APL2 is installed can make X Window System requests to your local workstation:

   ```
   xhost server_hostname
   ```

3. Log in to the remote server from your local workstation and invoke APL2 with the `-display` parameter:

   ```
   apl2 -display local_hostname:0.0
   ```

   Additional parameters can be included with the previous statement. Refer to Invocation Parameters for details.

© Copyright IBM Corporation 1994, 2017

Invoking APL2 on Windows Systems

This material describes the APL2 environment created on your desktop, techniques you can use to create that environment, parameters you can pass to the APL2 product, and environment variables that affect the way it behaves.

- APL2 on the Desktop
- Windows Opened by APL2
- Invocation Syntax
- Examples of APL2 Invocation

APL2 on the Desktop

An *IBM APL2* entry was created in the **Programs** menu of the **Start** action item during the installation process. When you select that entry, it opens the *IBM APL2* folder, where you will find documentation and program icons. APL2 is normally started by opening the *APL2win* program icon. The shared variable processor will be started automatically by APL2, or it can be started independently by opening the *SVP Monitor* program icon.

Invocation parameters for both APL2 and the shared variable processor can be specified in the Properties notebook for their program icons, if desired.

See Modifying the APL2 Desktop Object for more information on the customizing the desktop objects.

Windows Opened by APL2

When APL2 is started, two windows are typically created, though this can be affected by the way that you start the product. The windows are included on the Taskbar, and may or may not be displayed on the desktop:

1. The APL2 session manager. The session manager manages the interaction between the user and the APL2 language interpreter. See Using the Session Manager for details of its usage.

   This window is not created if the default `-sm on` invocation option is overridden.

2. The interpreter console. The interpreter console window is used to display output from commands invoked through `)HOST` or AP 100. Some informational and error messages may also be displayed in the console window.

   If APL2 is invoked with `-sm off`, but the input or output have not been redirected, then APL session input or output also goes through this window.

   The interpreter window is opened as a new command window, unless you have issued the APL2 command directly from an MS/DOS Command Prompt. In that case, the original command window is used as the console.

   Ctrl-Break can be used from this window to interrupt the interpreter, and closing this window terminates the APL2 session. However, since signals and session termination are normally controlled from the session manager window, cleanup may not be complete if this technique is used.

© Copyright IBM Corporation 1994, 2017

The interpreter window can be suppressed by using the `-hostwin off` invocation option. This option is useful for hiding the console from end users of an APL2 application, but since it also hides the `)HOST` output and messages, it should be used with care.

In addition, the *SVP Monitor* window is opened automatically if the invocation option `-svptrace` has been specified to the interpreter, or is opened explicitly by clicking on the *SVP Monitor* icon. This window can be used to monitor APL2 shared variable activity, to obtain information about processors and shared variables, initiate the programs used for cooperative processing, and to clean up after processors that may not have terminated normally. On-line help is provided within the window with details of its usage. For more information, see The SVP Monitor Facility.

Both the interpreter window and the *SVP Monitor* window are often minimized or hidden in normal usage. When APL2 is started from the *APL2win* icon, the interpreter window state is determined by the options specified in the Properties notebook for that icon. When the SVP monitor is started from the *SVP Monitor* icon, its state is determined by the options specified in the Properties notebook for that icon.

Additional windows can be opened by applications that are started automatically when APL2 is invoked.

Invocation Syntax

If APL2 is not started with a program icon, it can be started from an MS/DOS Command Prompt as follows:

```
apl2win [optional parameters]
```

to invoke the APL2 executable module directly, or

```
apl2 [optional parameters]
```

to use the command file APL2.BAT.

In addition to passing options as parameters to APL2, the `apl2.ini` file and environment variables may be used to control the APL2 session. For most session controls, any one of those methods may be used.

- Invocation Parameters
- APL2 Environment Variables
- System Environment Variables

Invocation Parameters

Parameter keywords can be entered in uppercase, lowercase, or mixed case, and must be marked with a leading slash or hyphen, followed immediately by the keyword.

When placed in the `apl2.ini` file, the same parameter keyword is used (without the leading slash or hyphen), followed immediately by = and the parameter value. See The `apl2.ini` Configuration File for more information.

© Copyright IBM Corporation 1994, 2017

Invocation options can also be set using environment variables. The environment variable names match the parameter keywords, but with a prefix of `APL`. For example, `APLID` is the environment variable name used for the `id` parameter. Environment variables must be set before invoking APL2.

Where parameters are specified they override the corresponding `apl2.ini` entries and environment variables. `apl2.ini` entries override their corresponding environment variables.

The following is the list of valid invocation parameters for APL2 on Windows:

`-207fl` *`path`*
: Path to the AP 207 vector font files.
If not specified, or if a requested font is not found in the specified path, standard operating system search order is used.

`-hostwin {`**`on`**`|off}`
: Controls the visibility of the interpreter console window.
If this option is set to `off`, the interpreter's console window will be hidden.
Note: If the interpreter window is hidden, output from `)HOST` and any other messages sent to the console will also be hidden.

`-id` *`procid`* `[,`*`parentid`* `[,`*`pparentid`*`]]`
: Sets user ID number, as reported by `⎕AI` and used when sharing variables with the session. Can also be used to specify a parent and grandparent ID for APL sessions that are to be subordinate to others.
If `-id` is not specified or *`procid`* is `0`, defaults to the first unused number greater than 1000. If *`procid`* is negative, requests the first unused number greater than or equal to the absolute value of *`procid`*.
The use of IDs of 1000 or less is not advised unless the session is being used as an auxiliary processor.
Note: On Windows, .BAT file processing removes commas between parameters. If you are using one of the forms of this parameter which contains commas, and you are passing options via the APL2.BAT file, surround the parameter value with double quotes to retain the commas. For example:

```
-id "127,1001"
```

`-input " '`*`line1`*`' ['`*`line2`*`']...."`
: Defines initial lines of input to be used when APL2 is invoked.
Each line of input is enclosed in single quotation marks and the group of one or more lines of input is enclosed in double quotes. All displayable characters can be included within the quoted lines. Use two single quote marks for each one that is to be treated as part of the input.
Note: If an environment variable is used, the outer set of double quotes is not necessary.

`-isol {cs|`**`rr`**`|rs|ur}`
: Sets the default isolation level for AP 127 and AP 227

`-javaxmx` *`heap_size`*
: Controls the maximum size, in bytes, of the memory heap used by the Java Virtual Machine started by Associated Processor 14. Append the letter k or K to indicate kilobytes, or m or M to indicate megabytes. For more information, consult the documentation of your Java implementation's `-Xmx` invocation option

`-lx {`**`on`**`|off}`
: Controls execution of latent expression
If `off` is specified, execution of the latent expression, `⎕LX`, will be suppressed when workspaces are loaded.

© Copyright IBM Corporation 1994, 2017

`-nlt` *`filename`*

Specifies which national language file to use for system commands and messages.
The files provided with APL2 are:

Language	**Filename**
US English	`En_US.nlt`
UK English	`En_UK.nlt`
German	`Gr_GR.nlt`
French	`Fr_FR.nlt`
Italian	`It_IT.nlt`

If the specified file is not found, or no file is specified, file `En_US.nlt` is used.
If `En_US.nlt` is not found, built-in US English tables are used.

`-odbc` *`library`*

Specifies the name of the shared library containing ODBC routines for AP 227.
If not specified, defaults to `odbc32.dll`.

`-p11` *`filespec`*

Processor 11 NAMES files. The value can be a single file name, or a series of file names separated by `;` (semicolon) characters. If multiple file names are provided, Processor 11 will search the files in order until a match is found.
If not specified, defaults to `aplnm011.nam`.

`-pw` *`printwidth`*

Specifies the number of characters in each line of interpreter output. This value will be the initial value of system variable `⎕PW`.
If *`printwidth`* is not a number between 30 and 254, the default value of 79 will be used.

`-quiet {on|`**`off`**`}`

Controls suppression of initial output.
If `on` is specified, interpreter output is suppressed until input is requested.
Input requests satisfied by the APL2 input stack do not reset `-quiet on`. Data is placed on the APL2 input stack either by the `-input` parameter or through AP 101.

`-rns function[:locator]`

Associates a name with and runs the niladic function `function` in the namespace identified by `locator`.

If `locator` is supplied, the following association will be performed:

```
'locator' 11 ⎕NA 'function'
```

If `locator` is omitted, the following association will be performed:

```
3 11 ⎕NA 'function'
```

`-run` *`dllname`*

Runs the workspace packaged into the Windows DLL identified by *`dllname`*.

`-sm {`**`on`**`|off|piped|`*`number`*`}`

Indicates the session manager to be used.

- `on` selects the default local session manager.
- `off` indicates no session manager. This setting would be used for line-by-line I/O devices, or programs which emulate that behavior, such as typing directly into an MS/DOS Command Prompt.

© Copyright IBM Corporation 1994, 2017

`piped` is used for batch processing, whether the session is driven by a file or by another process. See Running APL2 in Batch Mode for more information on batch processing.

`number` indicates a remote session manager. See Processor Network Identification for information on how to define the association between *`number`* and a remote session manager. See Running a Remote Session Manager for information on how to set up the remote session manager.

`-svplisten {on|`**`off`**`}`

Enables automatic startup of the SVP port server.

`-svptrace {on|`**`off`**`|log|both}`

Enables display and/or logging of shared variable processor events. Causes the *SVP Monitor* window to start, if not already started.

`on` causes the traces to be sent to the *SVP Monitor* window.

`off` disables tracing.

`log` directs tracing to the file identified by environment variable `APL2SVPLOG`.

`both` sends the trace message to both the window and the file.

`-tcl` *`tcl_library_name`*
`-tk` *`tk_library_name`*

Set the names of the Tcl and Tk libraries used by the `TCL` external function and the Tk_Init builtin Tcl command. For further information, see `TCL`.

`-ws` *`initial`*`[,`*`maximum`*`[,`*`increment`*`]]`

Controls workspace size.
Workspace size can be specified as a single value for a static size, or up to three values to request an expandable workspace. Each of the values provided are of the form:

`integer``[k|m]`

where `k` and `m` indicate kilobytes and megabytes, respectively.
If *`increment`*, or both *`maximum`* and *`increment`*, are omitted, they default to *`initial`*. The default for *`initial`* is 10M. The minimum value for *`initial`* is 32K. Values smaller than 32K will be rounded up to 32K.
The maximum allowed by APL2 for all three values is 2047M. However, the operating system may impose additional restrictions on the amount of storage it will allow APL2 to allocate, and sizes greater than the amount of available memory may cause performance degradation due to paging.

Here are some examples:

`-ws 5m`
The session is run with a 5 megabyte workspace. `WS FULL` is reported if this is not enough.
`-ws 2m,7m`
The session begins with a 2 megabyte workspace. Since the increment defaults to the initial value, this is expanded to 4 megabytes, 6 megabytes, and finally 7 megabytes as needed to avoid `WS FULL`.
`-ws 500k,1m,100k`
The session begins with a 500 kilobyte workspace. This is expanded as needed, 100 kilobytes at a time, but never exceeding 1 megabyte.

Note: On Windows, .BAT file processing removes commas between parameters. If you are using one of the forms of this parameter which contains commas, and you are passing options via the APL2.BAT file, surround the parameter value with double quotes to retain the commas. For example:

© Copyright IBM Corporation 1994, 2017

```
-ws "500k,1m,100k"
```

APL2 Environment Variables

Environment variables can be set prior to invoking APL2 to control certain aspects of the session. To set an environment variable, enter the following:

```
SET variable=value
```

Where *variable* is the name of the environment variable and *value* is the assigned content. Environment variables names are always folded to uppercase, but their values are case sensitive. Any auxiliary processors that are automatically started by the APL2 session inherit the environment variables set by the parent.

`SET` commands can be issued from the MS/DOS Command Prompt where APL2 will be started. They can also be added to the APL2.BAT invocation file or in the **System** notebook in **Control Panel**. See Specifying Invocation Parameters for more information.

In addition to the variables listed here, the interpreter invocation parameters can also be specified as environment variables. The variable names match the parameter keywords, but with a prefix of `APL`. Where parameters are specified, they override the corresponding environment variable.

APL2SVPLOG
: Name of the Shared Variable Processor trace file. This can be a simple file name or a fully qualified path and file name. The default is `apl2svp.trc`.

APL2SVPPARMS
: Name of the file that contains the Shared Variable Processor initialization parameters. This can be a simple file name or a fully qualified path and file name. The default is `apl2svp.prm`.

APL2SVPPRF
: Name of the user's Shared Variable Processor profile. This file contains entries that identify and authorize remote processors and independent local processors. This can be a simple file name or a fully qualified path and file name. The default is `apl2svp.prf`.

APL*nnnnn*
: Where *nnnnn* is a five-digit number such as 01234. Used to resolve library numbers with the system commands. A separate environment variable must be defined for each library accessed by number. Each of them is set to a directory path specification.
For more information, see Library Specification and the `LIBS` function.

Note:
APL2SVPLOG, APL2SVPPARMS and APL2SVPPRF can also be specified in the [Shared Variable Processor] section of `apl2.ini`.
APL*nnnnn* settings can be specified in the [Libraries] section of `apl2.ini`. See The `apl2.ini` Configuration File for more information.

System Environment Variables

The following list describes the operating system environment variables on which APL2 depends.

CLASSPATH

© Copyright IBM Corporation 1994, 2017

Location of Java class (.class) and archive (.jar) files used by Processor 14 and APL2PI. For more information, see Installing the APL2-Java Interface Classes.

PATH

Sequence of directories used to search for an executable file or command and for non-executable data files

Notes:

1. Previously, environment variable TZ was used to determine time zone. If set, TZ will still be used. However, as of Service Level 7 of Workstation APL2 2.0, if TZ is not present, Windows operating system services will be used to determine time zone. Since the operating system services reflect more accurately the latest policies on dates for changing to and from daylight savings time, we recommend that TZ not be set on Windows.
2. Previously, environment variable BOOKSHELF was required to locate the APL2 on-line documentation. If set, the paths in BOOKSHELF will still be searched. In addition, the APL2 installation directories are always searched.

Examples of APL2 Invocation

The following sections provide examples of how to invoke APL2.

- From the Desktop, Using a Mouse
- From a Command Window
- Driving an APL Application from the Desktop
- Running APL2 in Batch Mode

From the Desktop, Using a Mouse

1. Click on the **Start** button on the action bar, go to **Programs**, and select the *IBM APL2* entry.

   This opens the APL2 folder.

2. Double-click mouse button 1 on the *APL2win* icon within that folder.

   This starts an APL2 session using any previously assigned options. To change those options, single-click mouse button 2 on the *APL2win* icon and select **Properties**.

From a Command Window

1. Switch to the drive and directory that you want to use as the working directory during the APL2 session. Typically this is the directory where your private workspaces are stored. For example:

   ```
   [C:\]d:
   [D:\]cd wslib
   [D:\WSLIB]
   ```

2. Set any needed environment variables that are not already set, and are not set automatically by the command file you are going to use. (You cannot change these after entering APL2.) The following example defines APL libraries 50 and 1002, pointing library 1002 to the current directory.

© Copyright IBM Corporation 1994, 2017

```
[D:\WSLIB]set apl00050=e:\lib50
[D:\WSLIB]set apl01002=.\
[D:\WSLIB]
```

3. Start the APL2 session, providing any needed options. This example sets a workspace size of 5 megabytes and suppresses automatic execution of □LX expressions in workspaces that are loaded:

   ```
   [D:\WSLIB]apl2win -ws 5m -lx off
   ```

Driving an APL Application from the Desktop

The provider of the application can do the following:

1. Create a program icon. You can copy an existing one or drag the system's program template to another directory.
2. Fill in the full path and file name of the command file to be run, or if no command file is desired, the APL2 interpreter executable file, `apl2win.exe` in the `\BIN` subdirectory of the directory where the APL2 product was installed.
3. Select a working directory based on any needs the application may have.
4. Provide optional parameters as needed. For example:

   ```
   -quiet on -sm off -ws 4m -input "')LOAD ''applic.APL'''"
   ```

   Note that the complete filename form of the `)LOAD` command is used in the `-input` parameter. Using this form avoids any dependencies on environment variables. Drive and directory information can be included if the workspace is not stored in the working directory.

Assuming that the workspace contains a latent expression, it begins executing automatically when the user opens the program object.

Running APL2 in Batch Mode

Batch mode enables you to redirect APL2 session input and output. In this mode, input is redirected from text files containing what would normally be the keyboard input during an interactive session. The resulting output can then be redirected to another file or process. For example, suppose that the file `session.in` contains the following input lines:

```
2+2
)OFF
```

Suppose APL2 is invoked with this as redirected input, and the output is redirected to a file called `session.out`, using:

```
apl2win -sm piped < session.in > session.out
```

The `session.out` file then contains the session manager output:

© Copyright IBM Corporation 1994, 2017

```
 ...
CLEAR WS
      2+2
4
```

If the input file ends before an)OFF or)CONTINUE command is encountered in the input file, the session is terminated with an)OFF command.

© Copyright IBM Corporation 1994, 2017

Migrating from Version 1 to Version 2

- Features of Version 2
- Migrating Workspaces to Version 2
- Windows Considerations

© Copyright IBM Corporation 1994, 2017

Features of Version 2

APL2 Version 2 includes all the features of the APL2 Version 1 systems for AIX, Solaris and Windows. In addition, it includes a new APL2 system for Linux, and adds the following new features to all the systems:

- Namespaces

  This facility allows encapsulation of a saved APL2 workspace into a special format that can be accessed from the active workspace without bringing the entire namespace into the active workspace, and without exposing the active workspace to the entire set of names in the namespace.

  For more information, see Accessing Namespaces.

- On-line Documentation in PDF format

  In APL2 Version 1, on-line documentation was available only on Windows, in an IBM VisualAge format.

  With APL2 Version 2, the following on-line manuals are provided for all the systems in Adobe PDF format:

  - *APL2 User's Guide* (this manual)
  - *APL2 Language Summary*
  - *APL2 Programming: Language Reference*
  - *APL2 Programming: Developing GUI Applications* (for Windows only)
  - *APL2 Programming: Using SQL*
  - *APL2 Programming: Using APL2 with WebSphere*
  - *APL2 GRAPHPAK: User's Guide and Reference*

  The Adobe Acrobat Reader is required to view these manuals.

- The APL2 Runtime Library

  APL2 Version 2 includes a special runtime interpreter and a pre-packaged Runtime Library for each of the operating system platforms.

  The Runtime Library is a subset of the full APL2 product that will run applications that obey its restrictions. It can be re-distributed freely and used to run these applications on machines where a full APL2 system is not available.

  For more information on the APL2 Runtime Library, see The APL2 Runtime Library.

© Copyright IBM Corporation 1994, 2017

Migrating Workspaces to Version 2

Due to the addition of the namespaces function (described above), the internal format of the saved APL2 workspace has changed in Version 2.

When you)LOAD an APL2 workspace saved under Version 1 into a Version 2 system running on the same operating system, the workspace will automatically be converted to the new format. If you)SAVE it, it will be saved in the new format.

Once saved in the new format under Version 2, the workspace can no longer be loaded in a Version 1 APL2 system. If you wish to migrate the workspace back to Version 1, you can do so with the)OUT and)IN commands. This is the same methodology used to migrate workspaces between different operating systems. See Workspace Transfer Between APL2 Systems for details.

© Copyright IBM Corporation 1994, 2017

Windows Considerations

APL2 for Windows Version 1 had a companion product, *APL2 Runtime Environment*, which consisted of a Workspace Packager and Runtime Library. The Workspace Packager function `PACKAGE` encapsulated APL2 workspaces into Windows DLLs. These packaged workspaces could be run on the full APL2 for Windows or the Runtime Library using the `-run` invocation option.

In APL2 Version 2, the new namespaces facility and the new APL2 Runtime Library provide similar facilities, with the additional advantage of namescope isolation. It is expected that new applications will use the new facilities.

For upward compatibility, however, the `PACKAGE` external function is being shipped with the APL2 Version 2 system for Windows, and the `-run` invocation option is still supported by the APL2 Version 2 system for Windows:

1. You can run package DLLs created by the Workspace Packager under APL2 for Windows Version 1 on Version 2, with either full APL2 system or the APL2 Runtime Library.
2. You can re-package your workspaces under Version 2 if necessary. You can use the 2 PACKAGE workspace from your Version 1 system with Version 2, or use the `PACKAGE` external function directly.
3. Once a workspace has been re-packaged using Version 2, it can no longer be run on the Version 1 systems.
4. All application restrictions noted in the documentation provided with the Workspace Packager still apply under APL2 Version 2 when using `-run`.

© Copyright IBM Corporation 1994, 2017

General Information

- The APL2 Interpreter
- An Example of the Use of APL2
- Characteristics of APL2

© Copyright IBM Corporation 1994, 2017

The APL2 Interpreter

The APL2 *interpreter* takes one APL2 statement at a time, converts it to *machine instructions* (the computer's internal language), executes it, and then proceeds to the next line. In contrast to program compilers that convert an entire program to machine language before executing any statements, APL2 allows you a high degree of interaction with the computer. If something you enter is invalid, you get quick feedback on the problem before you go any further. This also yields high productivity gains.

© Copyright IBM Corporation 1994, 2017

An Example of the Use of APL2

A statement entered at the keyboard can contain numbers or symbols, such as + - × ÷, or names formed from valid combinations of letters (as described in *APL2 Programming: Language Reference*). The numbers and special symbols stand for the primitive objects and functions of APL2 - primitive in the sense that their meanings are permanently fixed, and therefore understood by the APL2 system without further definition. A name, however, has no significance until a meaning has been assigned to it.

Names are used for two major categories of objects. Names can be used for collections of data that are composed of numbers or characters; such a named collection is called a *variable*. Names can also be used for programs made up of sequences of APL2 statements; such programs are called *defined functions and operators*. Once they have been established, names of variables and defined operations can be used in statements by themselves or in conjunction with primitive functions and objects.

An Isolated Calculation

If the work to be done can be adequately specified simply by typing a statement made up of numbers and symbols, names are not required; entering the expression to be evaluated causes the result to be displayed. For example, suppose you want to compare the rates of return on money at a fixed interest rate but with different compounding intervals. For 1000 units at 6% compounded annually, quarterly, monthly, or daily for 10 years, the entry and response for the transaction (assuming a printing precision (⎕PP) equal to 6) looks like this:

```
      ⎕PP←6
      1000×(1+0.06÷1 4 12 365)*10×1 4 12 365
1790.85 1814.02 1819.4 1822.03
```

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

This example illustrates several characteristic features of APL2: familiar symbols such as + - × ÷ are used where possible; symbols are introduced where necessary (as the * for the power function); and a group of numbers can be worked on together.

Storing Functions and Data

Although many problems can be solved by typing the appropriate numbers and symbols, the greatest benefits of using APL2 occur when named functions and data are used. Because a single name can refer to a large array of data, using the name is far simpler than typing all of its numbers. Similarly, a defined function, specified by entering its name, can be composed of many individual APL2 statements that would be burdensome to type again and again.

Once a function has been defined, or data collected under a name, it is usually desirable to retain the significance of the names for some period of time - perhaps for just a few minutes, but more often for much longer, possibly months or years. For this reason APL2 systems are organized around the idea of a *workspace*, which might be thought of as a notebook in which all the data items needed during some piece of work are recorded together. An APL2 workspace thus contains defined functions, data structures, and a state indicator.

© Copyright IBM Corporation 1994, 2017

Characteristics of APL2

APL2 Programming: Language Reference describes APL2 in detail, giving the meaning of each symbol and discussing the various features of APL2. These details should be considered in light of the major characteristics of APL2, which can be summarized as follows:

- The primitive objects of the language are arrays (lists, tables, lists of tables, and so on). For example, `A + B` is meaningful for any conformable arrays `A` and `B`, the size of an array (`⍴A`) is a primitive function, and arrays can be indexed by arrays, as in `A[3 1 4 2]`.
- The syntax is simple:
  - There are only three statement types: name assignment, branch, or neither).
  - There is no function precedence hierarchy.
  - Functions have either one, two, or no arguments.
  - Primitive functions and defined functions (programs) are treated alike.
- The semantic rules are few:
  - The definitions of primitive functions are independent of the representations of data to which they apply.
  - All scalar functions are extended to other arrays in the same way - that is, item by item.
  - Primitive functions have no hidden effects (so-called *side-effects*).
- The sequence control is simple: one statement type embraces all types of branches (conditional, unconditional, computed, and so on), and the completion of the execution of any function always returns control to the point of use.
- External communications are established by means of variables, which are shared between an APL2 session and other processors. These processors can be APL or non-APL programs running on the same system or on another system. *Shared variables* are treated both syntactically and semantically like other variables. A subclass of shared variables - *system variables* - provides convenient communications between APL2 programs and their environment.
- The usefulness of the primitive functions is vastly expanded by *operators*, which modify their behavior in a systematic manner. For example, the primitive operator, *reduction* (denoted by `/`) modifies a function to apply over all elements of a list, as in `+/L` for summation of the items of `L`. The remaining primitive operators are:
  - *Scan* (running totals, running maxima, and so on)
  - The *axis* operator, which, for example, makes it possible to apply reduction and scan over a specified axis (rows or columns) of a table
  - The *each* operator, which applies a function to each of the elements of its arguments
  - The *outer product*, which produces tables of values as in `RATE∘.⋆YEARS` for an interest table
  - The *inner product*, a generalization of matrix product.
- The number of primitive functions and operators is sufficiently small that each is represented by a single, easily-read and easily-written symbol, yet the set of primitives embraces operations from simple addition to grading (sorting) and formatting.

© Copyright IBM Corporation 1994, 2017

Workspaces and Libraries

The common unit of storage in an APL2 system is the *workspace*. When in use, a workspace is said to be *active*, and is in main storage. Part of each workspace is set aside to serve the internal workings of the system; the rest is used, as required, to store items of information and to hold transient information generated during a computation.

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

Inactive workspaces are stored in *libraries* (that are operating system file directories), where they are identified by arbitrary names. The inactive workspaces are typically stored in `WSNAME.apl`. When required, copies of stored workspaces can be made active, or (if they are stored in an appropriate form) selected information can be copied from them into an active workspace.

Transfer files provide a way to store and transport APL objects outside of workspaces. They are particularly useful when transferring workspace objects between different APL2 implementations, because all APL2 products can read transfer files created by any other APL2 product. Transfer files are also used as an alternative to inactive workspaces. Transfer files can have arbitrary names, but the form `FILENAME.atf` is typically used.

Workspaces, libraries, and transfer files are managed by *system commands*, as described in *APL2 Programming: Language Reference*.

© Copyright IBM Corporation 1994, 2017

Library Specification

You can use environment variables or an `apl2.ini` configuration file to define the location of workspaces and transfer files. The variable names or `apl2.ini` file keywords have the form APL*nnnnn*, where *nnnnn* is the library number (a positive integer from 1 to 32767, left-padded to 5 digits with zeros). If the same library number is defined in `apl2.ini` and by an environment variable, the definition in `apl2.ini` takes precedence.

For more information on how to set environment variables or create the `apl2.ini` file, see the section on customizing APL2 in the Installing and Customizing APL2 section for your operating system.

Default Libraries

The following defaults for libraries are set by APL2:

Unix Systems:

The `apl2` shell script sets:

- `APL00001=$APL2/lib1`
- `APL00002=$APL2/lib2`

where `$APL2` is the directory into which APL2 was installed.
`APL01001` is set to the current working directory by the interpreter.

Windows Systems:

The APL2 installation process defines:

- `APL00001=C:\Program Files\IBMAPL2W\wslib1`
- `APL00002=C:\Program Files\IBMAPL2W\wslib2`
- `APL01001=C:\Program Files\IBMAPL2W\bin`

The directory path may be different if customized during installation.

User-Defined Libraries

Other APL*nnnnn* definitions can be created to support additional paths containing workspaces. For example, specifying:

```
APL02001=d:\hal\wkspaces
```

on Windows, or

```
APL02001=/u/hal/wkspaces
```

on Unix systems, causes APL2 to use the specified directory whenever a library number of 2001 is used. Then:

```
)WSID 2001 PLOT
)SAVE
```


© Copyright IBM Corporation 1994, 2017

Saves the active workspace in file `PLOT.apl` in that directory.

When system commands are specified with no explicit library number, a default library is determined as follows:

1. If a library definition APL*nnnnn* exists, where *nnnnn* matches the first element of `⎕AI`, its value is used as the workspace directory.
2. Else, if the library definition APL01001 is defined, its value is used.
3. Else, the APL01001 definition is automatically set to point to the current directory, which becomes the default library.

The `LIBS` function can be used to query the current library definitions and their associated paths.

Library definitions set in environment variables cannot be changed from within APL2. You must exit APL2, change the APL*nnnnn* environment variable, and restart APL2. Library definitions set in `apl2.ini` can be changed at any time by editing that file, even while APL2 is running. The changes will be recognized immediately.

© Copyright IBM Corporation 1994, 2017

Explicit File Specifications

As an alternative to library definitions, you can also specify a file name enclosed in single quotes in place of the workspace name and optional library number. For example:

```
)LOAD 'd:\hal\wkspaces\PLOT.apl'
)IN 'd:\hal\wkspaces\TEST.atf'
```

or

```
)LOAD '/u/hal/wkspaces/PLOT.apl'
)IN '/u/hal/wkspaces/TEST.atf'
```

Notes:

1. If a name enclosed in quotes is specified, it will be passed to the operating system exactly as given. The file type extension (`.apl` or `.atf`) must be included. The operating system may add path information if not given, according to its conventions.
2. If a name enclosed in quotes is set for the active workspace with `)LOAD` or `)WSID`, then a subsequent `)WSID` without parameters returns the same name (including the quotes).
3. The maximum length for a name enclosed in quotes is 248 characters (including the quotes).
4. On Unix systems, file names are case sensitive. By default, APL2 uses upper case letters for the workspace name and lower case letters for the file extension, as shown in these examples.

© Copyright IBM Corporation 1994, 2017

Using the Session Manager

The APL2 session manager:

- Allows you to carry on an interactive session with an APL2 interpreter. You can enter APL2 expressions and the results of their evaluation are displayed.
- Maintains a log of your input expressions and the interpreter's results. The log is retained from one APL session to the next. You can scroll forward and backward in the log, modify old expressions, and type new expressions.
- Can be used to control a remote APL2 interpreter. Consult Running a Remote Session Manager for further information.

In addition to the basic facilities described above, the session managers on each APL2 system have additional capabilities that exploit the windowing capabilities of their system.

- The APL2 Session Manager on Unix Systems
- The APL2 Session Manager on Windows

© Copyright IBM Corporation 1994, 2017

The APL2 Session Manager on Unix Systems

On Unix systems, the APL2 session manager is an X Window System-based application, and takes advantage of the distributed client-server model on which X Windows is based. This design allows the session manager to be run on a different workstation than the one to which the display is attached and the session manager is displayed on. In X terminology, the client is the workstation where the session manager is actually running and the server is the workstation to which the display is attached. In this environment the session manager can be installed on one workstation and support multiple X-stations and servers via the network. The X Server displaying the APL2 session manager must have the fonts that are included with the APL2 product installed or be able to obtain them from a font server.

The session manager is an auxiliary processor and is started automatically during APL2 invocation unless the `-sm piped` invocation option is specified.

Note: Do not enter the `kill -9 <pid>` command to end an APL2 session manager session, because interprocess resources are not released. If it is necessary to exit from the application in this manner, enter the `kill -15 <pid>` command.

- APL Characters and Fonts
- National Language Support (NLS)
- Screen Layout
- Customizing with Resources and Options
- Other Invocation Options and Associated X Resources
- Customizing the Keyboard

APL Characters and Fonts

The APL2 language uses a special set of characters. To enable you to type the APL characters, the session manager redefines each of the alphanumeric keys on your keyboard. Later sections describe how to redefine the keyboard.

Roman-style fonts are used to display the APL2 character set. The following APL fonts are available with the APL2 session manager: `apl6, apl10, apl12, apl14,` and `apl22`. By default `apl14` is selected at invocation. The font can be changed interactively during the session and the initial font can be specified either through command line parameters or resource settings.

The following table lists some X Windows software tools.

Note: If you are unable to find these tools on your system look in the directory `<X11*home>/Xamples/clients`. These are example programs shipped with X11 and might need to be compiled.

Name	Description
showrgb	Displays the names for the colors supported on the X Server.
xset	To change or query various characteristics of the current windowing session.

© Copyright IBM Corporation 1994, 2017

Name	Description
	In particular, you can change the fontPath resource that affect the following tools listed here since they work only with fonts that are in the current font path. To determine your current settings use: `xset q`
xfd	Displays the characters of a specified font.
xlsfonts	Lists all of the fonts available on your system.
xfontsel	Displays either a list of the fonts on your system or a representative number of characters from a specified font.

National Language Support (NLS)

The APL2 session manager is enabled to support single-byte character set (SBCS) languages. All messages and tables are externalized to allow translation. Since the session manager is an X-based product, it uses the X environment to provide NLS support, and the zero origin `⎕AV` values are simply indexes into the font. The languages currently supported are U.S. and U.K. English, German, French, and Italian.

Screen Layout

The session manager is a window consisting of a menu bar, work area, and message area.

- The menu bar provides various pull-down menus and dialog boxes.
- The work area is for user input and interpreter output.
- The message area is used by the session manager to provide information on the system/keyboard states and non-interpreter messages.

With each of these components there are associated X Window resources. By assigning values to these resources, you can customize the look and feel of the session manager.

The Menu Bar

The Menu Bar consists of four items. To select a menu bar item, move the mouse to the item and press the left mouse button. For each of the menu bar items, an associated pull-down menu appears with more selections. The items on the pull-down menu can then be selected using either the mouse or the cursor keys and enter key. To remove the pull-down menu, press the left mouse button when the mouse pointer is outside of the pull-down, and no action is taken.

Selections that are followed by an **"..."** or arrow create additional pull-down menus and pop-up dialogs. Selections that do not have these indicators take immediate action. When a pop-up dialog is presented, it has an **OK** and a **CANCEL** push button. Selecting **OK** performs the requested action and **CANCEL** aborts the action.

The following table lists the menu bar X resource names:

Resource	Description

© Copyright IBM Corporation 1994, 2017

Resource	Description
ap120*menubar	The Entire Menu Bar
ap120*menubar.button_0	The File Push Button
ap120*menubar.button_1	The Signals Push Button
ap120*menubar.button_2	The Options Push Button
ap120*menubar.button_3	The Help Push Button

File Menu

Resource	Description
ap120*file_menu	The Entire File Pull-Down
ap120*file_menu.button_0	The Log Push Button
ap120*file_menu.button_1	The Copy Push Button
ap120*file_menu.button_2	The Quit Push Button

Log Menu

Use the **Log** actions to:

- Load an existing log file replacing the active session log
- Rename the active log file
- Save the active session log to disk
- Clear the active session log
- Change the size of active session log

The default log name is `./apl2ses.log` and is stored relative to the directory from which the session manager is started. The default log size is `24K bytes` and the maximum size is `256K bytes`.

Resource	Description
ap120*log_menu	The Entire Log Pull-Down Menu
ap120*log_menu.button_0	The Name ... Push Button
ap120*log_menu.button_1	The Save ... Push Button
ap120*log_menu.button_2	The Load ... Push Button
ap120*log_menu.button_3	The Size ... Push Button
ap120*log_menu.button_4	The Clear ... Push Button

Copy Menu

Use the **Copy** actions to rename the current copy file and to turn copy on and off. When copy is `On`, information displayed in the session manager window is appended to the copy file. The default log name is `./apl2Copy.txt` and is stored relative to the directory from which the session manager is started. The default for Copy is `Off`.

© Copyright IBM Corporation 1994, 2017

Resource	Description
ap120*copy_menu	The Entire Copy Pull-Down
ap120*copy_menu.button_0	The Name ... Push Button
ap120*copy_menu.button_1	The On ToggleButton
ap120*copy_menu.button_2	The Off ToggleButton

Quit Menu

Use the **Quit** action to close the APL2 session with either a `)OFF` or `)CONTINUE`.

Resource	Description
ap120*quit_menu	The Entire Quit Pull-Down
ap120*quit_menu.button_0	The)OFF Push Button
ap120*quit_menu.button_1	The)CONTINUE Push Button

Signals Menu

This selection is used to send signals to the APL2 interpreter.

Resource	Description
ap120*signals_menu	The Entire Signals Pull-Down
ap120*signals_menu.button_0	The Suppress Push Button
ap120*signals_menu.button_1	The Attention Push Button
ap120*signals_menu.button_2	The Interrupt Push Button

Suppress
: stops all output coming from the interpreter output until the next input prompt. The `Pause` key also sends a suppress signal.

Attention
: causes the interpreter to halt when it completes executing the current line. The `Ctrl-Break` key also sends an attention signal.

Interrupt
: causes the interpreter to halt as soon as possible; typically at the end of the current primitive operation.

Options Menu

Use Options to configure the session manager.

Resource	Description
ap120*options_menu	The Entire Options Pull-Down
ap120*options_menu.button_0	The Scroll ... Push Button
ap120*options_menu.button_1	The Fonts ... Push Button

© Copyright IBM Corporation 1994, 2017

Resource	Description
ap120*options_menu.button_2	The APL Mode ... Push Button

Scroll
: controls whether the session manager scrolls the log as the interpreter produces output or stops after the page fills and waits for the Enter key to be pressed.

Fonts
: allows selection of a new font in the current session manager window.

APL Mode
: switches between the APL keyboard and the text keyboard definition.
The `Ctrl-Backspace` key also turns the APL keyboard on and off.

Resource	Description
ap120*scroll_menu	The Entire Scroll Pull-Down
ap120*scroll_menu.button_0	The Line ToggleButton
ap120*scroll_menu.button_1	The Page ToggleButton
ap120*font_menu	The Entire Fonts Pull-Down
ap120*font_menu.button_0	The apl6 ToggleButton
ap120*font_menu.button_1	The apl10 ToggleButton
ap120*font_menu.button_2	The apl12 ToggleButton
ap120*font_menu.button_3	The apl14 ToggleButton
ap120*font_menu.button_4	The apl22 ToggleButton
ap120*apl_mode_menu	Entire APL Mode Pull-Down
ap120*apl_mode_menu.button_0	On ToggleButton
ap120*apl_mode_menu.button_1	Off ToggleButton

Help Menu

The help facility provides help through dialogs and is divided into four general areas. These dialogs can remain active while using the session manager.

Resource	Description
ap120*help_menu	The Entire ... Help Pull-Down
ap120*help_menu.button_0	The General ... Push Button
ap120*help_menu.button_1	The Programmer ... Push Button
ap120*help_menu.button_2	The Keys ... Push Button

Status Area

The **Status** area is on the bottom of the screen and is used for:

- The system status
- Keyboard mode

© Copyright IBM Corporation 1994, 2017

- Non-interpreter messages

Resource	Description
ap120*status	The Entire Status Area
ap120*sys_mode	The Running/Input Label
ap120*kbd_mode	The [APL]/[Text] Label
ap120*sm_msg	The Session Manager Message Area

Main Work Area and Associated Color Resources

The **Main** work area is for user input, interpreter output, and scrolling of the session log.

Within the **Main** work area, the cut and paste functions allow you to copy text within a window or from one window to another.

To mark the text:

1. Position the mouse pointer at the beginning of the text to be selected.
2. Press and hold mouse `button 1`.
3. Move the mouse pointer to the end of the text to be selected.
4. Lift the mouse button.

To paste a copy of the marked text:

1. Move the cursor to the point where the text is to be inserted.
2. Press mouse `button 2`.

The following table lists the names of the color resources associated with the interpreter input and output classes. Other resources exist and are described in Other Invocation Options and Associated X Resources.

Resource	Description
ap120*backColor	The background color for the work area
ap120*fgDebugOutput ap120*bgDebugOutput	The foreground and background colors for the Debug Output Class
ap120*fgDelEditOut ap120*bgDelEditOut	The foreground and background colors for the Del Editor Output Class
ap120*fgDelEditTs ap120*bgDelEditTs	The foreground and background colors for the Del Editor Function Time Stamp Output Class
ap120*fgDelPrompt ap120*bgDelPrompt	The foreground and background colors for the Del Editor Line Input Class
ap120*fgErrorMsg ap120*bgErrorMsg	The foreground and background colors for the Error Messages
ap120*fgNormalOutput ap120*bgNormalOutput	The foreground and background colors for the Normal Output Class

© Copyright IBM Corporation 1994, 2017

Resource	Description
ap120*fgQuadOutput ap120*bgQuadOutput	The foreground and background colors for the □ Output Class
ap120*fgQuadOutput ap120*bgQuadOutput	The foreground and background colors for the □ Output Class
ap120*fgQuadPrompt ap120*bgQuadPrompt	The foreground and background colors for the □ Input Class
ap120*fgQuoteQuadInput ap120*bgQuoteQuadInput	The foreground and background colors for the ⍞ Input Class
ap120*fgQuoteQuadOutput ap120*bgQuoteQuadOutput	The foreground and background colors for the ⍞ Output Class
ap120*fgSysCommand ap120*bgSysCommand	The foreground and background colors for the System Commands Output Class
ap120*fgTraceOutputFns ap120*bgTraceOutputFns	The foreground and background colors for the Trace Output Class
ap120*fgTraceOutputVal ap120*bgTraceOutputVal	The foreground and background colors for the Trace Output Class line results
ap120*fgUserInput ap120*bgUserInput	The foreground and background colors for the user input
ap120*fgLineModified ap120*bgLineModified	The foreground and background colors for the user modified lines

Note: For a list of the valid color names use the `showrgb` command.

Customizing with Resources and Options

You can customize an APL2 session manager's look and feel. Using APL2 session manager resources, external files, and command line parameters, you can specify both system-wide and instance-specific variations for your session.

The following is the order of precedence for each level of customization within the APL2 session manager or within any X application:

1. Command line parameters
2. The user's `.Xdefaults` file found in the user's home directory
3. The system defaults file, `ap120`, located in the `$APL2/nls/<locale>` directory

Command line parameters override the customization specified in both the `.Xdefaults` file and application default entries. The `.Xdefaults` entries override any customization specified in the system's application defaults file.

APL2 session manager itself is constructed in a hierarchical fashion. Within the session manager application, there are a number of separate components that you can customize.

© Copyright IBM Corporation 1994, 2017

Syntax

Following is a brief overview for managing X Windows and Motif resources. For in depth information on the resource manager, refer to the X Windows and Motif publications.

A resource file is a standard ASCII text file that can be created or modified with a standard text editor. Within the resource file, you specify the resources for one or more widgets within one or more Motif client (application) programs. In this context, the session manager is the client program, and the components that make it up (for example, the menu bar) are widgets. The syntax for a resource file is simple, where lines in the resource file specify a resource name for a widget in a particular client along with a value for the resource. To do this, you need to know the instance names and instance hierarchy in client program. Every Motif client is composed of one or more widgets organized in a hierarchical manner. With each widget there is an associated instance name or class. For example, the **File** button on the menu bar is a widget that is a child of the menu bar widget. The instance name for **File** is `button_0` and the instance name for the menu bar is `menubar`.

The syntax for a resource entry is:

```
[object{.|*}subobject{.|*}subobject ... ]{.|*}attribute:value
```

where object and subobject can either be the instance name or class.

The object and subobject can be separated either by a period (.) or asterisk (*).. Using a period separator is referred to as tight binding and requires that you know the exact instance hierarchy in the client's widget instance tree. Using an asterisk separator is referred to as loose binding and gives you a *wildcard* capability in resource specification, and is the form most commonly used.

The last item in the resource specification is the attribute that is the name of the resource for which you are supplying the value, which is the last item.

For example, to specify a red background for all of the subcomponents of all menu bar and status subcomponents, enter:

```
ap120*background: red
```

To specify that only the background of all components called status are set to red, enter:

```
ap120*status.background: red
```

Command Line Parameters

Command line parameters are the final override in determining how resources are set (unless they are hardcoded in the session manager).

Most of the standard X command line parameters are supported, as well as a number of exclusive command line parameters that are interpreted only by the APL2 session manager. For example:

To invoke the session manager with a larger font, enter:

© Copyright IBM Corporation 1994, 2017

```
apl2 -fn apl22
```

To set a red background direct output to another display, enter:

```
apl2 -bg red -display xdisplay:0.0
```

This command specifies that output is being directed to another X display called `xdisplay:0.0`.

Note: The remote display can be any workstation or terminal that is running X. However, the fonts included in the APL2 session manager package must be installed on the remote display or available from a font server.

To invoke an APL2 session manager session with the background for error messages set to blue, you can use the `-xrm` command flag:

```
apl2 -xrm "ap120*sm.bgErrorMsg:blue"
```

This syntax differs from the standard syntax in that blank characters cannot follow the resource name. (For example, there are no blanks after `bgErrorMsg:` in the command shown above).

Customizing with the .Xdefaults File

If you are familiar with X applications, the APL2 session manager uses the standard X resource descriptions.

If you are not familiar with X applications, a brief example is provided of how to use the `.Xdefaults` file to modify APL2 session manager defaults.

The `.Xdefaults` file contains individual preferences for various X Window applications. These preferences are specified by resources. Each line in the `.Xdefaults` file sets a specific resource to a specified value.

APL2 session manager resource values may be used in the `.Xdefaults` file or the system application defaults file. The following sample `.Xdefaults` file entries specify:

1. A default session title is *APL2 Session Manager*.
2. The default background for the basescreen is black.
3. The default foreground color for interpreter error messages is red.
4. The default background color for interpreter error messages is black.

```
ap120.title:         APL2 Session Manager
ap120*sm.backColor:  black
ap120*sm.fgErrorMsg: Red
ap120*sm.bgErrorMsg: black
```

System Defaults

The application defaults for the X resources of the session manager and other X/Motif components can be found in the file `$APL2/nls/<locale>/ap120` where `$APL2` refers to the installation directory and `<locale>` is the setting of the `LANG` environment variable.

© Copyright IBM Corporation 1994, 2017

If you are planning on customizing the session manager, you can copy this file and use it as template.

Other Invocation Options and Associated X Resources

X Window System resources and options can be used along with the following APL2 session manager specific resources and options. Command line options are prefixed with a dash (-).

Option and Resource	Default Value(s)	Description
-aplmode aplmode	on	Initial APL2 keyboard mapping. Choices are `on` or `off`.
-bg background	X default background	Background color for menu bar and status components. Can be any valid color name.
-copy copy	off	Initial Copy mode. Choices are `on` or `off`.
-copyfile copyfile	./apl2Copy.txt	Initial Copy file name. Must be a valid file name that is writable.
-display	$DISPLAY	Sets the DISPLAY environment variable.
-fg foreground	X default foreground	Sets the foreground color for the menu bar and status components. Can be any valid color name.
-fn aplFontList	apl14	Sets initial session manager work area font. Choices are: `apl6` or `apl10` or `apl12` or `apl14` or `apl22`.
-geometry geometry	80x25+0+0	Sets the window width, height, xoffset, and yoffset.
-log logfile	./apl2ses.log	Initial log file name. Must be a valid file name that is writable. If it already exists then it must be a valid format for an APL2 session log.
-cr cursorColor	white	The color of the text cursor. Can be any valid color name.
-ms mouseColor	white	The color of the mouse pointer. Can be any valid color name.
-scroll scroll	line	Specifies scrolling mode. Can either be `line` or `page`.

Note: For a list of the valid color names use the `showrgb` command.

Customizing the Keyboard

This section explains what a translation is and provides several customization examples. It also lists the APL2 session manager functions that can be used in a translation, along with their default key mappings.

You can customize your APL2 session manager keyboard mapping by modifying the `.Xdefaults` file.

Translation Overview

© Copyright IBM Corporation 1994, 2017

A translation is the mapping of an event or sequence of events to one or more actions. The syntax is:

```
modifier_symbol modifier(s) detail: action
```

where the `modifier_symbol` is optional.

The following special symbols can be used to change the interpretation of an event sequence in a translation:

Symbol	Description
None	No modifiers can be specified.
!	Only the modifiers explicitly listed can be specified.
:	Apply the Shift modifier to the key event.
~	The modifier following the symbol cannot be specified.

Modifier keys are optional. They allow you to define more complex translations. If no modifier keys are specified for an event, any modifier key can be pressed along with the event.

If multiple modifiers are specified, they must be separated by at least one space.

Modifier keys are dependent upon the machine hardware and the implementation of X Window System on that hardware.

The modifiers listed in the following table may be available on various platforms. You can specify the modifier name or the corresponding abbreviation, if available.

Modifier	Abbreviation	Description
Ctrl	c	Control key is pressed
Shift	s	Shift key is pressed
Lock	l	Caps Lock is on
Alt	a	Alt key is pressed
Meta	m	Meta key is pressed
Mod1		Mod 1 key is pressed
Mod2		Mod 2 key is pressed
Mod3		Mod 3 key is pressed
Mod4		Mod 4 key is pressed
Mod5		Mod 5 key is pressed

`Button1, Button2, Button3, Button4,` and `Button5` can also be used as modifiers in translations. For example, the translation:

```
Button1<Key>Escape: enter()
```

invokes the **enter()** function when mouse `button 1` and the `Escape` key are pressed simultaneously.

© Copyright IBM Corporation 1994, 2017

You can use the X Window client program, `xmodmap`, to determine which modifiers are available on your system.

An event is primarily used to indicate that a keyboard key was pressed, a mouse button was pressed, or the mouse was moved. You can specify either the event type or the corresponding abbreviation.

An event must always be enclosed in angle brackets (< and >) in the translation. If multiple events are specified, they must be separated by commas.

The following list contains some types of events, but you can also use other predefined events for APL2 session manager translations.

Modifier	Abbreviation	Description
ButtonPress	BtnDown	Any mouse button is pressed
	Btn1Down	Mouse button 1 is pressed
	Btn2Down	Mouse button 2 is pressed
	Btn3Down	Mouse button 3 is pressed
ButtonRelease	BtnUp	Any mouse button is released
	Btn1Up	Mouse button 1 is released
	Btn2Up	Mouse button 2 is released
	Btn3Up	Mouse button 3 is released
KeyPress	Key	Any key is pressed
	KeyDown	Any key is pressed
KeyRelease	KeyUp	Any key is released
MotionNotify	Motion	The pointer is moved while any mouse button is pressed
	Btn1Motion	The pointer is moved while mouse button 1 is pressed
	Btn2Motion	The pointer is moved while mouse button 2 is pressed
	Btn3Motion	The pointer is moved while mouse button 3 is pressed

The detail parameter is optional. It specifies the key or mouse button that must be pressed to invoke the translation.

For example, to invoke the **pageup** function only if the `F2` key is pressed, you can enter:

```
<Key>F2: pageup()
```

where `F2` is the detail to the KeyPress event.

The text actually specified as the detail to a key event is known as *keysym*. All keysyms are defined in the `<X11/keysymdef.h>` file. You can review this file to determine the valid keysyms for your operating system. When specifying a keysym (detail) in a translation, the XK_ prefix is omitted. For example, if an entry in the `<X11/keysymdef.h>` file is:

```
#define XK_Escape 0xFF1B
```

A sample translation including this detail is:

© Copyright IBM Corporation 1994, 2017

```
<Key>Escape: undo()
```

To determine the keysyms that are bound to your keyboard keys, use the `xev` program, if it is available on your machine.

Detail can also be specified for the following mouse events: `BtnDown, BtnUp, ButtonPress, ButtonRelease, Motion,` and `MotionNotify`.

The detail for these events could be one of the following: `Button1, Button2, Button3, Button4,` and `Button5`. For example:

```
<ButtonPress>Button1: select-start()
```

where `Button1` is the detail.

This translation is equivalent to the following translations:

```
<BtnDown>Button1: select-start()
<Btn1Down>: select-start()
```

that demonstrate the use of event abbreviations.

The .motifbind File

Machines running the Motif window manager can use a .motifbind file. This file contains mappings of user defined keysym names to default X Window System keysym names that are found in the `<X11/keysymdef.h>` file.

When the X Window System and the Motif window manager are started on a machine that has a `.motifbind` file defined, the keysym names in the `.motifbind` file override their respective default X Window System keysym names for the entire X Window session.

The APL2 session manager program uses the default X Window System keysym names for all of its default translations. If any of these keysym names are remapped in a `.motifbind` file, then the default APL2 session manager translations that use them become undefined. You must create a new APL2 session manager translation in your `.Xdefaults` file using the remapped keysym name.

For example, if you are running the Motif window manager and the following entries appear in your `.motifbind` file:

```
osfInsert : <Key>Insert
osfDelete : <Key>Delete
```

then the default APL2 session manager translations:

```
<Key>Insert: insert()
<Key>Delete: delete()
```


© Copyright IBM Corporation 1994, 2017

become undefined. To redefine the APL2 session manager **insert()** and **delete()** functions to the same keys, you should add the following translations to your `.Xdefaults` file:

```
<Key>osfInsert: insert()
<Key>osfDelete: delete()
```

For more information about the `.Xdefaults` file, see Customizing with the .Xdefaults File.

An action is basically a function that is defined to the application that can be used in translations. APL2 session manager actions (functions) are listed in the section below. They include mouse functions, window functions, and APL2 session manager functions.

Action parameters are separated from the corresponding event sequences by a colon.

If multiple actions are specified, they must be separated by at least one space or multiple spaces.

When specifying an action in a translation, parenthesis must follow the function name. For example:

```
<Key>F2: undo()
```

where **undo()** is the action specified.

Some APL2 session manager actions accept arguments. For more information, see Translation Examples.

Translation Examples

1. To specify a simple translation where:
   - The **pageup** function is invoked when any key is pressed.
   - Any modifier key may be pressed at the same time.

   ```
   <Key>: pageup()
   ```

2. To add more detail to the simple translation where:
   - The **pageup** function is invoked when the `Escape` key is pressed.
   - Any modifier key can be pressed in conjunction with the `Escape` key.

   ```
   <Key>Escape: pageup()
   ```

3. To add modifiers where:
   - The **pageup** function is invoked when the `Shift` modifier and the `Escape` keys are pressed simultaneously.
   - Any other modifier key can be pressed in conjunction with the `Shift` and `Escape` keys.

   ```
   Shift<Key>Escape: pageup()
   ```

4. To add special modifier symbols where:
   - The **undo** function is invoked when the `F2` key is pressed without any modifiers.

© Copyright IBM Corporation 1994, 2017

```
None<Key>F2: undo()
```

- The **undo** function is invoked only when the `Ctrl` modifier and the `F2` keys are pressed simultaneously.
- No other modifiers except the `Ctrl` modifier can be pressed.

```
!Ctrl<Key>F2: undo()
```

- The **pagedown** function is invoked when the `F2` and the `Shift` keys are pressed.
- Any modifier key except `Ctrl` can also be pressed.

```
~Ctrl Shift<Key>F2: pagedown()
```

5. To specify multiple actions where:
   - Simultaneously pressing the `Ctrl` and the `a` key invokes the string function.
   - The string)LIB 1 is sent to the interpreter and the **enter** function is invoked.

```
Ctrl<Key>a: string(")LIB 1") enter()
```

6. To specify multiple events where pressing the `F2` key twice in succession without any other events occurring in between invokes the **attention** function.
   - In this example, if the `F2` KeyPress event is defined to an action, that action is invoked on the first `F2` KeyPress event. The **attention** function is invoked when the `F2` key is pressed a second time.

```
<Key>F2,<Key>F2: attention()
```

Default Keyboard Mapping

The session manager supports the following types of keyboard input:

Action Keys

Default Key	Function Name	Description
Enter or Return	enter()	Sends any modified lines to the interpreter
Ctrl-Backspace	aplonoff()	Switches APL keyboard support on and off
Pause	suppress()	Suppresses interpreter output
Insert	insert()	Toggle insert mode
Ctrl-Pause	attention()	Sends an Attention signal
Ctrl-Up Arrow	retrieve-back()	Move backward through prior inputs
Ctrl-Down Arrow	retrieve-forward()	Move forward through prior inputs

Cursor Movement

Default Key	Function Name	Description

© Copyright IBM Corporation 1994, 2017

Default Key	Function Name	Description
Ctrl-Enter	start-next-line()	Moves the cursor to the beginning of the next line
Alt-Enter	end-prior-line()	Moves the cursor to the end of the prior line
Left Arrow	left()	Move the cursor left.
Right Arrow	right()	Move the cursor right.
Down Arrow	down()	Move the cursor down one row
Up Arrow	up()	Move the cursor up one row
Tab	tab()	Move the cursor right one tab column
Shift-Tab	backtab()	Move the cursor left one tab column
Home	home()	Move the cursor to the beginning the line
End	end()	Move the cursor to the end of the line
Ctrl-Home	begin-log()	Move the cursor to the beginning of the log
Ctrl-End	end-log()	Move the cursor to the end of the log
Alt-Home	upper-left()	Move the cursor to the upper left corner of the screen
Alt-End	lower-right()	Move the cursor to the lower right corner of the screen
Alt-Left Arrow	scr-left1()	Scrolls the log left one column
Alt-Right Arrow	scr-right1()	Scrolls the log right one column
Alt-Up Arrow	scr-up1()	Scrolls the log up one line
Alt-Down Arrow	scr-down1()	Scrolls the log down one line
Page Up	page-up()	Scrolls the log up one page
Page Down	page-down()	Scrolls the log down one page
Ctrl-Page Up	page-left()	Scrolls the log left one page
Ctrl-Page Down	page-right()	Scrolls the log right one page
Ctrl-Left Arrow,	backward-word()	Move the cursor left one word
Ctrl-Right Arrow	forward-word()	Move the cursor right one word

Text Modification

Default Key	Function Name	Description
Esc	undo()	Selects a line for input or undo changes to the line
Backspace	backspace()	Move the cursor left one character and erase the character
Delete	delete()	Deletes the current character
Ctrl-Delete	erase2eol()	Erase to the end of the line

Default Function Keys

Default Key	Function Name	Description
F7	page-up()	Scrolls the log up one page

© Copyright IBM Corporation 1994, 2017

Default Key	Function Name	Description
F8	page-down()	Scrolls the log down one page
F9	current2top()	Move current (cursor) line to top of session log window
F12	<none>	Execute touched lines starting at the top of a new page
Shift-F8	<none>	Start another APL2 session running in parallel
Shift-F9	<none>	Prompt for an AIX command to execute with standard output to session log
Ctrl-F1	<none>	Prompt for an APL2 variable name; display its structure graphically
Ctrl-F2	<none>	Search for an idiomatic APL2 programming solution
Ctrl-F3	<none>	Cancel all Stop controls in the active workspace
Ctrl-F4	<none>	Cancel all Trace controls in the active workspace
Ctrl-F5	<none>	Set a Stop control on the first line of every function or operator
Ctrl-F6	<none>	Set Trace on first noncomment line of unlocked functions or operators
Ctrl-F7	<none>	Set Stop controls on every line of every function or operator
Ctrl-F8	<none>	Set Trace controls on all non-comment lines of functions or operators
Ctrl-F9	<none>	→⎕LC
Ctrl-F10	<none>	Print functions and variables from the active workspace
Ctrl-F11	<none>	Prompt for a string; search all FNS and OPS; report all occurrences
Ctrl-F12	<none>	Prompt for and edit a function, operator, or variable using EDITOR_2

Note: All of the functions that do not have an associated function name are defined by an entry in the `ap120` file found in the directory: `$APL2/nls/<locale>`

Mouse Actions

Default Key	Function Name	Description
Btn1Motion	select-adjust()	Adjust the current text selection.
Btn1Up	select-end()	End a text selection.
Btn1Down	select-start()	Start a text selection.
Btn2Down	insert-selection()	Insert the currently selected text at the cursor position

APL2 Session Manager Box Character Functions

Default Key	Description
Ctrl-Keypad 1	lower-left-box-corner()
Ctrl-Keypad 2	bottom-box-tee()
Ctrl-Keypad 3	lower-right-box-corner()
Ctrl-Keypad 4	left-box-tee()
Ctrl-Keypad 5	box-cross()

© Copyright IBM Corporation 1994, 2017

Default Key	Description
Ctrl-Keypad 6	right-box-tee()
Ctrl-Keypad 7	upper-left-box-corner()
Ctrl-Keypad 8	top-box-tee()
Ctrl-Keypad 9	upper-right-box-corner()
Ctrl-Keypad 0	vertical-box-bar()
Ctrl-Keypad .	horizontal-box-bar()

Other Session Manager Functions

Function Name	Description
hex()	This function translates hexadecimal to binary. For example, hex(414243) sends ABC to the session manager.
modlock()	This function can be used to create additional keyboard states that are locked. This is similar to Caps Lock in that no key must be held down to maintain the locked state. Modlock does this by toggling on or off the Modn modifier bit, where n is the argument passed to modlock. For example, pressing Ctrl-backspace sends a call to modlock(3), which enables the APL2 keyboard. After this call, all translation lookups are done with the Mod3 modifier bit on. Calling modlock(3) again disables the APL2 keyboard. For example, APL2 session manager's default translation table can contain a statement similar to the one below: `Mod3 Shift<Key>8: not-equal()` This specifies that the Shift-8 key should produce the APL2 not-equal character when the APL2 keyboard (Mod3) is active.
noop()	This indicates that no action is performed. It can be used to change (unmap) the default mapping of a key. To unmap the Escape key from the **Clear** function so that no action is taken when the Escape key is pressed, add the following line to your `.Xdefaults` file: `<Key>Escape: noop()`
string()	This function passes the character string as if it were entered using the keyboard. To map the F2 key to the string)OFF followed by the Enter key, put the following line in the translation specification of your `.Xdefaults` file as follows: `<Key>F2: string(")OFF") enter()`
stringa()	This function is similar to the string function, but it also accepts ASCII values above 128 (most of the special APL characters have ASCII values greater than 128). To map the F2 key to the string →⎕LC followed by the Enter key, either put the following

© Copyright IBM Corporation 1994, 2017

Function Name	Description
	line in the translation specification of your `.Xdefaults` file, as follows: `<Key>F2: stringa("→⎕LC") enter()`

APL2 Session Manager APL2 Character Functions

To turn APL2 characters on for your APL2 session manager session, either select the **APL2** button from the **Options** entry on the APL2 session manager menu bar or press the `Ctrl-Backspace` key (the default mapping for the APL2 session manager **aplonoff()** function).

To determine if there are any different translations for your operating system, browse the `apl2` resource file. Note that the APL2 character translations contain the Mod3 modifier. For example, the default translation:

`Mod3 Shift<Key>g: del()` generates the del symbol when the APL2 session manager session is toggled to APL2 mode and the Shift and g keys are pressed simultaneously.

The next table lists all of the APL2 session manager APL2 character functions and their mappings on an IBM RISC System/6000 101-key enhanced keyboard. This table does not list the APL2 symbols themselves. For a Union keyboard, there are additional definitions for the APL characters in `$APL2/nls/<locale>/apl`. The Union keyboard translations are in effect when the session manager is in text mode.

Function Name	Default Key
alpha()	Shift-A
circle-slope()	Alt-6
circle-stile()	Alt-5
del()	Shift-G
del-tilde()	Alt-2
delta-stile()	Alt-4
dieresis()	Shift-1
divide()	Shift-=
down-carat()	Shift-9
down-shoe()	Shift-V
down-tack()	Shift-B
epsilon()	Shift-E
equal-underbar()	\
iota()	Shift-I
jot()	Shift-J
left-bracket()	;
left-tack()	<none>
not-equal()	Shift-8

© Copyright IBM Corporation 1994, 2017

Function Name	Default Key
not-less()	Shift-6
overbar()	Shift-2
quad()	Shift-L
quad-jot()	Shift-`
quad-slope()	Alt-`
rho()	Shift-R
right-bracket()	'
right-tack()	<none>
slope()	Shift-/
squad()	]
tilde()	Shift-T
up-arrow()	Shift-Y
up-carat-tilde()	Alt-0
up-shoe-jot()	Alt-,
up-tack()	Shift-N
circle-bar()	Alt-7
circle-star()	Alt-8
circle()	Shift-0
del-stile()	Alt-3
delta()	Shift-H
diamond()	`
dieresis-dot()	Alt-\
down-arrow()	Shift-U
down-carat-tilde()	Alt-9
down-stile()	Shift-D
down-tack-jot()	Alt-;
epsilon-underbar()	Shift-\e
i-beam()	Alt-1
iota-underbar()	Shift-]
left-arrow()	[
left-shoe()	Shift-Z
minus()	Shift--
not-greater()	Shift-4
omega()	Shift-W
plus()	-

© Copyright IBM Corporation 1994, 2017

Function Name	Default Key
quad-divide()	Alt-=
quad-quote()	Alt-[
quote-dot()	Alt--
right-arrow()	Shift-[
right-shoe()	Shift-X
slash-bar()	Alt-/
slope-bar()	Alt-.
stile()	Shift-M
times()	=
up-carat()	Shift-0
up-shoe()	Shift-C
up-stile()	Shift-S
up-tack-jot()	Alt-'

© Copyright IBM Corporation 1994, 2017

The APL2 Session Manager on Windows

On Windows, the session manager provides facilities that let you:

- Open, rename, save, and change the size of the log
- Print all or portions of the log
- Cut, copy, and paste text between the clipboard and the log
- Search and optionally replace text in the log
- Select from a list of workspace objects and open edit windows
- Signal the interpreter to suppress output or halt
- Turn the APL keyboard on and off
- Select, create, and modify keyboard layouts
- Select different name, size, and style fonts
- Change the colors of different types of interpreter output
- Set function key definitions
- Control whether interpreter output is displayed immediately
- Control whether stacked input lines are displayed
- Select from a list of edit windows that have been opened

All of these facilities are accessible from the Session Manager's menubar.

The session manager also includes a built-in editor that can be used to edit variables, functions, and operators. You can double-click on an object name in the log or use the **Open Object** window to select an object for editing. See The Object Editor for more information on the object editor.

The session manager includes extensive online help. To display contextual help on any menu item or window, press **F1**. You can also use the choices on the **Help** menu to display the online help.

© Copyright IBM Corporation 1994, 2017

Editing APL Objects

APL2 provides several different facilities for editing APL objects.

The editing facilities include:

- The Object Editor which is built into the session manager on Windows.
- The Dialog Editor which is also built into the session manager on Windows.
- The Line Editor which is part of the interpreter.
- Using System Editors which are outside of APL2
- The EDIT Workspace which provides compatibility editors written in APL.

© Copyright IBM Corporation 1994, 2017

The Object Editor

Note: The Object Editor is not available on Unix systems.

Objects in the active workspace can be opened for editing from the session manager.

Session manager and editor windows all include the **Edit** menu choice **Open Object**. Use **Open Object** to display a list of the objects in the active workspace. Use the **Type of Object** buttons to list either the variables, functions, or operators in the workspace. Type or select an object name and press **OK** to open the object. If the object does not exist, the object is opened with the selected name class.

To open an object directly from a session manager or editor window, double click on the object's name.

The following objects can be edited:

- Functions
- Operators
- Character matrixes
- Vectors of character vectors
- Character vectors containing carriage return-linefeed delimited records
- Numeric matrixes
- Matrixes of character vectors

The following objects cannot be edited:

- Locked functions and operators
- External functions
- System variables and functions

When an object is opened, the session manager opens a window for editing the object. Editor windows can be left open while you continue to interact with the interpreter. Objects can be opened while the interpreter is running.

Session manager and editor windows all include the **Windows** menu choice **APL2 Window List**. Use **APL2 Window List** to display a list of the editor windows opened from the session manager. The list also includes the session manager window. Double click on a window name to switch to the window.

Editor windows have the following restrictions:

- If an object is open and you attempt to open the object again, another editor window is not opened. The editor window displaying the object is given focus.
- If an object is open, changes made to the object by the Line Editor, a system editor, or by execution of an APL expression are not displayed in the editor window. To display the changes made to the object, use the **Refresh** selection on the **Object** menu.

The object editor provides extensive online help. To display contextual help on any menu item or window, press **F1**. You can also use the choices on the **Help** menu to display the online help.

© Copyright IBM Corporation 1994, 2017

The Dialog Editor

Note: The Dialog Editor is not available on Unix systems.

You can use the Dialog Editor to create and modify dialog windows and also the controls and text within dialog windows. As you create the dialog window and its controls you see them on the screen as the user sees them when your program is run. You can place each dialog window and its controls where you want them on the screen. You can assign event handlers so your APL2 program will get control when events occur.

To enter the Dialog Editor, use the **Open Object** option in the **Edit** menu of the session manager. Select editing of a variable, and check the **Dialog Template** option. You can also enter the Dialog Editor by double-clicking on the name of an existing dialog template in the session log.

For more information on using the Dialog Editor, consult *APL2 Programming: Developing GUI Applications* and the Dialog Editor's online help.

© Copyright IBM Corporation 1994, 2017

The Line Editor

The Line Editor, also known as *Editor 1*, can be used to edit functions and operators.

To use the Line Editor to edit an object, first select the Line Editor for use as APL's editor, and then use ∇ followed by an object name or header. For example:

```
)EDITOR 1
∇name
```

Note: Unicode characters that are not in ⎕AV are displayed as Omega characters. If a line containing such characters is changed, the Omega characters are used and the Unicode characters are lost.

For further information about the Line Editor consult *APL2 Programming: Language Reference*.

© Copyright IBM Corporation 1994, 2017

System Editors

Editor programs outside of APL2 can be used to edit functions, operators, and character matrixes.

To use a system editor to edit an object, first select the editor for use as APL's editor, and then use ∇ followed by an object name. For example:

```
)EDITOR vi
∇name
```

When setting the editor, you may use a complete path if needed:

```
)EDITOR /usr/bin/e3
```

If any name in the path you specify has an imbedded blank, you can preserve the blank by surrounding the entire editor name in quotes:

```
)EDITOR 'C:\Program Files\Accessories\Wordpad.exe'
```

If the editor you select supports Unicode, you can tell APL2 to use Unicode data representation by specifying the `-u` option:

```
)EDITOR notepad -u
```

When the editor is invoked via ∇, the interpreter writes a copy of the object's definition to a temporary file, and then invokes the editor program with that file's name as its argument. On exit from the editor, the new definition is fixed in the workspace.

If there is an error in the temporary file which prevents the definition from being fixed in the workspace, a DEFN ERROR will be reported. In order to leave the editor, you must either correct the error, erase the temporary file, or change the file to consist of a single empty line. In that case, the old definition of the object will be preserved.

If the editor is a window program, it will start a separate window for the editing session. If it is not, the edit session will be opened in the APL2 interpreter's console window. Since that window is normally minimized, you may need to restore it to see the edit session.

Notes:

- In order to see APL characters when using a system editor, the editor must support the selection of an APL font for use during the edit session.
- In order to type APL characters when using a system editor, the editor must contain appropriate keyboard support.

  On Windows, the Apledit program, which is provided with APL2, has the needed keyboard support and can be used with this facility. See The File Editor for more information.

© Copyright IBM Corporation 1994, 2017

On Windows, the APL2 keyboard handler program provides support for the APL keyboard in many non-APL programs. See The APL2 Keyboard Handler for more information.

On AIX, the X resource file for the interpreter console window contains key mappings which can be used with editors like *vi*.

- Unless the `-u` option has been specified, Unicode characters in function and operator definitions that are not in `⎕AV` are converted to Omega characters as the temporary file is written. If the editor saves the file, the Omega characters are used in the new definition and the Unicode characters are lost. Variables containing Unicode characters outside of `⎕AV` can only be edited if the `-u` option is in effect.

© Copyright IBM Corporation 1994, 2017

The EDIT Workspace

The `EDITOR_2` function emulates the APL2 mainframe version's full-screen `)EDITOR 2`. It can be used to edit functions, operators and character matrixes.

The `EDIT` function is another alternative editor that can be used to edit functions and operators.

Note: `EDITOR_2` does not support characters that are not in `⎕AV`.

For further information, see The EDIT Workspace.

© Copyright IBM Corporation 1994, 2017

Editing Text Files

In order to see APL characters when using any editor, the editor must support the selection of an APL font for use during the edit session.

In order to type APL characters when using an editor, the editor must contain appropriate keyboard support.

On Windows, the Apledit program, described in The File Editor, has the needed keyboard support built in.

On Windows, the APL2 keyboard handler program provides support for the APL keyboard in many non-APL programs. See The APL2 Keyboard Handler for more information.

On AIX, the X resource file for the interpreter console window contains key mappings which can be used with editors like *vi*.

© Copyright IBM Corporation 1994, 2017

The File Editor

The Apledit program can be used to edit text files containing APL characters.

To invoke Apledit to edit a file, enter:

```
apledit filename
```

from an operating system prompt, or from APL2:

```
)HOST apledit filename
```

The maximum record length supported by Apledit is 128000.

Apledit provides extensive online help. To display contextual help on any menu item or window, press **F1**. You can also use the choices on the **Help** menu to display the online help.

© Copyright IBM Corporation 1994, 2017

The APL2 Library Manager

Note: The Library Manager is not available on Unix systems.

The APL2 Library Manager allows you to browse and compare APL2 saved workspaces, transfer files, and namespaces. The APL2 Library Manager can run either as a stand-alone program or as an external function. When run as a stand-alone program, the library manager runs independently of interactive APL2 sessions. When run as an external function, the library manager runs as part of an interactive APL2 session and can be used to browse the active workspace.

To start the APL2 Library Manager as a stand-alone program, use any of these methods:

- Click on the APL2 Library Manager icon in the APL2 folder,
- Click on the APL2 Library Manager button on the Session Manager toolbar,
- Run the apl2lm.exe program in the ibmapl2w\bin folder.
- Open an APL2 Saved Workspace, Transfer File, or Namespace with the Library Manager.

To start the APL2 Library Manager as an external function, associate and run the APL2LM external function:

```
      3 11 ⎕NA 'APL2LM'
1
      APL2LM
```

The `APL2LM` external function initially shows the contents of the active workspace.

© Copyright IBM Corporation 1994, 2017

The APL2 Library Manager Environment

The APL2 Library Manager uses the Calls to APL2 facility to start runtime APL2 interpreter sessions in the background. The stand-alone program starts one session to run the library manager itself, and both the stand-alone program and the external function start a session for each file that is opened. The environment for these sessions can be affected by using APL2 invocation options:

Workspace Size

> By default, the stand-alone program uses a -ws value of "10m,100m,50m" and the external function takes the workspace size from the APL2 session that invoked it. Pass the -ws parameter directly to the stand-alone program or to the APL2 session that will invoke the external function to set the workspace size for the background sessions.

Other Invocation Parameters

> All other APL2 invocation parameters passed to the stand-alone program or the APL2 session invoking the external function will be passed on to the background sessions started by the APL2 library manager. Parameters not applicable to the APL2 runtime environment will be ignored.

For more information about APL2 invocation options, consult Invocation Parameters.

© Copyright IBM Corporation 1994, 2017

Effects of External Names

The APL2 Library Manager examines saved workspaces, transfer files, and namespaces by copying their contents into hidden interpreters and querying attributes of the files' objects using APL primitives and system functions such as ⍴ and ⎕AT. As a result of these queries, resolution of objects which are defined as external associations may occur. This can cause side effects such as accessing additional APL2 namespaces, opening operating system files, or startup of external environments like Java. If external references cannot be resolved, information may be missing or only partially displayed in the APL2 Library Manager's object lists.

© Copyright IBM Corporation 1994, 2017

Your APL2 Libraries

When you start the APL2 Library Manager program, it creates a folder named **My APL2 Libraries** in the **My Documents** folder. It creates shortcuts in **My APL2 Libraries** for the currently defined APL2 library directories. Use these shortcuts in the **Open** dialog to quickly locate your workspaces, transfer files, and namespaces. For information on defining APL2 libraries, see Library Specification.

You can configure Windows so that the APL2 Library Manager is used to open saved workspaces, transfer files, and namespaces. Use the **File Associations** option to perform this configuration.

© Copyright IBM Corporation 1994, 2017

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 three major facilities within APL2's support for cooperative processing:

1. 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.

   - Processor Network Identification
   - Processor Profile Structure
   - Using the Port Server
   - Sending a Share Offer
   - Receiving a Share Offer
   - Processor Profile Syntax
   - Processor Profile Examples
   - Running a Remote Session Manager
2. 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.

3. 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.

© Copyright IBM Corporation 1994, 2017

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 non-negative 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. A *processor profile* is a file that provides a cross reference between the single processor number used by APL2 programs and a processor network identification.

A sample processor profile is shipped with the APL2 product.
On Unix systems, the sample is located in:

```
/usr/APL2/examples/svp/apl2svp.prf
```

On Windows systems, the sample is located in:

```
C:\Program Files\IBMAPL2W\samples\apl2svp.prf
```

Note: The directory path may be different on your system if customized during APL2 installation.

To create your own processor profile, copy the sample to your working directory and edit as desired. At execution time, the location of the processor profile can be specified by adding a definition for keyword SVPPRF in the [Shared Variable Processor] section of the `apl2.ini` file, or by setting environment variable APL2SVPPRF before starting APL2. If a location for the file is not specified, file `apl2svp.prf` in the current working directory is used.

The profile is used for both outgoing offers from a processor and for incoming offers from other processors. It is read for each applicable 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 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.

© Copyright IBM Corporation 1994, 2017

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.

© Copyright IBM Corporation 1994, 2017

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 a `: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 a `: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 123.45.6.78
: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
```


© Copyright IBM Corporation 1994, 2017

Using the Port Server

The APL2 *port server* is a program that manages TCP/IP interactions for cross-system sharing. The port server must be active for most scenarios of cross-system sharing to be possible.

Unix Systems: The APL2 port server can be started from an Xterm command window and run in the background as follows:

```
/usr/APL2/bin/apl2psrv &
```

Note: The directory path may be different on your system if customized during APL2 installation.

Only one port server program should be active on each workstation.

Windows: The APL2 port server can be started in any of the following ways:

- Specifying the `-svplisten on` parameter when invoking APL2 or any other independent processor.
- Adding the keyword definition `SVPLISTEN=on` to the [Invocation Options] section of the `apl2.ini` file.
- Setting the environment variable APLSVPLISTEN to `on` before invoking APL2.
- Specifying the `-listen on` parameter when starting the SVP monitor facility.
- Selecting **Cross System->Start** from the **Actions** menu of the SVP Monitor window. (For more information about the SVP Monitor window, see The SVP Monitor Facility.)

Note: No matter how the port server is started, TCP/IP must have been started *before* starting the SVP.

© Copyright IBM Corporation 1994, 2017

Sending a Share Offer

When a shared variable offer is extended, and the processor number from the left of `⎕SVO` is greater than 1000, it is matched against the data in `:svopid.` tags in the processor profile. If a match is found, the offer is extended to the processor described by the tag's `:processor.`, `:address.` and `:userid.` values.

If no match is found, the offer is extended to an independent processor identified by the given processor number, within the same SVP domain as the offerer (same IP address and user ID).

If the processor number is less than 1001, the offer is assumed to be to a processor dependent on the offerer. The parent is taken to be the offerer's processor number and the grandparent is taken to be the offerer's parent, if any. If this processor is not running, an attempt is made to start the processor. For example, if you are processor 1001 and issue the following share:

```
127 ⎕SVO 'VAR'
```

the offer is made to a processor 127 that is dependent on 1001. If processor 127 is not signed on, an executable named ap127 is searched for and started automatically, if found. Therefore an APL2 session can communicate with its dependent auxiliary processors without using the processor profile; dependent auxiliary processors are started automatically as needed. Independent auxiliary processors must be identified by processor numbers above 1000.

Note: Values already contained in the variable being offered are not sent to the remote partner. Only values specified after the share is completely coupled are transmitted.

© Copyright IBM Corporation 1994, 2017

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.`, `:userid.`, and `:processor.` tags of each ID entry in the processor profile. "Wildcard" support is provided as discussed in Using Asterisks in Processor Profile Entries.

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.

© Copyright IBM Corporation 1994, 2017

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
- Authorization Entries

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.

id can be coded as "0" in which case the entry identifies any remote processor. An unique id will be assigned to the processor when it signs on.

: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 used must be greater than 1000.

id can be coded as "*" in which case the entry identifies any remote processor with the corresponding `:svopid.`

:address.*addr*

This tag gives the domain name or IP address of the partners machine. A domain name is a character string identifier as defined by the TCP/IP Domain Name System (DNS). For example: `stl.ibm.com` or `STLAPL`. An IP address is written in "dotted decimal" notation consisting of four decimal numbers between 0 and 255 separated by periods. For example: `12.34.56.111`.

addr can be coded as "*" in which case the entry identifies processors from any address with the corresponding `:svopid.`

If `:address.` is omitted, the machine address of the local machine is used.

:userid.*userid*

© Copyright IBM Corporation 1994, 2017

This tag gives the character identification of the user ID of the partner.

The userid value is case sensitive.

Userid values longer than 8 characters will be tolerated by APL2, but only the first 8 characters will be used for APL2 cross-system communication. Userid values with imbedded blanks will also be tolerated by APL2, but only the characters preceding the first blank will be used for APL2 cross-system communication.

userid can be coded as "`*`" in which case the entry identifies processors from any user ID with the corresponding `:svopid.`

`:userid.` can be omitted if `:address.` is also omitted or specifies the local machine. In that case, the currently active userid of the local machine is used.

:crypt.*routine,library*

This tag allows the specification of a user exit to be called for encryption and decryption of shared variable data sent across the network.

library is the DLL containing the encryption routine.

routine is the routine name within the DLL. It must be compiled with 32-bit System linkage, and must use the following prototype:

```
long  _System cryptrtn(long encdecflag,
                       char * data,
                       long datalen,
                       char * buffer,
                       long * bufflenp);
```

A sample encryption routine is shipped with APL2. The sample contains complete documentation on the parameters and expected results of encryption routines.

Here is an example of a processor entry that defines □SVO argument `33586` as a remote user with encryption enabled.

```
* user BROWN at STLAPL
:svopid.33586
          :address.STLAPL
          :userid.BROWN
          :processor.1002
          :crypt.cryptrtn,svpexit
```

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*]]

© Copyright IBM Corporation 1994, 2017

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
```


© Copyright IBM Corporation 1994, 2017

Processor Profile Examples

The technical reference material you need to share variables between processors running on different machines was presented in Processor Network Identification and Processor Profile Structure. 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	Processor Number
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
- User to Auxiliary Processor Shared Variables
- Using Asterisks in Processor Profile Entries

User to User Shared Variables

Assume that User 1 wants to share variables with User 2. The information needed is as follows:

First, 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 7777. The following entry is required in User 1's profile:

```
:svopid.7777
   :address.9.10.11.222
   :userid.jsmith
   :processor.32739
```

This entry allows User 1 to extend an offer to User 2.

Next, User 1 needs to authorize User 2 to share variables. User 2 is already identified as 7777 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.7777
```

This authorizes remote processor 7777 to share variables with local processor 1001.

© Copyright IBM Corporation 1994, 2017

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.7777,8888
```

This entry authorizes the remote users associated with the numbers 7777 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
```


© Copyright IBM Corporation 1994, 2017

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.7777,8888,9127
```

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 in Using Asterisks in Processor Profile Entries.

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 is writing a server; there is no way to know who the server's potential clients might be. It is not possible to code separate identification entries for each client. In this case, User 1 can code this identification entry:

```
:svopid.0
   :address.*
   :userid.*
   :processor.*
```

In effect, this entry identifies any remote processor as number 0. This number is used for a special reason. A remote processor is always known to the APL2 session by the number used in the `:svopid.` tag. When the number `0` is seen by the APL2 shared variable processor, a new unique number is automatically assigned.

You have now handled the problem of identifying unknown processors, and can now deal with authorization. An asterisk can also be used in the `:rsvopid.` tag like this:

```
:procauth.1001
   :rsvopid.*
```

This authorizes shares from any processor.

© Copyright IBM Corporation 1994, 2017

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.

Notes:

1. When coding entries with wildcards it is important to remember that APL2 will stop scanning the processor profile as soon as it finds an entry that matches what it is looking for. If you have multiple entries in your file, it is a good idea to put the entries with wildcards at the end, so that if a more specific entry without wildcards is also a match, it will be used in preference to the one with the wildcards.
2. If an entry is coded with wildcards it cannot be used to initiate an outgoing offer, as it does not contain enough information to identify the share partner. Wildcard entries should only be used by servers who will not initiate offers, but rather only match offers sent to them.

© Copyright IBM Corporation 1994, 2017

Running a Remote Session Manager

The session manager is usually started automatically when the APL2 interpreter is invoked. In this case, the session manager and the interpreter run on the same machine.

The session manager can also be used to control a remote interpreter. In this way, an interactive session can be conducted with an interpreter on another machine across the room or around the world.

The session manager is implemented as auxiliary processor 120. Rather than providing applications with the ability to execute session manager commands, AP 120 processes requests that conform to the shared variable interpreter interface. The session manager shares a variable with the interpreter through which the session is conducted. On Windows, the session manager's Object Editor windows also share variables with the interpreter.

When attempting to use the session manager to control a remote session, the session manager should be manually started and run independently rather than automatically and dependently by an interpreter. The name of the program to run is `ap120.exe` on Windows, and `ap120` on Unix systems. The processor number that the session manager should use to sign on to the SVP should be passed as an argument. For example:

```
ap120 -id 1120
```

The session manager waits for any interpreter to offer it a variable. On Windows, a message window will appear indicating that the session manager is waiting. A cancel button is provided in this window.

In order to establish a shared variable connection with a remote interpreter, processor identification, and authorization entries are required for both the interpreter and the session manager. These entries are placed in the SVP profile file. For more information, see Processor Network Identification.

Note:

1. Before shares can be established, the SVP port server must be running. See Using the Port Server for details.
2. Since the session manager does not initiate an offer, the order in which the processes in this scenario are started is important. If the SVP on the machine where the session manager will run is not yet active when the offer from the remote interpreter comes in, that offer will be lost. The safest procedure to follow is to start the session manager process (and its port server) before starting the interpreter process.

- Workstation Interpreters
- APL2/370 Interpreters

Workstation Interpreters

When communicating with a remote workstation interpreter, the session manager actually shares a variable with processor 1, which runs under the interpreter. The session manager's processor profile entries might look like this:

```
:svopid.7001
   :address.234.45.65.78
   :userid.JOHNDOE
```


© Copyright IBM Corporation 1994, 2017

```
   :processor.1,1001
:procauth.1120
   :rsvopid.7001
```

For more information about processor profile entries, see Processor Profile Structure.

These entries would allow the session manager, running as processor 1120, to share variables with processor 1 running under interpreter 1001 on John Doe's machine. The session manager displays 7001 in the title bar when it receives an offer from the processor identified by the `:svopid.7001` tag.

The remote workstation interpreter's processor profile entries would look like this:

```
:svopid.9120
   :address.146.75.44.12
   :userid.JaneSmith
   :processor.1120
:procauth.1,1001
   :rsvopid.9120
```

These entries allow the processor 1 running under the interpreter 1001 to share variables with the session manager, running as processor 1120, on Jane Smith's machine.

To invoke a workstation interpreter and have it be controlled by a remote session manager, use the `-sm` invocation option to specify the processor identification number of the session manager that the interpreter should extend an offer to. For example:

```
apl2 -sm 9120
```

on Unix systems, and

```
apl2win -sm 9120
```

on Windows systems.

Before shares can be established, the SVP port server must be running. See Using the Port Server for details.

APL2/370 Interpreters

When communicating with an APL2/370 interpreter, the session manager shares a variable directly with the interpreter. The APL2/370 products do not include an auxiliary processor 1. The session manager's processor profile entries might look like this:

```
:svopid.7001
   :address.234.45.65.78
   :userid.JohnDoe
  :processor.1001
:procauth.1120
   :rsvopid.7001
```

These entries allow the session manager, running as processor 1120, to share a variable with interpreter 1001 running on John Doe's user ID. The session manager displays 7001 in the title bar when it receives an offer from the processor identified by the `:svopid.7001` tag.

© Copyright IBM Corporation 1994, 2017

The APL2/370 interpreter's processor profile entries would look like this:

```
:svopid.9120
   :address.146.75.44.12
   :userid.JaneSmith
   :processor.1120
:procauth.1001
   :rsvopid.9120
```

These entries allow interpreter 1001 to share a variable with the session manager, running as processor 1120, on Jane Smith's machine.

To invoke an APL2/370 interpreter and have it be controlled by a remote session manager, use the `sm` invocation option to specify the processor identification number of the session manager that the interpreter should extend an offer to. For example:

```
apl2 apnames(ap2x119(listen(0))) sm(9120)
```

Notes:

1. The Windows session manager's **Open Object** and **Edit** facilities are not supported when controlling an APL2/370 interpreter.
2. APL2/370 does not use as many different codes as the workstation interpreters to distinguish different types of output. Several types of output mentioned in the session manager's **Colors** window are identified differently by APL2/370.

© Copyright IBM Corporation 1994, 2017

Shared Variable Interpreter Interface

APL2's Shared Variable Interpreter Interface enables the interpreter to 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 from another APL2 session. This creates a number of interesting possibilities:

Automated testing
: A test control application can be built in APL that drives another application being tested. The control application can provide any desired input (even things like attention signals), and receives all output that would normally be sent to the application user. Unlike test drivers running within the same session, it has no difficulty recovering from application errors, or even system errors!

Remote control
: Since shared variables can be used to communicate between systems, It is possible for a user on one system to control an APL2 session running on a different system, even a different type of system. Facilities built into the APL2 session manager (see Running a Remote Session Manager) make it look to the user as if the remote session is actually running on his or her local system.

Asynchronous processing
: An APL application can be designed to exploit multitasking by driving one or more subordinate sessions, each of which is given part of the work to be done. The master session can pass data back and forth, or synchronize the execution as needed, using either the interpreter interface variable or other variables shared with the application.

Distributed computation
: This is really just a generalization of the asynchronous processing discussed above. Subordinate sessions can be started on any system in a network, thus utilizing available processing cycles wherever they may exist. The master session can hand off pieces of the work to the subordinate sessions, perhaps giving any part of the work to any processor as soon as it becomes available.

- General Concepts
- Shared Variable Protocols
- Interpreter Input Data
- Interpreter Output Data

© Copyright IBM Corporation 1994, 2017

General Concepts

The shared variable interpreter interface is activated on a workstation APL2 system by starting an APL2 session with the `-sm nnnn` option. On a mainframe system the session is started specifying `SMAPL(nnnn)`. The interpreter uses the numeric *nnnn* value as a 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 will it conflict with, variables or other names used by the application.

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 partner can pass character data as input to the session, just as if the data were being typed at a keyboard. It gets character data back, much like what would be displayed in a session manager window as session output. (There is some additional information provided with the data. See Interpreter Output Data.) The partner can also pass a number of special signals to the interpreter, and is given some additional information about system variables and data stacked using AP 101.

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 sends a control code indicating it is ready to receive input. The interpreter waits and/or processes requests until instructed to shutdown either by a shutdown control signal or an `)OFF` or `)CONTINUE` command.

To be able to use this interface, you must have an appropriate entry in your SVP profile (see Processor Profile Syntax). The following is an example of an entry which would allow the interface to be used between two APL2 sessions, where the shared variable interpreter session was started with id 2222:

```
:svopid.2222001 :processor.1,2222
```

(The `svopid` value can be anything you wish to use. It is the number to be specified by the controlling session when sharing variable `APL2` with the controlled interpreter.)

© Copyright IBM Corporation 1994, 2017

Shared Variable Protocols

- Only one variable can be shared with each APL2 session.
- The variable name is always `APL2`.
- The interpreter will offer or match the variable only if it was started with the appropriate `-sm` or `SMAPL` option.
- The variable can only be shared with the partner specified in the invocation option.
- If offered first by the partner, initial values are ignored by the interpreter.
- Input to the interpreter must be simple character vectors or integer vectors.
- Output from the interpreter is 3-element nested arrays:
    1. scalar integer return code
    2. 2-element integer type code
    3. data, of variable shape, rank, or depth
- If the interpreter's partner retracts the shared variable before the interpreter retracts it, the interpreter will not shut down; the interpreter will simply continue to execute the expression it is processing or wait for the next input.

© Copyright IBM Corporation 1994, 2017

Interpreter Input Data

Two types of data can be sent to the shared variable interpreter:

- Character vectors are sent to represent session input to the interpreter, in response to a □ or ⍞ request in an APL statement, or in response to an immediate execution or editor 1 prompt from the interpreter itself. Character vectors should only be sent when the interpreter has indicated it is ready for input. Character vectors sent at any other times are likely to be ignored, or may result in a shared variable interlock.
- Simple integer vectors may be sent as control signals. They include execution control signals which may be sent at any time, and output control signals which may only be sent when input has been requested.

The following control signal codes are defined:

- Execution Control Signal Codes

  `0 0` No operation (see note 1)
  `0 1` Attention
  `0 2` Interrupt
  `0 3` Suppress output
  `0 4` Shutdown

- Output Control Signal Codes

  `1 0` Set output to array mode (see note 2)
  `1 1` Set output to line mode
  `1 2 n` Set output to multi-line mode (see note 3)

Notes:

1. 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 will free the interlock condition and allow the interpreter to again reference and specify the variable.
2. Array mode output is unformatted, i.e. an arbitrary APL array produced as the result of an APL statement.

   Only the APL2/370 interpreter currently supports this mode; interpreters for the other systems ignore the `(1 0)` request.

3. Multi-line mode output is formatted as a vector of character vectors, with the number of lines limited to the third value given (shown here as `n`). This can provide significant performance improvements over single line output. Only output of a single type (see the second item of Output Data) is grouped together in one array.

   The APL2/370 interpreter does not currently support this mode; it ignores the `(1 2 n)` request.

© Copyright IBM Corporation 1994, 2017

Interpreter Output Data

The shared variable interpreter sends messages and arrays in 3-item arrays:

1. 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.
2. A 2 element integer vector, which is either `(0 0)`, or provides additional information about the type of output. When non-zero, the first element of this item indicates what type of output or input prompt is involved. The second element indicates whether the output is complete. If the second element is `1`, then more information will be sent at a later time. `1` is used for ⎕ output from an APL statement, and also for input requests when prompt data is provided.
3. Output data, if any. Its format and content depends on the first two items.

Here are the types of output that may appear. In the following definitions, `i` may be `1` indicating the output is incomplete, or `0` indicating it is complete.

`1 (0 0) array`	Normal output. `array` is either a character vector, a vector of character vectors, or an arbitrary array, depending on the output mode in effect. See Signal Codes under Input Data.
`0 ⎕ET ⎕EM`	An error was raised during execution. **Note:** Only APL2/370 systems currently return this form. On other systems, errors result in several lines of class 8 (error message) output.
`¯1 (0 i) 'text'`	System message. **Note:** Only APL2/370 systems currently return this form.
`¯1 (1 1) ' '`	APL (immediate execution) prompt. The interpreter is ready for normal input. The data is the normal six blank prompt. The "incomplete" flag is set because the prompt is not a full output line.
`¯1 (2 1) ' '`	Quad prompt. The interpreter is ready for ⎕ input. The data is the normal six blank prompt. The interpreter does not provide the ⎕:
`¯1 (3 1) '[n] '`	Function definition prompt. The interpreter is ready for function definition input. The line number prompt does not have to be echoed in the value returned. A quick way to copy small functions from the controlling session to the controlled session is to use a statement like this:

© Copyright IBM Corporation 1994, 2017

	`APL2←'2 ⎕TF ',2 ⎕TF 'function'`
`¯1 (4 i) '[n] data'`	Function line display.
`¯1 (5 i) ' ∇ date time'`	Function time stamp.
`¯1 (7 i) 'name[n] '`	Function name line number (stop or trace).
`¯1 (8 i) 'text'`	Error message.
`¯1 (9 i) 'text'`	System command message.
`¯1 (10 i) 'data'`	Traced result.
`¯1 (11 i) 'data'`	Quad output.
`¯1 (12 i) 'data'`	Quad-Prime output.
`¯1 (13 i) 'data'`	Debug output.
`¯1 (14 i) 'data'`	Quad-Prime prompt. The interpreter is ready for ⍞ input. The data may be empty or include ⍞ output. **Note:** The class 12 or 14 prompt data should normally be returned along with the response. The system assumes that the response overlays data from the beginning of the line.
`¯2 (0 0) (⎕PW ⎕PP PBS ⎕PR ⎕FC)`	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. `PBS` represents the current setting of the APL2/370 `)PBS` command, and is either `0` (for `PBS OFF`) or `1`. Interpreters other than APL2/370 always provide `0` for this item.
`¯3 (0 0) 'line'`	Stacked Input. This value is sent whenever the interpreter requests input and the AP 101 stack is not empty. `line` is the first character vector in the stack. It has been removed from the stack, but not given to the interpreter. It should (normally) be returned on the following input request.

© Copyright IBM Corporation 1994, 2017

Transferring Workspaces and Files

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.

Files created by the APL2 file processors, AP 210 and AP 211, can also be transferred across systems, or accessed on shared media.

- Workspace Transfer Between APL2 Systems
- Migration of TryAPL2 Workspaces
- Migration of VS APL Workspaces
- AP 210 Files
- AP 211 Files

© Copyright IBM Corporation 1994, 2017

Workspace Transfer Between APL2 Systems

In general, APL2 workspaces must be sent to other APL2 systems as *transfer form* files. On workstation platforms, these files have an extension of `.atf`. On mainframe platforms, these files have an extension or filetype of `apltf`. Note that on some workstations the names of transfer files and workspaces and their extensions are case sensitive.

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. 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 the mainframe systems use an EBCDIC character encoding, while the workstation systems 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
```

When transferring a workspace from a mainframe system to a workstation system, there is a possibility that there may be APL names in the workspace containing underscored alphabetic characters. These characters are not supported on the workstation platforms, and will cause errors on `)IN`. The `ATFUSTOLC` function

© Copyright IBM Corporation 1994, 2017

(described in The MIGRATE Workspace) can be used to remove the underscored characters before the `)IN` process.

Note: Although it may be possible to run the older APL2/PC for DOS on the same system where you have installed this new version of APL2, the workspace formats used by the two programs are different. Workspaces must be migrated between the two systems using `.atf` files as described above.

© Copyright IBM Corporation 1994, 2017

Migration of TryAPL2 Workspaces

Workspaces saved under TryAPL2 can be read by all the workstation APL2 systems. The function `TRYLOAD` in the FILE workspace can be used to read these files.

© Copyright IBM Corporation 1994, 2017

Migration of VS APL Workspaces

VS APL workspaces can be migrated to the workstation platforms using the `VSCOPY` function from the MIGRATE workspace.

The saved VS APL workspaces should have a clear state indicator. This can be ensured before you `)SAVE` the workspace by using the `)SI` command to check the state indicator, and entering → one time for each `*` in the display from `)SI`.

Download each saved workspace in binary form to a file. The `VSCOPY` function assumes that the file extension used for downloaded VS APL workspaces is `.VWS` (renamed from `.VSAPLWS` on VS APL). Then, run the `VSCOPY` function against each workspace:

```
)CLEAR
)COPY 2 MIGRATE VSCOPY
VSCOPY 'filename'
)ERASE VSCOPY
)SAVE wsid
```


© Copyright IBM Corporation 1994, 2017

AP 210 Files

Files created by AP 210 are portable between all workstation APL2 systems. Files should be transmitted in binary mode.

In addition, files created by AP 210 on APL2/PC can be read by the workstation APL2 systems. Writing back to these files is allowed, but is not recommended for code `A` files. The format used for code `A` files is different from APL2/PC, and objects take a different amount of storage. File corruption is likely. In order to create a writable file, each object in the APL2/PC file should be read and rewritten to a new file.

© Copyright IBM Corporation 1994, 2017

AP 211 Files

Files created by AP 211 are portable between APL2 systems running on CMS, TSO, and workstations. 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 the workstation APL2 systems. Writing back to these files is not allowed. The function `REBUILD211` in the FILE workspace can be used to permanently convert the APL2/PC file to the new format if desired.

© Copyright IBM Corporation 1994, 2017

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 ⎕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.

The following sections discuss external names and associated processors in general:

- Applications of External Names
- Managing External Names from APL
- Environmental Considerations

The following associated processors are provided with the APL2 system:

- Processor 10 provides facilities through with programs written in Rexx may be created, manipulated, and executed. It also provides several routines for easy access to operating system files.
- Processor 11 provides facilities through which compiled programs written in languages other than APL can 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 text 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.
- Processor 14 provides facilities through which Java programs and variables may be accessed.
- Processor 15 provides facilities for accessing structured storage.

© Copyright IBM Corporation 1994, 2017

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 and what data structures they require in arguments. Once the information has been provided, you can then use external names 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 host information can vary widely between platforms, 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 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 matter. 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

© Copyright IBM Corporation 1994, 2017

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 in a single workspace. Namespaces, supported through processor 11, provide an alternative solution to this problem since each namespace contains its own namescope. 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.

© Copyright IBM Corporation 1994, 2017

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
- Invoking an External Name
- Querying an Associated Name
- Avoiding Name Conflicts

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 'BEEP'
1
```

causes the name BEEP 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 still may be associated with the processor in a workspace that was previously saved. If the workspace is subsequently reloaded, the processor will 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 will be 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:

© Copyright IBM Corporation 1994, 2017

```
      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 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
      (⊂3 11) ⎕NA¨'GEORGE' 'SALLY'
1 0
      SALLY 'GO ROUND THE ROSES'
VALENCE ERROR
      SALLY 'GO ROUND THE ROSES'
      ^
```


© Copyright IBM Corporation 1994, 2017

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 of 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'
```


© Copyright IBM Corporation 1994, 2017

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 auxiliary processors as described in Auxiliary Processors and Writing Your Own Auxiliary Processors.

© Copyright IBM Corporation 1994, 2017

Processor 10 - Communicating with Rexx

Processor 10 provides:

- Use of Rexx functions as APL functions
- Creation of Rexx routines from APL arrays
- Access to operating system files

Processor 10 is designed to be used with the Object Rexx language, originally developed by IBM and now managed by the Rexx Language Association. Either *IBM Object Rexx* or the 32-bit version of *Open Object Rexx* (also known as ooRexx) must be installed to access Rexx using Processor 10. Open Object Rexx is available at *http://www.oorexx.org*.

The operating system file utilities provided with the workstation version of Processor 10 do not actually require Rexx services. They may be used even where Rexx is not available.

This chapter provides a very brief introduction to Rexx, and some examples of using the Rexx language, but does not attempt to provide a tutorial on Rexx. Detailed information on Rexx can be found in the manuals supplied with the Rexx products.

Note: The mainframe version of Processor 10 also supports access to variables and constants in the Rexx variable pool. The workstation version of Processor 10 does not support access to the variable pool.

Using Rexx Functions as APL Functions

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 using dyadic `⎕NA` (See Associating Names with Processor 10). The function thus established is monadic, and its argument is either a character vector or vector of character vectors (See Constructing the Rexx Arguments). The function may be built-in to Rexx, part of a Rexx function package, or a Rexx program in a file. 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).

As an example, consider the use of the Rexx built-in functions DELWORD, WORDPOS and WORDS that operate on blank-delimited substrings of vectors, called *words*.

DELWORD(*string*,*offset*,*length*)
: deletes *length* words from *string* beginning at *offset*

WORDPOS(*target*,*string*)
: returns the offset in *string* of *target*

WORDS(*string*)
: returns the number or words in *string*

© Copyright IBM Corporation 1994, 2017

The following example illustrates the basic technique for building APL applications that use Rexx functions. Although the example limits itself to built-in Rexx functions, you may also access Rexx external programs. By writing your own functions in Rexx, you can enhance the power of APL with the string handling and system access of Rexx.

```
      ⍝ 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'
¯40
     ⍝ DELWORD requires two arguments
     ⍝ Now for a more perennial example
     (⊂3 10) ⎕NA¨'WORDPOS' 'WORDS'
1 1
     HERBS←'PARSLEY, SAGE, ROSEMARY AND THYME'
     SUBHERBS←'SAGE, ROSEMARY'
     WORDPOS SUBHERBS HERBS
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 (WORDPOS SUBHERBS HERBS) (WORDS SUBHERBS)
PARSLEY, AND THYME
     ⍝ Combine APL and Rexx
      ∇Z←PHRASE REMOVE_FROM STRING;POSITION
[1]    Z←STRING                                  ⍝ Assume not found
[2]    →('0'=POSITION←WORDPOS PHRASE STRING)/0   ⍝ Done if not found
[3]    Z←DELWORD STRING(POSITION)(WORDS PHRASE)  ⍝ Delete PHRASE
[4]    ∇
       SUBHERBS REMOVE_FROM HERBS
PARSLEY, AND THYME
```

Associating Names with Processor 10

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 that 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. For example:

```
      3 10 ⎕NA 'SUB substr'
1
```

associates name SUB with the Rexx built-in function **substr**, and

```
      3 10 ⎕NA 'ΔFV'
1
```

associates name ΔFV with the Processor 10 built-in routine of the same name.

© Copyright IBM Corporation 1994, 2017

Because APL names cannot contain the characters `#.$@!&?` , in order to access Rexx functions whose names contain those characters, you must use a surrogate name. For example:

```
      3 10 ⎕NA 'FRED fred.rex'
1
```

associates APL name `FRED` with the Rexx program in file `fred.rex`.

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.

Constructing the Rexx Arguments

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 `(⍳0)` - an empty numeric vector, or `(0⍴⊂'')` - an empty enclosed character vector. The vector is enclosed because empty character vectors are valid Rexx strings.

For example, the function TIME has no required arguments and could be invoked in a Rexx program as `TIME()`. 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 'TIME'
1
      TIME ⍳0
16:06:58
      TIME ''
¯40
```

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
      ∧
      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')
```


© Copyright IBM Corporation 1994, 2017

```
      ^
```

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 specified function, Rexx generates error 40, which is returned to APL as a numeric return code.

```
      SUBWORD 1⍴⊂'FRED'
¯40
```

This can sometimes happen inadvertently if you forget that when simple scalars are juxtaposed, you get a simple vector, not a nested vector.

```
      3 10 ⎕NA 'DELWORD'
1
      DELWORD 'A B' '2'
A
      DELWORD 'A ' '2'
A
      DELWORD 'A' '2'
¯40
      '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 or ⎕EC) or the CTN external function to convert from character to numeric format.

Some Rexx functions might return a character vector that is really several strings joined by a separator character; the newline character (x'0a'), for example. You can use the partition function (⊂) to convert such a partitioned string into an APL vector of vectors.

Other Rexx functions are capable of returning a vector that might represent a structure of data containing both numbers and characters. You can often use the RTA external function 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.

© Copyright IBM Corporation 1994, 2017

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 ERROR, RANK ERROR or LENGTH ERROR (See Constructing the Rexx Arguments).

Rexx errors are typically some sort of syntax error in the Rexx program, or a problem in locating and calling the program. 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.

In addition to the numeric result, Rexx messages indicating more about the error will often be produced in the APL2 interpreter's window. The interpreter window is minimized in normal operation. If you have received a Rexx error and you wish to check for Rexx messages, you will need to find the interpreter window in the operating system task bar or task list and restore it to view them.

Creating Rexx Routines from APL Arrays

APL arrays can be executed as Rexx programs using the Processor 10 built-in function ΔEXEC.

Before using ΔEXEC, dyadic ⎕NA must first be used to associate its name with Processor 10:

```
      3 10 ⎕NA 'ΔEXEC'
```

See Associating Names with Processor 10 for more details.

```
r←array ΔEXEC arguments
```

Executes the Rexx program contained in array with the items of arguments as the argument strings.

array is a character vector, matrix or vector of vectors containing the Rexx program.

arguments is a vector of 1 to 20 character vectors that are the strings to be passed to the Rexx program as arguments. (See Constructing the Rexx Arguments)

The right argument of ΔEXEC does not include the "name" of the program. In effect, the temporary program contained in the left argument array is invoked in the same way as described in Using Rexx Functions as APL Functions.

The following example demonstrates the use of ΔEXEC:

```
      ⍝ NORMALTIME is a character matrix
       NORMALTIME
Parse arg year month day rest
if length(month) = '1' then month = '0'month
if length(day) = '1' then day = '0'day
return DATE('N',year||month||day,'S')
       3 10 ⎕NA 'ΔEXEC'
1
       NORMALTIME ΔEXEC ⍎⎕TS
9 Feb 1999
```


© Copyright IBM Corporation 1994, 2017

Accessing Operating System Files

Three built-in file utilities are provided with Processor 10. ∆F returns information about files. ∆FV and ∆FM read and write text files as APL character arrays.

Before using the file utilities, dyadic ⎕NA must first be used to associate the name of the function with Processor 10. For example:

```
3 10 ⎕NA '∆FV'
```

See Associating Names with Processor 10 for more details.

∆FV and ∆FM are equivalent in function to the APL functions of the same name in public workspace FILE, except that:

1. The APL functions are available on all the APL2 workstation systems and on the DOS APL2/PC system. They are not available on the APL2 mainframe systems.
   The Processor 10 functions are available on the APL2 mainframe systems and workstation systems. They are not available on the DOS APL2/PC system.
2. The Processor 10 functions are generally faster than the APL functions.
3. The Processor 10 functions are less likely to generate WS FULL than the APL functions.

The ∆F Function

```
fileinfo←∆F filespec
```

returns information about a file.

filespec is a character vector containing the file name. The name may include path information. If path is not included, the file will be located using standard operating system search order.

fileinfo is a nine-element nested array:

fileinfo[1]	The full file name, including path
fileinfo[2]	File attributes (read, write, execute)
fileinfo[3]	Length of longest record
fileinfo[4]	Number of records
fileinfo[5]	Total file size (bytes)
fileinfo[6]	Time of last modification
fileinfo[7]	Drive number
fileinfo[8]	'?' (unused)
fileinfo[9]	'?' (unused)

© Copyright IBM Corporation 1994, 2017

If the operation is unsuccessful, `fileinfo` contains the return code from the operating system file routine. The `CHECK_ERROR` function can be used to obtain more information.

The ∆FV Function

```
r←∆FV filespec
```

Reads a file into an APL vector of character vectors. Trailing blanks are removed from the end of each record. The file's line delimiters are removed. Delimiters of CR/LF or simple LF are handled.

If the operation is successful, `r` contains the APL array. If it is unsuccessful, `r` contains the return code from the operating system file routine.

```
r←array ∆FV filespec
```

Writes an APL array to a file with variable-length records. `array` may be either a character matrix or a vector of character vectors. Each row of the matrix or character vector is written as a file record. Trailing blanks are removed from the end of each record. Line delimiters are added to the end of each record.

If `array` is empty, the file is erased.

`r` contains the return code from the operating system file routine. `0` indicates success.

The ∆FM Function

```
r←∆FM filespec
```

Reads a file into an APL character matrix. Each row of the matrix is padded with blanks to the length of the longest record in the file. The file's line delimiters are removed. Delimiters of CR/LF or simple LF are handled.

If the operation is successful, `r` contains the APL array. If it is unsuccessful, `r` contains the return code from the operating system file routine.

```
r← array ∆FM filespec
```

Writes an APL array to a file with fixed-length records. `array` may be either a character matrix or a vector of character vectors. Each row of the matrix or character vector is written as a file record. Trailing blanks are not removed. For vectors of character vectors, each record is padded with blanks to the length of the longest vector. Line delimiters are added to the end of each record.

If `array` is empty, the file is erased.

© Copyright IBM Corporation 1994, 2017

`r` contains the return code from the operating system file routine. `0` indicates success.

© Copyright IBM Corporation 1994, 2017

Processor 11 - External Routines and Namespaces

Processor 11 provides facilities that allow access to objects outside the active workspace. These objects can be routines written in languages other than APL2, or APL2 objects in *namespaces*.

APL2 and processor 11 manage the necessary housekeeping, and argument and result conversion, based upon descriptive information provided for each routine by the user.

- Accessing Non-APL Routines
- Accessing Namespaces
- Name Association Left Argument Syntax
- NAMES Files and Routine Descriptors
- Routine Descriptor Tags
- Array Patterns

See Managing External Names from APL for more information on the characteristics of external names in general.

Accessing Non-APL Routines

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. Once access to a routine is established through processor 11, the routine is treated like a locked APL function. 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 reside in DLLs (Windows) or executable modules (Unix systems). Processor 11 must know the name of the routine, and the name of the DLL or module it resides in.

All of this information provided to Processor 11 in a *routine descriptor*. The routine descriptor can be passed to processor 11 directly in the left argument of ⎕NA, or stored in an external NAMES file (see NAMES Files and Routine Descriptors).

Accessing Namespaces

Facilities are provided with APL2 that allow a saved workspace to be encapsulated and converted to a namespace file. Such namespace files can be subsequently accessed through Processor 11. Access to the APL objects (arrays, functions and operators) in a namespace is provided through the use of ⎕NA.

© Copyright IBM Corporation 1994, 2017

To convert a saved workspace to a namespace, the saved workspace must be processed by the external routine CNS.

Information used to locate the objects 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, or provided in the left argument of dyadic ⎕NA (see NAMES Files and Routine Descriptors).

Once external objects are declared 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, 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 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. The saved workspace is converted to a namespace using the external routine CNS that is provided with APL2:

```
      3 11 ⎕NA 'CNS'
1
      'SETUP' 'RUN' CNS 'REPORT'
REPORT.ans
```

   The right argument to CNS is the name of the workspace to be converted. The optional left argument is a list of names to be made available for access. Because a name list was provided as the left argument to CNS in this example, external access to the namespace will be restricted to the names SETUP and RUN only.

   CNS returns as its result the name of the file it has created.

© Copyright IBM Corporation 1994, 2017

3. The SETUP and RUN functions may then be accessed as normal APL functions through the use of ⎕NA:

```
      'REPORT' 11 ⎕NA 'SETUP'
1
     SETUP 'INITIAL TEST'
SETUP COMPLETE
      'REPORT' 11 ⎕NA 'RUN'
1
     RUN 'INITIAL TEST'
RUN COMPLETE
```

Characteristics of Namespaces

Namescopes

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 namespace. The functions and operators in a namespace can call other functions and operators or access arrays within the same 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)LOADed 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.

Namescope Switching

You can declare a name to be 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 corresponding 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.

The system remains in the alternate 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.

Operators in Namespaces

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.

For example, the following defined operator in a namespace returns the canonical representation of FUNCTION from the *caller's* namescope.

© Copyright IBM Corporation 1994, 2017

```
      ⍝ In the called namescope
      ∇RESULT←(FN MOP) ARGUMENT
[1]   RESULT←FN ARGUMENT
[2]   ∇
      ⍝ In the calling namescope
      4 11 ⎕NA 'MOP'
1
      ⎕CR OPERATOR 'FUNCTION'
```

Querying the Namescope

The external routine 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.

Accessing the Caller's Namescope

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 expressions, system functions, defined functions, or defined operators. An application in a namespace, for example, might wish to obtain the value for ⎕IO from the caller's namescope, or might wish 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 several ways:

1. Reaching back to the primary namescope (active workspace) is accomplished with a special form of ⎕NA:

   ```
         '' 11 ⎕NA 'object'
   1
   ```

   This form is valid only when issued from a namespace, and always accesses the primary namescope, regardless of how many namescopes are available.

2. Reaching back to the previous namescope (your caller's namescope) can be done by using the external routine EXP, supplied with the APL2 system. For example:

   ```
         3 11 ⎕NA 'EXP'
   1
         ⍝ Reference caller's ⎕IO
         EXP ⊂'⎕IO'
   0
   ```

   EXP causes an implicit switch 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 call to EXP), this namescope switch accesses the namescope that last issued an explicit call to the current namescope.

3. Reaching back to a specific target namescope when there is more than one alternate namescope available can be accomplished by designing the application namescope to be entered through a defined

© Copyright IBM Corporation 1994, 2017

function or operator with an argument that identifies the target namescope. That argument can subsequently be used with ⎕NA to access names in the target namescope. For example, before calling the namespace, 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 namespace application (and in turn passed by it to other namespaces):

```
      'NSAPPL' 11 ⎕NA 'FUNCTION'
1
      FUNCTION CURRENT
```

The namespace application can then use the value passed to it as a left argument to ⎕NA:

```
      ∇Z←FUNCTION TARGET;OBJLIST
[1]   TARGET ⎕NA 'NL ⎕NL'
[2]   OBJLIST←NL 2 3 4
[3]   ∇
```

System Variables in Namespaces

Each namespace contains its own copy of the system variables, except for ⎕NLT, ⎕PW and ⎕TZ, which are session variables and have only a single value in the user's session. System variables such as ⎕IO, ⎕PP, ⎕RL, etc. can have different values in a namespace than they have in the user's active workspace.

Storage Consumption

Objects in the namespace do not consume permanent space in the user's workspace unless their definitions are modified during execution in the namespace's namescope. If an object is so modified or if new objects are dynamically created in the namespace's namescope, those new definitions consume space in the user's workspace.

In addition, on the first ⎕NA to a namespace, a copy of the name table in the namespace is made in the active workspace. Any changes to objects in the namespace are recorded in this copy of the name table and such changes are local to the user who issued the ⎕NA.

Effect of)SAVE and)LOAD

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 saved with)SAVE. The information required to form an association is also produced by)OUT for use by)IN.

© Copyright IBM Corporation 1994, 2017

Objects that have been modified or created during execution in the namespace's namescope are saved along with the user's workspace when a)SAVE command is issued, along with sufficient information to re-access names referenced by the namespace. The namespace itself is not saved.

If a saved workspace that contains names associated with one or more namespaces is loaded, the namespaces are not accessed 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 namespaces are no longer available when the re-association occurs, attempts to access the objects fail.

If, after a)LOAD command, it is found that the namespace has been modified (that is, recreated from the saved workspace) since its last use, a warning message is produced and the user is given access to objects in the new namespace. Data or objects in the namespace that were created or modified through previous use of the namespace, and saved when the user's workspace was saved, are lost.

Portability of Namespace Applications

In general, APL2 code that uses namespaces is quite compatible across the systems which support them. However, there are some differences. In order to write applications which can be run on multiple systems the programmer must take these differences into consideration.

1. The process of creating a namespace is different on different systems. On mainframes, external function PACKAGE is used to create the namespace file, and additional steps may be required to create an operating system load module from that file. On workstations, external function CNS is used to create the namespace file, and no additional steps are required.
2. The namespace file itself is not portable across platforms. Namespaces are built from saved workspaces, and like saved workspaces, they are specific to the type of hardware on which they were generated.
3. On each system, association to objects in the namespace is supported by several different forms of the left argument to ⎕NA. Not all of these forms are supported on all systems. This difference can be handled by placing the object descriptor in an external names file appropriate to the system. and using the common association syntax:

   ```
   0 11 ⎕NA 'OBJECT'
   ```

   See NAMES Files and Routine Descriptors for more information on using a names file.

4. On mainframe systems, after a namespace object has been associated, changes to the valence or name class of the object are not tolerated. If such changes are made, VALENCE ERROR or SYNTAX ERROR will result when using the object. On workstation systems, changes to the valence or name class are handled in the same way as other changes to the object definition. The modified object is stored in the user's workspace and the name table for the namespace points to the modified version, which can be used normally. Applications written to run on both mainframe and workstation systems should not make changes to the valence or name class of associated objects.

Name Association Left Argument Syntax

The APL2 system function ⎕NA is used to establish an association between a workspace name and an external object that processor 11 manages. The syntax of the ⎕NA expression for processor 11 can take six forms.

© Copyright IBM Corporation 1994, 2017

1. The first form causes processor 11 to look for the routine descriptor in an external NAMES file located by the value of the `p11` session option. The session option can be specified as APL2 invocation parameter `-p11`, as a P11 keyword entry in the [Invocation Options] section of the `apl2.ini` file, or as environment variable APLP11.

   ```
   class 11 ⎕NA 'object'
   ```

   `class` indicates the expected name class of `object` (1, 2, 3 or 4, as defined for the system function `⎕NC`, or 0 to mean any class is accepted.) If `class` is non-zero, it is compared against the actual name class of the object. If they do not match, the association fails.

   Note: Name classes 1, 2 and 4 are supported only for namespaces.

   The search for `object` progresses as follows:

   1. Processor 11 looks for the value of session option `p11`.
      The value can contain a single file name or a series of file names separated by the path delimiter character (on Windows, the semicolon; on Unix systems, the colon). The file names can be fully qualified. The files are processed in order until a match is found.
      If a value was not specified, the system proceeds using the simple name `aplnm011.nam` on Windows; the name `/usr/APL2/defaults/aplnm011.nam` on Unix systems.
   2. Processor 11 attempts to open the specified file using standard operating system facilities. If the file is not found, the search proceeds to the next file in the list. If there are no more files to be searched, the association fails.
   3. Processor 11 searches the NAMES file for an entry for `object`. If an entry is not found, the search proceeds to the next file in the list. If there are no more files to be searched, the association fails.
2. The second `⎕NA` form causes processor 11 to use an environment variable other than `APLP11` to locate the NAMES file.

   ```
   '(envvar)' 11 ⎕NA 'object'
   ```

   The search follows the same progression, using the environment variable `envvar` rather than `APLP11`.

3. The third `⎕NA` form supplies processor 11 with the file name of the NAMES file or files directly.

   ```
   '<filenames>' 11 ⎕NA 'object'
   ```

   The search, using `filenames`, follows the same progression, bypassing the use of an environment variable.

4. The fourth `⎕NA` form supplies processor 11 with the routine descriptor directly.

   ```
   'routine_descriptor' 11 ⎕NA 'object'
   ```

   The first character of the routine descriptor must be a colon (`:`). Routine descriptors are discussed in NAMES Files and Routine Descriptors.

© Copyright IBM Corporation 1994, 2017

5. The fifth form is valid only for namespaces. The filename of the namespace file (created by the `CNS` external routine) is provided directly.

   ```
   'namespace' 11 ⎕NA 'object'
   ```

   A path and the file extension may be provided if desired. If not provided, the current working directory and the extension `.ans` will be used.

6. The sixth form is valid only from within a namespace.

   ```
   '' 11 ⎕NA 'object'
   ```

   This syntax specifies that the object is to be located in the user's active workspace and not in a namespace. Attempts to issue this form from the active workspace fail and return `0`.

NAMES Files and Routine Descriptors

This section discusses the format and contents of the NAMES file and the routine descriptor.

NAMES files are text files that contain comment records (an `*` in column 1), case-insensitive keywords that start with a "`:`" and end with "`.`" (known as *tags*), and descriptive information (anything following the tag, up to the next tag or end of record, except for trailing blanks). The syntax of each noncomment record is:

```
:keyword.Descriptive text.
```

More than one keyword/text pair can be on each non-comment line.

The description of each routine is located by a `:nick.` tag with the descriptive information being the name by which processor 11 knows it, and ends at the next `:nick.` tag or the end of the file.

If the routine descriptor is to be passed in the left argument of `⎕NA` rather than through a names file, the `:nick.` tag is not used. Rather, the set of descriptor tags that would follow the `:nick.` entry for the routine are passed. The first character of the routine descriptor character vector must be a colon (`:`).

The descriptive text for each tag can contain colons and periods provided that they cannot be interpreted as tags. It is not valid to use any sequence consisting of a colon, a letter, optionally one or more additional letters or numbers, and finally a period.

The maximum record length allowed in a NAMES file is 254 characters. In some cases, however, text of more than 254 characters in length may be required for a tag. To accommodate this requirement, the text can be written in successive records in a NAMES file, each of which is prefixed with an appropriate tag. (Use preceding blanks in additional descriptive text if blanks are required between phrases.) For example:

```
:rarg.(G0 1 3) (1 E8 *)
:rarg. (I4 2 * 10)
:rarg. (I4 2 10 *)
```

The limit on the total length for all the descriptive text associated with a tag is 4095 characters.

© Copyright IBM Corporation 1994, 2017

Routine Descriptor Tags

The following tags are supported for both non-APL routines and namespaces:

- :nick.
- :desc.
- :link.
- :lib.
- :path.
- :macro.

The following additional tags are supported for non-APL routines:

- :proc.
- :valence.
- :rarg.
- :rslt.

:nick.*name*

Specifies the name that will be used by APL2 for the routine.

This tag is used in a NAMES file to associate the routine description following the tag with the specified name. The same name must be used as the right argument of the name association.

If the routine descriptor is passed as part of the left argument of `⎕NA` , this tag is not used. In that case, the routine name is implied from the right argument.

:desc.*description*

(Optional) Allows inclusion of descriptive text in a routine descriptor.

Comments can also be included in the NAMES file by placing an asterisk in column 1 of comment records.

:link.*type*

Specifies the type of program linkage to be used when calling the routine. *type* can have one of the following three values:

`APL` indicates an APL namespace. If no `:link.` tag is provided, this is the default.

`SYSTEM` is the default linkage supported by most languages, also known as `_System` or `__stdcall` on Windows, or `CDECL` on Unix systems. Arguments are converted by Processor 11 from internal APL format to the format expected by this type of routine. The result, if any, is converted back to internal APL format by Processor 11.

See Using Prebuilt DLLs for more information on using `:link.SYSTEM` linkage to call existing routines on Windows. See Creating SYSTEM Linkage Routines for more information on writing

© Copyright IBM Corporation 1994, 2017

new routines with `:link.SYSTEM` linkage.

Notes:

- `:link.SYSTEM` does not support dyadic functions.
- `:link.SYSTEM` supports up to 64 parameters.

`FUNCTION` describes the linkage used for routines written expressly to be called by APL2. APL objects are passed directly to the routine as arguments and the routine must build an APL object to return as the result.

See Creating FUNCTION Linkage Routines for more information on writing routines with `:link.FUNCTION` linkage.

:lib.*file*

Names the file that contains the namespace or non-APL routine. The name can be fully qualified.
For non-APL routines on Windows, it must be a `DLL`.

:path.*fullpath*
:path.(*envvar*)

Path used to locate the file specified in the `:lib.` tag.

If the path is enclosed in parentheses, it is evaluated as the name of an environment variable that gives a path to locate the file.

If a `:path.` tag is not specified, normal operating system search order is used.

:macro.*name definition*

Allows definition of shorthand notations for frequently used expressions.

`:macro.` is valid only in NAMES files. Macros must be defined before the first `:nick.` tag, and are invoked by $(*name*).

Once a macro name is defined, all the subsequent instances of $(*name*) in the same file are replaced with the definition portion of the line. For example, a macro defined as:

```
:macro.CHARP (G4 0) (S1 1 *)
```

Can be used later on as:

```
:rslt.$(CHARP)
```

which is interpreted as

© Copyright IBM Corporation 1994, 2017

```
:rslt.(G4 0) (S1 1 *)
```

The `:macro.` facility allows 20 levels of nested macros. Any tags appearing in the definition are treated as normal text. Leading and trailing spaces are ignored.

:proc.*entrypoint*

The entry point name in the executable file specified by the `:lib.` tag.

The entry point name is case sensitive.

If not specified, defaults to the name used in the right argument of the name association.

Note: This tag is valid for non-APL routines only. If specified for a routine in an APL namespace, it is ignored.

:valence.*er fv ov*

Specifies the valence attributes of the routine.

er specifies whether the routine produces an explicit result. If *er* is 0, the routine does not produce an explicit result. If *er* is 1, the routine may produce an explicit result.

fv specifies whether the routine is niladic or monadic. If *fv* is 0, the routine is niladic. If *fv* is 1, the routine is monadic. If *fv* is 2, the routine is dyadic.
Note: Dyadic functions are only supported for routines with `:link.FUNCTION`.

ov must be 0.

If not specified, the valence attributes default to 1 1 0.

Note: This tag is valid for non-APL routines only. If specified for a routine in an APL namespace, it is ignored.

:rarg.*pattern*

Specifies a pattern for the right argument of the external routine with linkage type `:link.SYSTEM`. See Array Patterns for details on patterns.

Note: This tag is valid for non-APL routines with system linkage only. If specified for an APL namespace or function linkage routine, it is ignored.

:rslt.*pattern*

Specifies a pattern for the result of the external routine with linkage type `:link.SYSTEM`. See Result Patterns for details.

Notes:

1. `>` and `*` cannot be used in `:rslt.` patterns.

© Copyright IBM Corporation 1994, 2017

2. This tag is valid for non-APL routines with system linkage only. If specified for an APL namespace or function linkage routine, it is ignored.

Array Patterns

The `:rarg.` and `:rslt.` tags are used to specify the expected arguments and result for an external routine with linkage type `:link.SYSTEM`.

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.

An *array pattern* is a keyword value or character vector that describes the structure and type of an array. In addition to using array patterns to describe routine arguments and results, array patterns are used with several external routines supplied with APL2. Array patterns are given as the left argument of the external routines `RTA` and `ATR`, and are produced by the external routine `PFA`. The external routine `SIZEOF` can be used to determine the number of bytes described by a pattern. Array patterns are also used in Associated Processor 15 associations.

- Array Item Patterns
- Nested or Mixed Arrays
- Array Patterns for Non-APL Programs
- Array Patterns for ATR, PFA, RTA, SIZEOF, and Associated Processor 15

Array Item Patterns

The general form of an array item within an array pattern is as follows:

```
([count][>]type rank[shape])
```

Where:

`(` An optional indicator for readability. Inclusion or omission has no effect on processing. A left parenthesis can be used only at the beginning of an item pattern, and cannot be followed by a blank.

`count` An optional integer or `*`, indicating the number of items in the array. This field can be omitted if the number of items is fully specified by the *`shape`* below. It can be coded as `*` if the number of items is fully specified by the shape, or if a variable number of items is permitted.

`>` An optional update indicator of an argument that is set by the function. Such an argument is passed by name, as seen by the caller. The data provided by the calling program must be a character vector containing the name of an APL array. The content of that array is passed to the called program, and any updates to its argument are returned in that variable.
Note: The "result indicator" (<) defined for APL2/TSO and APL2/CMS is not implemented on any

© Copyright IBM Corporation 1994, 2017

other platforms.

type A required data representation indicator. The following codes are supported:

```
Type Description                 Storage per item
B1   Boolean                     1 bit
B8   Integer byte (unsigned)     1 byte
B16  Unsigned short integer      2 bytes
B32  Unsigned long integer       4 bytes
I2   Short integer (signed)      2 bytes
I4   Long integer (signed)       4 bytes
E4   Real single precision       4 bytes
E8   Real double precision       8 bytes
J8   Complex single precision    8 bytes
J16  Complex double precision    16 bytes
S1   Null terminated char string 1 byte, + 1 for string
S2   Null terminated char string 2 bytes, + 2 for string
C1   Character                   1 byte
C2   Extended character          2 bytes
C4   Extended character          4 bytes
X0   Skipped data                1 byte
G0   General object              none (inline structure)
G4   General object              4 byte pointer
```

rank A required integer showing the number of dimensions.

shape A set of integers, one per dimension (empty if *rank* is zero). If varying shapes are accepted, one or more items of the shape can be specified as `*` in the `:rarg.` tag.

`)` A readability indicator that can be provided at the end of an item pattern. Inclusion or omission has no effect on processing. If used, it must immediately follow the rank and shape information without intervening blanks.

The elements of the pattern must be separated from one another by one or more blanks or parentheses.

Array Item Pattern Examples

Array	**Array Pattern**
`'ABC'`	`(3 C1 1 3)` or `C1 1 3`
`2 3 4`	`(3 I4 1 3)` or `I4 1 3`

Nested or Mixed Arrays

The pattern for a nested or mixed array is a recursive structure in which the first subpattern has the type `G0` or `G4` (general object). Each subpattern can be surrounded by parentheses to make it easier to read. Parentheses do not indicate nesting; nesting can only be designated by using a `G0` or `G4` type. Each subpattern can be either an array item pattern or a group of items as defined here.

The following are examples of arrays with nested or mixed array patterns:

Array	**Array Pattern**
`'ABCD',2 3 4`	`(7 G0 1 7)(C1 0)(C1 0)(C1 0)(C1 0)(I4 0)(I4 0)(I4 0)` or `G0 1 7 C1 0 C1 0 C1 0 C1 0 I4 0 I4 0 I4 0`
`'ABCD' (2 3 4)`	`(2 G0 1 2)(4 C1 1 4)(3 I4 1 3)` or `G0 1 2 C1 1 4 I4 1 3`

© Copyright IBM Corporation 1994, 2017

As is shown in the examples above, a general object implies a recursive pattern where the first item is of type G0 (or G4). The count field in that item determines the number of additional array patterns that must follow.

Note: The functions ATR, RTA, and SIZEOF do not permit all the forms of array patterns. See Array Patterns for ATR, PFA, RTA, SIZEOF, and Associated Processor 15 for restrictions that apply.

Array Patterns for Non-APL Programs

Array patterns are most frequently used to define the interface to a program written in some language other than APL. These patterns are provided at the time of the name association, as discussed in Name Association Left Argument Syntax.

Processor 11 uses argument patterns for two purposes:

- Conformability checks on a routine's argument
- For routines with linkage :link.SYSTEM, to determine how to pass the argument to the external routine. The rest of the discussion of patterns in this chapter is directed at that type of routine.

It is useful to start by thinking of how arguments are passed to the external routine and to work backward from there to argument patterns and finally to the APL data structure needed to satisfy the patterns.

Consider first a routine that accepts a 4-byte integer and a 4-byte floating point number as arguments. To write the pattern for this, begin by saying that there are two arguments, and then list them:

```
:rarg. (2 G0 ...) (1 I4 ...) (1 E4 ...)
```

Focus on the type descriptors (G0, I4, and E4) and the number to their left, which is the number of items being described. The numbers to the right of the type descriptor deal with the data as viewed by APL.

Now consider what happens if the second parameter is an array of floating point numbers. Some languages make it possible to pass such an array directly, but more typically it is passed as a pointer to the array. Here is the pattern for a parameter list containing the integer as before, but also including a pointer to an array of 6 floating point numbers:

```
:rarg.(2 G0 ...) (1 I4 ...) (1 G4 ...) (6 E4 ...)
```

The first pattern descriptor still says there are two parameters, but the second parameter is now a G descriptor. Every G descriptor is followed by, and includes, as many subdescriptors as the number of items it defines.

Some languages treat "string" as a special data type, but both APL and C treat it essentially as an array of characters. Like other arrays it is most commonly pointed to rather than passed directly. In this example, a character string and a floating point array are passed:

```
:rarg.(2 G0 ...) (1 G4 ...) (20 S1 ... ) (1 G4 ...) (6 E4 ...)
```

Because the S1 is subordinate to the first G4, the two of them together count for only one of the two items defined by the initial G0.

© Copyright IBM Corporation 1994, 2017

An argument being passed can also be a structure. Such structures are typically pointed to rather than being passed directly. A G0 is used to define the structure, but a G4 is also used (as above) to specify the pointer to it. Consider a program that expects a single argument, which is a pointer to a structure containing an integer and an 8-character name:

```
:rarg.(1 G0 ...) (1 G4 ...) (2 G0 ...) (1 I4 ... ) (8 C1 ...)
```

You cannot replace the (1 G4) followed by the (2 G0) with a single (2 G4).(2 G4) implies two pointers, rather than a single pointer to two items.

Structures can contain pointers to other structures, which are represented by G4 entries within the structure definition, or they can contain other structures imbedded within them. In the next example, the 8-character name has been replaced by a date composed of day (within the month), a 3-character abbreviated month name, and a year. The day and year are both expressed as 2-byte integers. Assume that the program is expecting the year to start on an even-byte boundary, so there is an unused byte in front of it:

```
:rarg.1 G0... 1 G4... 2 G0... 1 I4... 3 G0... 1 I2... 3 C1... 1 X0... 1 I2...
```

These examples can help you generate patterns for any interface definition you might encounter; except for all those unexplained periods.

The first number after the type descriptor is always the rank as seen by the APL program, and is normally set to whatever is most natural for the APL programmer. For data entries (all except G0 or G4) this is completely determined by the nature of the data.

For G0 entries, this is always a vector (rank 1), except that when a single parameter is being passed the first G0 is often more conveniently defined as a scalar (rank 0). G4 entries usually represent a single pointer, and can best be represented as scalars. The exception is an array of pointers, which is of whatever rank is most appropriate to the data (usually rank 1).

Given the rank and the total number of items, you can deduce the shape information. Revisiting the examples above, all multi-item pattern descriptors are most naturally represented as vectors except for the 6-item floating point array, which could be a matrix. It is assumed to be a 2 by 3 matrix below. The first example becomes:

```
:rarg. (G0 1 2) (I4 0) (E4 0)
```

The numbers used earlier to the left of the type descriptor are optional, because they can be computed from the information on the right, so they have been omitted here for brevity. Viewed from the APL standpoint this is a 2-item vector, each item being a number. APL programs make no distinction between integers and floating point numbers, so this is a simple numeric vector (depth 1).

The second example (viewed beginning with the G0) is also a 2-item vector, but this time the second item is an array. Here is what the completed pattern looks like along with sample data for the call:

```
:rarg.(G0 1 2) (I4 0) (G4 0) (E4 2 2 3)
75 (⊂2 3⍴1.2 2.3 3.4 4.3 3.2 2.1)
```


© Copyright IBM Corporation 1994, 2017

Note: The array needs to be enclosed because the `(G4 0)` tells APL to expect a scalar.

The third example is very similar. Note in the sample data below that only 11 characters are passed even though the field was defined as 20 bytes long. This is permitted for null-terminated strings.

Note: There must be room at the end of the data for the null terminator. So at most 19 characters can be passed in this 20-byte string.

For null-delimited strings that are not updated by the non-APL program, there is no need to set a specific size ahead of time. Such strings are often defined with a length of `*`.

```
:rarg.(G0 1 2) (G4 0) (S1 1 20) (G4 0) (E4 2 2 3)
func (⊂'Hello there')(⊂2 3⍴1.2 2.3 3.4 4.3 3.2 2.1)
```

The `(G4 0)` implies not only a pointer as seen by the non-APL program, but also a level of nesting as seen from APL. But rules in this area can be a bit slippery, as the next example shows. Two changes have been introduced: a structure and a single parameter. Assume for the moment that the initial `G0` specifies a 1-item vector:

```
:rarg.(G0 1 1) (G4 0) (G0 1 2) (I4 0) (C1 1 8)
```

You might want to say that the following is valid data:

```
13 'Thomas  '
```

But this is a two-item array, and the pattern calls for a single item. Data such as that is appropriate for a routine that expected two arguments rather than one, so the data needs to be enclosed. That yields to a scalar value, and the pattern calls for a 1-item vector. Thus, given the pattern as assumed, correct data is:

```
,⊂⊂13 'Thomas  '
```

Now the reason for the earlier comment about a scalar `G0` becomes clear. A simpler form of the interface is:

```
:rarg.(G0 0) (G4 0) (G0 1 2) (I4 0) (C1 1 8)
⊂⊂13 'Thomas  '
```

It is typical of single-parameter interfaces that the APL program needs to explicitly enclose the data.

A simple case requiring an explicit enclose is that of a routine that accepts a pointer to a character string as its only parameter:

```
:rarg.(G0 0) (G4 0) (S1 1 *)
⊂⊂'string'
```

Although the first enclose of the character string seems reasonable, because it implies "pointer to," the second enclose seems a bit unnatural, and is due to the strict conformance to the rule used so far in the examples: The argument descriptor is started with a `G0` tag that identifies the number of parameters.

© Copyright IBM Corporation 1994, 2017

In the case of single-parameter arguments, it is often desirable to completely elide the `(G0 0)` descriptor from the `:rarg.` tag:

```
      :rarg.(G4 0) (S1 1 *)
      ⊂'string'
```

This simplification is most commonly seen in the case of a single parameter that is a simple number. For example:

```
      :rarg.(G0 0) (I4 0)
```

Is simplified to:

```
      :rarg.I4 0
```

Updated Arguments

Routines in some compiled languages 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 non-APL routine that expects two integer vectors as arguments, the second of which is to be updated in place, would be described:

```
      :rarg.(G4 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:

© Copyright IBM Corporation 1994, 2017

```
RESULT←3⍴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. Items that are to be updated must always be passed using a pointer, so there must always be a `G4` level ahead of any descriptor that contains a >.

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 routine.

Result Patterns

The `:rslt.` tag can be specified on routines with an explicit result to describe the form of the supplied result.

If supplied, `:rslt.` must be specified with a pattern. The pattern takes the same form as argument patterns described above, with the following exceptions:

1. `*` cannot be used in the pattern.
2. The > optional update indicator cannot be used in the pattern.

If the routine produces an explicit result and a `:rslt.` tag is supplied, the pattern is used to format the routine's result. If a `:rslt.` tag is not supplied, the right argument is returned. The indirect values in the argument might have been updated.

If the routine does not produce an explicit result, the `:rslt.` tag is ignored.

Explicit Results

The first value of the `:valence.` tag is used to specify whether the routine returns an explicit result. This value is used to specify what `1 ⎕AT` should return when it is used to query the valence of the routine.

If the `:valence.` tag specifies that the routine returns an explicit result and no `:rslt.` tag is supplied, then the right argument is returned as the result. The routine may have updated the argument.

If the `:valence.` tag specifies that the routine does not return an explicit result, then the `:rslt.` tag is ignored.

Function Valence

Function valence is controlled by the second value coded in the `:valence.` tag.

Niladic functions do not accept arguments. If a `:rarg.` tag is supplied for a niladic routine, it is ignored.

Monadic functions with linkage `:link.SYSTEM` require a `:rarg.` tag.

Functions with linkage `:link.FUNCTION` cannot have a `:rarg.` tag.

© Copyright IBM Corporation 1994, 2017

Array Patterns for ATR, PFA, RTA, SIZEOF, and Associated Processor 15

External routines ATR and RTA accept array patterns as arguments, and external routine PFA returns an array pattern as its result. The external routine SIZEOF calculates the amount of storage described by a pattern. Associated Processor 15 uses patterns to determine how to convert data between bytes in storage and arrays in the APL2 workspace.

The following restrictions apply to patterns used as arguments to these routines:

1. A G0 descriptor can be used as needed, but the G4 descriptor is not supported.
2. The > optional update indicator is not permitted.
3. A single * can appear in either the *count* or *shape* field if its value can be computed from the other field. (*count* is the product of values in *shape*.)

Patterns returned by PFA also meet these restrictions, and always have *count* and *shape* fully specified.

© Copyright IBM Corporation 1994, 2017

Processor 12 - Files as Arrays

Processor 12 provides access to 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 (residing in a DLL) 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.

  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 uses the 'A' and 'D' file formats of the file auxiliary processor, AP 210. Files written by processor 12 can be read by AP 210, and vice versa. However, Processor 12 variables are quite different from variables shared with AP 210.

- 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.

The next sections discuss the following Processor 12 topics:

- Name Association Syntax for Processor 12
- Supported Primitive Operations
- APL Files as External Variables
- Text Files as External Variables
- Format Descriptors for External Variables
- Processor 12 Errors
- Portability of Processor 12 Applications

Name Association Syntax for Processor 12

© Copyright IBM Corporation 1994, 2017

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'`
: APL files. Arbitrary APL arrays are stored in the file in APL2 CDR format. This is equivalent to an AP 210 code 'A' file.

`'F'`
: Text files. Character arrays are stored in the file in ASCII form. This is equivalent to an AP 210 code 'D' file.

The types of access are:

`'W'`
: The file may be read and/or written. Exclusive control of the file is to be gotten and held for as long as the association is active. If write access cannot be obtained, the `⎕NA` returns `0`.

`'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, or opened if it does exist. If either `C` or `W` are specified alone, then the file must exist (`W`) or not exist (`C`) before the operation.

`'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` (`SYNTAX ERROR`, `Invalid operation in context`.) If `R` is specified together with `C` or `W`, it is accepted but ignored.

`'D'`
: The file is to be deleted on completion of processing. The 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` must be used together with one or more of the other access codes. It cannot be specified alone.

Notes:

1. The above access codes may be given in any order, and may be separated by blanks as desired.
2. The mainframe syntax for Processor 12 allows an additional parenthesized expression after the `C` code, to specify the record length and file size limit for the file. This expression will be tolerated, but ignored, by the workstation version of Processor 12.

Examples of type items:

© Copyright IBM Corporation 1994, 2017

'FW'
: Gain read/write access to an existing text file.

'ACW'
: Create an APL file, or gain read/write access to it if it already exists.

'FRD'
: Read a text file and then delete it.

'A C'
: Create an APL file, rejecting the association if it already exists.

'FCDRW'
: Create a text file, or accept an existing one. Allow read/write access, and delete it when the connection is erased.

`locator` A character vector indicating where the file is located. A complete path may be specified. If a complete path is not specified, the path of the current working directory is used.

`format` A character vector that defines the format in which the data is to be viewed by the application. The vector must be empty if the file class is `A`. See Format Descriptors for External Variables 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 displayed in the APL2 interpreter window.

Supported Primitive Operations

The following primitive operations are defined for external variables supported by Processor 12:

Each	`function¨file`
	`var function¨file`
	`file function¨var`
	`file1 function¨file2`

Outer Product	`var∘.function file`
	`file∘.function var`
	`file1∘.function file2`

Pick	`i⊃file`
Indexing	`file[i]` `i⌷file`
Indexed Assignment	`file[i]←⊂array`

Selective Assignment	`(i⊃file)←array`
	`(↑i↓file)←⊂array`
	etc.

Catenate	`file←file,⊂array`
Shape	`⍴file`
Compress/Replicate	`i/file`
First	`↑file`
Take	`i↑file`
Drop	`i↓file`

Notes:

© Copyright IBM Corporation 1994, 2017

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. In some circumstances, Compress, Replicate, Expand, First, Take and Drop can return prototype or fill elements. If the file is not empty, the prototype is created using the attributes of the first item in the file. If the file is empty, the prototype will be a character object whose dimensions are based on the format descriptor from the name association. If the length attribute in that descriptor was given as `*`, or the format descriptor was null, the length of the prototype object will be zero.

APL Files as External Variables

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 type, depth and shape.

The name association syntax for APL files is:

```
('A { C[(len size)] | R | W | D }...' 'filename' '') 12 ⎕NA 'name'
```

where `{ | | }...` and `[ ]` indicate choices, repetition, and options, but these symbols are not coded in the argument.

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.

Notes:

1. The optional parenthesized expression after `C` is not used by the workstation version of Processor 12. It is tolerated for compatibility with the mainframe version of Processor 12.
2. The mainframe version of Processor 12 allows library numbers to be included with the `filename` parameter. The workstation version of Processor 12 does not support library numbers.

Examples:

```
      ('ACWD' 'C:\TEMPFILE.P12' '') 12 ⎕NA 'TEMP'
1
      ('AR' 'APLFILE' '') 12 ⎕NA 'VAR'
1
```

Text Files as External Variables

Text files are viewed by the APL application as a vector of arrays in which the subarrays are always character vectors or character matrices. Each character vector, or each row of a character matrix, represents one record in the file.

© Copyright IBM Corporation 1994, 2017

The name association syntax for text files is:

```
('F { C[(len size)] | R | W | D }...' 'filename' { 'format' | ''} 12 ⎕NA
'name'
```

where `{ | | }...` and `[ ]` indicate choices, repetition, and options, but these symbols are not coded in the argument.

If the last sub-item of the first item in the left argument is empty, the default view for text files is used. This view is a vector of character vectors with one variable-length file record in each item.

Alternate views of the file are also possible, by specifying a non-null format descriptor. See Format Descriptors for External Variables for a discussion of this topic.

Notes:

1. The optional parenthesized expression after `C` is not used by the workstation version of Processor 12. It is tolerated for compatibility with the mainframe version of Processor 12.
2. PC compatible systems (such as Windows) generally use the two-character record delimiter `CR/LF` for text files, while Unix systems (such as AIX and Solaris) use the one-character record delimiter `LF`. Processor 12 will handle files with either type of record delimiter when reading. When writing, Processor 12 uses the convention of the system on which it is currently running. If Processor 12 is running on one type of system, but the file follows the opposite convention (it is remotely mounted, or has been transferred between systems in binary mode), it is possible to end up with a file in which some records have `CR/LF` and others have `LF`. Processor 12 can process files with mixed records, but other programs may or may not be able to process them correctly.

Examples:

```
      ('FCW' 'TEXTFILE.P12' 'G0 1 * C1 1 *') 12 ⎕NA 'TEXT'
1
      ('FR' 'C:\CONFIG.SYS' '') 12 ⎕NA 'CONFIG'
1
```

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 supported descriptors are:

© Copyright IBM Corporation 1994, 2017

Vector of Character Vectors

```
'G0 1 * C1 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. End-of-line delimiters are assumed to exist in the file to indicate where the records end. The delimiters will be removed from the data by Processor 12 when reading, and will be added automatically when writing to the file. Trailing blanks are not deleted when writing data to the file. When replacing records, the new record need not match the length of the old record.

Note: The mainframe version of Processor 12 does not allow for replacement of text records unless the lengths match exactly.

If specified as a number, the file will be viewed as having fixed-length records of the length given. Line delimiters will not be used to determine record length, and will not be removed or added by Processor 12. When adding or replacing records, the data given must match the record length exactly.

Vector of Character Matrices

```
'G0 1 * C1 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 increase 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 defaulted to the length of each record by specifying `*`.

If specified as `*`, the width of each matrix is the width of the longest record in that set. Shorter records in the set are padded with blanks to the length of the longest. End-of-line delimiters are assumed to exist in the file to indicate where the records end. The delimiters will be removed from the data by Processor 12 when reading, and will be added automatically when writing to the file. Trailing blanks are deleted from each row of the matrix before writing it to the file as a record. When replacing records, the new record need not match the length of the old record.

Note: The mainframe version of Processor 12 does not allow for replacement of text records unless the lengths match exactly. For this form of the descriptor, however, it will strip trailing blanks from

© Copyright IBM Corporation 1994, 2017

individual rows of the matrix as necessary to make them the correct length for the file.

If specified as a number, the file will be viewed as having fixed-length records of the length given. Line delimiters will not be used to determine record length, and will not be removed or added by Processor 12. When adding or replacing items, the width of the data must match the record length exactly.

Simple Data Vectors

```
'C1 1 *'
'C2 1 *'
'C4 1 *'
'I4 1 *'
'E8 1 *'
'J16 1 *'
```

These views of the file are as vectors of simple data elements. The file is treated as a binary data stream. It is divided into individual elements and interpreted based on the element length and type specified in the pattern:

Pattern	Data Interpretation	Bytes per item
C1	Character	1
C2	Unicode Character	2
C4	Unicode Character	4
I4	Integer	4
E8	Floating Point	8
J16	Complex	16

Control characters, including line delimiters, are not handled in any special way. They are considered part of the binary data stream.

Note: The mainframe version of Processor 12 does not allow these formats.

Processor 12 Errors

Processor 12 uses the following rules for handling errors:

- If ⎕NA cannot be completed successfully its explicit result is 0. A secondary error message may be displayed in the interpreter window to explain the failure, but 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. In addition, for some types of errors (typically, errors accessing the file) a secondary error message may be displayed in the interpreter window.
- On some operating systems, file updates may be performed to a temporary cache area, and errors in this process may not be known for some time after the variable assignment that caused the records to be

© Copyright IBM Corporation 1994, 2017

written. These may include things like out-of-space conditions as well as other I/O errors on the disk media. If detected during a subsequent access, errors of this sort are normally signaled as follows:

```
⎕ET=1 5 (SYSTEM LIMIT, Interface unavailable)
```

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 error may be signaled on some line of the function that looks at its argument.

- In some cases errors may not be detected until the association is deactivated. Processor 12 closes the file at this time, and any pending writes will take place. Secondary messages will be displayed in the interpreter window for any file I/O errors that occur.

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`
Insufficient storage for working areas

`1 9` `SYSTEM LIMIT, Array size`
The catenation of an additional data element would either cause the array to exceed the maximum number of elements allowed in an APL2 array, or cause the file to exceed the maximum size allowed for the file system in use.

`1 12` `SYSTEM LIMIT, Interface representation`
Class `A` has been selected, but the file does not contain valid APL objects.

`2 4` `SYNTAX ERROR, Invalid operation in context`
Attempt has been made to modify a variable associated with a read-only file, or a selective specification operation was used that is not supported for external variables.

`5 5` `INDEX ERROR`
An item has been referenced that is beyond the end of the file.

The following errors are reported only for text 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.

`5 3` `DOMAIN ERROR`
An item assigned to the variable does not contain character data, or contains extended characters, or is itself a nested array.

Portability of Processor 12 Applications

© Copyright IBM Corporation 1994, 2017

In general, Processor 12 is quite compatible across the APL2 systems which support it. However, there are some differences. In order to write applications which can be run on multiple systems the programmer must take these differences into consideration.

1. The `C` access code in the mainframe version of Processor 12 allows for record length and file size limits to be specified, and in some cases requires them. The workstation Processor 12 does not use these parameters. It does tolerate them if present, however, so a common expression can be used on all systems.
2. File name specifications are generally not compatible across operating systems. In a multiple-system application, the file name specification will probably need to be in a system-dependent expression or variable.
3. The mainframe Processor 12 supports library numbers for APL files. The workstation Processor 12 does not.
4. The APL file format is not the same on mainframe and workstation systems. On the mainframe, AP 121 files are used. On the workstation, AP 210 files are used. Thus, while the syntax to access the APL files is the same, the files themselves cannot be directly transferred between mainframe and workstation systems. Transfer files (.atf) and AP 211 files are two possible vehicles for sending APL data between mainframe and workstation systems.
5. The workstation Processor 12 allows replacement of text records with records of different sizes than those they replace. The mainframe Processor 12 does not.
6. The workstation Processor 12 allows text records of length 0. The mainframe Processor 12 does not.
7. The simple data vector views for type F files are not supported on the mainframe.
8. The mainframe Processor 12 supports the following expression to replace all the items in a file:

   ```
   (((⍴FILE)⍴1)/FILE)←DATA
   ```

   The workstation Processor 12 supports these additional expressions to accomplish the same task:

   ```
   ((⍴FILE)↑FILE)←DATA
   (0↓FILE)←DATA
   FILE[]←DATA
   ```


© Copyright IBM Corporation 1994, 2017

Processor 14 - Calls to Java

Processor 14 provides access to Java. Using Processor 14, APL2 programs can:

- Reference and specify static fields
- Call static methods
- Instantiate Java objects
- Reference and specify instance fields
- Call instance methods

Java methods called through Processor 14 can also call back to APL2.

Processor 14 uses the APL2-Java interface. This interface requires Java 2 Version 1.4 or later. Consult Installing Java and Installing the APL2-Java Interface Classes for detailed information about Java requirements and installation of the APL2-Java interface.

Java Language Overview

The section provides a very brief overview of the Java programming language. If you already know Java, you may want to skip this section.

Java programs are composed of Java classes. Each class performs a service such as file IO, database access, or user interface services. Some classes are built-in to Java; some are built by application developers.

A Java class is similar to an APL workspace. Like a workspace, a class resides in a single file, can contain variables and functions, and can have both global and local variables. In addition, functions can also be local or global.

Java uses different terms than APL for some programming concepts. In Java,

- *Public* means global
- *Private* means local
- *Field* means variable
- *Method* means function
- *Member* means either a field or a method.

Here is a sample Java class:

```
public class Sample
{
   public static int StaticField = 0 ;
   public static int StaticMethod(int Argument) {
      return Argument + StaticField ;
   }
   public int InstanceField ;
   public Sample(int InitialValue) {
      this.InstanceField = InitialValue ;
      return ;
   }
   public int InstanceMethod(int Argument) {
      return Argument + this.InstanceField ;
```


© Copyright IBM Corporation 1994, 2017

```
    }
}
```

The first line of the class definition supplies the name of the class, in this case Sample. Everything within the curly braces constitutes the class's definition.

The keyword public indicates that a field or method is accessible from outside the class.

The first two members are declared static. This means that they exist as soon as the class is loaded and can always be used. They correspond to global variables and functions in an APL workspace.

The next member, InstanceField, is not declared static; this means that it is an "instance" member and can only be used in the context of an instance of the class.

The fourth item is a special type of method called a constructor. Constructors are easily identified; they always have the same name as the class. When a constructor is called, an instance of the class, also called an object, is created. Even though the method is defined without the static keyword, because it is a constructor, it is accessible from outside the context of any instance.

Once you have an instance of the class, you can use the instance to access instance fields and methods. Within the context of a constructor or instance method, the word `this` refers to the instance.

To illustrate, here is some Java code that uses the Sample class:

```
//* Set the static field
Sample.StaticField = 24 ;
//* Use the static method
int Added = Sample.StaticMethod(47) ;
//* Create an instance of the Sample class, a Sample object
Sample Object = new Sample(999) ;
//* Call the instance method
int Sum = Object.InstanceMethod(456) ;
```

Notice for static members, you use the name of the class followed by a period and the member name. For instance members, you use the name of a field containing a reference to the instance followed by a period and the member name.

For further information about Java, consult the Sun Microsystems Java web site, http://java.sun.com. New users may find *The Java Tutorial*, http://java.sun.com/docs/books/tutorial particularly helpful.

Name Association Syntax for Processor 14

The general syntax for name association through Processor 14 is:

```
(class 'signature') 14 ⎕NA 'surrogate member'
```

`class` A class identifier. If a static member is to be associated, then `class` is a character vector containing the name of the class. If an instance member is to be associated, then `class` is an instance integer returned by a constructor.

© Copyright IBM Corporation 1994, 2017

`signature` A character vector description of the member. If the member is a field, then the signature is a description of the field's datatype. If the member is a method, then the signature is a description of the method's arguments and result. The signature is coded using the standard Java internal type signature syntax. For more information, see Java Signatures.

`member` The name of the Java method or field. The name must match the member name exactly. If the member name is not a valid APL2 name, then a surrogate must be used.

`surrogate` A name to be used within the APL workspace to refer to the member. The particular name used has no significance to Processor 14, and bears no required relationship to the name of the member with which it is associated.

In addition to providing access to Java fields and methods, Processor 14 includes several built-in functions. They can be accessed like this:

```
3 14 ⎕NA 'DeleteLocalRef'
3 14 ⎕NA 'GetApl2interp'
```

Further information about these functions can be found in Managing Java Objects and Calling Back to APL2 from Java.

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 displayed in the APL2 interpreter window.

Java Signatures

Java signatures are character vectors that describe the datatype of a field or the datatypes of a method's arguments and result. The signature of a field is a single datatype. The signature of a method is a list of one or more datatypes enclosed in parentheses followed by a datatype for the result.

Java uses the following characters to describe native data types:

Code	Type
`Z`	boolean
`B`	byte
`C`	char
`S`	short
`I`	int
`J`	long
`F`	float
`D`	double

© Copyright IBM Corporation 1994, 2017

`V` void

A left square bracket indicates an array. For example `'[I'` indicates an array of integers.

A class name is indicated by preceding the name with an L and following it with a semi-colon. For example, this is the signature for the Java String class: `'Ljava.lang.String;'`.

Signature Examples:

An integer field	`'I'`
A String field	`'Ljava/lang/String;'`
A method that takes 2 integer arguments and returns a double	`'(II)D'`
A method that takes a String argument and returns an integer array	`'(Ljava/lang/String;)[I'`

The Java SDK tool javap can be used to extract the signatures of Java class fields and methods. For example,

```
javap -s com.ibm.apl2.Apl2interp
```

Supported Primitive Operations for Fields

Processor 14 provides access to fields by maintaining an image of the field as an array that appears to reside in the active workspace. This is analogous to the behavior of Processor 12 for files. That processor can create an image of a file that appears to reside in the active workspace. Neither the file (for Processor 12) nor the field (for Processor 14) is actually within the workspace. This has the following implications for Processor 14 fields:

Very large fields can be accessed, fields 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/field`
Each	`process¨field`
Pick	`i⊃field`
Selective assignment	`(i⊃field)←value`

It should also be noted that fields have an existence independent of the workspace. Assigning a value to a Processor 14 variable causes an immediate and permanent change to the field. This is not affected by later expunging the variable.

The following primitive operations are defined for fields associated by Processor 14:

Each	`function¨field`
	`var function¨field`
	`field function¨var`

© Copyright IBM Corporation 1994, 2017

	`field1 function¨field2`

Outer Product	`var∘.function field`
	`field∘.function var`
	`field1∘.function field2`

Pick	`i⊃field`
Indexing	`field[i]` `i⌷field`
Assignment	`field←array`
Indexed Assignment	`field[i]←⊂array`

Selective Assignment	`(i⊃field)←array`
	`(↑i↓field)←⊂array`
	etc.

Shape	`⍴field`
Compress/Replicate	`i/field`
First	`↑field`
Take	`i↑field`
Drop	`i↓field`

Notes:

1. Operations other than those defined here either attempt to bring the entire field 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. In some circumstances, Compress, Replicate, Expand, First, Take and Drop can return prototype or fill elements. If the field is not empty, the prototype is created using the attributes of the first item in the field. If the field is empty, the prototype is created using the field's signature.

Creating Java Objects

To create a Java object, first associate a name with a constructor method. Constructors are always named `<init>`. Surrogates must always be used when associating names with constructors because `<init>` is not a valid APL2 workspace object name.

Call the constructor method to instantiate an object. Although constructors are always declared to return void, when called by APL2, a constructor returns an integer. The integer can be used to associate names with instance members.

Sample Associations

The simplest way to learn about associating names with Java members is to look at some examples.

© Copyright IBM Corporation 1994, 2017

Here is another sample Java class:

```
//* Declare a class named Sample
public class Sample {
   //* Declare a static field
   //* Static fields exist as long as the class is loaded
   static int IncrementAmount = 1 ;
   //* Declare a static method
   //* Static method can be used as long as the class is loaded
   static int Increment(int Value) {
      return Value + IncrementAmount ;
      }
   //* Declare a field which exists for each instance
   int Value ;
   //* Declare a constructor which is used to create, or instantiate, an object
   static Sample(int Initial) {
      this.Value = Initial ;
      return ;
      }
   //* Declare a method
   static boolean IsEqual(int NewValue) {
      return Value == NewValue ;
      }
}
```

The following session log demonstrates associating names with members in the sample class.

```
      ⍝ Associate a name with the static field
      ('Sample' 'I') 14 ⎕NA 'IncrementAmount'
1
      ⍝ Set the static field
      IncrementAmount←5
      ⍝ Associate a name with the static method
      ('Sample' '(I)I') 14 ⎕NA 'Increment'
1
      ⍝ Call the static method
      Increment 34
39
      ⍝ Associate a name with the constructor
      ('Sample' '(I)V') 14 ⎕NA 'CONSTRUCT <init>'
1
      ⍝ Call the constructor
      Instance←CONSTRUCT 34
1
      ⍝ Associate a name with the instance field
      (Instance 'I') 14 ⎕NA 'Value'
1
      ⍝ Set the instance field
      Value←123
      ⍝ Associate a name with the instance method
      (Instance '(I)Z') 14 ⎕NA 'IsEqual'
1
      ⍝ Call the instance method
      IsEqual 45
0
```

Managing Java Objects

Objects instantiated from APL2 are not subject to Java garbage collection until they are deleted. Processor 14 includes a built-in function named `DeleteLocalRef` that is used to delete objects created through Processor 14 so their resources can be freed by the Java garbage collector. The following sample illustrates how to associate a name with the `DeleteLocalRef` function and delete an instance.

```
      ⍝ Associate a name with the Processor 14 DeleteLocalRef built-in function
```


© Copyright IBM Corporation 1994, 2017

```
      3 14 ⎕NA 'DeleteLocalRef'
1
      ⍝ Delete the instance
      DeleteLocalRef Instance
```

Once an instance has been deleted, Java can free the object's resources.

Note:

Not only constructor methods return objects. Many other methods return objects that consume resources and should eventually be deleted using the `DeleteLocalRef` function. For example, the `java/util/Hashtable` class includes a method named `get` that returns a `java/lang/Object` object. Proper use of the `get` method and the `DeleteLocalRef` function is demonstrated by the `DEMO_HASH` function in the DEMOJAVA workspace.

Managing Associations with Instance Methods and Fields

Names associated with instance methods and fields cannot be used after `)LOAD` and `)COPY` commands. This is because these names are associated with objects which no longer exist after the workplace is replaced.

Java class constructors return integers that are pointers to the internal representation of the constructed objects. These integers are used in associations with instance members. However, once the objects are deleted, the integers are no longer valid pointers and the associations cannot be reactivated. Attempts to use names associated with members of deleted instances return `VALENCE ERROR`.

Names associated with instance methods and fields should always be localized so they are not inadvertently saved in the workspace. In addition, the Processor 14 built-in `DeleteLocalRef` function should always be used to delete objects when they are no longer needed. This will free the resources used by the objects and ensure that associations can no longer be made with the objects' instance members.

Calling Back to APL2 from Java

Java methods called through Processor 14 can call back to APL2.

APL2 includes a Java class named Apl2interp. The Apl2interp class is used to manage and use APL2 interpreters from Java. Detailed information about using the Apl2interp class is available in Calling APL2 from Java.

The Processor 14 function `GetApl2interp` is used to retrieve the Apl2interp object owned by the APL2 interpreter. This object can be passed to a Java method. The method can then use the Apl2interp object to call back to APL2.

Here is a sample that demonstrates retrieving the Apl2interp object and passing it to a Java method:

```
      ⍝ Associate a name with the Processor 14 GetApl2interp built-in function
      3 14 ⎕NA 'GetApl2interp'
1
      ⍝ Associate a name with a Java method
      ('Class' '(Lcom/ibm/apl2/Apl2interp;)V') 14 ⎕NA 'Method'
1
      ⍝ Call the method and pass the Apl2interp object as a parameter
      Method GetApl2interp
```


© Copyright IBM Corporation 1994, 2017

Here is a sample method that uses an Apl2interp parameter:

```
public void Method(Apl2interp Apl2) {
   //* Create a workspace object from a Java integer array
   Apl2object Vector1 = new Apl2object(Apl2,new int[] {1,2,3,4,5,6});
   Apl2object Vector2 = new Apl2object(Apl2,new int[] {7,8,9,10,11,12});
   //* Call the function; it returns another workspace object
   Apl2object Result = Apl2.Execute(Vector1,"+",Vector2);
   //* Query the result's type and rank
   int Type = Result.type();
   int Rank = Result.rank();
   //* Free the workspace objects
   Vector1.Free();
   Vector2.Free();
   Result.Free();
   return;
}
```

Handling Errors and Interrupts

If a Java method encounters an error and throws an exception, Processor 14 displays the exception description in the interpreter window, signals a □ET error, and then clears the exception. The Apl2exception class can be used to throw an error with a specific event type.

Java methods called through Processor 14 are not automatically interrupted if the APL2 user signals an attention or interrupt. Long running methods can use the Apl2interp class's IsInterruptPending method to determine if an interrupt is pending. For example,

```
   int i;
   for(i=0;i<10000;i++)
      {
      if(Apl2interp.Apl2.IsInterruptPending())
         throw new Apl2exception(Apl2exception.TYPE_RESOURCE,
                                 Apl2exception.CODE_RESOURCE_INTERRUPT,
                                 "Interrupt Pending") ;
      }
```

The IsInterruptPending method is only useful for detecting interrupts during methods called through Processor 14. Slave interpreters, as described in Calling APL2 from Java do not support attentions and interrupts.

Handling Signals

When Java is first accessed from APL2, Processor 14 normally starts a Java virtual machine (JVM). Processor 14 uses the invocation option -Xrs to reduce the JVM's use of signals. When this option is in effect, the JVM handles asynchronous signals (SIGBREAK, SIGINT, SIGTERM, SIGHUP, and SIGQUIT) only when it is in direct control. When APL2 is in control these signals are handled by the signal handlers established by APL2.

However, if APL2 itself has been called by Java, Processor 14 uses the JVM already running, and Java's signal handlers will take precedence for those signals.

Java's signal handlers for synchronous signals, such as SIGSEGV, always take precedence over APL2's.

DEMOJAVA Workspace

The DEMOJAVA workspace contains functions that demonstrate how to call Java from APL2.

© Copyright IBM Corporation 1994, 2017

Processor 15 - Access Structured Storage

Processor 15 provides access to storage outside the APL2 workspace using variable names in the workspace. The APL2 data structures used by Processor 15 to format the storage can be based on a user-provided pattern, or implied from the structure of the data assigned to the variable.

Processor 15 variables that are associated with patterns are *strongly typed*. Processor 15 enforces the pattern established at name association whenever the variable is assigned. If the data assigned to the variable does not match the pattern, an error is signaled at assignment time.

Processor 15 can provide an advantage to APL2 programmers who need to more closely control the structure of their data. For example,

- Many non-APL routines in compiled libraries (DLLs on Windows) use C-style structures for arguments and results. Because these structures can be quite complex, building the appropriate APL2 array to match the structure's pattern can be prone to error. By using Processor 15, the data can be validated as correct when assigned rather than by Processor 11 when the routine is called.

  In addition, if the same structure is to be used in multiple routine calls, it can be more efficient to build it once and reuse it.

- Some APL2 applications also require data that conforms to specific structures and types. Specification of data with incorrect structure or type can lead to application errors at execution time, when it may be more difficult to identify the source of the error. Processor 15 variables can be used to force validation of the data at assignment.
- In addition to routines, some compiled libraries export variables. You can use Processor 15 to access these variables.

Processor 15 also provides the capability to establish debugging monitors on its variables.

The next sections discuss the following Processor 15 topics:

- Name Association Syntax for Processor 15
- Syntax Examples
- Supported Primitive Operations for Structured Storage
- Usage Guidelines
- Accessing Variable Addresses
- Monitoring Variable Changes
- Calling External Functions with Structured Storage
- Using Structured Storage to Access Exported Variables
- Using Structured Storage to Validate APL2 Data

Name Association Syntax for Processor 15

```
('pattern' [address | 'library']) 15 ⎕NA 'name'
pattern
```


© Copyright IBM Corporation 1994, 2017

A pattern describing how the storage should be processed.

Use an array pattern to enable type checking during specifications. The patterns for Processor 15 are a restricted version of those defined for Processor 11 names files in Array Patterns. In general, patterns which describe variable-length data are not allowed. Array Patterns for ATR, PFA, RTA, SIZEOF, and Associated Processor 15 describes the restrictions in more detail.

Use an empty pattern, `''`, to disable type checking.

`address`
The numeric address of previously allocated storage. If `address` is zero or if both `address` and `library` are omitted, Processor 15 will allocate the required storage, and it will be retained until the variable is expunged.

`address` is not supported if `pattern` is empty,

`library`
The name of a compiled library in which to find an exported variable.

`library` is not supported if `pattern` is empty,

The following rules are used to locate the library:

Unix Systems

If `library` contains a slash character, `library` is used directly, and no directories are searched. Otherwise, the LIBPATH or LD_LIBRARY_PATH environment variables can be used to specify a list of directories which are searched for the library.

Windows

If no file name extension is specified, the default library extension .dll is appended. However, the file name string can include a period to indicate that the module name has no extension. When no path is specified, Processor 15 searches for loaded libraries whose base name matches the base name of the library to be loaded. If the name matches, the load succeeds. Otherwise, Processor 15 searches for the file in the following sequence:

1. The directory from which the application loaded.
2. The current directory.
3. The system directory. Use the GetSystemDirectory function to get the path of this directory.
4. The 16-bit system directory. There is no function that obtains the path of this directory, but it is searched.
5. The Windows directory. Use the GetWindowsDirectory function to get the path of this directory.
6. The directories that are listed in the PATH environment

`name`
A name to be used within the APL workspace to refer to the storage.

© Copyright IBM Corporation 1994, 2017

If `library` is supplied, the name of an exported variable from the library. The name must match the exported variable exactly. If the exported variable name is not a valid APL2 name, then a surrogate must be used.

Processor 15 maintains attributes for each external variable. They can be accessed like this:

```
(attribute 'name') 15 ⎕NA 'attrvar'
```

`name`

Is the name of an external variable previously associated with Processor 15.

`attribute`

Is either `'ADDRESS'` or `'MONITOR'`.

Further information about these attributes can be found in Accessing Variable Addresses and Monitoring Variable Changes.

`attrvar`

Is a name to be used within the APL workspace to refer to the attribute.

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 displayed in the APL2 interpreter window.

Syntax Examples

A vector of 10 single byte characters in Processor 15-allocated storage:

```
      'C1 1 10' 15 ⎕NA 'VAR'
1
      VAR←'Abcdefghij'
```

A vector of 10 million characters at a pre-allocated address:

```
      ('C1 1 10000000' Address) 15 ⎕NA 'VAR'
1
      (10↑VAR)←'AbcdeFghij'
```

A null terminated vector of up to 9 two byte characters in Processor 15-allocated storage:

```
      'S2 1 10' 15 ⎕NA 'VAR'
1
      VAR←'Abc'
```

A 4 byte integer and a vector of 10 single byte characters:

```
      Pattern←'(G0 1 2)(I4 0)(C1 1 10)'
      Pattern 15 ⎕NA 'VAR'
1
      VAR←45 'AbcdeFghij'
```


© Copyright IBM Corporation 1994, 2017

A 4 byte integer and two vectors of 10 single byte characters at an address:

```
      Pattern←'(G0 1 3)(I4 0)(C1 1 10)(C1 1 10)'
      (Pattern Address) 15 ⎕NA 'VAR'
1
      VAR←45 'AbcdeFghij' '1234567890'
```

Supported Primitive Operations for Structured Storage

The following primitive operations are defined for variables associated with Processor 15:

Each	`function¨p15var`
	`var function¨p15var`
	`p15var function¨var`
	`p15var1 function¨p15var2`

Outer Product	`var∘.function p15var`
	`p15var∘.function var`
	`p15var1∘.function p15var2`

Pick	`i⊃p15var`
Indexing	`p15var[i]` `i⌷p15var`
Assignment	`p15var←array`
Indexed Assignment	`p15var[i]←⊂array`

Selective Assignment	`(i⊃p15var)←array`
	`(↑i↓p15var)←⊂array`
	etc.

Shape	`⍴p15var`
Compress/Replicate	`i/p15var`
First	`↑p15var`
Take	`i↑p15var`
Drop	`i↓p15var`

Notes:

1. Operations other than those defined here either attempt to bring the entire array 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. In some circumstances, Compress, Replicate, Expand, First, Take and Drop can return prototype or fill elements. The prototype is created using the attributes of the first item in the array.

Usage Guidelines

© Copyright IBM Corporation 1994, 2017

- Using Processor 15 to perform structure and type validation imposes a performance penalty. You may want to use it during development and test, but remove it before putting your application into production.
- By definition, Processor 15 can access storage that is larger than the workspace. For simple arrays with types corresponding exactly to the APL2 internal datatypes (`C1`, `C4`, `I4`, `E8` and `J16`) this definition holds true. For nested arrays and arrays using types requiring data conversion, the pattern validation and conversion processes will cause temporary copies to be made in the workspace, and WS FULL can result if there is insufficient space for the copies.
- Data in storage has an existence independent of the workspace. Assigning a value to a Processor 15 variable causes an immediate and permanent change to the storage. If the storage is pre-allocated, this is not affected by later expunging the variable.
- When a name is successfully associated with Processor 15, the association, name class, and valence remain in effect until the object is deleted from the workspace. They remain in effect even after using the commands `)SAVE`, `)OUT`, `)LOAD`, `)COPY`, or `)IN`. However, if the name association included an address, the address may no longer be valid. It is good practice to expunge names associated with addresses before saving the workspace or transfer file.

Accessing Variable Addresses

Use the `ADDRESS` attribute to retrieve the address of the storage associated with a variable.

When a name associated with an external variable's address attribute is referenced, the address of the storage associated with the external variable is returned. The address can then be passed as an argument when calling external routines.

This example demonstrates retrieving the address of an automatically allocated 4 byte integer:

```
      ⍝ Associate a variable named VAR
      'I4 0' 15 ⎕NA 'VAR'
1
      ⍝ Associate a name with VAR's address attribute
      ('ADDRESS' 'VAR') 15 ⎕NA 'ADDR'
1
      ⍝ Retrieve the address of the storage associated with VAR
      ADDR
4076800
```

Notes:

- Names associated with variables' address attributes are read-only.
- If the variable was associated with an empty pattern, the value of the address attribute changes each time the variable is assigned.

Monitoring Variable Changes

Use the `MONITOR` attribute to reference and specify a variable's monitor expression. When the variable's value is changed, the monitor expression is executed. The result is displayed in the session log.

Notes:

- Only one name may be associated with each variable's monitor expression.
- Each monitor expression may contain up to 8190 characters.

© Copyright IBM Corporation 1994, 2017

- A variable's monitor expression is executed after the variable's value has been changed.
- If an error occurs during execution of a monitor expression, the error is signaled at the point of the assignment.
- A variable's monitor expression may not change the variable's value.

This example demonstrates using the `DISPLAY` function in a monitor expression.

```
      ⍝ Associate a variable named VAR
      '' 15 ⎕NA 'VAR'
1
      ⍝ Associate a name with VAR's monitor expression
      ('MONITOR' 'VAR') 15 ⎕NA 'MON'
1
      ⍝ Associate DISPLAY
      3 11 ⎕NA 'DISPLAY'
1
      ⍝ Set VAR's monitor expression
      MON←'DISPLAY VAR'
      ⍝ Change VAR
      VAR←2 4 7
┌→─────┐
│2 4 7 │
└~─────┘
      ⍝ Change VAR again
      VAR←3 4⍴⍳12
┌→─────────┐
↓1  2  3  4 │
 5  6  7  8 │
 9 10 11 12 │
└~─────────┘
      ⍝ Turn off monitoring
      MON←''
      A←67
      A
67
```

Calling External Functions with Structured Storage

The following example demonstrates using a Processor 15 C-style structure with an external function that updates a parameter structure:

```
      ⍝ Define pattern for SYSTEM_INFO structure:
      WORD←'(I2 0)'
      LPVOID←DWORD_PTR←DWORD←'(I4 0)'
      PATTERN←'(G0 1 11)'
      PATTERN←PATTERN,WORD      ⍝ wProcessorArchitecture;
      PATTERN←PATTERN,WORD      ⍝ wReserved;
      PATTERN←PATTERN,DWORD     ⍝ dwPageSize;
      PATTERN←PATTERN,LPVOID    ⍝ lpMinimumApplicationAddress;
      PATTERN←PATTERN,LPVOID    ⍝ lpMaximumApplicationAddress;
      PATTERN←PATTERN,DWORD_PTR ⍝ dwActiveProcessorMask;
      PATTERN←PATTERN,DWORD     ⍝ dwNumberOfProcessors;
      PATTERN←PATTERN,DWORD     ⍝ dwProcessorType;
      PATTERN←PATTERN,DWORD     ⍝ dwAllocationGranularity;
      PATTERN←PATTERN,WORD      ⍝ wProcessorLevel;
      PATTERN←PATTERN,WORD      ⍝ wProcessorRevision;
      ⍝ Associate a name using the pattern
      PATTERN 15 ⎕NA 'SYSINFO'
1
      ⍝ Associate another name with the variable's address attribute
      ('ADDRESS' 'SYSINFO') 15 ⎕NA 'ADDR'
1
      ⍝ Initialize the structure
      SYSINFO←11⍴0
```


© Copyright IBM Corporation 1994, 2017

```
      ⍝ Associate a name with the Windows GetSystemInfo routine
      ⍝ Notice that the argument pattern is a 4 byte integer; this is for the structure
address.
      ⍝ Although the function will update the structure, the pattern does not include >.
      ':link.SYSTEM :lib.kernel32 :proc.GetSystemInfo :valence.0 1 0 :rarg.I4 0' 11 ⎕NA
'GetSystemInfo'
1
      ⍝ Call the routine with the address of the SYSINFO structure
      GetSystemInfo ADDR
      ⍝ Reference the structure
      SYSINFO
0 0 4096 65536 2147418111 3 2 586 65536 15 1034
```

Using Structured Storage to Access Exported Variables

On Windows, APL2 includes a library named aplwin.dll that exports a 4 byte Boolean variable named DisplayLogo that controls whether the session manager displays the product information dialog during invocation. The following example demonstrates how to access this variable:

```
      ⍝ Associate name
      ('I4 0' 'aplwin') 15 ⎕NA 'DisplayLogo'
1
      ⍝ Reference value
      DisplayLogo
0
```

Using Structured Storage to Validate APL2 Data

Suppose your application has a global array with this structure:

```
      A←4.5 'Abcde' (3 4 5 6)
```

Also suppose that somewhere in your application you have code to access the array's last element, like this:

```
      3 4⊃A
6
```

Further suppose that somewhere in your program you made a mistake and typed this:

```
      A←4.5 'Abcde' (3 4 5)
```

Run the program. At the place where it accesses the array's last element, APL2 will signal an Index Error.

```
      3 4⊃A
INDEX ERROR
      3 4⊃A
      ^  ^
```

Now, imagine trying to find the cause of the error. You know that the contents of the array are invalid, but how did they get that way? The incorrect assignment could be anywhere. How would you find it?

But, suppose you used structured storage and did this at the beginning of the program:

```
      '(3 G0 1 3)(1 E8 0)(5 C1 1 5)(4 I4 1 4)' 15 ⎕NA 'A'
1
```

Now, the incorrect assignment will cause an error:

```
      A←4.5 'Abcde' (3 4 5)
Expecting pattern:
(3  G0 1 3)
   (1  E8 0)
   (5  C1 1 5)
   (4  I4 1 4)
Current data pattern is:
(3 G0 1 3)(1 E8 0)(5 C1 1 5)(3 I4 1 3)
LENGTH ERROR
      A←4.5 'Abcde'(3 4 5)
      ^^
```


© Copyright IBM Corporation 1994, 2017

Since Processor 15 signals the error at the time of the incorrect assignment, you don't need to work to locate the mistake. You can just fix it and resume working.

© Copyright IBM Corporation 1994, 2017

Supplied External Routines

The following table lists the external routines supplied with APL2 that are accessed using Processor 11 and depend on the default names file, `aplnm011.nam`. For example:

```
      3 11 ⎕NA 'FILE'
1
      A←FILE 'data.bin'
```

In addition to the routines listed in the table,

- Three file utility routines are provided as built-in functions of Processor 10.
  See Accessing Operating System Files for more information on these routines.
- External namespaces are provided as alternatives for several of the supplied workspaces:
  FILE Namespace
  GRAPHPAK Namespace
  GUITOOLS Namespace (Windows only)
  GUIVARS Namespace (Windows only)
  MATHFNS Namespace
  SQL Namespace

APL2PIA	APL2 Programming Interface for APL2
ATR	Array to Record
ATS	Array to SCAR
BEEP	Sound a beep
CHECK_ERROR	Get System Error Text
CNS	Create Namespace
COPY	Copy Workspace Objects
CPUCOUNT	Get Number of Processors
CTN	Character to Numeric
CTUTF	Character to UTF
DATATYPE	Array Data Type
DIR	Get directory listing
DISPLAY	APL2 Array Structures
DISPLAYC	APL2 Array Structures
DISPLAYG	APL2 Array Structures
EDITOR_2	APL2 Editor 2
ERASE	Remove a file
EXP	Execute in Previous Namescope
FDELETE	File Delete

© Copyright IBM Corporation 1994, 2017

FILE	File Read or Write
FSTAT	File Status
GETENV	Get Environment Variable
GETLIB	Get Path for Library Number
GMTSTAMP	Get or Set Timestamp
HOST	Issue host system commands
IDIOMS	APL2 Idiom Library
LEXP	Lexical Parse
LIB	List Library Contents
LIBS	Get APL2 Library Definitions
LTM	Tcl List to APL2 Matrix
MD5	Encode Data to MD5
MKDIR	Create new directory
MTL	APL2 Matrix to Tcl List
OPTION	Query or Set Session Options
PCOPY	Protected COPY
PEACHP	Parallel Each using processors
PEACHT	Parallel Each using threads
PFA	Pattern from Array
PIPE	Redirect system command
QNS	Query Namescope
RENAME	Rename system file
REPLACEX	Replace Strings
RF	RowFind
RMDIR	Remove directory
ROUNDC	Commercial Rounding
RTA	Record to Array
SCANDIR	List Files in a Directory
SI	State Indicator
SIS	State Indicator with Statements
SIZEOF	Size of Array
STA	SCAR to Array
TCL	Tool Command Language Interface
TIME	Application Performance Analysis
UNZIP	Object decompression
UNZIPWS	Workspace decompression
UTFTC	UTF to Character

© Copyright IBM Corporation 1994, 2017

WSCOMP	Workspace compare
ZIP	Object compression
ZIPWS	Workspace compression

The following additional routines are supplied on Windows systems only:

APL2CFG	Configure APL2
APL2LM	APL2 Library Manager
COM	Component Object Model Interface
COMBROWSE	COM Browser
CTK	Character to Kanji
IDIOMSG	APL2 Idiom Library with Graphical User Interface
KTC	Kanji to Character
PRINTWSG	Print Workspace with GUI interface

© Copyright IBM Corporation 1994, 2017

APL2CFG - Configure APL2

```
APL2CFG
```

This function opens a properties window that shows the contents of the apl2.ini file.

Note: This function is available only on Windows systems.

Use `APL2CFG` to add, change, and delete the following settings:

- Default values for invocation options
- APL library definitions
- Session manager tool bar buttons
- SVP profile file names

For further information about using `APL2CFG` consult the online help.

For detailed information about the apl2.ini file, consult The `apl2.ini` Configuration File.

© Copyright IBM Corporation 1994, 2017

APL2LM - APL2 Library Manager

```
APL2LM
```

This function allows you to browse and compare the active workspace with saved workspaces and transfer files. When `APL2LM` is started, it displays the current contents of the active workspace; `APL2LM` does not display changes made to the active workspace with the object editor.

The APL2 Library Manager provides extensive online help. To display contextual help on any window, press **F1**. You can also use the choices on the **Help** menu to display the online help.

For more information, consult the online help or The APL2 Library Manager.

© Copyright IBM Corporation 1994, 2017

APL2PIA - APL2 Programming Interface for APL2

```
[result ←] [control] APL2PIA command [parameter ...]
```

This function allows you to start and stop an APL2 interpreter session, manage APL2 objects within that session, and execute APL2 expressions and functions in that session.

For detailed syntax and usage information on this interface, see Calling APL2 from APL2.

`control`
: A Boolean indicator of what the function's behavior should be for error conditions.
If omitted or `0`, errors will be reported as regular APL2 errors and execution will be suspended on error.
If `1`, errors will be reported as part of the result.

`command`
: A character vector containing one of the following commands:

`'START'`	Start an APL2 session
`'STOP'`	Stop an APL2 session
`'PUT'`	Create an object
`'GET'`	Reference an object
`'FREE'`	Expunge an object
`'EXECUTE'`	Execute expression or function
`'EXTOKEN'`	Execute using array tokens

`parameter`
: Additional parameters as defined for the specified command:

`'START'`	`sessionid ← 'START' [ options ]`
`'STOP'`	`'STOP' sessionid`
`'PUT'`	`arrayid ← 'PUT' sessionid array`
`'GET'`	`array ← 'GET' sessionid arrayid`
`'FREE'`	`'FREE' sessionid arrayid`
`'EXECUTE'`	`array ← 'EXECUTE' sessionid [array] {function | expression} [array]`
`'EXTOKEN'`	`arrayid ← 'EXTOKEN' sessionid [arrayid] {functionid | expressionid} [arrayid]`

`result`
: If `control` is omitted or `0`, a result as defined for the specified command.
If `control` is `1`, three-element nested array:

`[1]` APL2 error codes as defined for `⎕ET`.

`[2]` Boolean indicator of whether a result was created.

`[3]` If the first element is `0 0` and the second element is `1`, the result from the command.
If the first element is not `0 0` and the second element is `1`, a character matrix containing the error message and execution stack at the time of the error.
Otherwise, a null character vector.

© Copyright IBM Corporation 1994, 2017

ATR - Array to Record

```
record ← [pattern] ATR array
```

This function converts an APL array into a character vector containing structured data, based on a pattern that describes the desired format. It can be used to map APL arrays into records or structures expected by other languages, or expected by services called through AP 145.

`array`
: An APL2 array whose structure and data types should be compatible with the pattern. The internal APL storage form need not match the pattern exactly, so long as it is possible to convert it to the pattern format using APL system tolerance.

`pattern`
: A character vector containing a formalized description of the fields within the record. Array patterns are defined in Array Patterns. The pattern may not contain a `*` unless it can be resolved to an integer based on other information in the item description. The pattern may not contain the > mark.

  If `pattern` is omitted, APL2 Common Data Representation (CDR) format is used to create the result.

`record`
: A character vector produced by converting the data in the array according to the pattern. `DOMAIN ERROR` is signaled if an impossible conversion is implied by the pattern. Incompatible data structures result in `LENGTH ERROR` or `RANK ERROR`.

Note: External function `RTA` is the inverse of `ATR`.

© Copyright IBM Corporation 1994, 2017

ATS - Array to SCAR

```
data ← ['A'|'B'] ATS array
```

This function converts an APL2 array into a SCAR object.

`array`
: Any arbitrary APL2 array.

`'A'`
: Indicates that the SCAR object should be encoded into a printable ASCII representation.

`'B'`
: Indicates that the object should returned in its binary form.
If no left argument is supplied, binary form is used.

`data`
: A character vector containing the SCAR object.

The SCAR ("Self-Contained Array") format is a data interchange format defined by Insight Systems, Inc. to allow non-like APL systems to exchange data directly without the overhead of the numeric-to-character conversions required by transfer form. For more information on the SCAR format, visit `http://www.insight.dk/scardesign`.

Once created, the character vector containing the SCAR object can be sent across a TCP/IP network, written to a file, or stored in a database. It can be received and processed by APL2 or another APL system which supports the SCAR format.

In APL2, the `ATS` and `STA` external functions are used to create and interpret SCAR objects.

In Dyalog APL/W, the `SQAApl2Scar` and `SQLScar2Apl` external functions are used to create and interpret SCAR objects. These functions use file CNDYA30.DLL which is part of SQAPL. The DLL is loaded and initialized by function `SQAInit` in workspace SQAPL.DWS.

In APL+Win, APL functions `TOSCAR` and `FMSCAR` in workspace SCAR are used to create and interpret SCAR objects.

© Copyright IBM Corporation 1994, 2017

BEEP - Sound a Beep

```
rc ← BEEP frequency duration
```

This function sounds a beep.

`frequency`
: Cycles per second, in the range from 37 to 32767.
Not all operating systems support a user-specified frequency. If not supported, this parameter will be ignored.

`duration`
: The length of the sound in milliseconds.
Not all operating systems support a user-specified duration. If not supported, this parameter will be ignored.

`rc`
: 0 if successful, or the system return code.

© Copyright IBM Corporation 1994, 2017

CHECK_ERROR - Get System Error Text

```
msg←CHECK_ERROR errno
```

This function converts the system error number specified by the right argument to a text error message.

`errno`
: A numeric system error code. The error number can be a number returned by an operating system command, or a return code from an APL2 external function or auxiliary processor that is not defined for that function or processor (for example, non-zero return codes from the `FILE`, `∆F`, `∆FM` and `∆FV` external functions, or positive return codes from AP 210).

`msg`
: The message text associated with the specified error number. The C subroutine `strerror` is used to obtain the error message text.

© Copyright IBM Corporation 1994, 2017

CNS - Create Namespace

```
result ← [namelist] CNS wsname [outdir]
```

This function creates a namespace from a saved workspace.

`wsname`
: A character vector containing the name of the saved workspace to be converted.
The workspace name may be preceded by a library number. Explicit file specification of workspace names (with complete path, enclosed in quotes) is also supported. For example:

```
CNS 'MYAPPL'

CNS '2 GRAPHPAK'

CNS '''D:\WSDIR\MYAPPL.APL'''
```

`outdir`
: The directory path into which the namespace file should be written. If not specified, the current working directory will be used. For example:

```
CNS '2 GRAPHPAK' 'D:\WSDIR'
```

`namelist`
: A list of names of APL objects in the resulting namespace that will be accessible by using `⎕NA`. If a name list is not specified, all APL objects in the resulting namespace are accessible. `namelist` can be a simple character scalar or vector representing one name, or a matrix or vector of vectors representing a list of names. All names in the list must be valid APL2 names. For example:

```
('PLOT' 'OPENGP' 'CLOSEGP') CNS '2 GRAPHPAK'
```

`result`
: The name of the file (including path) that contains the resulting namespace.

Note: Workspaces to be converted to namespaces must have been saved using APL2 Version 2.0 or later.

© Copyright IBM Corporation 1994, 2017

COM - Component Object Model Interface

```
[result ←] [control] COM 'COMMAND' [arguments]
```

The COM external function provides access to Microsoft Component Object Model (COM) objects. The COM function supports several commands:

QUERY	Query information about COM classes and objects
CREATE	Create instances of COM classes
CONNECT	Connect to running COM programs
METHOD	Invoke COM object methods
PROPERTY	Specify and reference COM object properties
HANDLERS	Specify and reference COM object event handlers
WAIT	Wait for COM events
RELEASE	Release references to COM objects
CONFIG	Set configuration options

This section includes the following topics:

- COM Overview
- COM Error Handling
- Querying COM Classes and Objects
- Creating COM Objects
- Connecting to Running COM Objects
- Referencing and Specifying COM Properties
- Invoking COM Methods
- Period Delimited Member Names
- Indexed Properties
- Positional, Named, and Omitted Method Arguments
- Output Arguments
- By Reference Arguments
- Calling the Evaluate method
- Enumerations
- Managing COM Objects
- Handling COM Events
- Configuring the COM Interface
- Data Conversion Between COM and APL2
- COM Microsoft Agent Example
- COM Excel Example
- COM Internet Explorer Example
- COM Word Example
- COM Office Visual Basic Example

Note:

© Copyright IBM Corporation 1994, 2017

The COM function is only available on Windows.

COM Overview

The Microsoft Component Object Model (COM) is a system for creating software components that can interact. COM is the foundation technology for Microsoft's OLE (compound documents), ActiveX (Internet-enabled components), and others.

Many applications, such as Microsoft's Office suite of products and elements of Windows, are distributed as COM components. Through the COM external function, these applications can be used from APL2.

COM is an object-oriented system. This means that COM components are distributed as *classes* and operate as *objects*. An application, such as APL2, creates an instance of a COM class to create a COM object. Once an object is created, the application can use *methods* and *properties* within the object. When the application has finished using the object, it releases the object to free its resources.

This document describes how to access COM components from APL2. It does not attempt to describe the capabilities of any individual components. To learn how to use an individual component, consult the component's documentation. Information about many of Microsoft's components such as Excel can be found at the Microsoft Developers Network (MSDN), http://msdn.microsoft.com.

COM Error Handling

By default, if the COM function encounters an error calling a Component Object Model service, it signals a SYSTEM LIMIT and sets the first row of ⎕EM to the service's error message. Use the COM function's left argument to control the COM function's error handling. An omitted or 0 left argument produces the default behavior. A left argument of 1 causes the COM function to always return a three element result:

```
[1] - ⎕ET
[2] - Error message
[3] - The result, or '' if none
```

More information may be printed to standard error in the interpreter's console window.

Example:

```
      (ET EM RESULT)←1 COM 'CREATE' 'Excel.Application'
```

Note:

Unless otherwise stated, the examples in this documentation use the default error handling.

Querying COM Classes and Objects

The QUERY command supports several types of requests:

CLASSES	Query list of classes installed on the local machine
MEMBERS	Query list of members available in an object
EVENTS	Query list of events supported by an object

© Copyright IBM Corporation 1994, 2017

ENUMERATIONS Query list of enumerations available in an object
CONSTANTS Query list of constants and their values in an enumeration

Note:

Object information may not be available for all classes.

```
CLASSES←COM 'QUERY' 'CLASSES'
```

The CLASSES request returns the list of classes installed on the local machine. The result is a seven column matrix:

[;1] - Version independent program identifiers.
[;2] - Version dependent program identifiers.
[;3] - User-type names
[;4] - Globally unique class identifiers (CLSID)
[;5] - Booleans indicating classes that are programmable through the IDispatch interface
[;6] - Booleans indicating classes are ActiveX controls
[;7] - Booleans indicating classes that are insertable in container applications

You can use either type of program identifier or the CLSID in the CREATE and CONNECT commands.

Note:

> The 'QUERY' 'CLASSES' request returns all the classes that are listed in the Windows registry and have program identifiers. However, not all COM classes can be used from APL2; only programmable classes can be used from APL2. In addition, the installation programs for COM classes do not always update the registry to indicate whether the classes are programmable. To determine whether a class is actually programmable, use the CREATE command which will fail if the class is not programmable.

The following query requests can be used to query information about objects returned by the CREATE and CONNECT commands and by object methods.

```
MEMBERS←COM 'QUERY' 'MEMBERS' OBJECT
```

The MEMBERS request returns the list of methods and properties supported by the object. The result is a five column matrix containing information about the members:

© Copyright IBM Corporation 1994, 2017

[;1] - Member name

[;2] - Member type METHOD, PROPERTY, or PROPERTY() for indexed properties

[;3] - Prototype For more information, see Prototypes.

[;4] - Description A text description of the member (may be null)

[;5] - Help file name The name of the member's help file (may be null)

```
EVENTS←COM 'QUERY' 'EVENTS' OBJECT
```

The EVENTS request returns the list of events supported by the object. The result is a five column matrix containing information about the events:

[;1] - Event name

[;2] - Event type METHOD, PROPERTY, or PROPERTY(). Usually METHOD.

[;3] - Prototype For more information, see Prototypes.

[;4] - Description A text description of the event (may be null)

[;5] - Help file name The name of the event's help file (may be null)

```
ENUMS←COM 'QUERY' 'ENUMERATIONS' OBJECT
```

The ENUMERATIONS request returns a list of the object's enumeration names. The result is a vector of character vectors. For more information, see Enumerations.

```
CONSTANTS←COM 'QUERY' 'CONSTANTS' OBJECT 'enumeration name'
```

The CONSTANTS request returns the list of constants in an enumeration. The result is a two column matrix:

[;1] - Constant name

[;2] - Constant value

Prototypes

© Copyright IBM Corporation 1994, 2017

The 'QUERY' 'MEMBERS' and 'QUERY' 'EVENTS' commands return prototypes which describe the types and names of a member's and event's arguments and results. Prototypes use the following syntax:

```
TYPE MemberName(ARGUMENTS)
```

Where: ARGUMENTS is a comma delimited list of arguments of the form:

```
TYPE ArgumentName[options]
```

Where options is a comma-delimited list of any of the following words:

in	The argument is used for input to the member
out	The argument is used for output of the member
lcid	The argument is the locale identifier of the application
retval	The argument is the return value of the member
defaultvalue	The argument has a default value
optional	The argument is optional

TYPE can be any of the following words:

BOOL	2 byte integer
BSTR	Pointer to Unicode string
CY	Currency amount
DATE	Date
DISPATCH	Pointer to IDispatch interface
ERROR	Error code
HRESULT	API result code
I1	1 byte signed integer
I2	2 byte signed integer
I4	4 byte signed integer
I8	8 byte signed integer
INT	4 byte signed integer
LPSTR	Pointer to Unicode string
LPWSTR	Pointer to Unicode string
R4	4 byte floating point value
R8	8 byte floating point value
UI1	1 byte unsigned integer
UI2	2 byte unsigned integer
UI4	4 byte unsigned integer
UI8	8 byte unsigned integer
UINT	4 byte unsigned integer
UNKNOWN	Pointer to IUnknown interface

© Copyright IBM Corporation 1994, 2017

VARIANT	Variant
VOID	C-style void
Unrecognized	While generating the prototype, a type was encountered that is defined by the Component Object Model but is unrecognized by the COM external function

Any other type name is a user-defined type.

Any type can have an asterisk suffix indicating the value is the address of a value of the indicated type.

Any type can be surrounded by SAFEARRAY() indicating the value is a COM Safe Array containing elements of the indicated type.

Any type can be suffixed by brackets, such as [lb...ub] indicating the value is a C-style array of elements of the indicated type. The array's lower and upper bounds are indicated by the values (lb) and (ub).

Creating COM Objects

Use the CREATE command to create a COM object:

```
⍝ Create an object
OBJECT←COM 'CREATE' 'Identifier'
```

Identifier is a character vector containing a program or class identifier.

The CREATE command creates an instance of the specified class and returns an integer handle that can be used to query the object's members and enumerations, specify and reference the object's properties, invoke the object's methods, query the object's events, specify the object's event handlers, and wait for events.

Example:

```
EXCEL←COM 'CREATE' 'Excel.Application'
```

Note:

The handle returned by the CREATE command is a pointer to the object's IDispatch interface. If you have obtained an object's IDispatch interface by another means, you may use it as if it were obtained using the CREATE command.

Connecting to Running COM Objects

Use the CONNECT command to connect to COM object classes that are already running.

```
⍝ Connect to a running object
OBJECT←COM 'CONNECT' 'Identifier'
```


© Copyright IBM Corporation 1994, 2017

`Identifier` is a character vector containing a program or class identifier.

The `CONNECT` command connects to an object that has registered itself and is in the system's running objects table. The command returns an integer handle that can be used like handles returned by the `CREATE` command to query the object's members and enumerations, specify and reference the object's properties, invoke the object's methods, query the object's events, specify the object's event handlers, and wait for events.

If the object is not running, the COM function will fail. It is good practice to specify the function should return errors so the application can detect this condition.

Example:

```
      (ET EM EXCEL)←1 COM 'CONNECT' 'Excel.Application'
```

Referencing and Specifying COM Properties

Use the `PROPERTY` command to reference and specify COM object properties:

```
⍝ Reference a property
RESULT←COM 'PROPERTY' OBJECT 'PropertyName' [indices]

⍝ Specify a property
COM 'PROPERTY' OBJECT 'PropertyName' [indices] newvalue
```

`PropertyName` is a character vector containing the case-sensitive name of the property to be specified or referenced.

If `newvalue` is not supplied, the property's value is referenced and returned as `COM`'s explicit result.

If `newvalue` is supplied, the property's value is specified. No result is returned.

Example:

```
      ⍝ Reference Excel's Visible property
      COM 'PROPERTY' EXCEL 'Visible'
0
      ⍝ Set Excel's Visible property
      COM 'PROPERTY' EXCEL 'Visible' 1
```

Note:

Some properties are read-only and cannot be specified.

See Period Delimited Member Names for information about using multiple property names in PropertyName.

See Indexed Properties for information about the use of indices.

© Copyright IBM Corporation 1994, 2017

Invoking COM Methods

Use the METHOD command to invoke COM object methods:

```
⍝ Invoke a method
RESULT←COM 'METHOD' OBJECT 'MethodName' [indices] [args]
```

MethodName is a character vector containing the case-sensitive name of the method to be invoked.

args is an optional variable length list of arguments for the method.

COM returns the method's result or '' if none.

Example:

```
Name←COM 'METHOD' EXCEL 'InputBox' 'Enter your name' 'APL2 Sample'
```

See Period Delimited Member Names for information about using multiple property names in MethodName.

See Indexed Properties for information about the use of indices.

See Positional, Named, and Omitted Method Arguments for information about naming and omitting arguments.

Period Delimited Member Names

Periods can be used to delimit multiple member names. For example:

```
DATA←COM 'PROPERTY' EXCEL 'ActiveSheet.UsedRange.Value'
```

In the example, COM retrieves the ActiveSheet property of the Excel object to retrieve a subobject. COM then retrieves the UsedRange property of the subobject to get another subobject. COM finally references the Value property of the last subobject. This corresponds to the following sequence of separate commands:

```
SUB1←COM 'PROPERTY' EXCEL 'ActiveSheet'
SUB2←COM 'PROPERTY' SUB1 'UsedRange'
COM 'RELEASE' SUB1
DATA←COM 'PROPERTY' SUB2 'Value'
COM 'RELEASE' SUB2
```

Any number of period delimited member names can be used before the final method or property name.

Indexed Properties

Some COM objects support properties that can only be accessed using one or more indices. These properties typically contain an array of values; the indices select which values should be returned or set when the property

© Copyright IBM Corporation 1994, 2017

is referenced or specified. For example, Excel spreadhseets support several indexed properties which are used to access specific cells or ranges of cells in the spreadsheets.

Member names can include paired parentheses to indicate that a property is an indexed property. For example:

```
      COM 'PROPERTY' EXCEL 'Cells.Item().Value' (1 3)
David
      COM 'PROPERTY' EXCEL 'Range().Value' (⊂'A1:B2')
13 12
16 15
```

For each pair of parentheses indicating an indexed property, an array of indices must be supplied. To illustrate, the following example uses the indices `1 3` for `PropA` and `(⊂'A1:B2')` for `PropB`.

```
      COM 'PROPERTY' OBJECT 'PropA().PropB().Value' (1 3) (⊂'A1:B2')
13 12
16 15
```

Notice the use of enclose (⊂) to ensure that the character vector is passed as a single index rather than a list of scalar indices.

Note:

The paired parentheses syntax can also be used to call methods in period delimited member names. For example, the following expressions show using a Word Styles object's Item method with a constant as an argument.

```
      WORD←COM 'CREATE' 'Word.Application'
      DOCUMENT←COM 'METHOD' WORD 'Documents.Open' 'c:\sample.doc'
      ⍝ The WdBuildinStyle enumerator's wdStyleNormal constant is ¯1
      COM 'PROPERTY' DOCUMENT 'Styles.Item().Font.Name' ¯1
Times New Roman
```

Positional, Named, and Omitted Method Arguments

The default use of the `METHOD` command requires positional arguments; each of the method's arguments must be listed in the order specified in the method's prototype. However, it is sometimes inconvenient, or even incorrect, to supply all the arguments in methods' prototypes. Use brackets, `[]`, after the method name and name-value pairs to name individual arguments:

```
      COM 'METHOD' OBJECT 'Method[]' ('ArgNameA' ValueA) ('ArgNameB' ValueB)
```

If the method named is followed by `[]`, subsequent data must be a vector of 2 element nested arrays. The first element of each array is an argument name; the second element is the argument's value.

For example, consider the Add method that is supported by Excel WorkSheets objects:

```
DISPATCH Add(VARIANT Before[in, optional],VARIANT After[in, optional],
             VARIANT Count[in, optional],VARIANT Type[in, optional])
```

To add a sheet after a particular sheet, you must supply the `After` argument, but omit the `Before` argument. The following expressions demonstrate how to use a named argument to add a new sheet after an existing sheet:

© Copyright IBM Corporation 1994, 2017

```
SHEET2←COM 'PROPERTY' EXCEL 'Worksheets.Item()' 2
AFTER←COM 'METHOD' EXCEL 'Worksheets.Add[]' ('After' SHEET2)
```

It is also possible to use both positional and named arguments. Empty argument names indicate the argument is positional. For example, the following expression adds 3 sheets before an existing sheet:

```
BEFORE←COM 'METHOD' EXCEL 'Worksheets.Add[]' ('Before' SHEET2) ('Count' 3)
```

Because the `Before` argument is the first argument in the method's prototype, it can be passed positionally by passing a null vector as the name:

```
BEFORE←COM 'METHOD' EXCEL 'Worksheets.Add[]' ('' SHEET2) ('Count' 3)
```

Notes:

All positional arguments must be provided before any named arguments.

All positional arguments must be provided in the order specified by the method's prototype.

Named arguments can be provided in any order.

Output Arguments

Some methods support arguments that produce output; the methods update arguments with new values. These arguments are identified in the members' prototypes by the **out** option. To retrieve the value of updated arguments, use the `METHOD` command's positional or named argument syntax:

```
(Result Arg1 ValueA)←COM 'METHOD' OBJECT 'Method[]' ('=' Arg1) ('=NameA' ValueA)
```

For positional arguments, use `'='` rather than `''`. For named arguments, use = before the argument names.

The first element of the `COM` function result is the method's result. Subsequent elements are updated argument values.

By Reference Arguments

Some methods require that arguments are passed by reference rather than by value. This means that the address of the argument must be passed rather than the argument itself. These arguments are identified in the members' prototypes by an asterisk. To cause the COM external function to pass an argument by reference, use the `METHOD` command's positional or named argument syntax:

```
(Result Arg1 ValueA)←COM 'METHOD' OBJECT 'Method[]' ('*' Arg1) ('*NameA' ValueA)
```

For positional arguments, use `'*'` rather than `''`. For named arguments, use `*` before the argument names. To pass output arguments by reference, prefix the argument name with both `*` and = in any order.

Calling the Evaluate Method

© Copyright IBM Corporation 1994, 2017

Some objects support an Evaluate method. For example, Excel supports an Evaluate method that converts a name to an object or a value.

The documentation for many objects uses a Visual Basic notation containing brackets for calling the Evaluate method. In Visual Basic, using square brackets (for example, "[A1:C5]") is identical to calling the Evaluate method with a string argument.

To make it easier to use this documentation to write APL2 applications, the `COM` external function supports a similar notation using brackets for calling Evaluate.

To use the bracket notation in APL2, replace the member name with the method's argument surrounded by square brackets.

For example, the following expressions call the Evaluate method using the normal syntax:

```
      WORKBOOK←COM 'METHOD' EXCEL 'Workbooks.Open' 'd:\test.xls'
      WORKSHEET←COM 'PROPERTY' WORKBOOK 'WORKSHEETS()' (⊂'SHEET1')
      CELL←COM 'METHOD' WORKSHEET 'Evaluate' 'A1'
      COM 'PROPERTY' CELL 'Font.Bold' 1
```

The following single expression performs the same operation:

```
      COM 'PROPERTY' WORKBOOK 'WORKSHEETS().[A1].Font.Bold' (⊂'SHEET1') 1
```

The advantage of using square brackets is that the code is shorter. The advantage of using the Evaluate method explicitly is that the argument is a character vector, so you can either construct the character vector in your code or use a variable.

Enumerations

Some COM objects include lists of predefined constants. Each list is called an enumeration. Each enumeration contains a list of named constant values.

Enumeration constants are typically used as indices for indexed properties. For example, Excel has an indexed property named International. The elements of the XlApplicationInternational enumeration are used as indices with the International indexed property. One of the XlApplicationInternational enumeration's constants is named xlGeneralFormatName and has the value 26. The following log demonstrates using these names and values:

```
      ⍝ Create an instance of Excel
      EXCEL←COM 'CREATE' 'Excel.Application'
      ⍝ Query the enumerations supported by the Excel object
      ENUMS←COM 'QUERY' 'ENUMERATIONS' EXCEL
      ⍝ Check the name of Excel's second enumeration
      2⊃ENUMS
XlApplicationInternational
      ⍝ Query the constants in the enumeration
      CONS←COM 'QUERY' 'CONSTANTS' EXCEL (2⊃ENUMS)
      ⍝ Check the name and value of the 20th element in the enumeration
      CONS[20;]
 xlGeneralFormatName 26
      ⍝ Use the constant with the International indexed property
      COM 'PROPERTY' EXCEL 'International()' 26
General
      ⍝ Release Excel
```


© Copyright IBM Corporation 1994, 2017

```
COM 'RELEASE' EXCEL
```

Managing COM Objects

Use the RELEASE command to release a COM object:

```
⍝ Release an object
COM 'RELEASE' OBJECT
```

COM objects are not automatically released when the COM external function is expunged. In fact, some COM objects such as Excel may continue running even after APL2 is shut down if you do not first close and release them appropriately. Use the RELEASE command to release objects after your application no longer needs them.

Note:

Not only the CREATE command returns objects. COM properties and methods can return objects that should be released after use. For example, the Open method in the following command returns a Workbook object that should be released after the file is closed.

Example:

```
WORKBOOK←COM 'METHOD' EXCEL 'Workbooks.Open' 'd:\test.xls'
COM 'RELEASE' WORKBOOK
COM 'RELEASE' EXCEL
```

Handling COM Events

Some COM objects signal events to notify applications that something has happened. For example, a spreadsheet may signal an event when the focus has moved to a new cell.

Use the 'HANDLERS' command to indicate that your application should be notified when objects signal events.

```
⍝ Set the event handlers for an object
COM 'HANDLERS' OBJECT HANDLERS

⍝ Query an object's event handlers
HANDLERS←COM 'HANDLERS' OBJECT
```

The handlers array is a two column matrix:

[;1] - Character vector - Event names as returned by the 'QUERY' 'EVENTS' command

[;2] - Arbitrary array - Event handler returned when the event is signaled

© Copyright IBM Corporation 1994, 2017

Event handlers are arbitrary arrays. The structure of your application dictates what type of array you will use. For example, you might use character vectors as event handlers and execute them when they are signaled. Or, you might choose to use labels and branch to them when the events are signaled.

You do not need to specify event handlers for all an object's events. Events for which no handler is specified are ignored.

Assign a zero element handler array to remove all event handlers.

When an event is signaled, if an event handler has been specified for it, the `COM` function queues the events until the application waits for them.

Use the `'WAIT'` command to retrieve queued events or wait for new events.

```
⍝ Wait for events
EVENTS←COM 'WAIT' TIMEOUT
```

The `TIMEOUT` argument is the number of seconds to wait for an event. `WAIT` returns `''` if no event has occurred in the specified number of seconds. If an event has occurred, `WAIT` returns a 6 or 7 element vector that describes the event:

[1] Scalar integer - Handle of object that signaled the event

[2] Character vector - Event name

[3] Arbitrary array - Event handler associated with the event using the `'HANDLERS'` command.

[4] Two integers - Position of the mouse pointer at the time the event occurred

[5] Scalar integer - Time the event occurred (in milliseconds since system start)

[6] 2 column array - Named parameters. Columns are:

[;1] Character vector - Parameter name

[;2] Array - Parameter value

[7] Vector of arrays - Positional arguments not supplied as named arguments

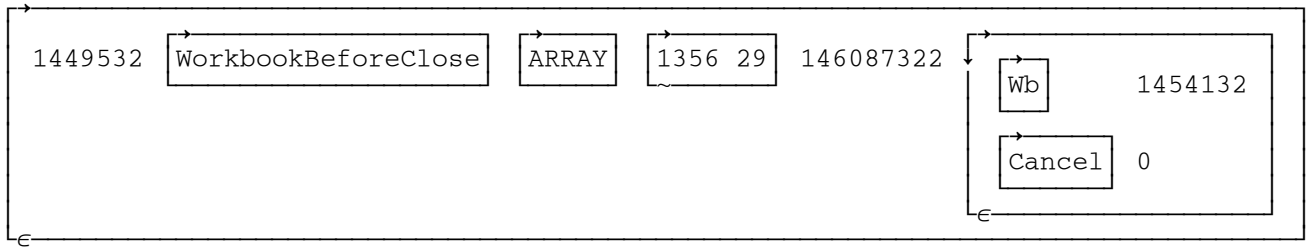
Notes:

- The object handle returned for each event must be released.
- Scalar parameters that are IDispatch or IUnknown interface pointers must be released.
- Parameters that contain arrays of IDispatch or IUnknown interface pointers are invalid and cannot be used or released.
- All event handlers must be removed by assigning a zero element handler array to completely release an object.

The following log demonstrates using COM events:

© Copyright IBM Corporation 1994, 2017

```
      ⍝ Start Excel
      EXCEL←COM 'CREATE' 'Excel.Application'
      ⍝ Add a workbook
      WORKBOOK←COM 'METHOD' EXCEL 'Workbooks.Add'
      ⍝ Notice the Excel and workbook handles may later be returned in events
      EXCEL WORKBOOK
1443852 1450804
      ⍝ Query Excel's events
      EVENTS←COM 'QUERY' 'EVENTS' EXCEL
      ⍝ The seventeenth event is a close event
      ⊃EVENTS[17;]
WorkbookBeforeClose
METHOD
VOID WorkbookBeforeClose(Workbook* Wb[in], BOOL* Cancel[in])
C:\Program Files\Microsoft Office\Office10\VBAXL10.CHM
      ⍝ Set a handler for the close event
      COM 'HANDLERS' EXCEL (1 2⍴'WorkbookBeforeClose' 'ARRAY')
      ⍝ Make Excel visible
      COM 'PROPERTY' EXCEL 'Visible' 1
      ⍝ Wait for an event
      RESULT←COM 'WAIT' ¯1
      DISPLAY RESULT
```




```
      ⍝ Parse the event result
      (OBJECT EVENT HANDLER CURSORPOS TIME NAMED)←6↑RESULT
      POSITIONAL←6↓RESULT
      ⍝ Release the handles returned in the event
      COM 'RELEASE' OBJECT
      COM 'RELEASE' NAMED[1;2]
      ⍝ Indicate COM should stop queuing events
      COM 'HANDLERS' EXCEL ''
      ⍝ Make Excel invisible
      COM 'PROPERTY' EXCEL 'Visible' 1
      ⍝ Prevent Excel from prompting when the file is closed
      COM 'PROPERTY' EXCEL 'DisplayAlerts' 0
      ⍝ Invoke a method to close the file
      COM 'METHOD' WORKBOOK 'Close'
¯1
      ⍝ Release the workbook and Excel
      COM 'RELEASE' WORKBOOK
      COM 'RELEASE' EXCEL
```

Configuring the COM Interface

COM objects can support multiple national languages and use the application's current locale to determine how to interpret member names and arguments.

Use the `CONFIG` command to set the current locale used by the `COM` external function:

```
⍝ Retrieve the current locale information
(prim sub sort)←COM 'CONFIG' 'LOCALE'

⍝ Set the locale information
```


© Copyright IBM Corporation 1994, 2017

```
COM 'CONFIG' 'LOCALE' prim sub sort
```

Where `prim`, `sub`, and `sort` are scalar integers. `prim` is a primary language identifier, `sub` is a sublanguage identifier, and `sort` is a sort identifier. Valid values for these identifiers can be found at the Microsoft Developers Network, http://msdn.microsoft.com.

The `COM` function initializes the current locale with the user's default locale which corresponds to a setting of `0 1 0`.

Notes:

If you intend to provide your application to users who may have their systems configured to use different locales than your development systems, during application initialization you should configure the `COM` function to use your development systems' locale to ensure your application will run on your users' machines.

COM objects that do not support multiple national languages may ignore the locale setting.

Use of different languages requires that both the operating system's and the COM object's support for those languages be installed.

Data Conversion Between COM and APL2

COM components and APL2 use different internal data types. When moving data between the environments, the `COM` external function performs automatic data conversion.

Some COM objects support user-defined data types much like C structures. The `COM` external function does not support user-defined data types.

The `COM` external function uses the Visual Basic to APL2 interface to perform data conversion. See Data Conversion Between Visual Basic and APL2 for further information.

COM Microsoft Agent Example

The following function demonstrates using COM to control Microsoft Agent:

```
∇DEMO_MERLIN;AGENT;COM;CURSORPOS;EM;ET;EVENT;HANDLER;MERLIN;NAMED;OBJECT;POSITIONAL;RC;REQ
UEST;RESULT;TIME
[1]   ⍝
[2]   ⍝ Use the Microsoft Agent Control's Merlin character to demonstrate using COM events.
[3]   ⍝
[4]
[5]   ⍝ Associate the COM function
[6]    ⎕ES(1≠3 11 ⎕NA 'COM')/'COM unavailable'
[7]
[8]   ⍝ Create a Microsoft Agent control
[9]    AGENT←(⎕IO+2)⊃1 COM 'CREATE' 'Agent.Control'
[10]
[11]  ⍝ Tell the agent to connect
[12]   COM 'PROPERTY' AGENT 'Connected' 1
```


© Copyright IBM Corporation 1994, 2017

```
[13]
[14] ⍝ Load the merlin character
[15]  REQUEST←COM 'METHOD' AGENT 'Characters.Load' 'Merlin' 'Merlin.acs'
[16]  COM 'RELEASE' REQUEST
[17]
[18] ⍝ Tell the agent to use merlin
[19]  MERLIN←COM 'METHOD' AGENT 'Characters.Character' 'Merlin'
[20]
[21] ⍝ Move the character
[22] ⍝ Note: Character methods return Request objects which must be release
[23]  REQUEST←COM 'METHOD' MERLIN 'MoveTo' 100 200
[24]  COM 'RELEASE' REQUEST
[25]
[26] ⍝ Show the character
[27]  REQUEST←COM 'METHOD' MERLIN 'Show'
[28]  COM 'RELEASE' REQUEST
[29]
[30] ⍝ Tell the character to speak
[31]  REQUEST←COM 'METHOD' MERLIN 'Speak' 'Hello from the APL Products and Services group
at IBM.'
[32]  COM 'RELEASE' REQUEST
[33]
[34] ⍝ Set an event handler for the agent's Click event
[35] ⍝ Note: Use a label as the event handler so we can simply branch to it
[36] ⍝       when the event is signaled.
[37]  COM 'HANDLERS' AGENT(1 2⍴'Click' CLICK)
[38]
[39] LOOP: ⍝ Wait for events
[40]  (ET EM RC)←1 COM 'WAIT' ¯1
[41]  →(~ET≡0 0)/ERROR
[42]  →(RC≡'')/LOOP
[43]  (OBJECT EVENT HANDLER CURSORPOS TIME NAMED)←6↑RC
[44]  POSITIONAL←6↓RC
[45] ⍝ Release the object returned with the event
[46]  COM 'RELEASE' OBJECT
[47]  →HANDLER
[48]
[49] CLICK:
[50]
[51] ERROR:
[52] ⍝ Hide the character
[53]  REQUEST←COM 'METHOD' MERLIN 'Hide'
[54]  COM 'RELEASE' REQUEST
[55]
[56] ⍝ Release merlin
[57]  COM 'RELEASE' MERLIN
[58]
[59] ⍝ Unload merlin
[60]  0 0⍴COM 'METHOD' AGENT 'Characters.Unload' 'Merlin'
[61]
[62] ⍝ Reset the handlers to release internal references to the agent
[63]  COM 'HANDLERS' AGENT ''
[64]
[65] ⍝ Release the agent
[66]  COM 'RELEASE' AGENT
[67]  ∇
```

COM Excel Example

The following log demonstrates using COM to control Microsoft Excel:

```
      3 11 ⎕NA 'COM'
1
      ⍝ Create an instance of Excel
      EXCEL←COM 'CREATE' 'Excel.Application'
      ⍝ Invoke a method to open a file
```


© Copyright IBM Corporation 1994, 2017

```
      WORKBOOK←COM 'METHOD' EXCEL 'Workbooks.Open' 'c:\sample.xls'
      ⍝ Specify a property to make the Excel window visible
      COM 'PROPERTY' EXCEL 'Visible' 1
      ⍝ Reference a property to get the contents of the spreadsheet
      DATA←COM 'PROPERTY' EXCEL 'ActiveSheet.UsedRange.Value'
      DATA
 David 12 13
    14 15 16
      ⍝ Specify a property to set the value of the spreadsheet cells
      DATA←⌽DATA
      COM 'PROPERTY' EXCEL 'ActiveSheet.UsedRange.Value' DATA
      ⍝ Reference an indexed property to get the value of a single cell
      COM 'PROPERTY' EXCEL 'Cells.Item().Value' (1 3)
David
      ⍝ Reference another indexed property to get a range of cells
      COM 'PROPERTY' EXCEL 'Range().Value' (⊂'A1:B2')
13 12
16 15
      ⍝ Reference a specific range of cells
      DATA←COM 'PROPERTY' EXCEL 'Cells().Resize().Value' (2 3) (4 5)
      DATA
772490 231068 555695 549374 314042
371448 913758 527378 630692  33735
 13214  81415 327089 380452 241366
719720 326696 772742 465291 129836
      ⍝ Specify a range of cells
      DATA←4 5⍴⍳20
      COM 'PROPERTY' EXCEL 'Cells().Resize().Value' (2 3) (4 5) DATA
      ⍝ Set a property to make Excel terminate when we release it
      COM 'PROPERTY' EXCEL 'Visible' 0
      ⍝ Prevent Excel from prompting when the file is closed
      COM 'PROPERTY' EXCEL 'DisplayAlerts' 0
      ⍝ Invoke a method to close the file
      COM 'METHOD' WORKBOOK 'Close'
¯1
      ⍝ Release the objects we acquired
      COM 'RELEASE' WORKBOOK
      COM 'RELEASE' EXCEL
```

COM Internet Explorer Example

The following function demonstrates using COM to control Microsoft Internet Explorer:

```
      ∇DEMO_IE;COM;CURSORPOS;EM;ET;EVENT;HANDLER;IEXPLORER;NAMED;OBJECT;POSITIONAL;RC;TIME
[1]   ⍝
[2]   ⍝ Demonstrate using COM to control Internet Explorer
[3]   ⍝
[4]    ⎕ES(~3 11 ⎕NA 'COM')/5 1
[5]
[6]   ⍝ Create an instance of Internet Explorer
[7]    IEXPLORER←COM 'CREATE' 'InternetExplorer.Application'
[8]
[9]   ⍝ Tell the explorer to display a web site
[10]   0 0⍴COM 'METHOD' IEXPLORER 'Navigate2' 'http://www.ibm.com'
[11]
[12]  ⍝ Set the size and position of the explorer window
[13]   COM 'PROPERTY' IEXPLORER 'Height' 800
[14]   COM 'PROPERTY' IEXPLORER 'Width' 800
[15]   COM 'PROPERTY' IEXPLORER 'Top' 50
[16]   COM 'PROPERTY' IEXPLORER 'Left' 50
[17]
[18]  ⍝ Set an event handler
[19]  ⍝ Note: Use a label as the event handler so we can simply branch to it
[20]  ⍝       when the event is signaled.
[21]   COM 'HANDLERS' IEXPLORER(1 2⍴('OnQuit' ONQUIT))
[22]
```


© Copyright IBM Corporation 1994, 2017

```
[23] ⍝ Show the explorer
[24]  COM 'PROPERTY' IEXPLORER 'Visible' 1
[25]
[26] LOOP: ⍝ Wait for events
[27]  ⎕←(ET EM RC)←1 COM 'WAIT' ¯1
[28]  →(~ET≡0 0)/ERROR
[29]  →(RC≡'')/LOOP
[30]  (OBJECT EVENT HANDLER CURSORPOS TIME NAMED)←6↑RC
[31]  POSITIONAL←6↓RC
[32] ⍝ Release the object returned with the event
[33]  COM 'RELEASE' OBJECT
[34]  →HANDLER
[35]
[36] ONQUIT:
[37] ERROR:
[38]  COM 'RELEASE' IEXPLORER
[38]  COM 'RELEASE' IEXPLORER
[39]  ∇
```

COM Word Example

The following log demonstrates using COM to control Microsoft Word:

```
      3 11 ⎕NA 'COM'
1
      ⍝ Start Microsoft Word
      WORD←COM 'CREATE' 'Word.Application'
      ⍝ Open a document
      DOCUMENT←COM 'METHOD' WORD 'Documents.Open' 'c:\sample.doc'
      ⍝ Get the builtin style constants
      CONS←COM 'QUERY' 'CONSTANTS' WORD 'WdBuiltinStyle'
      ⍝ Get the value of the constant for level 1 heading style
      INDEX←CONS[CONS[;1]⍳⊂'wdStyleHeading1';2]
      ⍝ Get the font used for level 1 headings
      COM 'PROPERTY' DOCUMENT 'Styles.Item().Font.Name' INDEX
Verdana
      ⍝ Set the font used for level 1 headings
      COM 'PROPERTY' DOCUMENT 'Styles.Item().Font.Name' INDEX 'Arial'
      ⍝ Save the changes
      COM 'METHOD' DOCUMENT 'Save'
      ⍝ Close the file
      COM 'METHOD' DOCUMENT 'Close'
      ⍝ Release the file
      COM 'RELEASE' DOCUMENT
      ⍝ Close Word
      COM 'METHOD' WORD 'Quit'
      ⍝ Release Word
      COM 'RELEASE' WORD
```

COM Office Visual Basic Example

The following function demonstrates using Visual Basic code in a Microsoft Office program.

```
VDEMO_VB_CODE;CODEMODULE;COM;COMPONENT;COUNT;COUNTOFLINES;EM;ET;EXCEL;INDEX;LINES;NAME;TYP
E;VBCOMPONENTS;VB_CODE;WORKBOOK;vbext_ct_ClassModule;vbext_ct_StdModule
[1]  ⍝
[2]  ⍝ Demonstrate how to work with Visual Basic code in a Microsoft Office program.
[3]  ⍝
[4]  ⍝ Note:
[5]  ⍝
[6]  ⍝    Microsoft provides little documentation about this topic.
[7]  ⍝    However, good documentation from third parties is available on the web.
```


© Copyright IBM Corporation 1994, 2017

```
[8]  ⍝    For more information, search the internet for VBPROJECT.
[9]  ⍝
[10]
[11]  ⎕ES(~3 11 ⎕NA 'COM')/1 5
[12]
[13] ⍝ Start Excel and add an empty workbook
[14]  EXCEL←COM 'CREATE' 'Excel.Application'
[15]  WORKBOOK←COM 'METHOD' EXCEL 'Workbooks.Add'
[16]
[17] ⍝
[18] ⍝ By default, Microsoft Office programs do not allow applications to access
[19] ⍝ Visual Basic macro source code.  If access is not authorized, the following
[20] ⍝ property reference will fail.
[21] ⍝
[22] ⍝ To authorize access, start Excel, or the desired Office program,
[23] ⍝ pull down the Tools menu, select Macro, and then select Security....
[24] ⍝ In the Security dialog, select the Trusted Sources tab and check
[25] ⍝ Trust access to Visual Basic Project.  Then press Ok and close
[26] ⍝ the program.
[27] ⍝
[28] ⍝ Get the handle of Excel's VBComponents object
[29]  (ET EM VBCOMPONENTS)←1 COM 'PROPERTY' WORKBOOK 'VBProject.VBComponents'
[30]  →(~ET≡0 0)/CLEANUP
[31]
[32] ⍝ Display the possible component types
[33]  'Component types:'
[34]  COM 'QUERY' 'CONSTANTS' VBCOMPONENTS 'vbext_ComponentType'
[35]
[36] ⍝ Define component type constants.
[37]  vbext_ct_StdModule←1
[38]  vbext_ct_ClassModule←2
[39]
[40] ⍝⍝⍝
```

```
[77] ⍝ Get the component's name and type
[78]  NAME←COM 'PROPERTY' COMPONENT 'Name'
[79]  TYPE←COM 'PROPERTY' COMPONENT 'Type'
[80]
[81]  'Component Index: ',(⍕INDEX),' Type: ',(⍕TYPE),' Name: ',⍕NAME
[82]
[83]  →(~TYPE∊vbext_ct_ClassModule vbext_ct_StdModule)/DOES_NOT_CONTAIN_CODE
[84]
[85] ⍝ Get the component's CodeModule object
[86]  CODEMODULE←COM 'PROPERTY' COMPONENT 'CodeModule'
[87]
[88] ⍝ Get the number of lines in the codemodule
[89]  COUNTOFLINES←COM 'PROPERTY' CODEMODULE 'CountOfLines'
[90]
[91] ⍝ Get the source code
[92]  LINES←COM 'PROPERTY' CODEMODULE 'Lines()'(1 COUNTOFLINES)
[93]
[94]  'Count of lines: ',⍕COUNTOFLINES
[95]  'Lines:'
[96]  LINES
[97]
[98] ⍝ Release the code module
[99]  COM 'RELEASE' CODEMODULE
[100]
[101] DOES_NOT_CONTAIN_CODE:
[102] ⍝ Release the component
[103]  COM 'RELEASE' COMPONENT
[104]
[105]  →LOOP INDEX←INDEX+1
[106]
[107] DONE:
[108] ⍝ Release the components collection
[109]  COM 'RELEASE' VBCOMPONENTS
[110]
[111]
⍝⍝⍝
```

COMBROWSE - COM Browser

```
[owner] COMBROWSE {class | object | class object}
```

The COMBROWSE external function is an interactive tool for browsing the members, enumerations, and events supported by Component Object Model classes and objects.

owner
: Integer scalar - Handle of COMBROWSE window's owner

class
: Character vector - COM class such as 'Word.Application'. If object is not supplied, an instance of the class is created.

object
: Integer scalar - Handle of COM object returned by either the COM external function or the GUITOOLS function CALLCOM.

© Copyright IBM Corporation 1994, 2017

COPY - Copy Workspace Objects

```
result ← COPY specification
```

This function is a program interface to the APL2 system command)COPY.

specification
: A character vector containing the specification of the workspace to copy from and optional list of objects to copy. The syntax is the same as for)COPY:
  `[library] wsname [object ...]` or
  `'[path]filename' [object ...]`

result
: A seven item array:

  [1] Return Code
  - 0 - Success.
  - 1 - Success, but some objects were not copied. See items 4 through 7 for object lists.
  - 2 - Error. See item 2 for reason code.

  [2] Reason Code
  - 0 - Success
  - 1 - Incorrect command
  - 2 - Improper library reference
  - 3 - Workspace not found
  - 4 - Workspace locked
  - 5 - Library unavailable
  - 6 - Library I/O error
  - 7 - System limit
  - 8 - Workspace invalid

  [3] Workspace timestamp, if Return Code is 0 or 1

  [4] List of objects not found, if object names were specified.

  [5] Not used. (Null)

  [6] Not used. (Null)

  [7] List of objects not copied due to insufficient workspace size.

© Copyright IBM Corporation 1994, 2017

CPUCOUNT - Get Number of Processors

```
count ← CPUCOUNT
```

This function returns the number of CPU processors in the machine. This value can be used to determine whether PEACHP or PEACHT may provide a performance benefit.

count
: A scalar integer containing the number of CPU processors in the machine.

Note:

CPUCOUNT returns the number of processors in the system as reported by the operating system. Even if CPUCOUNT returns a value greater than 1, PEACHP and PEACHT may not provide improved performance. For example, the CPU processor may support hyperthreading but the operating system may not exploit running tasks asynchronously.

© Copyright IBM Corporation 1994, 2017

CTK - Character to Kanji

```
data ← [codepage] CTK characters
```

This function converts APL2 characters to the Multibyte (*Shift-JIS*) format.

Note: This function is available only on Windows systems.

`characters`
: An APL2 character vector. There may be characters present in the vector from outside of `⎕AV`

`codepage`
: The codepage to use for the conversion. If not given, the active codepage on the running machine is used.

`data`
: A character vector containing data in the Shift-JIS format.

For more information on using extended characters in APL2, see Double-Byte Character Set Support.

© Copyright IBM Corporation 1994, 2017

CTN - Character to Numeric

```
numbers ← CTN characters
```

This function converts a character vector or matrix to a numeric vector or matrix.

`characters`
: A character vector or matrix that contains the formatted representation of one or more numbers. Only numeric formats considered valid by the C `strtod` subroutine are accepted, except that the APL high minus character will be accepted and replaced with the regular minus character before calling `strtod`.

  ```
  [ +|-|¯ ] [ digits ] [ . ] digits [ e|E [ +|-|¯ ] digits ]
  ```

  If a matrix is passed, each row of the matrix must produce the same number of elements in its result. LENGTH ERROR will be produced if the matrix does not conform.

`numbers`
: A numeric vector or matrix that is formed by processing `characters` with the `strtod` routine. A null vector is returned if the argument contained any invalid numeric representations.

© Copyright IBM Corporation 1994, 2017

CTUTF - Character to UTF

```
result ← [bits] CTUTF data
```

Encodes APL2 character data into UTF-7 and UTF-8 formatted data.

UTF-7 and UTF-8 are Universal Character Set Transformation Formats. They are used for transmitting Unicode data across networks which represent character data as bytes and use either 7 or 8 bits in each byte. UTF-7 and UTF-8 are described by the Unicode standard and in the Internet Request for Comments (RFC) 2152 and 2279.

`bits`
: The number of bits of significant data within individual encoded bytes. If `bits` is 7, `result` contains UTF-7 data. If `bits` is 8, `result` contains UTF-8 data. If `bits` is omitted, `result` contains UTF-8 data.

`data`
: A character vector of any length containing the data to be encoded. CTUTF will automatically convert the data to Unicode if necessary before encoding into the transformation format.

`result`
: A character vector containing the encoded data.

© Copyright IBM Corporation 1994, 2017

DATATYPE - Array Data Type

```
code ← DATATYPE array
```

This function returns a numeric data type code for an APL2 array.

`array`
: Any APL2 data object.

`code`
: One of the following numeric codes:

`2`	Extended Character
`1`	Byte Character
`0`	Nonhomogeneous or Nested
`¯1`	Boolean
`¯2`	Integer
`¯3`	Floating Point
`¯4`	Complex
`¯5`	Arithmetic Progression Vector

© Copyright IBM Corporation 1994, 2017

DISPLAY, DISPLAYC and DISPLAYG - APL2 Array Structures

The DISPLAY, DISPLAYC, and DISPLAYG functions are useful in showing the structure of nested and mixed arrays. The external function versions are equivalent to the APL functions found in the DISPLAY workspace.

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

Z is a character matrix representing the array X.

DISPLAY and DISPLAYG use box characters. DISPLAYG is identical to DISPLAY and is included for compatibility with the DISPLAYG function distributed with APL2 on mainframe systems. DISPLAYC uses characters that display on all implementations. This is functionally equivalent to the DISPLAY function in the APL2 mainframe system.

The following characters are used to convey shape information:

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

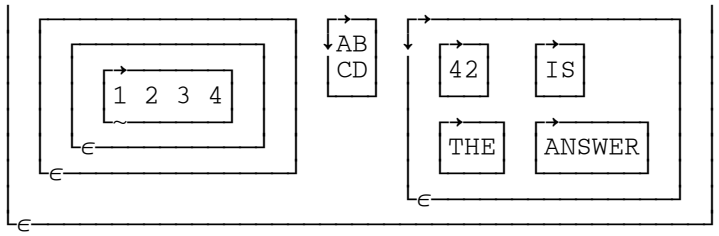
The following characters are used to convey type information:

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

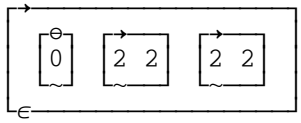
To use each of the DISPLAY functions:

```
      (⊂3 11)⎕NA¨'DISPLAY' 'DISPLAYC' 'DISPLAYG'
1 1 1
      (DISPLAY ⍳5) (DISPLAYC ⍳5) (DISPLAYG ⍳5)
┌→─────────┐ .→--------. ┌→─────────┐
|1 2 3 4 5| ¦1 2 3 4 5¦ |1 2 3 4 5|
└~─────────┘ '~---------' └~─────────┘
      X←⊂⍳5
      (DISPLAY X) (DISPLAYC X) (DISPLAYG X)
┌∊────────┐ .∊----------. ┌∊────────┐
| ┌→─────┐ | | .→------. | | ┌→─────┐ |
| |1 2 3 4 5| | | ¦1 2 3 4 5¦ | | |1 2 3 4 5| |
| └~─────┘ | | '~-------' | | └~─────┘ |
└─────────┘ '∊----------' └─────────┘
      X←(⊂⊂⍳4)(2 2⍴'ABCD')(2 2⍴'42' 'IS' 'THE' 'ANSWER')
      DISPLAY X
┌→──────────────────────────────────────┐
```

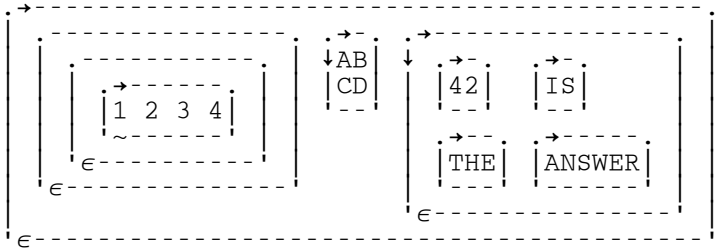

© Copyright IBM Corporation 1994, 2017



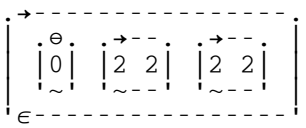

```
      ρX
3
      ρ¨X
   2 2  2 2
      DISPLAY ρ¨X
```




```
      DISPLAYC X
```




```
      ρX
3
      ρ¨X
   2 2  2 2
      DISPLAYC ρ¨X
```



© Copyright IBM Corporation 1994, 2017

EDITOR_2 - APL2 Editor 2

```
EDITOR_2 name
```

EDITOR_2 is functionally equivalent to Editor 2 (available through)EDITOR 2) on APL2 mainframe systems. Editor 2 is fully described in *APL2 Programming: Language Reference*.

To use this function, access it using ⎕NA and then invoke it with a right argument of the name of the object to be edited:

```
3 11 ⎕NA 'EDITOR_2'
EDITOR_2 'object_name'
```

This is equivalent to entering the mainframe APL2 commands:

```
)EDITOR 2
∇object_name
```


© Copyright IBM Corporation 1994, 2017

EXP - Execute in Previous Namescope

This routine is designed to be used in namespaces. It provides access to names in the namescope of the function or operator that caused entry into the current namescope.

Notes:

1. If the EXP routine is run in a namescope where there is no previous namescope (for instance, in your active workspace rather than in a namespace), it operates in the current workspace.
2. Processing a function or operator declared with ⎕NA causes an explicit change to the namescope of the function or operator. Processing the EXP function or an operand to an external operator causes an implicit namescope switch. If the EXP function is run in a namescope that was entered implicitly, the namescope switches to the one that originally caused explicit entry into the current namescope.

Functions processed under control of EXP operate in the same manner as those processed under control of ⎕EC, and exhibit the following behavior:

- Requests for quad input are handled the same as quad input under ⎕EC.
- Errors generated during processing do not cause suspension of the function being processed and are reported against EXP.
- Stop control vectors (S∆) are ignored.
- An attention signal does not cause suspension; an interrupt signal causes the EXP function to be interrupted.
- Branch escape (→) causes the EXP function to run, but its callers are not abandoned.

EXP can perform four different actions, depending on how the right argument is constructed:

```
result ← EXP ⊂expr
```

Processes an APL expression in the previous namescope.

`expr`
: A character scalar or vector containing an expression to be processed in the previous namescope. If a vector, it must be enclosed (with ⊂) to create a single item.
If the expression is nothing more than the name of a variable or niladic function in the previous namescope, then you are referencing the named item.

`result`
: The result of processing `expr`.

Example:

```
⍝ Get value of ⎕IO from previous namescope
PREV_IO←EXP ⊂'⎕IO'
⍝ (0 1) or (1 2) depending on ⎕IO in previous namescope
IOTA2←EXP ⊂'⍳2'
```


© Copyright IBM Corporation 1994, 2017

```
result ← EXP fn_name value
```

Processes a monadic function in the previous namescope using an argument from the current namescope.

`fn_name`
: A character scalar or vector containing the name of a monadic function in the previous namescope. The function can be a defined function or a system function.

`value`
: The right argument (from the current namescope) to be supplied to the monadic function.

`result`
: The result of executing the named monadic function in the previous namescope.

Example:

```
      ⍝ Create function NEWFN in previous namescope
      Z←EXP '⎕FX' ('R←NEWFN RA' 'R←RA')
```

```
result ← EXP lvalue fn_name rvalue
```

Processes a dyadic function in the previous namescope using arguments from the current namescope.

`fn_name`
: A character scalar or vector containing the name of a dyadic function in the previous namescope. The function can be a defined function or a system function.

`lvalue`
: The left argument (from the current namescope) to be supplied to the dyadic function.

`rvalue`
: The right argument (from the current namescope) to be supplied to the dyadic function.

`result`
: The result of executing the named dyadic function in the previous namescope.

```
result ← EXP vname '←' value
```

Assigns a value from the current namescope to a variable in the previous namescope.

`vname`
: A character scalar or vector that contains the name of the variable in the previous namescope.

`value`
: The value from the current namescope that is to be assigned to `vname`.

`result`
: The same as `value`.

Example:

```
      ⍝ Restore ⎕IO in previous namescope
      T←EXP '⎕IO' '←' PREV_IO
```


© Copyright IBM Corporation 1994, 2017

FDELETE - File Delete

```
rc ← FDELETE filespec
```

This function deletes a system file.

On Windows, the deleted file will be sent to the Recycle Bin (subject to the Recycle Bin's properties).

`filespec`
: A character vector that identifies a file. It may be a simple file name, or may include a path specification.

  On Windows, `filespec` may contain Unicode characters.

`rc`
: Return code. On Windows, the code is from the `SHFileOperation` routine. On Unix systems, it is the `errno` value from the call to the `unlink` routine.

© Copyright IBM Corporation 1994, 2017

FILE - File Read or Write

```
rdata ← FILE filespec

rc ← wdata FILE filespec
```

This function reads or writes a system file in binary mode. The monadic form reads a file, while the dyadic form writes it. FILE is a stream read/write function. It is not record-oriented, and it does not perform data conversion.

`filespec`
: A character vector that identifies a file. It may be a simple file name, or may include a path specification.

  On Windows `filespec` may contain Unicode characters.

`rdata`
: The contents of the file as a simple character vector. Any line or record separators are retained as they existed in the file.

  Any error conditions encountered while reading a file are reported by returning a numeric result in place of the data.

`wdata`
: The data to be written to the file.

  If `wdata` is a simple character object, it is raveled and written to the file exactly as given. If the file already exists, the old contents are replaced by the new data. If the character object is null, a 0-length file will be written.

  If `wdata` is a non-character null, the file will be erased.

`rc`
: The return code from writing a file; normally zero. Nonzero return codes are system dependent. The `CHECK_ERROR` function can be used to obtain more information.

Hint: You can use the APL membership (∈) and partition (⊂) functions to parse an ASCII text file into a vector of vectors.

© Copyright IBM Corporation 1994, 2017

FILE Namespace

The FILE namespace consists of several groups of cover functions that use the file auxiliary processors AP 210 and AP 211.

The namespace objects, shipped in file `FILE.ans`, are equivalent to the objects found in the FILE workspace. See FILE Workspace for a list of the objects provided and the syntax for their usage.

Because of the large number of objects in the FILE namespace and name conflicts between its objects and other supplied external routines, nicknames for these external objects are not provided in the APL2 default names file (`aplnm011.nam`). Instead, associations to the objects can be made using the namespace file name in the left argument of `⎕NA`:

```
      'FILE' 11 ⎕NA 'REBUILD211'
1
      'FILE' 11 ⎕NA ⊃'WOPEN' 'READ' 'WRITE' 'CLOSE'
1 1 1 1
```

Surrogate names may be used to avoid name conflicts with objects already in your workspace:

```
      'FILE' 11 ⎕NA ⊃'X_OPEN WOPEN' 'X_READ READ' 'X_WRITE WRITE' 'X_CLOSE CLOSE'
1 1 1 1
      X_OPEN 'MYFILE'
```


© Copyright IBM Corporation 1994, 2017

FSTAT - File Status

```
fileinfo ← FSTAT filespec
```

This function returns information about a system file. The information is taken from the structure returned by the C routine `stat`.

`filespec`
: A character vector that identifies a file. It may be a simple file name, or may include a path specification.

  On Windows `filespec` may contain Unicode characters.

`fileinfo`
: An eleven-element nested array:

`fileinfo[1]`	Device ID of files's directory entry
`fileinfo[2]`	File serial number (Unix)
`fileinfo[3]`	File mode information. The information is returned as a 16-element Boolean array. Each element of the array corresponds to a file attribute. An element value of `1` indicates the attribute exists. The following table shows the correspondence between the array elements and file attributes: `mode[1]` Regular file `mode[2]` Directory `mode[3]` Character device `mode[4]` Fifo (Unix) `mode[5]` User ID set on execution (Unix) `mode[6]` Group ID set on execution (Unix) `mode[7]` Unused `mode[8]` Read permission `mode[9]` Write permission `mode[10]` Execute permission `mode[11]` Read permission - group (Unix) `mode[12]` Write permission - group (Unix) `mode[13]` Execute permission - group (Unix) `mode[14]` Read permission - other (Unix) `mode[15]` Write permission - other (Unix) `mode[16]` Execute permission - other (Unix)
`fileinfo[4]`	Number of links (always 1 on Windows)
`fileinfo[5]`	User ID (Unix)
`fileinfo[6]`	Group ID (Unix)
`fileinfo[7]`	Device ID of file (always the same as `fileinfo[1]` on Windows)
`fileinfo[8]`	Size of file in bytes
`fileinfo[9]`	Time of last access (in `⎕TS` format)

© Copyright IBM Corporation 1994, 2017

`fileinfo[10]`	Time of last modification (in `⎕TS` format)
`fileinfo[11]`	Time of file creation (in `⎕TS` format)

If the operation is unsuccessful, `fileinfo` contains the `errno` code from the `stat` routine.

The following constants can be used with the `BITWISE` operator to examine the file mode bits:

```
S_IFDIR←16384    ⍝ Directory
S_IFCHR←8192     ⍝ Character device
S_IFREG←32768    ⍝ Regular file
S_IREAD←256      ⍝ Read permission
S_IWRITE←128     ⍝ Write permission
S_IEXEC←64       ⍝ Execute/search permission
```


© Copyright IBM Corporation 1994, 2017

GETENV - Get Environment Variable

```
vector ← GETENV env_variable
```

This function returns the value of an environment variable as a character vector.

`env_variable`
: An enclosed character vector that identifies the environment variable. Depending on the platform, environment variable names can be case sensitive. For maximum application portability, use the exact case for the name of the environment variable.

`vector`
: An enclosed character vector that represents the value of the environment variable. If the environment variable is not set, then `vector` is `⊂⍳0`.

© Copyright IBM Corporation 1994, 2017

GETLIB - Get Path for Library Number

```
path ← GETLIB number
```

Returns the directory path associated with an APL2 library number.

`number`
: A whole number between 1 and 32767.

`path`
: A character vector containing the directory path associated with the library number. If the given library number is not defined, `path` is a null character vector.

Libraries can be defined in the `apl2.ini` file or by environment variable. See Library Specification for more information.

© Copyright IBM Corporation 1994, 2017

GMTSTAMP - Get or Set Timestamp

```
result ← [newts] GMTSTAMP name
```

The monadic form of this function gets an object's timestamp. The dyadic form allows you to set the timestamp of a defined function or operator.

This function operates in GMT (Greenwich Mean Time). Input timestamps are assumed to be in GMT and are stored as given. Result timestamps are reported in GMT. Depending on the value of `⎕TZ`, the output from the monadic form of `GMTSTAMP` and the output from `2 ⎕AT` for the same object may be different.

Attempts to set the timestamp of a variable, locked function or external function will result in `DOMAIN ERROR`.

`name`
: Character vector function or operator name.

`newts`
: Numeric vector in `⎕TS` format, to be used to set a new timestamp for the named object.

`result`
: Numeric vector in `⎕TS` format, the current timestamp for the named object.

© Copyright IBM Corporation 1994, 2017

GRAPHPAK Namespace

GRAPHPAK is a powerful general-purpose graphical library that uses the AP 207 Universal Graphics auxiliary processor to display results.

The namespace objects, shipped in namespace `GRAPHPAK.ans`, are equivalent to the objects found in the GRAPHPAK workspace. See GRAPHPAK Workspace for an overview of the workspace and notes on workstation usage. For a full description of GRAPHPAK, refer to *APL2 GRAPHPAK: User's Guide and Reference*.

Because of the large number of objects in the GRAPHPAK namespace and name conflicts between its objects and other supplied external routines, nicknames for these external objects are not provided in the APL2 default names file (`aplnm011.nam`). Instead, associations to the objects can be made using the namespace file name in the left argument of `⎕NA`:

```
      'GRAPHPAK' 11 ⎕NA 'DEMO'
1
      'GRAPHPAK' 11 ⎕NA ⊃'CHART' 'ERASE' 'bw'
1 1 1
```

Surrogate names may be used to avoid name conflicts with objects already in your workspace:

```
      'GRAPHPAK' 11 ⎕NA ⊃'G_CHART CHART' 'G_ERASE ERASE' 'G_bw bw'
1 1 1
      G_bw←.5
      'F' G_CHART 1 2 3
      G_ERASE
```


© Copyright IBM Corporation 1994, 2017

GUITOOLS Namespace

Note: This namespace is only provided on Windows.

The GUITOOLS namespace contains tools for building GUI applications. For information about building GUI applications, consult *APL2 Programming: Developing GUI Applications*.

The objects shipped in the GUITOOLS namespace are identical to the objects of the same name found in the GUITOOLS workspace. However, unlike the GUITOOLS workspace, the GUITOOLS namespace does not support execution of event handlers. Therefore, the following objects are not available in the namespace

- `CHECK_EVENTS`
- `DEFAULTPROC`
- `EXECUTEDLG`
- `EXECUTEDLGX`

Applications that use the GUITOOLS namespace must use the `WAIT_EVENT` function to wait for events.

When a variable shared with AP 145 is retracted, the AP destroys all windows and releases all other resources acquired using the shared variable. Applications that use the GUITOOLS workspace often localize the `SV145` shared variable to cause AP 145 to release resources when the application terminates. This technique does not work with the GUITOOLS namespace because the shared variable is in the namespace and cannot be localized.

To force the GUITOOLS namespace's shared variable to be retracted, use the following GUITOOLS functions to explicitly share and retract the shared variable:

- `GUISHARE`
- `GUIRETRACT`

Alternatively, expunge all associations with the GUITOOLS namespace during application termination. Localize the associated names to cause them to be automatically expunged.

Because of the large number of objects in the GUITOOLS namespace, nicknames for these external objects are not provided in the APL2 default names file (`aplnm011.nam`). Instead, associations to the objects can be made using the namespace file name in the left argument of `⎕NA`:

```
      'GUITOOLS' 11 ⎕NA ⊃'FILEDLG' 'GUIRETRACT'
1 1
      FILEDLG 1 1 ''
C:\Program Files\ibmapl2w\readme.txt
      GUIRETRACT
```

Surrogate names may be used:

```
      'GUITOOLS' 11 ⎕NA 'SETPROP SET_PROPERTY'
1
```


© Copyright IBM Corporation 1994, 2017

GUIVARS Namespace

Note: This namespace is only provided on Windows.

The GUIVARS namespace contains variables that define a set of common constants used in GUI programs. For information about building GUI applications and the constants supplied in GUIVARS, consult *APL2 Programming: Developing GUI Applications*.

The objects shipped in the GUIVARS namespace are identical to the objects found in the GUIVARS workspace.

Because of the large number of objects in the GUIVARS namespace, nicknames for these external objects are not provided in the APL2 default names file (`aplnm011.nam`). Instead, associations to the objects can be made using the namespace file name in the left argument of `⎕NA`:

```
      'GUIVARS' 11 ⎕NA 'HWND_DESKTOP'
1
      'GUIVARS' 11 ⎕NA ⊃'VK_F3' 'AF_CHAR' 'AF_VIRTUALKEY'
1 1 1
```

Surrogate names may be used:

```
      'GUIVARS' 11 ⎕NA 'OKCANCEL MBID_OKCANCEL'
1
```


© Copyright IBM Corporation 1994, 2017

Host System Utilities

The host system utilities are functions for performing operating system dependent tasks.

The external function versions are equivalent to the APL functions found in the host workspaces (AIX, LINUX, SOLARIS and WINDOWS). By using the external functions, you can create applications which are common across the Workstation APL2 platforms. The links to the external functions will be resolved at runtime to use the correct versions for the operating system in use. For example:

```
      3 11 ⎕NA 'ERASE'
1
      ERASE 'temp.fil'
0
```

Except where noted, these are monadic functions. All functions return either `0` or data if successful. They return a numeric error code if an error occurs.

```
Z←DIR path
```

Returns a directory list. This is equivalent to the `dir` command on Windows, and the `ls` command on Unix systems. The right argument, `path`, is the path leading to the directory whose contents are to be displayed.

```
Z←ERASE filename
```

Erases a file. This is equivalent to the `delete` command on Windows, and the `rm` command on Unix systems. `filename` is the name of the file to be erased, and can include a path definition.

```
Z←HOST cmd
```

Issues command `cmd` to the host operating system through AP 100. When `HOST ''` is executed (`cmd` is null), it returns the name of the currently running operating system.

```
Z←MKDIR path
```

Creates a new subdirectory.

```
Z←cmd PIPE data
```


© Copyright IBM Corporation 1994, 2017

Provides a means to pipe APL2 data to an operating system command and returns any output generated by the command.

The right argument, `data`, must be either a simple character vector, a character matrix, or a vector of character vectors specifying the input to be passed to the command as *stdin* (standard input). If this is an empty vector, no input is passed to the command. The left argument, `cmd`, must be a simple character vector specifying the command to be executed. The result is a vector of vectors containing the output *stdout* (standard output) generated by executing the command. Some examples of the use of the `PIPE` function:

To get names of APL workspaces and transfer files, sorting by date:

```
      ⊃'dir /b/od *.atf *.apl' PIPE ''
```

To run batch scripts through the interpreter:

```
      JOB1←')LOAD 1 DISPLAY' 'DISPLAY ''APL2'' 123'
      JOB2←')LOAD 2 WINDOWS' '∇PIPE[⎕]∇'
      OUTPUT←(⊂'apl2 -sm piped -quiet on')PIPE¨JOB1 JOB2
```

```
Z←RENAME oldnew
Z←old RENAME new
```

Renames a file. When called monadically, the right argument, `oldnew`, has the form `'oldname newname'` where `oldname` is the name of the file to be renamed (and may include a path definition), and `newname` is the new name by which the file is to be known. `RENAME` can also be used dyadically, with the old name as the left argument and the new name as the right argument.

```
Z←RMDIR path
```

Removes the subdirectory pointed to by `path`.

© Copyright IBM Corporation 1994, 2017

IDIOMS and IDIOMSG - APL2 Idiom Library

APL2 is a very powerful and concise language. Although experienced APL2 programmers can produce working solutions to complex problems in a very short time, learning the APL2 language can take years. The novice is usually entranced with the power of APL2, but may have a hard time thinking in vector notation. APL2 algorithms are not always obvious!

In order to speed up the learning process of APL2, IDIOMS was developed. With over 600 distinct APL2 phrases, sorted into 24 general categories, IDIOMS represents a fairly complete list of solutions to common application problems and lets you take advantage of algorithms that many others have developed.

```
list ← IDIOMS
```

`IDIOMS` provides a text-based interface to the APL2 Idiom Library. `IDIOMS` is available on all platforms. For further information about using `IDIOMS` consult the online help or the IDIOMS workspace.

```
IDIOMSG
```

`IDIOMSG` provides a graphical user interface to the library. `IDIOMSG` is only available on Windows. For further information about using `IDIOMSG` consult the online help.

© Copyright IBM Corporation 1994, 2017

KTC - Kanji to Character

```
characters ← [codepage] KTC data
```

This function converts from the Multibyte (*Shift-JIS*) format to APL2 characters.

Note: This function is available only on Windows systems.

`data`
: A character vector containing data in the Shift-JIS format.

`codepage`
: The codepage to use for the conversion. If not given, the active codepage on the running machine is used.

`characters`
: An APL2 character vector. There may be characters present in the vector from outside of `⎕AV`

For more information on using extended characters in APL2, see Double-Byte Character Set Support.

© Copyright IBM Corporation 1994, 2017

LEXP - Lexical Parse

```
integers ← LEXP statement
```

LEXP analyzes a character vector using APL2's syntax rules.

statement
: A character vector containing an APL2 statement.

integers
: An integer vector with the same length as statement. The integers increase wherever the corresponding character in statement begins a new token. Zeros are returned for syntactically insignificant blanks.

LEXP can be used in combination with partition to separate a character vector into words. For example:

```
      STATEMENT←'ABC ∘.× ⍳ 34.56+⍴45 56    ⍝ HELLO'
      INTEGERS←LEXP STATEMENT
      INTEGERS
1 1 1 0 2 3 4 0 5 0 6 6 6 6 6 7 8 9 9 0 10 10 0 0 0 0 11 11 11 11 11 11 11
      DISPLAY INTEGERS⊂STATEMENT
┌→──────────────────────────────────────────────────────────────────────┐
│ ┌→─┐ ┌→┐ ┌→┐ ┌→┐ ┌→┐ ┌→────┐ ┌→┐ ┌→┐ ┌→┐ ┌→┐ ┌→──────┐ │
│ │ABC│ │∘│ │.│ │×│ │⍳│ │34.56│ │+│ │⍴│ │45│ │56│ │⍝ HELLO│ │
│ └──┘ └─┘ └─┘ └─┘ └─┘ └─────┘ └─┘ └─┘ └─┘ └─┘ └───────┘ │
└∊──────────────────────────────────────────────────────────────────────┘
```


© Copyright IBM Corporation 1994, 2017

LIB - List Library Contents

```
result ← LIB specification
```

This function is a program interface to the APL2 system command)LIB.

specification
: A character vector containing the specification of the library to list and optional filters for the list. The syntax is the same as for)LIB:
  `[ libno | 'path' ] [ initial | [first]-[last] ...] [ .atf | .apl ]`

  initial indicates that only files whose names begin with the specified characters will be returned.

  [first]-[last] indicates that files whose names are in the specified range will be returned. If first or last is omitted the range is from the beginning or to the end, respectively.

  .atf or .apl indicates that only files with the specified extension will be returned. If neither is specified, the default is to return both .atf and .apl when a library number has been used, or to return all files if a path specification has been given.

  Any number of filename filters may be specified. Only one extension filter may be specified.

result
: A three item array:
  [1] Return Code
    0 - Success. See item 3 for library list.
    2 - Error. See item 2 for reason code.
  [2] Reason Code
    0 - Success
    1 - Incorrect command
    2 - Improper library reference
    3 - Workspace not found
    4 - Workspace locked
    5 - Library unavailable
    6 - Library I/O error
    7 - System limit
    8 - Workspace invalid
  [3] Character matrix containing the list of files meeting the specifications.

Examples:

```
LIB ''                    ⍝ .apl and .atf in the default library
LIB '.atf'                ⍝ .atf in the default library
LIB '1'                   ⍝ .apl and .atf in library 1
LIB '1 M'                 ⍝ .apl and .atf beginning with letter M in library 1
LIB '1 M-S'               ⍝ .apl and .atf from M to S in library 1
```


© Copyright IBM Corporation 1994, 2017

```
LIB '1 M- .apl'             ⍝ .apl from M to z in library 1
LIB '1 -M'                  ⍝ .apl and .atf from A to M in library 1
LIB '''C:\MYWS'''           ⍝ All files in C:\MYWS
LIB '''C:\MYWS'' M-S .apl'  ⍝ .apl from M to S in C:\MYWS
```


© Copyright IBM Corporation 1994, 2017

LIBS - Get APL2 Library Definitions

```
liblist ← LIBS
```

Returns the list of active library definitions.

`liblist`
: A character matrix giving the definition of each valid library number for this APL2 session. If no libraries are defined, `path` contains a null character matrix.

Libraries can be defined in the `apl2.ini` file or by environment variable. See Library Specification for more information.

© Copyright IBM Corporation 1994, 2017

LTM - Tcl List to APL2 Matrix

```
matrix ← LTM list
```

Converts a Tcl list to an APL2 matrix

`list`
: A nested vector containing data retrieved from a Tcl array using the `TCL` external function.

`matrix`
: The APL2 matrix represented by the Tcl array.

The Tcl Tktable package provides support for displaying data in tabular form. Tktable uses Tcl array variables to represent tabular data. The `LTM` function can be used to convert this type of data to an APL2 matrix.

For example:

```
      LIST←1 TCL⊂'array' 'get' 'mat'
      DISPLAY LIST
┌→──────────────────────────────────────────────────────┐
│ ┌→───┐  ┌→───┐  ┌→───┐  ┌→───┐  ┌→───┐  ┌→───┐      │
│ │0,0 │ 1│0,1 │ 2│0,2 │ 3│1,0 │ 4│1,1 │ 5│1,2 │ 6    │
│ └────┘  └────┘  └────┘  └────┘  └────┘  └────┘      │
└∊──────────────────────────────────────────────────────┘
      MATRIX←LTM LIST
      DISPLAY MATRIX
┌→─────┐
↓1 2 3 │
│4 5 6 │
└~─────┘
```

Element names which are not of the form `Row,Column` are discarded. Missing elements are replaced with null vectors.

The `DEMO_TABLE` function in the TCL workspace demonstrates using `LTM`.

© Copyright IBM Corporation 1994, 2017

MATHFNS Namespace

The MATHFNS namespace contains functions that perform specialized mathematical operations.

The namespace objects, shipped in namespace `MATHFNS.ans`, are equivalent to the objects found in the MATHFNS workspace. See MATHFNS Workspace for a list of the objects provided and the syntax for their usage.

Nicknames for these external objects are not provided in the APL2 default names file (`aplnm011.nam`). Instead, associations to the objects can be made using the namespace file name in the left argument of `⎕NA`:

```
      'MATHFNS' 11 ⎕NA 'EIGEN'
1
      'MATHFNS' 11 ⎕NA ⊃'POLYZ' 'FFT' 'IFFT'
1 1 1
```

Surrogate names may be used to avoid name conflicts with objects already in your workspace:

```
      'MATHFNS' 11 ⎕NA ⊃'M_POLYZ POLYZ' 'M_FFT FFT' 'M_IFFT IFFT'
1 1 1
      M_POLYZ ¯2 1
0.5
```


© Copyright IBM Corporation 1994, 2017

MD5 - Encode Data to MD5

```
result ← MD5 data
```

Encodes data into a fixed-length ASCII character vector using the RSA Data Security, Inc. MD5 Message-Digest Algorithm.

`data`
: A character vector of any length containing the data to be encoded. Typically, the data is ASCII characters.

`result`
: A 32-element character vector. The argument data is encoded into 16 bytes according to the MD5 algorithm and is returned as the 32-byte ASCII hexadecimal representation of those 16 bytes.

© Copyright IBM Corporation 1994, 2017

MTL - APL2 Matrix to Tcl List

```
list ← MTL matrix
```

Converts an APL2 matrix to a Tcl list.

`matrix`
: An APL2 matrix.

`list`
: A nested vector which can be assigned to a Tcl array using the `TCL` external function.

The Tcl Tktable package provides support for displaying data in tabular form. Tktable uses Tcl array variables to represent tabular data. The `MTL` function can be used to convert an APL2 matrix to the required Tcl format.

For example:

```
      LIST←MTL (2 3⍴⍳6)
      DISPLAY LIST
┌→──────────────────────────────────────────────────────────────────────┐
│ ┌→──┐  ┌→──┐  ┌→──┐  ┌→──┐  ┌→──┐  ┌→──┐                            │
│ │0,0│ 1 │0,1│ 2 │0,2│ 3 │1,0│ 4 │1,1│ 5 │1,2│ 6                          │
│ └───┘  └───┘  └───┘  └───┘  └───┘  └───┘                            │
└∊──────────────────────────────────────────────────────────────────────┘
      TCL⊂'array' 'set' 'mat' LIST
```

The `DEMO_TABLE` function in the TCL workspace demonstrates using `MTL`.

© Copyright IBM Corporation 1994, 2017

OPTION - Query or Set Session Options

```
result ← [value] OPTION key
```

You can use this function monadically to query the current state of certain processing options, or dyadically to change those options.

`key`
: Character vector containing the name of an APL2 invocation or session option. It can be in any case, and have leading and/or trailing blanks, but must not be abbreviated. A null character vector in `key` requests all the command line parameters passed to the APL2 session.

`value`
: Character vector containing a value to be assigned to the option. It can be in any case, and have leading and/or trailing blanks, but must not be abbreviated.

`result`
: If `key` is non-null, character vector containing the value of the option. In the dyadic form, the value returned is the one in effect *before* the `value` argument was applied. If `key` is null, a vector of character vectors containing the items passed in the command line that invoked APL2.

The following keys and values are supported for the dyadic form of `OPTION`:

Key	Values	Function
LX	OFF ON	Equivalent to `-lx` invocation option. Suppresses or restores execution of the latent expression on `)LOAD`.
QUIET	OFF ON	Equivalent to `-quiet` invocation option. Restores or suppresses session output.
CHECKTRACE	OFF SERVICE STMT EXEC SYNT FREE NS	Equivalent to `)CHECK TRACE` system command. Controls tracing of internal interpreter events.
CHECKWS	OFF ON ALL NOW SLOP	Equivalent to `)CHECK WS` system command. Controls internal validation of the workspace.

For the monadic form of the function, the key passed can be null, one of the keys from the above table, or any APL2 invocation option or environment variable, with the leading dash, slash or `APL` removed. For example:

```
      OPTION 'QUIET'  ⍝ Get current setting of output suppression
OFF
      OPTION 'WS'     ⍝ Get value of ws option
```


© Copyright IBM Corporation 1994, 2017

```
50m
      OPTION 'SM'     ⍝ Get value of sm option
      OPTION 'P11'    ⍝ Get value of p11 option
C:\Program Files\IBMAPL2\bin\APLNM011.NAM
      OPTION ''       ⍝ Get the APL2 command line parameters
 apl2win -ws 20m -quiet on
      ⍴⍨OPTION ''
 7  3  3  6  2
```

If the option was not specified as a parameter when APL2 was invoked, there is no entry for the option in the `apl2.ini` file, and the environment variable has no value, a null character vector is returned. If the option was specified in more than one way, invocation parameters take precedence, followed by `apl2.ini` file entries and then environment variables.

For a complete list of options supported by APL2, see Invoking APL2.

Notes:

1. Allowing programs to set and reset QUIET gives applications significantly more control over what information is displayed on the screen. As in the past, an implicit QUIET OFF action occurs any time a screen input request is issued.
2. The CHECKTRACE and CHECKWS settings are cumulative. Each dyadic call to `OPTION` adds the specified trace or validation. The result is a list of all active settings. OFF removes all settings.
3. Use of the CHECKTRACE and CHECKWS options can create large amounts of output and significantly degrade performance. These are best used under the direction of your IBM support personnel.

© Copyright IBM Corporation 1994, 2017

PCOPY - Protected COPY

```
result ← PCOPY specification
```

This function is a program interface to the APL2 system command)PCOPY. Objects are not copied if their names conflict with objects already in the workspace.

`specification`
: A character vector containing the specification of the workspace to copy from and optional list of objects to copy. The syntax is the same as for)PCOPY:
`[library] wsname [object ...]` or
`'[path]filename' [object ...]`

`result`
: A seven item array:

`[1]` Return Code
- 0 - Success.
- 1 - Success, but some objects were not copied. See items 4 through 7 for object lists.
- 2 - Error. See item 2 for reason code.

`[2]` Reason Code
- 0 - Success
- 1 - Incorrect command
- 2 - Improper library reference
- 3 - Workspace not found
- 4 - Workspace locked
- 5 - Library unavailable
- 6 - Library I/O error
- 7 - System limit
- 8 - Workspace invalid

`[3]` Workspace timestamp, if Return Code is 0 or 1

`[4]` List of objects not found, if object names were specified.

`[5]` List of objects not copied due to name conflicts.

`[6]` Not used. (Null)

`[7]` List of objects not copied due to insufficient workspace size.

© Copyright IBM Corporation 1994, 2017

PEACHP and PEACHT - Parallel Each Operators

```
result ← [larg] (function PEACHP (options [processors])) rarg
result ← [larg] (function PEACHT options) rarg
```

The PEACHP and PEACHT operators have an effect similar to the ¨ primitive operator. They apply a function to individual argument elements. However, PEACHP and PEACHT process the arguments' elements in separate asynchronous threads. On machines with multiple CPU processors, PEACHP and PEACHT can provide significant performance improvements.

PEACHP and PEACHT automatically start asynchronous APL2 interpreters for processing the arguments' elements. The definition of function is copied into the APL2 interpreter sessions. The function is then applied to each of the arguments' elements.

PEACHP uses one or more instances of auxiliary processor 200 to start the asynchronous APL2 interpreters. The interpreters run in threads in the auxiliary processors' processes. PEACHT starts the asynchronous APL2 interpreters in separate threads in the calling interpreter's process. The usage and performance effects of the two operators are similar, except that:

- PEACHP supports distributing work to other machines; PEACHT does not.
  To distribute work to other machines, use the processors operand element.
- PEACHT supports sharing of resources in the calling interpreter's process; PEACHP does not.
  To share resources, pass the handles and addresses of the resources in the function's arguments.

larg
rarg

Left and right arguments similar to arguments for the ¨ primitive operator.

Like the ¨ primitive operator, if larg is supplied, PEACHP and PEACHT applies the function between corresponding pairs of elements of larg and rarg. If larg is omitted, PEACHP and PEACHT applies the function to each element of rarg.

Unlike the ¨ primitive operator, larg and rarg must not be empty.

PEACHP supports simple homogeneous arguments; PEACHT does not. For PEACHT, each argument must be nonhomogeneous or have depth greater than one, unless it has exactly one element.

function

Character vector containing the name of the function to apply.

options

Vector of character vectors containing invocation options for the asynchronous interpreters. Code '' to use the default APL2 invocation options. See Invoking APL2 for information about default values.

© Copyright IBM Corporation 1994, 2017

`processors`

Integer vector containing processor identification numbers. Each number must identify a processor running an instance of auxiliary processor 200. `PEACHP` shares one variable (i.e. starts one APL2 interpreter) with each processor number. To run multiple APL2 interpreters on one processor, specify its number multiple times.

To use processors on other machines, define processor identification numbers in the APL2 SVP processor profile file. See Cooperative Processing for information about processor profiles.

If `function` identifies an external function, the processors identified by `processors` must also have access to the external function.

If `processors` is omitted, `PEACHP` uses the local instance of AP 200 and shares one variable for each CPU processor on the machine.

Usage Guidelines

- `PEACHP` and `PEACHT` provide the greatest performance improvement for algorithms that are very computation intensive. When using `PEACHP` and `PEACHT`, the arguments' elements must be copied from the caller's workspace to the asynchronous interpreters' workspaces. Depending on the size of the arguments and the type of computation being done in the function, this extra step may take more time than is saved by using multiple threads.
- Unless `processors` is supplied, `PEACHP` and `PEACHT` use one thread per CPU processor. If the machine has only one processor, `PEACHP` and `PEACHT` use only one thread and will not provide any better performance than the ¨ primitive operator, To determine whether the machine has more than one processor, use the CPUCOUNT external function.
- Use `PEACHP` and `PEACHT` for algorithms that can be performed without access to objects in the caller's workspace. Because each arguments' elements are processed in separate APL2 interpreters, access to the caller's workspace is not available to the applied function.
- `PEACHP` passes arguments and results to the asynchronous interpreters through shared memory. If not enough shared memory is available, `PEACHP` will issue a `SYSTEM LIMIT` with `⎕ET` set to `1 7`, interface capacity. See The SVP Parameter File for information about how to increase the amount of shared memory.

Example

The following example compares using the ¨ primitive operator and `PEACHP` to call the `FFT` function from the MATHFNS namespace to calculate Fast Fourier Transforms.

```
      ⍝ Determine whether the machine has more than one processor
      3 11 ⎕NA 'CPUCOUNT'
1
      CPUCOUNT
2
      DATA←⊂[2]?20 65536⍴1000
      START←⎕AI
      'MATHFNS' 11 ⎕NA 'FFT'
1
      ⍴FFT¨DATA
20
      ⎕AI-START
0 6209 6443 218
```


© Copyright IBM Corporation 1994, 2017

```
      START←⎕AI
      4 11 ⎕NA 'PEACHP'
1
      ⍴('FFT' PEACHP '')DATA
20
      ⎕AI-START
0 47 3541 203
```

The second and third elements of ⎕AI are the compute and connect times. Notice that by simply changing the code to use PEACHP rather than the ¨ primitive operator, the connect time was reduced by 45 percent. The compute time was also reduced to nearly zero, but this is only because CPU time used by the asynchronous interpreters is not accumulated in the calling interpreter's account information.

© Copyright IBM Corporation 1994, 2017

PFA - Pattern From Array

```
pattern ← PFA array
```

This function creates an array pattern from an array.

`array`
: Any APL array.

`pattern`
: A character vector containing a formalized description of the array. Array patterns are defined in Array Patterns. The patterns produced by `PFA` always contain the optional parentheses and the count field. They never contain the > or < markers or an `*`.

`PFA` may be used as a tutorial on how to write patterns. Because the types selected in the pattern reflect the internal storage mechanism in use at the moment, the same array may give different patterns at different times or on different platforms.

© Copyright IBM Corporation 1994, 2017

PRINTWSG - Print Workspace with GUI Interface

```
PRINTWSG
```

This function prompts for printing options and prints the current workspace.

To use PRINTWSG,

```
)LOAD your-workspace
3 11 ⎕NA 'PRINTWSG'
PRINTWSG
```

Note: This function is available only on Windows systems.

© Copyright IBM Corporation 1994, 2017

QNS - Query Namescope

```
result ← QNS 0
```

This function queries the current namescope.

`result`
: The left argument to `⎕NA` for the function or operator that caused entry into the current namescope.

  If the `QNS` is processed in the user's active workspace, it returns `'' 11`.

© Copyright IBM Corporation 1994, 2017

REPLACEX - Replace Strings

```
output ← (old new [old new] ... ) REPLACEX input
```

This function is a high-performance routine to replace instances of a string or strings in a character vector.

The left argument is a vector of character vectors. The items in the left arguments are processed in pairs, in order from left to right.

`input`
: A character vector.

`old`
: A character vector containing a string to be replaced wherever it is found in `input`.

`new`
: A character vector to be used to replace all instances of `old` in `input`. If `new` is null, all instances of `old` will be removed from `input`.

`output`
: The character vector resulting from the specified replacements.

Examples:

```
      3 11 ⎕NA 'REPLACEX'
1
      ('DOG' 'CAT') REPLACEX  'DOGS ARE GREAT PETS. DOGS PROVIDE FUN AND COMPANIONSHIP.'
CATS ARE GREAT PETS. CATS PROVIDE FUN AND COMPANIONSHIP.
       ('ABC' '' 'A' 'AN ' 'GE' 'G E') REPLACEX 'AABCINTABCERESTINGEVABCENABCT'
AN INTERESTING EVENT
```


© Copyright IBM Corporation 1994, 2017

ROUNDC - Commercial Rounding

```
rounded ← [power] ROUNDC numeric
```

Implicit argument: ⎕CT

ROUNDC is a high-performance routine for rounding numbers to the nearest power of ten. If a number is exactly midway between two multiples of the power of ten, it is rounded away from zero. ROUNDC is useful in commercial applications.

The right argument is any array containing only real numbers.

The left argument is a scalar integer that specifies the power of ten to which the numbers should be rounded. The default is zero. If power is positive, it indicates how many digits to the left of the decimal point are insignificant. If power is negative, it indicates how many digits to the right of the decimal point are significant.

ROUNDC is pervasive.

Examples:

```
      3 11 ⎕NA 'ROUNDC'
1
      3 ROUNDC ¯45678 ¯12345 12345 45678
¯46000 ¯12000 12000 46000
      ¯2 ROUNDC ¯1.235 ¯1.2349 1.2349 1.235
¯1.24 ¯1.23 1.23 1.24
```

Definition:

The ROUNDC external function implements the following definition:

```
      ∇
[0]   RESULT←POWER ROUNDC VALUE;MULTIPLIER;SIGN
[1]   MULTIPLIER←10*-POWER
[2]   SIGN←×VALUE
[3]   RESULT←SIGN×(⌊0.5+MULTIPLIER×SIGN×VALUE)÷MULTIPLIER
      ∇
```


© Copyright IBM Corporation 1994, 2017

RF - RowFind

```
indices ← matrix1 RF matrix2
```

This function returns the index of the first row in `matrix1` of each row of `matrix2`. The function is equivalent to the expression:

```
indices ← (⊂[1+⎕IO]matrix1)⍳⊂[1+⎕IO]matrix2
```

The arguments must be rank 2 character, integer or nested arrays that have the same number of columns. Both arguments must be the same type. If the arguments are nested, their depth must be 2 or less, and all of their subitems must be character or integer arrays.

This function may provide a performance improvement over the APL2 expression for some types of data.

© Copyright IBM Corporation 1994, 2017

RTA - Record to Array

```
array ← [pattern] RTA record
```

This function converts a character vector containing structured data into an APL array, based on a pattern that describes the format of the data. It can be used to map records or structures produced by other languages, or expected by services called through AP 145, into APL arrays of any required depth.

`record`
: A character vector containing data that was stored by some other program, in its original format. No check is made for incorrect lengths. The result is unpredictable if the record is shorter than the structure described in the pattern.

`pattern`
: A character vector containing a formalized description of the fields within the record. Array patterns are defined in Array Patterns. The pattern may not contain an asterisk (`*`) unless it can be resolved to an integer based on other information in the item description. The pattern may not contain the > mark.

  If `pattern` is omitted, the right argument is treated as an APL2 Common Data Representation (CDR) object. The CDR formats of both mainframe and workstation APL2 systems are handled.

`array`
: An APL2 array whose format is determined by the pattern, and whose content is taken from the record.

Note: External function `ATR` is the inverse of `RTA`.

© Copyright IBM Corporation 1994, 2017

SCANDIR - List Files in a Directory

```
result ← SCANDIR pathspec
```

This function returns the list of files in the specified directory. In contrast to DIR,

1. SCANDIR does not use the DOS or Unix shells to execute a DIR or ls command, so it avoids any character translations performed by those shells.
2. Filters may not be specified to subset the list of files. The complete list of files found in the directory is always returned, including any subdirectories.

`pathspec`
: A character vector containing the path to search. If a null or all-blank vector is specified, the current path will be searched.

`result`
: A character matrix containing the list of files found in the specified directory.

Example:

```
      3 11 ⎕NA 'SCANDIR'
1
      SCANDIR 'C:\MYDIR'
.
..
letter.txt
picture.jpg
program.exe
```


© Copyright IBM Corporation 1994, 2017

SI - State Indicator

```
result ← SI number
```

This function is a program interface to the APL2 system command)SI.

The state indicator is a list of

- The calling sequence of defined functions and defined operators (and their pertinent line numbers).
- Asterisk(s) for all immediate execution expressions that did not complete, either because of error in the expression or because the function invoked by the expression is pendent or suspended.

SI returns each line in the state indicator.

number
: An integer indicating the maximum number of levels of the state indicator to be returned. To return all levels of the state indicator, specify a null vector (⍳0 or '').

result
: A character matrix containing the state indicator levels requested.

Example:

```
      3 11 ⎕NA 'SI'
1
      SI ''
GN[1]
FN[2]
*
*
      SI 2
GN[1]
FN[2]
```


© Copyright IBM Corporation 1994, 2017

SIS - State Indicator with Statements

```
result ← SIS number
```

This function is a program interface to the APL2 system command `)SIS`.

The state indicator is a list of

- The calling sequence of defined functions and defined operators (and their pertinent line numbers).
- Asterisk(s) for all immediate execution expressions that did not complete, either because of error in the expression or because the function invoked by the expression is pendent or suspended.

`SIS` returns each line in the state indicator and the statement that was being executed at the time the line was added to the state indicator. Carets shown on the line below the statement indicate how much of the statement had been executed.

`number`
: An integer indicating the maximum number of levels of the state indicator to be returned. To return all levels of the state indicator, specify a null vector (`⍳0` or `''`).

`result`
: A character matrix containing the state indicator levels requested.

Example:

```
      3 11 ⎕NA 'SIS'
1
      SIS ''
GN[1]  Z←3÷0
         ^ ^
FN[2]  Z←2×GN
            ^
*   FN
    ^
*   3⍳
    ^^
```


© Copyright IBM Corporation 1994, 2017

SIZEOF - Size of Array

```
size ← SIZEOF pattern
```

This function calculates the amount of storage described by a pattern. It can be used to determine how large an array to supply to a routine which will update a parameter.

`pattern`
: A character vector containing a formalized description of the fields within an array. Array patterns are defined in Array Patterns. The pattern may not contain an asterisk (`*`) unless it can be resolved to an integer based on other information in the item description. The pattern may not contain the > mark.

`size`
: The number of bytes required to hold an array described by the pattern.

© Copyright IBM Corporation 1994, 2017

SQL Namespace

The SQL namespace enables you to execute SQL statements and retrieve information from databases using AP 127 (the DB2 processor) or AP 227 (the ODBC processor).

The namespace objects, shipped in file `SQL.ans`, are equivalent to the objects found in the SQL workspace. See SQL Workspace for an overview of the workspace. For complete information about the SQL workspace, AP 127 and AP 227, see APL2 Programming: Using Structured Query Language

Because of the large number of objects in the SQL namespace and name conflicts between its objects and other supplied external routines, nicknames for these external objects are not provided in the APL2 default names file (`aplnm011.nam`). Instead, associations to the objects can be made using the namespace file name in the left argument of `⎕NA`:

```
      'SQL' 11 ⎕NA 'QUERY'
1
      'SQL' 11 ⎕NA ⊃'SQL_AP' 'CONNECT' 'SQL'
1 1 1
```

Surrogate names may be used to avoid name conflicts with objects already in your workspace:

```
      'SQL' 11 ⎕NA ⊃'Q_SQL_AP SQL_AP' 'Q_CONNECT CONNECT' 'Q_SQL SQL'
1 1 1
      Q_SQL_AP←227
      Q_CONNECT 'DSN=APL2TEST'
0 0 0 0 0   SQL02011 APL2TEST MYUSERID
```


© Copyright IBM Corporation 1994, 2017

STA - SCAR to Array

```
array ← [translate] STA data
```

This function converts a SCAR object into an APL2 array.

`data`
: A character vector containing a SCAR object.

`translate`
: A 256-element numeric vector containing the Unicode values to use for translation of single-byte character data in the SCAR object.

  `STA` contains built-in translation tables for the character sets of APL2, Dyalog APL/W, APL+Win and SHARP APL. A translate table should only be provided if it is necessary to override the built-in translation.

`array`
: An APL2 array built from the SCAR object. If the SCAR contains data in representations not supported by APL2 (for example, 64-bit integers or enclosed scalars) data conversion will take place, and precision or structure may be lost.

The SCAR ("Self-Contained Array") format is a data interchange format defined by Insight Systems, Inc. to allow non-like APL systems to exchange data directly without the overhead of the numeric-to-character conversions required by transfer form. For more information on the SCAR format, visit `http://www.insight.dk/scardesign`.

The character vector containing the SCAR object can be received from a TCP/IP network, read from a file, or retrieved from a database. It can be sent by APL2 or another APL system which supports the SCAR format.

In APL2, `ATS` and `STA` external functions are used to create and interpret SCAR objects.

In Dyalog APL/W, the `SQAApl2Scar` and `SQLScar2Apl` external functions are used to create and interpret SCAR objects. These functions use file CNDYA30.DLL which is part of SQAPL. The DLL is loaded and initialized by function `SQAInit` in workspace SQAPL.DWS.

In APL+Win, APL functions `TOSCAR` and `FMSCAR` in workspace SCAR are used to create and interpret SCAR objects.

© Copyright IBM Corporation 1994, 2017

TCL - Tool Command Language Interface

```
[control] TCL cmd_array
```

Executes one or more Tool Command Language (Tcl) commands.

Tcl is a popular scripting language that is available for many operating systems. Tcl provides a wide variety of routines including tools for string parsing and file IO. There are also numerous extensions available including packages for building graphical user interfaces (GUI), database access, object-oriented programming, and network access.

Like APL2, Tcl is an interpreted language. A separate Tcl interpreter is started for each unique name associated with the `TCL` external function. Calls to a name are passed to its Tcl interpreter. The interpreter is deleted when the association is expunged.

`cmd_array`
: Contains one or more Tcl commands. The array can be:

  - A character vector containing one or more commands. For example:

    ```
    TCL 'set a 12'
    TCL 'set a 12;set b 34'
    ```

  - A vector of character vectors each containing one or more commands. For example:

    ```
    TCL 'set a 12' 'set b 34;set c 56'
    ```

    The `TCL` external function catenates a linefeed character to each vector, enlists the array, and passes it to the Tcl interpreter for evaluation.

  - A vector of arrays where each array is either a character vector, or a vector of Tcl command tokens. For example:

    ```
    TCL ⊂ 'set' 'a' 12
    TCL ('set' 'a' 12) ('set 'b' (34 56 78)) 'set c 587'
    ```

    The `TCL` external function passes each element of the vector of arrays to the Tcl interpreter for evaluation one at a time. If an element is a vector of tokens, the tokens are converted to Tcl objects, catenated into a list, and the list is evaluated. Any rank 0 or 1 array can be used as a command token except that complex numbers are not supported.

`control`
: Controls the kind of result returned by `TCL`.

  Every Tcl command produces a return code and some data. If a command array contains more than one command, the return code and the result of the last command in the array are returned. If the last return

© Copyright IBM Corporation 1994, 2017

code is non-zero, the data is an error message. The `control` argument controls whether this return code and data are returned to APL2. `control` can be any of these values:

0 Neither the return code nor data are returned to APL2. If the Tcl interpreter returns a non-zero return code, the data is written to the APL2 interpreter's window and a `SYSTEM LIMIT` is signaled. This is the default behavior.

1 If the return code is zero, the data is converted to an APL2 array and returned. Otherwise, the data is written to the APL2 interpreter's window and a `SYSTEM LIMIT` is signaled.

2 The data is converted to an APL2 array and the return code and data are returned to APL2 as a two element array. The first element is the return code and the second element is the data. Tcl defines the following return codes:

0 - Success, the data is the command result

1 - Error, the data is an error message

2 - The return command was encountered. The data is null.

3 - The break command was encountered. The data is null.

4 - The continue command was encountered. The data is null.

Applications can define other return codes.

`TCL` automatically converts scalar and vector results to APL2 arrays. `TCL` can convert Tcl Boolean, Integer, Double, String, Unicode, and List values to APL2 arrays. Other results are returned in a Unicode character vector representation.

The Tcl Environment

After starting a Tcl interpreter, the `TCL` external function creates several Tcl variables in the interpreter's global namespace:

argc
: The number of arguments entered on the command line when APL2 was invoked.

argv
: A list of the arguments entered on the command line when APL2 was invoked.

env
: An array of environment variable values. The name of the environment variable is the array index, eg. env(PATH), and the array element contains the environment variable value.

The `TCL` external function also defines several Tcl commands in the global namespace:

apleval
: The apleval command calls the APL2 interpreter to evaluate the arguments. apleval takes 1, 2, or 3 arguments:

1 argument	An expression to be executed
2 arguments	The name of a monadic function and a right argument.

© Copyright IBM Corporation 1994, 2017

3 arguments	A left argument, the name of a dyadic function, and a right argument.

If the evaluation produces a result, the Tcl return code will be set to TCL_OK and the APL2 result will be converted to a Tcl object and returned as the result of apleval. If the evaluation does not return a result, the Tcl return will be set to TCL_OK and apleval will return a null character string result. If an error is encountered during evaluation of the arguments, the Tcl return code will be set to TCL_ERROR and the result will be a message describing the APL2 error.

For example:

```
      ⍝ Define an addition function
      ∇RESULT←LA PLUS RA
[1]   RESULT←LA + RA∇
      ⍝ Adding two numbers works.
      ⍝ (The Tcl command expr converts the characters 3 and 5 to numbers.)
      DISPLAY 2 TCL 'apleval [expr 3] PLUS [expr 5]'
┌→──┐
│0 8│
└~──┘
      ⍝ But without expr, character vectors are passed to APL2.
      DISPLAY 2 TCL 'apleval 3 PLUS 5'
┌→──────────────────┐
│ ┌→──────────────┐ │
│1│DOMAIN ERROR   │ │
│ └───────────────┘ │
└⊂──────────────────┘
```

In the above examples, 2 was passed as the left argument of `TCL`. This caused `TCL` to return to APL2 both the Tcl return code and the command result. If 0 or 1 had been passed in the last statement, `TCL` would have signaled an error and written the DOMAIN ERROR to the interpreter window.

Note that automatic type conversion within Tcl may cause unexpected results. For example:

```
      ⍝ Display a null numeric vector
      DISPLAY ⍳0
┌⊖┐
│0│
└~┘
      ⍝ Assign the null numeric vector to a Tcl variable
      TCL⊂'set' 'a' (⍳0)
      ⍝ Display the Tcl variable.
      ⍝ Tcl converts all zero length lists to character.
      DISPLAY 1 TCL 'set $a'
┌⊖┐
│ │
└─┘
```

exit

`TCL` replaces the Tcl exit command. The `TCL` exit command does nothing. In particular, exit does not destroy the Tcl interpreter. The Tcl interpreter is destroyed when the external function is expunged.

Tk_Init

Tk is a Tcl extension package which provides support for building GUI applications. The Tk_Init command initializes the Tk environment and creates a top level window.

Customizing APL2 for Tcl

© Copyright IBM Corporation 1994, 2017

Before using the TCL external function you must install Tcl. The TCL external function supports Tcl and Tk version 8.1 and higher. After installing Tcl, you must customize the APL2 environment.

During name association, the TCL external function loads Tcl. The name of the library containing Tcl must be specified in the APL2 invocation option -tcl, by a TCL keyword definition in the [Invocation Options] section of the apl2.ini configuration file, or in the APLTCL environment variable. For example, on Windows the option might be coded as:

```
-tcl tcl84.dll
```

On Unix systems the option might be coded as:

```
-tcl libtcl8.4.so
```

If TCL is unable to load the library, the association fails.

The Tcl command Tk_Init loads Tk. To use Tk_Init, the name of the library containing Tk must also be specified in the APL2 invocation option -tk, by a TK keyword definition in the [Invocation Options] section of the apl2.ini configuration file, or in the APLTK environment variable. For example, on Windows the options might be coded as:

```
-tcl tcl84.dll -tk tk84.dll
```

On Unix systems the options might be coded as:

```
-tcl libtcl8.4.so -tk libtk8.4.so
```

If Tk_Init is unable to load the library, a Tcl error is generated.

The TCL workspace contains demonstration and utility functions.

License Information

Portions of this program were derived from Tcl source code distributed by Scriptics Corporation and were originally subject to the license shown below. The program is now subject to the IBM Program License Agreement included with APL2. The IBM license overrides the original license.

Tcl license terms:

This software is copyrighted by the Regents of the University of California, Sun Microsystems, Inc., Scriptics Corporation, and other parties. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF

© Copyright IBM Corporation 1994, 2017

THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

GOVERNMENT USE: If you are acquiring this software on behalf of the U.S. government, the Government shall have only "Restricted Rights" in the software and related documentation as defined in the Federal Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2). If you are acquiring the software on behalf of the Department of Defense, the software shall be classified as "Commercial Computer Software" and the Government shall have only "Restricted Rights" as defined in Clause 252.227-7013 (c) (1) of DFARs. Notwithstanding the foregoing, the authors grant the U.S. Government and others acting in its behalf permission to use and distribute the software in accordance with the terms specified in this license.

© Copyright IBM Corporation 1994, 2017

TIME - Application Performance Analysis

```
report ← [namelist] TIME n
```

The TIME performance monitoring facility provides the capability to measure a running application and determine the CPU time used by each defined program, each line within each defined program, or both.

The facility works by associating with each line of each specified program a pair of counters to record the number of times the line is executed and the total CPU time consumed by the line.

`namelist`
: An optional list of program names. If specified, the scope of the current operation is limited to the names listed. If not specified, the operation applies to all programs currently defined in the name space.

  The list can be a character scalar or vector containing a single name or a vector of character vectors each containing one name. All names listed must currently exist in the name space.

`n`
: An integer specifying the operation to be performed.

  `0` Enable timing and create counters for all lines in the specified programs. The counters are set to zero.

  `1` Fetch times for all specified programs that have accumulated timing information.

  `2` Fetch times for all lines of the specified programs that have accumulated timing information.

  `3` Fetch times for all lines of the specified programs even if no timing has been accumulated.

  `4` Fetch timing state

  `¯1` Enable timing. If timing has been disabled, timing is resumed. A left argument is not allowed for `TIME ¯1`.

  `¯2` Disable timing. Stops the accumulation of timing data. A left argument is not allowed for `TIME ¯2`.

  `¯3` Deletes the space used by the counters for the specified programs.

`report`
: `TIME 0`, `¯1`, `¯2` and `¯3` return an empty matrix.
  `TIME 1` returns a 4-column matrix.
  `TIME 2` and `3` return a 5-column matrix.

Column 1	Number of times the line or program was actually processed.
Column 2	Accumulated processing time (in seconds) of the line or program.
Column 3	Percentage of the total time used by the line or program.
Column 4	Name of the program
Column 5	Program line, preceded by the line number.

  `TIME 4` returns a Boolean: 0 - timing off, 1 - timing on

Notes:

© Copyright IBM Corporation 1994, 2017

1. Use of the timing facility requires space in the workspace for the counters and also increases running time by some small amount. Thus, in general you should not)SAVE after doing a time analysis.
2. Reported timings are approximate and should only be used for relative comparisons, not absolute times.
3. When a program is erased, its counters are deleted. When a program is created or changed, its counters are not preserved.

Performance Analysis Using the TIME Function

The theory behind performance tuning through the TIME facility is that very often a small amount of code within an application consumes the majority of CPU cycles. By quickly identifying these "hot spots", the programmer can focus his attention on optimizations that provide the greatest effect in reducing overall application CPU time. The following APL2 session demonstrates the use of TIME:

```
      )LOAD COSTEST
SAVED 1993-09-19 14.13.20 COSTEST
      3 11 ⎕NA 'TIME'      ⍝ Access TIME function
1
      TIME 0               ⍝ Zero time counters
      ESTIMATE 10          ⍝ Run the application
COMPLETED... SEE ¨COST_REPORT¨
      ⎕←T←TIME 1           ⍝ Fetch time summary
500 31.19    72.83469164 PRODCOST
500   7.4    17.28043341 CHARGE
 10 3.627    8.469747566 CALC
500 0.208   0.4857202905 EVAL
 10 0.196    0.457697966 PC
 10 0.103   0.2405249515 STORE
  1  0.05   0.1167596852 ESTIMATE
  1 0.044    0.102748523 TIME
 10 0.004 0.009340774817 FINDMAX
 10 0.001 0.002335193704 GETNEXT
  1     0              0 CLOSE
  1     0              0 OPEN
      +/T[;2]              ⍝ Compute total CPU time
42.823
      'PRODCOST' TIME 2    ⍝ Fetch time detail
500 31.028  99.48060276 PRODCOST[2] PCOST←SCHG∘.×⍳WEEKS
500  0.162 0.5193972427 PRODCOST[1] SCHG←CHARGE N
```

The TIME 1 report identifies the function PRODCOST as the major CPU consumer, attributable to about 73% of the total CPU time (31.19 of the total CPU 42.823 seconds for this small sample run). Further analysis of the PRODCOST function with the TIME 2 report shows that the mathematical calculation performed in line 2 is the best target for potential performance improvement.

© Copyright IBM Corporation 1994, 2017

UTFTC - UTF to Character

```
result ← [bits] UTFTC data
```

Decodes UTF-7 and UTF-8 formatted data to APL2 character data.

UTF-7 and UTF-8 are Universal Character Set Transformation Formats. They are used for transmitting Unicode data across networks which represent character data as bytes and use either 7 or 8 bits in each byte. UTF-7 and UTF-8 are described by the Unicode standard and in the Internet Request for Comments (RFC) 2152 and 2279.

`bits`
: The number of bits of significant data within individual encoded bytes. A value of 7 indicates `data` contains UTF-7 data. A value of 8 indicates `data` contains UTF-8 data. If not given, `data` is assumed to contain UTF-8 data.

`data`
: A character vector of any length containing either UTF-7 or UTF-8 data to be decoded.

`result`
: An APL2 character vector containing the decoded data.

© Copyright IBM Corporation 1994, 2017

WSCOMP - Workspace Compare

WSCOMP

Use these functions to compare workspaces.

To use WSCOMP,

```
)LOAD your-workspace
3 11 ⎕NA 'WSCOMP'
WSCOMP
```

WSCOMP prompts for the following information:

- Two workspace names. Use the same syntax as you would for the system commands)IN or)COPY. Library numbers are accepted. If specifying a complete path and file name, you must enclose it in quotes.
- Whether the workspaces named are transfer files. This option controls whether)IN or)COPY is used to access the workspace.

On UNIX, the initial prompt will also ask you what name classes you want to compare.

A report is produced showing the results of the comparison.

On Unix systems, the report takes the form of four variables containing lists of names:

WSC_ONLY_1
: Objects in workspace 1 but not in workspace 2.

WSC_ONLY_2
: Objects in workspace 2 but not in workspace 1.

WSC_DIFF
: Objects with different definitions in the two workspaces.

WSC_SAME
: Objects with the same definition in both workspaces.

The variable WSCOMP_RESULTS is also set; it contains all four lists.

On Windows, a dialog is presented with the four lists of names. Additional features include the ability to open an object's definition and dynamically modify the lists to include or remove name classes.

© Copyright IBM Corporation 1994, 2017

ZIP, UNZIP, ZIPWS, and UNZIPWS - Compression Utilities

The zip compression utilities are functions for compressing and decompressing objects and workspaces.

There are four compression utilities: ZIP, UNZIP, ZIPWS, and UNZIPWS. The ZIP and UNZIP functions are used to compress and decompress arbitrary arrays. The ZIPWS and UNZIPWS functions are used to compress and decompress the current workspace.

The zip compression utilities use the APL2 Java classes and require that Java is installed and configured for use from APL2. For more information about installing Java, see Installing Java.

```
cv←ZIP array
```

Compresses an arbitrary array and returns a character vector.

```
array←UNZIP cv
```

Uncompresses an arbitrary array from a character vector produced by ZIP.

```
ZIPWS 'name'
```

Compresses the contents of the current workspace and places the result in the variable named in the right argument. ZIPWS ignores shared variables, locked functions, and objects named ZIPWS and UNZIPWS. ZIPWS erases any user objects included in the compressed result. ZIPWS includes the current values of the following system variables in the compressed result: ⎕CT, ⎕FC, ⎕IO, ⎕LX, ⎕PP, ⎕PR, and ⎕RL. The value of ⎕LX is reset to null.

```
UNZIPWS 'name'
```

Reestablishes the definitions and associations of the objects stored by ZIPWS in the variable named in the right argument. UNZIPWS replaces any objects in the current workspace that have the same names as objects in the zipped workspace.

UNZIPWS erases the variable named in the right argument.

© Copyright IBM Corporation 1994, 2017

Auxiliary Processors

Auxiliary processors are programs that provide asynchronous interfaces between the APL2 interpreter and various system services. Shared variables are used to communicate between an APL2 program running in the interpreter and an auxiliary processor. Commands and data are passed to the processor from the program, and return codes and data are passed back to the program by the processor.

Since auxiliary processors are running asynchronously from 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. Tools to assist in setting up a connection using the proper protocol are discussed in Using the Share-Offer Utilities. For a complete description of the SVP system functions, refer to *APL2 Programming: Language Reference*.

The auxiliary processors (APs) supplied with IBM APL2 are:

- AP 100 - Host Command Processor
- AP 101 - Alternate Input (Stack) Processor
- AP 119 - Socket Interface Processor
- AP 120 - Session Manager Interface
- AP 124 - Text Display Processor
- AP 127 - DB2 Processor
- AP 144 - X Window Services Processor
- AP 145 - GUI Services Processor
- AP 200 - Calls to APL2
- AP 207 - Universal Graphics Processor
- AP 210 - File Auxiliary Processor
- AP 211 - APL2 Object Library Processor
- AP 227 - ODBC Processor
- AP 488 - GPIB Support Processor

Note: AP 144 is provided only on Unix systems. AP 120, AP 145 and AP 488 are not provided on Unix systems.

© Copyright IBM Corporation 1994, 2017

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 AP 124, AP 210 and AP 488, which 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 'C' or 'CTL' and the data variable name begins with 'D' or '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 letter 'C' (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.

SVOFFER Examples

```
⍝ 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
```


© Copyright IBM Corporation 1994, 2017

SVOPAIR Examples

```
⍝ 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
⍝ Offer improper control variable
      210 SVOPAIR 'A1' 'D1'
1 1
```


© Copyright IBM Corporation 1994, 2017

AP 100 - Host Command Processor

AP 100 is an auxiliary processor that allows operating system commands or programs to be executed. AP 100 itself imposes no specific limit on the number of concurrent shared variables that you can use. It accepts a shared variable of any name. For example:

```
      100 SVOFFER 'SHR100'
2
```

offers variable `SHR100` to AP 100 and sets access control. See Using the Share-Offer Utilities for a description of the `SVOFFER` function (from workspace 1 UTILITY).

After sharing is established, a valid operating system command or program can be executed by assigning the command to the shared variable. For example, to list the files in the current directory:

```
      SHR100←'DIR'
```

Notes:

1. If AP 100 is started automatically by the interpreter (the default), then any output generated by the execution of the command appears in the window associated with the interpreter. This window is often minimized, because it is not normally used for APL2 session input and output.
2. If an empty vector (`''` or `⍳0`) is given as the command, the shared variable returns with a character string containing the name of the operating system.
3. Commands issued through AP 100 do not affect the interpreter environment. For example, issuing a change directory command through either AP 100, or through `)HOST`, does not change the directory that your current APL2 session is using.

AP 100 Return Codes

Code	Meaning
`0`	Success
`1`	Invalid command
`444`	Invalid shared variable value

Other error codes are those returned as the exit status of the called routine.

© Copyright IBM Corporation 1994, 2017

AP 101 - Alternate Input (Stack) Processor

The alternate input (stack) processor (AP 101) can be used to create a stack of programmable input to APL2.

Conceptually, the alternate-input stack is a vector of character vectors. Each item is one stacked input line of variable length.

Although any one instance of APL2 can maintain only one stack, several shared variables can be used to add entries to the stack. Entries are stacked in first-in, first-out (FIFO) order (the default) or last-in, first-out (LIFO) order.

An input stack is normally created within a defined function. The top entry (the first item) in the stack is used when the APL2 interpreter requests input. This occurs when:

- An APL2 statement prompts for user input (⍞ or ⎕)
- The APL2 line editor (selected 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∆)

Share a variable with AP 101. For example:

```
      101 SVOFFER 'SHR101'
2
```

offers variable SHR101 to AP 101 and sets access control. See Using the Share-Offer Utilities for a description of the SVOFFER function.

After sharing is established, entries can be added to the stack by assigning the text to the shared variable (SHR101 in this example):

```
      SHR101←'any character string'
```

The string can contain new-line characters (⎕TC[⎕IO+1]), which causes each portion of the string delimited by a new-line to be treated as a separate entry on the stack.

- AP 101 Commands
- AP 101 Return Codes

AP 101 Commands

The stack can be purged of all or some entries, and the order (LIFO/FIFO) in which the entries are used can be selected with the following commands:

SHR101←0
: Purges the entire stack.

SHR101←0,n
: Purges n entries from the stack. If n>0 these are dropped from the LIFO end, and if n<0 these are dropped from the FIFO end.

© Copyright IBM Corporation 1994, 2017

`SHR101←10`
: Queries the state of the stack. Returns `1` if it is LIFO or `¯1` if it is FIFO.

`SHR101←10 1`
: Sets the stack to LIFO

`SHR101←10 ¯1`
: Sets the stack to FIFO (the default)

The maximum size of the stack is 1000 entries.

AP 101 Return Codes

Code	Meaning
`0`	Success
`12`	Stack overflow
`444`	Invalid object

© Copyright IBM Corporation 1994, 2017

AP 119 - Socket Interface Processor

The socket interface processor is used to pass requests to the Transmission Control Protocol/Internet Protocol (TCP/IP) product. TCP/IP provides communication facilities across networks. See your TCP/IP Programmer's Reference for more information about TCP/IP.

Notes:

1. Commands are passed as nested vectors. The first element of the value assigned to the variable determines the type of commands being issued. For the workstation, the command is `'TCPIP'` for commands to TCP/IP.
2. The general form of the result is a three-element vector:
   - AP 119 return code
   - TCP/IP return code or secondary AP 119 return code
   - Data returned by the command
3. No AP invocation options are required or defined.
4. Commands that reference a machine on the network refer to the machine with an *ip address* or a *domain name*. An ip address is a dotted decimal number like `'12.34.555.6'`. A domain name is a dotted string of characters like `'stl.ibm.com'`. If you use a domain name, AP119 will contact the domain name server (DNS) defined in your TCP/IP configuration to find the equivalent ip address. The ip address will work in a command even if the equivalent domain name is not known by the domain name server.

- Blocking
- Using AP 119 - The TCPIP commands
- AP 119 Return Codes
- Sample AP 119 Session

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 to receive. 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 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 non-blocking mode with the AP 119 FCNTL command. If this is done, AP 119 receives control on subsequent calls and returns the EWOULDBLOCK return code instead of blocking.

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. The examples assume that a variable named `SV119` has already been shared with AP 119.

© Copyright IBM Corporation 1994, 2017

- ACCEPT
- BIND
- CLOSE
- CONNECT
- FCNTL
- GETHOSTID
- GETHOSTBYADDR
- GETHOSTBYNAME
- GETHOSTNAME
- GETPEERNAME
- GETSOCKNAME
- GETSOCKOPT
- LISTEN
- READ
- RECV
- RECVFROM
- SELECT
- SEND
- SENDTO
- SETSOCKOPT
- SHUTDOWN
- SOCKET
- WRITE

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 the socket is in blocking mode.

Note: AP 119 imposes a limit of 256 active sockets per shared variable.

```
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:

© Copyright IBM Corporation 1994, 2017

```
      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 or domain name. 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
      SV119←'TCPIP' 'BIND' 4 1044 'stl.ibm.com'
      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:

© Copyright IBM Corporation 1994, 2017

```
      SV119←'TCPIP' 'CLOSE' 4
      SV119
0 0 0
```

CONNECT

Completes the binding necessary for a socket if `BIND` has not been 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 or domain name

`CMDRC`
: is `0`.

Example:

```
      SV119←'TCPIP' 'CONNECT' 3 1002 '9.113.14.90'
      SV119
0 0 0
      SV119←'TCPIP' 'CONNECT' 4 1022 'tac.org.de'
      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.
: The possible values for `cmd` are `'F_GETFL'` and `'F_SETFL'`.

© Copyright IBM Corporation 1994, 2017

cdata
: is the data associated with the command.
The possible values for cdata are 0 and 'FNDELAY'. See Blocking for more information on use of 'FNDELAY' to set non-blocking status for a socket.

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.

Example:

```
      SV119←'TCPIP' 'FCNTL' 3 'F_SETFL' 'FNDELAY'
      SV119
0 0 0
```

GETHOSTBYADDR

Returns the domain name for the specified ip address.

```
SV119←'TCPIP' 'GETHOSTBYADDR' ia
(APRC TCPIPRC dn)←SV119
```

Where:

dn
: is the domain name.

ia
: is the IP address.

Example:

```
      SV119←'TCPIP' 'GETHOSTBYADDR' '9.112.12.92'
      (APRC TCPIPRC CMDRC)←SV119
      CMDRC
stl.ibm.com
```

GETHOSTBYNAME

Returns the IP address for the specified host.

```
SV119←'TCPIP' 'GETHOSTBYNAME' dn
(APRC TCPIPRC ia)←SV119
```

Where:

dn

© Copyright IBM Corporation 1994, 2017

is the domain name.

`ia`
is the IP address.

Example:

```
      SV119←'TCPIP' 'GETHOSTBYNAME' 'stl.ibm.com'
      (APRC TCPIPRC CMDRC)←SV119
      CMDRC
9.112.12.92
```

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
```


© Copyright IBM Corporation 1994, 2017

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.

© Copyright IBM Corporation 1994, 2017

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.

The following table provides the options and levels that are defined for the AP 119 `GETSOCKOPT` and `SETSOCKOPT` calls. Not all values may be supported on all operating systems. TCP/IP will return an error if an unsupported option is used.

Option	**Level**
SO_BROADCAST	SOL_SOCKET
SO_DEBUG	SOL_SOCKET
SO_DONTROUTE	SOL_SOCKET
SO_ERROR	SOL_SOCKET
SO_KEEPALIVE	SOL_SOCKET
SO_LINGER	SOL_SOCKET
SO_OOBINLINE	SOL_SOCKET
SO_RCVBUF	SOL_SOCKET
SO_RCVLOWAT	SOL_SOCKET
SO_RCVTIMEO	SOL_SOCKET
SO_REUSEADDR	SOL_SOCKET
SO_SNDBUF	SOL_SOCKET
SO_SNDLOWAT	SOL_SOCKET
SO_SNDTIMEO	SOL_SOCKET

© Copyright IBM Corporation 1994, 2017

Option	Level
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 2 integers, representing the linger option on/off status and the timeout value in seconds. For all other options, the third item is a single integer.

LISTEN

Waits for a connection to a given 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.

`READ` is the same as `RECV` with `flg` set to `0`

```
SV119←'TCPIP' 'READ' sn type
(APRC TCPIPRC CMDRC)←SV119
```

Where:

© Copyright IBM Corporation 1994, 2017

sn
: is the socket number.

type
: is one of:

  'B' No conversion of data

  'E' Translate character data from EBCDIC to native format

  'A' Translate character data from ASCII to native format
  Note: Because workstation systems' native format is ASCII, no translation is done for option 'A'.

CMDRC
: is the data received from the socket. Up to 32767 bytes of data can be received in one call. If the length of the data is 0, the connection has been closed.

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 the receive option flag and is one of:

  0 No special options

  1 MSG_OOB option - reads any out-of-band data on the socket

  2 MSG_PEEK option - peeks at the data present on the socket; the data is returned but not consumed, so that a later receive operation sees the same data.

  3 Both MSG_OOB and MSG_PEEK

tp
: is the type and is one of:

  'B' No conversion of data

  'E' Translate character data from EBCDIC to native format

  'A' Translate character data from ASCII to native format
  Note: Because workstation systems' native format is ASCII, no translation is done for option 'A'.

© Copyright IBM Corporation 1994, 2017

CMDRC
: is the data received from the socket. Up to 32767 bytes of data can be received in one call. If the length of the data is 0, the connection has been closed.

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 receive option flag and is one of:

  - `0` No special options
  - `1` MSG_OOB option - reads any out-of-band data on the socket
  - `2` MSG_PEEK option - peeks at the data present on the socket; the data is returned but not consumed, so that a later receive operation sees the same data.
  - `3` Both MSG_OOB and MSG_PEEK

tp
: is the type and is one of:

  - `'B'` No conversion of data
  - `'E'` Translate character data from EBCDIC to native format
  - `'A'` Translate character data from ASCII to native format
    Note: Because workstation systems' native format is ASCII, no translation is done for option `'A'`.

dat
: is the data received from the socket. Up to 32767 bytes of data can be received in one call. If the length of the data is 0, the connection has been closed.

fam
: is the family (always `2` - AF_INET).

rp
: is the remote port.

ra
: is the remote IP address.

© Copyright IBM Corporation 1994, 2017

Example:

```
      SV119←'TCPIP' 'RECVFROM' 4 0 'B'
      (APRC TCPIPRC CMDRC)←SV119
      CMDRC
(data sent by partner) 2 1003 9.113.14.90
```

SELECT

Monitors read, write, and exception status on a set of sockets.

Control returns when something happens on one of the sockets, or when a timeout occurs.

There are two forms of the SELECT command - one that uses integer vectors of socket numbers to identify the sockets of interest and one that uses Boolean masks to identify the sockets of interest. The form with Boolean masks is closer to the standard C-socket SELECT definition. The form with integer vectors is more convenient for the APL2 programmer.

- SELECT - with Integer Vectors
- SELECT - with Boolean Masks

SELECT - with Integer Vectors

The first argument is zero. This is the indication that integer vectors are being used. The next three arguments are vectors of integers containing the socket numbers to be monitored for read, write, and exception status. For example, if the read vector contains socket number 3, this socket will be checked.

When something happens to one of the indicated sockets, the SELECT call returns three vectors of integers listing the sockets that are ready. For example, if 3 is returned in the read vector of the result, then that socket has data ready to read.

If the connection to a listening socket is completed, the socket number is returned in the read vector and an ACCEPT call can be made, even though data is not available for reading.

If your partner closes the connection, the socket is also marked ready to read, but when you read it, you will get zero bytes of data.

A socket is normally always ready to write.

The exception condition is noted if out of band data is received.

The timeout value specifies the number of seconds to wait for the call to complete. A value of zero means to wait indefinitely.

```
SV119←'TCPIP' 'SELECT' 0 rv wv xv to
(APRC TCPIPRC CMDRC)←SV119
(rv wv xv)←CMDRC
```


© Copyright IBM Corporation 1994, 2017

Where:

`rv`
: Sockets to check for ready to read status

`wv`
: Sockets to check for ready to write status

`xv`
: Sockets to check for exceptional conditions

`to`
: is the timeout value

Example:

```
      R_V←2 5 1024
      W_V←⍳0
      X_V←⍳0
      SV119←'TCPIP' 'SELECT' 0 R_V W_V X_V 0
      (APRC TCPIPRC DATA)←SV119
      DATA
 5 2
```

The variable `DATA` above is a three-item vector with (5 2) as the first item, the read vector. The second and third items, the write and exception vectors, are empty.

SELECT - with Boolean Mask

The first argument indicates how many elements of each mask vector to use. This would normally be 1 plus the largest socket number allocated. The masks must be at least as long as this value. A one in the mask specifies a corresponding socket to check.

Note: Mask positions are 0-origin. For example, if socket number 3 is to be monitored, the fourth mask bit should be set.

Three masks are returned as soon as something happens on one of the selected sockets. For example, the read mask has a one in the fourth position if socket number 3 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. If your partner closes the connection, the socket is also marked ready to read, but when you ready it, you will get zero bytes of data.

A socket is normally always ready to write.

The exception mask is set if out of band data is received.

The timeout value specifies the number of seconds to wait for the call to complete. A value of zero means to wait indefinitely.

```
SV119←'TCPIP' 'SELECT' ns rm wm xm to
(APRC TCPIPRC DATA)←SV119
(rm wm xm)←DATA
```


© Copyright IBM Corporation 1994, 2017

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 DATA)←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 have been 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 options flag and is one of:
  - `0` No special options
  - `1` MSG_OOB option - sends out-of-band data.
  - `4` MSG_DONTROUTE option - data should not be subject to routing.
  - `5` Both MSG_OOB and MSG_DONTROUTE

`tp`
: is the type and is one of:
  - `'B'` No conversion of data
  - `'E'` Translate character data from EBCDIC to native format
  - `'A'` Translate character data from ASCII to native format
    Note: Because workstation systems' native format is ASCII, no translation is done for option `'A'`.

© Copyright IBM Corporation 1994, 2017

`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 flg tp dat fm rp ra
(APRC TCPIPRC CMDRC)←SV119
```

Where:

`sn`
: is the socket number.

`flg`
: is the send options flag and is one of:
  - `0` No special options
  - `1` MSG_OOB option - sends out-of-band data.
  - `4` MSG_DONTROUTE option - data should not be subject to routing.
  - `5` Both MSG_OOB and MSG_DONTROUTE

`tp`
: is one of:
  - `'B'` No conversion of data
  - `'E'` Translate character data from EBCDIC to native format
  - `'A'` Translate character data from ASCII to native format
    Note: Because workstation systems' native format is ASCII, no translation is done for option `'A'`.

`dat`
: is 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 or domain name

`CMDRC`
: is the number of characters sent

© Copyright IBM Corporation 1994, 2017

Example:

```
      (FAM R_PORT R_ADDR)←2 '9.113.12.92' 1002
      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.

The values and levels that are defined for the `GETSOCKOPT` and `SETSOCKOPT` calls are listed in GETSOCKOPT.

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:

© Copyright IBM Corporation 1994, 2017

`sn`
: is the socket number.

`how`
: is one of:

  `0` to end communication from socket `sn`

  `1` to end communication to socket `sn`

  `2` to end communication both to and from socket `sn`

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.

Note: AP 119 imposes a limit of 256 active sockets per shared variable.

```
SV119←'TCPIP' 'SOCKET' [type]
(APRC TCPIPRC CMDRC)←SV119
```

Where:

`type`
: is the socket type desired. Valid values are `'STREAM'` for a stream socket, and `'DGRAM'` for a datagram socket. If no type is specified, a stream socket is allocated.

`CMDRC`
: is the socket number allocated.

Note: Stream sockets provide sequenced, duplex byte streams that are reliable and connection-oriented. Datagram sockets provide connectionless message exchange with no guarantees of reliability and limited message size. Datagram sockets should only be used by those with experience handling this type of communication.

Examples:

```
      SV119←'TCPIP' 'SOCKET'
      SV119
0 0 3
      SV119←'TCPIP' 'SOCKET'  'DGRAM'
      SV119
0 0 4
```

WRITE

Writes data to a given socket.

© Copyright IBM Corporation 1994, 2017

WRITE is the same as SEND with flg set to 0

```
SV119←'TCPIP' 'WRITE' sn tp dat
(APRC TCPIPRC CMDRC)←SV119
```

Where:

sn
: is the socket number.

tp
: is one of:

  'B' No conversion of data

  'E' Translate character data from EBCDIC to native format

  'A' Translate character data from ASCII to native format
  Note: Because workstation systems' native format is ASCII, no translation is done for option 'A'.

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
```

AP 119 Return Codes

Code	Meaning
0	Success
1	Incorrect command
2	Wrong type
3	Wrong rank
4	Wrong shape
5	Item is wrong type. Second element of result is zero-origin index to item in error.
6	Item is wrong rank. Second element of result is zero-origin index to item in error.
7	Item is wrong shape. Second element of result is zero-origin index to item in error.
8	Item data is wrong. Second element of result is zero-origin index to item in error.
10	AP 119 subsystem support error. Second element indicates type of error. 1 - No more sockets available through this variable 2 - Insufficient storage to process the command
11	TCP/IP error occurred. Second element of result is TCP/IP return code.

© Copyright IBM Corporation 1994, 2017

Sample AP 119 Session

In this example, User 1 on system 87.65.43.21 communicates with User 2 on system 12.34.56.78.

```
      ⍝ 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 has been 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 the 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 not successful.

```
      ⍝ 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

      ⍝ User 2 allocates a socket
      B←'TCPIP' 'SOCKET'
      B
0 0 17

      ⍝ User 2 binds the socket to a port
      B←'TCPIP' 'BIND' 17 1055 '0.0.0.0'
      B
0 0 0

      ⍝ User 2 connects to user 1
      B←'TCPIP' 'CONNECT' 17 1023 '87.65.43.21'
      B
0 0 0

      ⍝ User 1 accepts the connection
      A←'TCPIP' 'ACCEPT' 3
```


© Copyright IBM Corporation 1994, 2017

```
      A
 0 0   4 1055 12.34.56.78
```

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 'A' 'SOME DATA'
      A
0 0 9
```

The `A` means that ASCII characters are being sent. In this case, specifying type `A` is the same as type `B` since both sides are already using ASCII. If the receiving side was EBCDIC-based then type `E` on the SEND would cause translation to EBCDIC before the data is sent.

Note that you can send non-character 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' 17 0 'A'
      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
      ⍝ arrives, it is not known when a
      ⍝ receive should be done.  User 1 issues
      ⍝ a SELECT that requests that he be
      ⍝ informed when data arrives on socket 4.
      SOCKET←4
      R_MASK←W_MASK←X_MASK←(SOCKET+1)⍴0
      R_MASK[SOCKET+⎕IO]←1
      TIMEOUT←0
      A←'TCPIP' 'SELECT' (SOCKET+1) R_MASK W_MASK X_MASK TIMEOUT
```

Because the masks are numbered in 0-origin, the masks, and the number passed to specify the length of the masks, must have a length which is 1 greater than the largest socket number. In this example, the 1 in the fifth position of 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 SELECT is a blocking call, if User 1 now references `A`, execution 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.

© Copyright IBM Corporation 1994, 2017

```
      ⍝ User 2 sends some data
      B←'TCPIP' 'SEND' 17 0 'A' 'DATA BACK TO YOU'
      B
0 0 16

      ⍝ 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 'A'
      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' 17
      B
0 0 0
```


© Copyright IBM Corporation 1994, 2017

AP 120 - Session Manager Interface

Note: This processor is not provided on Unix systems.

AP 120 provides a programming interface to the APL2 Session Manager.

AP 120 imposes no limit on the names or number of concurrent shared variables. The following descriptions assume that a variable named `SV120` has been shared with AP 120. For example:

```
      120 SVOFFER 'SV120'
2
```

shares the variable `SV120` with AP 120 and sets access control. See Using the Share-Offer Utilities for a description of the `SVOFFER` function (from workspace 1 UTILITY).

After a variable is shared, assign any AP 120 command to the variable. Reference the shared variable to retrieve the command's return code and result. For example, to open an object named `Foo`:

```
      SV120←'OPEN' 'Foo'
      (RC RESULT)←SV120
      RC
0
```

Note: Most AP 120 commands simply return `''` as a result. Those which return something else are identified below.

AP 120 supports the following commands:

- CLOSE
- MESSAGE
- OPEN
- REFRESH
- WAIT
- WATCH

AP 120 uses the following return codes:

Code	Meaning
`0`	Success
`1`	Object was already open
`2`	Object is not open
`3`	No event occurred in timeout seconds
`-1`	Invalid command
`-2`	System error

CLOSE

© Copyright IBM Corporation 1994, 2017

```
SV120←'CLOSE' 'objectname'
```

The CLOSE command closes an object editor window.

Where:

`objectname`

Character vector - the name of the object to close.

Example

Close the object named Foo:

```
      SV120←'CLOSE' 'Foo'
      (RC RESULT)←SV120
      RC
0
```

MESSAGE

```
SV120←'MESSAGE' 'objectname' 'text' [beep]
```

The MESSAGE command displays a message in the session manager's or an object editor's status area and optionally sounds a beep.

Where:

`objectname`

Character vector - the name of the object editor window in which to display the message. Use `''` to display the message in the session manager window.

`text`

Character vector - the text of the message.

`beep`

Boolean scalar - 0 is silent, 1 sounds a beep.

Example

Display Hello World in the session manager's status area and sound a beep:

© Copyright IBM Corporation 1994, 2017

```
      SV120←'MESSAGE' '' 'Hello World' 1
      SV120
0
```

OPEN

```
SV120←'OPEN' 'objectname'
```

The OPEN command opens an object.

Where:

`objectname`

Character vector - the name of the object to open.

Example

Open an object named Foo.

```
      SV120←'OPEN' 'Foo'
      (RC RESULT)←SV120
```

REFRESH

```
SV120←'REFRESH' 'objectname'
```

The REFRESH command refreshes the definition shown in an object editor window.

Where:

`objectname`

Character vector - the name of the object to refresh.

Example

Refresh the object named Foo:

```
      SV120←'REFRESH' 'Foo'
      (RC RESULT)←SV120
      RC
0
```

WAIT

© Copyright IBM Corporation 1994, 2017

```
SV120←'WAIT' timeout
```

The WAIT command waits for events in object editor windows.

Where:

`timeout`

Positive real number - The number of seconds to wait.

If the WAIT command detects an event, the result is a two element array. The first element is the event name. The second element is the name of the object that signaled the event. AP 120 supports the following event names:

Event	Meaning
`'Change'`	The user changed the definition.
`'Close'`	The user closed the object editor window.
`'Open'`	The user opened an object. The object name is the requested object.
`'Refresh'`	The user refreshed the object definition.
`'Save'`	The user saved the definition.
`'Save and Close'`	The user saved the definition and closed the object editor window.

Notes

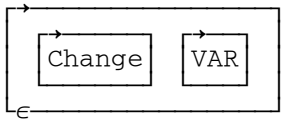
The WAIT command only returns events in object editor windows that were opened with the OPEN command using the shared variable to which the WAIT command is specified.

The session manager starts saving events as soon as the WAIT command is issued. One event is returned for each call to the WAIT command. Events are discarded when the variable shared with AP 120 is retracted.

Example

Wait for 10 seconds:

```
      SV120←'WAIT' 10
      (RC RESULT)←SV120
      RC
0
      DISPLAY RESULT
```



WATCH

© Copyright IBM Corporation 1994, 2017

The WATCH command manages the windows that monitor APL2 objects and expressions.

```
SV120←'WATCH' 'NAME' objectname
```

The NAME subcommand opens a watch window for a variable or function by name. This is equivalent to selecting Watch from the Popup Object Menu.

Where:

`objectname`

Character vector - the name of an APL2 object.

```
SV120←'WATCH' 'EXPRESSION' expression
```

The EXPRESSION subcommand opens a watch window for an APL2 expression. This is equivalent to pressing Ctrl-Enter when the cursor is on an APL2 expression.

Where:

`expression`

Character vector - an APL2 expression.

```
SV120←'WATCH' 'CLOSE' {objectname | expression}
```

The CLOSE subcommand closes an open watch window.

Where:

`objectname`

Character vector - the name of an APL2 object.

`expression`

Character vector - an APL2 expression.

The object name or expression must match exactly to the title of an open watch window.

```
SV120←'WATCH' 'CLOSEALL'
```


© Copyright IBM Corporation 1994, 2017

The CLOSEALL subcommand closes all open watch windows for the current APL2 session.

```
SV120←'WATCH' 'LIST'
(RC RESULT)←SV120
```

The LIST subcommand requests a list of all open watch windows for the current APL2 session.

`RESULT` is a character matrix with one row for each open watch window.

```
SV120←'WATCH' 'DISPLAY' {'ON' | 'OFF' | ''}
(RC RESULT)←SV120
```

The DISPLAY subcommand controls the setting **Use DISPLAY in Watch Windows**.

Where:

`'ON'`

Sets the option value on. The DISPLAY function is used in watch windows.

`'OFF'`

Sets the option value off. The DISPLAY function is not used in watch windows.

`''`

Requests the current value of the option. `RESULT` is a character vector containing `'ON'` or `'OFF'`.

```
SV120←'WATCH' 'ALWAYSONTOP' {'ON' | 'OFF' | ''}
(RC RESULT)←SV120
```

The ALWAYSONTOP subcommand controls the setting **Watch Windows Always on Top**.

Where:

`'ON'`

Sets the option value on. Watch windows are always on top.

`'OFF'`

Sets the option value off. Watch windows follow normal zorder rules.

`''`

© Copyright IBM Corporation 1994, 2017

Requests the current value of the option. `RESULT` is a character vector containing `'ON'` or `'OFF'`.

```
SV120←'WATCH' 'WSSIZE' {wssize | 'DEFAULT' | ''}
(RC RESULT)←SV120
```

The WSSIZE subcommand controls the setting **Watch Workspace Size**.

Where:

`wssize`

Character vector - workspace size to be used for watch windows' APL2 sessions. Syntax for `wssize` is the same as for the `-ws` invocation parameter to APL2.

`'DEFAULT'`

Resets the watch workspace size to the default value (currently 10m).

`''`

Requests the current value of the watch workspace size. `RESULT` is a character vector containing the workspace size in use.

Examples

Open a watch window for the variable `Foo`

```
      SV120←'WATCH' 'NAME' 'Foo'
      (RC RESULT)←SV120
      RC
0
```

Open a watch window for the expression `Foo+⍳10`

```
      SV120←'WATCH' 'EXPRESSION' 'Foo+⍳10'
      (RC RESULT)←SV120
      RC
0
```

Get a list of the currently open watch windows:

```
      SV120←'WATCH' 'LIST'
      (RC RESULT)←SV120
      RC
0
      DISPLAY RESULT
┌→──────┐
↓HOW    │
│Foo+⍳10│
│Foo    │
```


© Copyright IBM Corporation 1994, 2017

```
└──────┘
```

Close the watch window for variable HOW:

```
      SV120←'WATCH' 'CLOSE' 'HOW'
      (RC RESULT)←SV120
      RC
0
```

Close all open watch windows:

```
      SV120←'WATCH' 'CLOSEALL'
      (RC RESULT)←SV120
      RC
0
```

Query the current setting of the watch workspace size:

```
      SV120←'WATCH' 'WSSIZE' ''
      (RC RESULT)←SV120
      RC
0
      DISPLAY RESULT
┌→─┐
│10m│
└──┘
```

Turn the **Use Display in Watch Windows** setting off:

```
      SV120←'WATCH' 'DISPLAY' 'OFF'
      (RC RESULT)←SV120
      RC
0
```


© Copyright IBM Corporation 1994, 2017

AP 124 - Text Display Processor

AP 124 enables you to control a window from within an APL2 defined function. It enables your application to:

- Write to the formatted screen
- Read from the formatted screen
- Produce colors, highlighting, reverse video, and so on
- Control the keyboard translation
- Enter "Inkey" mode to monitor all keyboard activity
- Define up to 255 fields

AP 124 is discussed in detail in the following sections:

- AP 124 Operation
- Understanding Screen Management
- AP 124 Commands
- Summary of AP 124 Commands
- AP 124 Return Codes

AP 124 Operation

AP 124 requires two shared variables: a data variable and a control variable. They can be offered in any order. The name of the data variable must always begin with the letter `'D'` or the letters `'DAT'`, and the control variable must begin with the letter `'C'` or the letters `'CTL'`. The remaining characters in both names (possibly none) must be the same, because the coupling of both variables is recognized by their name. Examples of valid pairs are: `C` and `D`, `C1` and `D1`, and `CXjj` and `DXjj`. Also accepted as valid pairs (for compatibility with APL2/370) are names such as `CTL` and `DAT` or `CTL1` and `DAT1`. The control variable is used to select the operation to perform and to control each input/output operation. For example:

```
      124 SVOPAIR 'C124' 'D124'
2 2
```

offers variables `C124` and `D124` to the auxiliary processor. See Using the Share-Offer Utilities for a description of the `SVOPAIR` function (from the 1 UTILITY workspace).

AP 124 supports a single window for all operations. On most APL2 systems, AP 124 allows the user to share multiple pairs of variables, with all the variables affecting that one window. On Windows systems, AP 124 allows only one set of variables to be shared. Programs written for use on multiple operating systems should not use techniques that require more than one set of variables to be shared with AP 124.

Understanding Screen Management

To use the Text Display auxiliary processor, you should understand the screen and its attributes. This section gives an overview of how the auxiliary processor logically views the screen.

- AP 124 Usage
- Screen Size

© Copyright IBM Corporation 1994, 2017

- Screen Fields
- Field Types and Attributes

AP 124 Usage

In the following, the "physical screen" refers to what is actually displayed in the AP 124 window. The "logical screen" is a buffered image of that window stored in memory. Operations are normally performed on the logical screen. The information stored in the logical screen is transferred to the physical screen (the physical screen is refreshed) by certain calls ("Read and Wait" and "Immediate Write").

Screen Size

The physical screen (window) size can be controlled by the application program, by use of an AP 124 command code, and by the application user through standard interactive window resizing techniques supported by the system window manager.

The initial size when the window is opened is automatically determined by the number of rows and columns defined by the logical screen format array, with a minimum default of 25x80.

The size of the active window can be queried or set at any time by the application program, by use of command code 14 (see Set Window Attributes).

Screen Fields

The text display auxiliary processor views the screen in terms of rectangular areas called *screen fields*. You can enter or display data only in these areas. Each screen field has a starting location, a width, and a height that you define when you format 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 Types and Attributes

Each screen field has associated with it, a *field type* and a *field attribute* that qualify its content. For instance, a field type may indicate that a field is to contain alphabetic or numeric data or that user input to the field is to be allowed, and the field attribute may specify that the field is to be displayed as red characters on a white background.

AP 124 Commands

The following section describes each AP 124 command in detail. The commands are summarized in Summary of AP 124 Commands.

Each time the command-control variable (`C124` in the following examples) is set, AP 124 attempts to perform the requested operation and resets the command-control variable to indicate the degree of success or failure. A value of `0` indicates that the operation was successful; anything else indicates that a problem has been encountered - these values are listed in AP 124 Return Codes.

- Clear Screen
- Format the Screen into Fields

© Copyright IBM Corporation 1994, 2017

- Reformat the Screen
- Push Format Array
- Pop Format Array
- Immediate Write of Data to Screen
- Read and Wait or Read and Test
- Delayed Write of Data to Screen
- Get Data from the Logical Screen (as a Matrix)
- Update Field Types
- Update Field Attributes
- Control and Information Request
- Get Format Table
- Get the Current Logical Screen
- Sound a Beep to Alert User
- Set the Cursor
- Set Window Attributes
- Query Window Attributes
- Get Data from the Logical Screen (as a Vector)
- Get Field Attributes
- Clear Screen (VS APL Compatible)
- Set Title Bar Text
- Hide or Show Menubar

Clear Screen

Syntax:

```
C124←0
```

This request clears both the physical screen and the logical field contents on the next operation requiring a screen update.

Format the Screen into Fields

Syntax:

```
D124←numeric_format_array
C124←1
```

This call permits you to divide your logical screen into rectangular "fields". Each field is defined in terms of its offset from the top left-hand corner of the screen, its depth, and its width. You can also indicate whether the field is output only or input/output; and its display "attribute".

`numeric_format_array` is 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, a numeric vector can be passed, and 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 four elements of each row of the matrix (assigned to `D124`) are defined as:

© Copyright IBM Corporation 1994, 2017

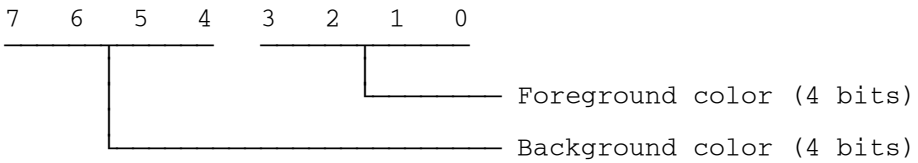
1. Start row of the field
2. Start column of the field
3. Field height
4. Field width

The fifth and sixth elements of each row of the matrix are optional and are defined as:

5. Field type:

0	Input/output/selectable
1	Numeric input only/any output/selectable
2	Output only (the default)
3	Output only/selectable

6. See the description of the call to Set the Cursor for the meaning of the "selectable" attribute.
7. Field attribute: An integer between 0 and 255. The default field attribute is 1, which normally gives blue characters on a black background. The following diagram shows the meanings of the bits of the display attribute byte on color display adapters:



The combinations of colors available are:

Code	Bits	Color
0	0 0 0 0	Black
1	0 0 0 1	Blue
2	0 0 1 0	Green
3	0 0 1 1	Cyan
4	0 1 0 0	Red
5	0 1 0 1	Magenta
6	0 1 1 0	Yellow
7	0 1 1 1	Gray
8	1 0 0 0	Light Gray
9	1 0 0 1	Light Blue
10	1 0 1 0	Light Green
11	1 0 1 1	Light Cyan
12	1 1 0 0	Light Red
13	1 1 0 1	Light Magenta
14	1 1 1 0	Brown

© Copyright IBM Corporation 1994, 2017

Code	Bits	Color
15	1 1 1 1	White

The format request clears any previously defined fields and establishes a new screen definition with one field for each row of the format array given in `D124`. These fields are given field numbers, starting from one, corresponding to the row number of the field in the format array. These field numbers are used to identify each individual field in subsequent calls to AP 124.

Example:

```
D124←1 6⍴10 5 1 6 0 7
C124←1
```

defines a field in the tenth row, fifth column, one high, six wide. The field has a type of input/output/selectable (`0`) and the attribute (`7`) specifies gray characters on a black background.

Notes:

1. A row in the format array can be all zeros. This can be used as a place holder, and the corresponding field can be defined later with a reformat screen request (`C124←1,Field_number(s)`) as described below.
2. Initially, the display screen contains only one field that covers the entire screen area.

Reformat the Screen

Syntax:

```
D124←numeric_format_array
C124←1,Field_number(s)
```

The reformat screen request modifies one or more existing field definitions. The number of rows given in the format array (`D124`) must be the same as the number of `Field_number(s)` given in the control variable (`C124`).

`numeric_format_array` is a four-, five-, or six-column numeric matrix with one row for each field to be formatted.

Note: If a field is formatted so that it overlaps another, and then is reformatted to a smaller size so that more of the overlapped field is revealed, the color of the newly revealed area will not be refreshed until its field is affected by a format, reformat or change of attribute. The text in the area will not be refreshed until its field is affected by a format or write.

Push Format Array

Syntax:

```
D124←numeric_format_array
C124←1 ¯1
```


© Copyright IBM Corporation 1994, 2017

The push format array request saves the current screen format, data and cursor type on a last-in first-out (LIFO) stack. The screen is then formatted according to the new format array, and the cursor type reset to normal.

Any areas of the screen not defined in the new format array retain their previous contents.

`numeric_format_array` is a four-, five-, or six-column numeric matrix with one row for each field to be formatted.

Pop Format Array

Syntax:

```
C124←1 ¯2      ⍝ Pop last pushed format array
C124←1 ¯2,n    ⍝ Pop last n format arrays
```

The pop format array request restores the last saved screen format, data and cursor type from the stack. If `n` is specified, this process is repeated `n` times. If the stack is empty, the current format, data and cursor type are retained and no error is given.

Immediate Write of Data to Screen

Syntax:

```
D124←array_of_character_data
C124←2,Field_number(s)
```

This call permits you to write data to the logical screen, which is then immediately transferred to the physical screen.

If a single field is being updated, the data can be passed as a character vector or matrix. If a matrix is passed, it will be treated as if it were raveled.

If multiple fields are being updated, the data can be passed as a character matrix with one row for each field, or as a nested vector with one element for each field. Each of the vector elements can be a character vector or matrix as described above for a single item.

Read and Wait or Read and Test

In normal operation, the physical screen is refreshed. Then, either the auxiliary processor waits for a certain key to be pressed, or it returns to APL2 with information on whether a key was pressed.

Syntax	Description
`C124←3` `C124←3 0`	Allow interactive input and return to APL2 when a special key is pressed. Assigning the control variable a scalar value of `3`, or the two element vector `3 0`, are completely equivalent.
`C124←3 1`	Return to APL2 when any key is pressed.
`C124←3 2`	Test for a key pressed, and return immediately.

© Copyright IBM Corporation 1994, 2017

Syntax	Description
`C124←3 3`	Return to APL2 when any key except a cursor movement key is pressed. ("Semi-inkey" mode.)
`C124←3 4`	Test for a key pressed, and return immediately. If a normal (non-special) key has been pressed, it will be echoed to the display, and the field containing it will be marked as having been updated.

If any of the above calls has three elements, this instructs the auxiliary processor not to refresh the physical screen. (The value of the third element is ignored.)

After the return code is checked in `C124`, the `D124` variable contains a vector of five or more elements. They are:

`D124[1 2]`
: For calls `3`, `(3 0)`, and `(3 4)`, a code indicating the special key pressed to return to APL. See Special Key Code Processing for the list of codes.

  For calls `(3 1)`, `(3 2)`, and `(3 3)`, the character code and extended function code returned by the APL2 keyboard routine. See Extended Key Code Processing for the list of codes.

  For call `(3 2)`, if no key has been pressed, return is `(¯1 ¯1)`. Similarly, for call `(3 4)`, if no *special* key has been pressed, return is `(¯1 ¯1)`.

`D124[3]`
: Field number where the cursor was located at return to APL2, or 0 if the cursor was not in a defined field.

`D124[4 5]`
: Cursor position (row,column) within that defined field. If field number is 0, these elements give the offset from top-left of the logical screen. Positions are given in origin one, for example, `1 1` specifies the top-left corner of the field.

`D124[6...]`
: List of fields updated during this call.

Special Key Code Processing

The codes returned for the `3`, `(3 0)`, and `(3 4)` commands are:

Code	Description
`0 0`	Enter (New-line key)
`1,N`	F-key, where `N (1 ≤ N ≤ 48)` is the number of the key that was depressed. 1-12: normal F-keys 13-24: F-keys in shift mode 25-36: F-keys in Ctrl mode 37-48: F-keys in Alt mode **Note:** On Unix systems, some of the F-keys in Alt mode (particularly Alt-F1 through Alt-F4) may be used by the desktop manager to perform windowing functions such as focus switching, minimization, and window close. For these keystroke combinations, control will not be given to AP 124, and thus no key codes will be returned to the APL2 application. For cross-system portability, it is best not to rely

© Copyright IBM Corporation 1994, 2017

Code	Description
	on these key combinations in your AP 124 applications.
`4 1`	Esc
`6 1`	Home
`6 2`	End
`6 3`	PageUp
`6 4`	PageDown
`6 5`	Ctrl-PageUp
`6 6`	Ctrl-PageDown
`6 7`	Ctrl-Left
`6 8`	Ctrl-Right
`6 9`	Ctrl-Up
`6 10`	Ctrl-Down

While AP 124 is in the (`3 0`) "Read and Wait" state, you can type on the screen. The cursor movement keys can be used in the normal way. Three other keys have special functions:

Ctrl-Backspace
: Toggles the keyboard between APL2 and national modes.

Ctrl-End
: Clears to the end of the field. However, if the field has less than 80 columns, only the current line is cleared to its end.

Ctrl-Home
: Clears from the cursor to the start of the field. However, if the field has less than 80 columns, only the current line is cleared to its beginning.

Extended Key Code Processing

The codes returned for the `(3 1)`, `(3 2)`, and `(3 3)` commands are:

- If `D124[2]` is `0` and `D124[1]` is non-zero, then `D124[1]` is the zero origin `⎕AV` index of the character corresponding to the key pressed.
- If `D124[1]` is `0`, then `D124[2]` is an extended function code:

Code	Description
`0`	Ctrl-Break or Ctrl-Backspace
`3`	Null character
`15`	Shift-Tab
`16-25`	Alt-q, w, e, r, t, y, u, i, o, p (national keyboard only)
`30-38`	Alt-a, s, d, f, g, h, j, k, l (national keyboard only)
`44-50`	Alt-z, x, c, v, b, n, m (national keyboard only)
`59-68`	Function keys 1-10
`71`	Home

© Copyright IBM Corporation 1994, 2017

Code	Description
72,73	Cursor Up, PageUp
75,77	Cursor Left, Right
79	End
80,81	Cursor Down, PageDown
82,83	Insert, Delete
84-93	Shift-Function keys 1-10
94-103	Ctrl-Function keys 1-10
104-113	Alt-Function keys 1-10 **Note:** On Unix systems, some of the F-keys in Alt mode (particularly Alt-F1 through Alt-F4) may be used by the desktop manager to perform windowing functions such as focus switching, minimization, and window close. For these keystroke combinations, control will not be given to AP 124, and thus no key codes will be returned to the APL2 application. For cross-system portability, it is best not to rely on these key combinations in your AP 124 applications.
114	Ctrl-PrintScreen
115,116	Ctrl-Cursor Left, Right
117,118	Ctrl-End, Ctrl-PageDown
119	Ctrl-Home
120-131	Alt-1, 2, 3, 4, 5, 6, 7, 8, 9, 0, -, = (national keyboard only)
132	Ctrl-PageUp
133,134	Function keys 11, 12
135,136	Shift-Function keys 11, 12
137,138	Ctrl-Function keys 11, 12
139,140	Alt-Function keys 11, 12
141,142	Ctrl-Cursor Up, Ctrl-Keypad -
143,144	Ctrl-Enter, Ctrl-Keypad +
145,146	Ctrl-Cursor Down, Ctrl-Insert
147,148	Ctrl-Delete, Ctrl-Tab
149,150	Ctrl-Keypad /, Ctrl-Keypad *
151,152	Alt-Home, Alt-Cursor Up
153	Alt-PageUp
155,157	Alt-Cursor Left, Right
159,160	Alt-End, Alt-Cursor Down
161	Alt-PageDown
162,163	Alt-Insert, Alt-Delete
164	Alt-Keypad /
165	Code 165 was originally defined in this table for the Alt-Tab keystroke. However, that keystroke is not returned to AP 124 by the operating system. The operating system uses Alt-Tab to change window focus. This behavior overrides any behavior defined by another program, including AP 124.

© Copyright IBM Corporation 1994, 2017

Code	Description
166	Alt-Enter

Delayed Write of Data to Screen

Syntax:

```
D124←array_of_character_data
C124←4,Field_number(s)
```

This call permits you to write data to the logical screen, which is displayed at the next refresh of the physical screen.

In all other respects, this service is identical to the immediate write service (see Immediate Write of Data to Screen).

Get Data from the Logical Screen (as a Matrix)

Syntax:

```
C124←5,Field_number(s)
```

This call enables you to read data from the logical screen. It returns a matrix in `D124` with one row for each data field requested and as many columns as the maximum field length (field length is the total number of characters in a field).

If a field contains more than one row, its data is raveled into a single row of the matrix. If multiple fields are requested, the data from each is padded to the length of the longest.

Update Field Types

Syntax:

```
D124←New_field_type(s)
C124←6,Field_number(s)
```

where each item of `New_field_type` is a number:

0 Field is input/output (selectable)
1 Field is numeric input only/any output (selectable)
2 Field is output only (non-selectable)
3 Field is output only (selectable)

See the description of the call to Set the Cursor for the meaning of the "selectable" attribute.

This call updates column five of the format array previously specified for the indicated fields.

© Copyright IBM Corporation 1994, 2017

Update Field Attributes

Syntax:

```
D124←Attribute(s)
C124←7,Field_number(s)
```

Each item of `Attribute` is an integer from 0 to 255 as defined in Format the Screen into Fields.

It is also possible to define the attributes for each character position of a field. To do this, the `D124` variable is set to a vector of attributes, and only a single `Field_number` is specified. A value of `¯1` in the vector of attributes specifies that this character position is to use the currently-defined field attribute.

Control and Information Request

Syntax:

```
C124←8          ⍝ Set to APL2 keyboard
C124←8 0        ⍝ Set to APL2 keyboard
C124←8 1        ⍝ Set to national keyboard
C124←8 2        ⍝ Return status
```

Returns in `D124`:

`D124[1]` Keyboard in APL2 mode
`D124[2]` Reserved (always `0`)
`D124[3]` Reserved (always `1`)
`D124[4]` Beep request pending
`D124[5]` Reserved (always `0`)
`D124[6]` Cursor mode (`0` = normal, `1` = field)

Get Format Table

Syntax:

```
C124←9
```

Returns a `n` by 6 numeric matrix in `D124`, where `n` is the current number of fields defined.

This call returns the current format array stored by AP 124.

If no format array currently exists then `D124` is set to:

```
1 1, No_of_rows, No_of_columns, 2 1
```

This feature can be used to determine the number of rows and columns that the window is capable of displaying.

© Copyright IBM Corporation 1994, 2017

Note: On Unix systems, the default format array is a dummy array that may not be used for output. A format array must be defined with command `1` before any input or output can be performed. On Windows and on the mainframe APL2 systems, the default format array can be used for output. Applications being written to run on multiple systems, however, should not take advantage of this feature.

Get the Current Logical Screen

Syntax:

```
C124←10 1
```

Returns the logical screen as a character matrix in the `D124` variable.

Sound a Beep to Alert User

Syntax:

```
C124←11        ⍝ Delayed Beep
C124←11 0      ⍝ Delayed Beep
C124←11 1      ⍝ Immediate Beep
C124←11 2      ⍝ Cancel previous delayed Beep
```

If the beep is delayed, it occurs at the next "Read and Wait" or "Read and Test" operation. To find out whether a beep is pending, specify call `8 2` and examine the fourth element.

Set the Cursor

Syntax:

```
D124←Field_no, Row_offset, Column_offset
C124←12
```

Sets the cursor to a specific screen location. `Field_no` is the number of a defined field. If it is zero, then row and column are considered as coordinates from the top left corner of the screen. `D124` must be a three element numeric vector. Positions are given in origin one, for example, `1 1` specifies the top-left corner of the field.

```
C124←12 0
```

Sets cursor type to normal (the default).

```
C124←12 1
```

Sets cursor type to field. Any fields with a field type that is "selectable" (field types 0, 1 or 3) are displayed in reverse video whenever the cursor is situated within the field.

Set Window Attributes

The following section describes the syntax for setting window attributes.

© Copyright IBM Corporation 1994, 2017

- Set Background Color
- Set Window Size and Placement

Set Background Color

Syntax:

```
C124←14,color
```

Sets the screen background (all areas of the window not currently defined by any field in the format array during the previous "reformat screen" operation) to the color specified. For example:

```
C124←14 16
```

Sets the background to blue.

Set Window Size and Placement

Syntax:

```
C124←14,row,column,height,width
```

Requests the window size and placement, through standard negotiation with the window manager. Logical fields defined totally outside the window are not displayed, and fields that exceed the window boundaries are clipped. Any parameter can be specified with a value of `¯1` to request default handling. The row and column positions are zero origin offsets from the top left corner of the Desktop window.

Query Window Attributes

Syntax:

```
C124←14
```

Returns a 5-element integer vector in `D124`:

```
background_color,row, column, height, width
```

Get Data from the Logical Screen (as a Vector)

Syntax:

```
C124←15,Field_number(s)
```

Returns a vector of arrays with as many elements as the number of fields requested. Each array element is either a character vector (if the field height is 1) or a a character matrix with dimensions equal to the field height and

© Copyright IBM Corporation 1994, 2017

width (if the field height is not 1). Returns a vector of character arrays in `D124` with as many elements as the number of fields requested.

This call enables you to read data from the logical screen.

Get Field Attributes

Syntax:

```
C124←17,Field_number(s)
```

Returns in `D124`, a character matrix (one row for each `Field_number` specified) containing the attribute character for each character position of the field. The `⎕AV` system variable can be used to convert these attribute characters to the equivalent attribute integers.

Clear Screen (VS APL Compatible)

Syntax:

```
C124←20
```

This is the same as call `0`, and is provided purely for compatibility with the VS APL AP 124.

Set Title Bar Text

Syntax:

```
D124←'text'
C124←21 1
```

Sets the title bar of the AP 124 window to the specified text.

Hide or Show Menubar

Syntax:

```
C124←22        ⍝ To hide menubar
C124←22 0      ⍝ To hide menubar
C124←22 1      ⍝ To show menubar
```

Hides or shows the AP 124 window's menubar. If command 22 is not used, the AP 124 window contains a menubar.

Note: This command is ignored on Unix systems.

Summary of AP 124 Commands

© Copyright IBM Corporation 1994, 2017

Operation	Control Variable	Data Variable	Response in Data Variable
Clear screen	0		
Reformat screen	1	format	
Push format array	1, -1	format	
Pop format array	1, -2 [,n]		
Reformat fields	1, field_nos	format	
Immediate write	2, field_nos	data	
Read and wait	3 3 0 3 1 3 3		key, cursor, field_no
Read and test	3 2 3 4		key, cursor, field_no
Delayed write	4, field_nos	data	
Get data	5, field_nos		data
Update field type	6, field_nos	type	
Update field attribute	7, field_nos	attribute	
Set APL2 keyboard	8 8 0		
Set national keyboard	8 1		
Return status	8 2		keyboard, beep, cursor
Get format table	9		format
Get the current logical screen	10 1		data
Sound delayed beep	11 11 0		
Sound immediate beep	11 1		
Cancel delayed beep	11 2		
Set the cursor	12	position	
Set cursor type to normal	12 0		
Set cursor type to field	12 1		
Query window attributes	14		attributes
Set window background color	14, color		
Set window placement and size	14, row, column, height, width		
Data as a vector of arrays	15, field_nos		
Get field attributes	17, field_nos		attributes
Clear screen	20		
Set title bar text	21 1	data	

© Copyright IBM Corporation 1994, 2017

Operation	Control Variable	Data Variable	Response in Data Variable
Hide menubar	22 22 0		
Show menubar	22 1		

AP 124 Return Codes

Code	Meaning
0	Success
11	Control variable rank error
12	Control variable length error
13	Control variable domain error
14	Invalid call
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
25	Data variable value error
26	Data variable too large
30	Invalid field number
32	Defined field extends beyond the window
37	Invalid field type
38	Invalid attribute
53	Required storage not available
69	Window has been closed by user
89	Data variable interlocked

© Copyright IBM Corporation 1994, 2017

AP 127 - DB2 Processor

AP 127 allows you to use the Structured Query Language (SQL) to access IBM DATABASE 2 (DB2).

Supplied workspace SQL is a companion to AP 127.

The tables in the next sections provide a summary of AP 127 commands with their corresponding workspace functions, and the AP 127 return code structure. See APL2 Programming: Using Structured Query Language for complete information about AP 127 and the SQL workspace.

- AP 127 Commands
- AP 127 Return Codes

AP 127 Commands

Operation Code and Syntax	Workspace Function
'CALL' *name* [*values*]	CALL
'CLOSE' *name*	CLOSE
'COMMIT' ['RELEASE']	COMMIT
'CONNECT' *database-identifier*	CONNECT
'DECLARE' *name* ['HOLD'|'NOHOLD']	DECLARE
'DESCRIBE' *name* [*type*]	DESC
'EXEC' *stmt*	EXEC
'FETCH' *name* [*options..*]	FETCH
'GETOPT'	GETOPT
'ISOL' [*setting*]	ISOL
'MSG' *rcode*	MESSAGE
'NAMES'	NAMES
'OPEN' *name* [*values*]	OPEN
'PREP' *name stmt*	PREP
'PURGE' *name*	PURGE
'PUT' *name values*	PUT
'ROLLBACK' ['RELEASE']	ROLLBACK
'SETOPT' *options..*	SETOPT
'SQLCA'	SQLCA
'SQLSTATE'	SQLSTATE
'SSID' [*subsystem*]	SSID
'STATE' *name*	STATE
'STMT' *name*	STMT

© Copyright IBM Corporation 1994, 2017

Operation Code and Syntax	Workspace Function
'TRACE' [*(module level)..*]	TRACE
'XMLSIZE' [*size*]	XMLSIZE

AP 127 Return Codes

Return Code Vector	Meaning
0 0 0 0 0	Normal return. All operations completed. Result table retrieved by a FETCH request is complete.
0 0 1 0 0	Normal return, but a result table may not have been completely retrieved.
1 0 0 1 *msgn*	Error in auxiliary processor. *msgn* is the number of the auxiliary processor error message.
1 0 0 2 *msgn*	Error detected in the database system. *msgn* gives the SQL return code (SQLCODE).
1 0 0 3 *msgn*	Error detected in an SQL workspace function. *msgn* gives the message number.
0 1 0 *n msgn*	Warning message. For example, FETCH has no more rows to retrieve, a DELETE statement deletes nothing, or the value-list is longer than the highest vector index.
1 1 0 *n msgn*	Transaction backout. All changes made to tables since the last COMMIT or ROLLBACK have been discarded. Application must restore processing to point of last COMMIT or ROLLBACK. All locks are released and all cursors closed.

© Copyright IBM Corporation 1994, 2017

AP 144 - The X Window System Interface Processor

Note: This processor is provided only on Unix systems.

AP 144 provides an interface to the X Window System library (Xlib). It enables a very large set of the X Window System Xlib calls and data structures to be used from the APL2 environment, and in so doing, enables APL2 to use a true windowing environment.

Before using AP 144 to exploit X Window System support within your application, it is recommended that you first study the AP 207 interface, to determine if it satisfies your windowing requirements. AP 207 offers the advantage of a high-level portable graphics interface and provides more extensive error handling capability. While AP 144 offers an APL2 application programmer the extensive set of Xlib services available to programmers writing windowing applications in other languages, it uses a *pass through* command interface to the subroutine library, which is a much lower-level programming interface than that offered by AP 207.

To use AP 144, you must be able to read and write programs in APL2 and C, to use the underlying operating system's facilities, and should have a good general knowledge of X Window System.

Since AP 144 uses a fast-path shared variable interface, it operates much like a subroutine call to the Xlib functions. This implies that the APL2 application must ensure the validity of structures and parameters it passes to the Xlib routines - AP 144 does a minimum amount of auditing of this data. Since many of the subroutines use pointers as arguments, it is possible for the APL2 application to pass parameters to an Xlib routine that look like valid addresses to AP 144, but can cause the Xlib subroutine to fail, and, if the error is severe enough, can even abort the APL2 session.

AP 144 itself imposes no specific limit on the number of concurrent shared variables that you may use. A variable can be shared with AP 144 using the `SVOFFER` function from the 1 UTILITY workspace. For example:

```
    144 SVOFFER 'SHR144'
2
```

offers variable `SHR144` to AP 144 and sets access control. See Using the Share-Offer Utilities for a description of the `SVOFFER` function.

After sharing is established, X Window System calls can be made by assigning a command and any appropriate parameters to the variable `SHR144`:

```
      SHR144←'command' [parm] [parm] ...
```

`command`
: The name of the X Window System call to be invoked, specified as an APL2 character vector. The name is case-sensitive and must be given exactly as required.

`[parm]`
: All but a few of the X Window System calls require additional input parameters to be specified. These are given after the name of the call itself, in the same order as listed in the X Window System documentation.

© Copyright IBM Corporation 1994, 2017

To reference the result of the call:

```
      (rc [results])←SHR144
```

`rc`
: The operation return code

`[results]`
: Results include the X Window System explicit result (if any), as well as any implicit results passed back as output parameters given on the call.

Note It is recommended that AP 144 commands are passed to the AP using the `XWIN` function supplied in the AP144 workspace. This simplifies the conversion of APL2 X Window System applications to use APL2 to X Window System interfaces other than that provided by AP 144.

For a tutorial on the use of the AP 144 X Window System interface, see Using the X Window System Interface.

- AP 144 Commands and Structures
- AP 144 Return Codes

AP 144 Commands and Structures

X Window System Xlib Commands

The list of available X Window System Xlib commands can be generated using the AP 144 system command

```
      SHR144←')Cmds' 'Xlib'.
```

The result that is returned in the shared variable for the `')Cmds'` command is a two item vector. The first item of this is a scalar return code (which should be `0`), and the second item is a nested `n` by `1` array containing `n` nested three column matrices (`n=1` for the case given above of a single parameter of `'Xlib'`). The first column of each matrix contains the names of the supported X Window System Xlib commands. The second and third columns of the matrix identify the types of the input and output parameters (respectively) that are used with the call.

The following table explains the character-type codes:

Type	Category	Description
B B1 B8 I I2 I4	Booleans Integers	Used for integer parameters. Pointers are often in this category, as are the various X Window System identifiers. On input, specify these as integers or Booleans. The interface attempts to convert any input or output parameters to the type requested.
E8	Floating Point	Used for floating point parameters. The interface attempts to convert any input or output parameters to the type requested.
S	Strings	Used for character strings, (*char* * in C terms). Specify as a simple-character vector.
C	Characters	Used for single characters. Specify as a character scalar.

© Copyright IBM Corporation 1994, 2017

Type	Category	Description
X X2 X4	Hex Flags	Used by X Window System flag parameters that typically modify the behavior of a specific function. Specify either as a number, as a hexadecimal character string, or as a Boolean vector. On return, this type code results in an integer (packed bits) being returned.
P	Pointers	Indicates that the given parameter is a pointer to a structure that also is defined in (and thus available via) the interface. Specify pointers as for integers described above.
G	Arrays	On input, accept anything and store it unchanged. The called function can then perform whatever additional verification is needed. On output, results are specified by the called program.

The following table shows the parameter type code prefix and suffix characters:

Type	Category	Description
_	Ignore one	This parameter is ignored. This type code is used on output for those X Window System calls that don't produce any explicit results, but do pass back information as output parameters.
?	Optional (prefix)	The parameter following is an optional one. If it is specified, it must match the type given.
⋆	Indirection (prefix)	One or more ⋆ codes can be used to indicate parameter indirection. This is equivalent to the use of a "*" when defining variables in C.
...	Repeat last (suffix)	This code can only be used at the very end of the list of codes. It is an indication that the last definition is repeated to accommodate the number of incoming parameters.
[] ;[n]	Array Indexes (suffix)	Array indexes can be specified with or without dimensions. If without, that dimension is unbound and accepts any number of elements. If with, the dimensions of the incoming data must match the specified dimension. Multiple dimensions can be specified by separating each with a ";", (for example: [1;2;3]).
⋆	Ignore rest (suffix)	This code can only be used at the very end of the list of codes. It indicates that parameters can follow the ones specified in the command so far and are to be ignored.

The interface performs any conversions needed such as conversion of Booleans to integers or vice versa.

The last digit in certain of the type codes specifies the number of bytes (or bits for the Boolean B types) used to store the value.

X Window System Xlib Structures

The list of available X Window System Xlib structures can be generated using the AP 144 system command

```
      SHR144←')Structs' 'Xlib'
```

The following Xlib XEvent structures are defined:

```
XAllowEvent             XDestroyWindowEvent  XMapRequestEvent
XAnyEvent               XErrorEvent          XMotionEvent
```


© Copyright IBM Corporation 1994, 2017

```
XButtonEvent           XEvent               XNoExposeEvent
XButtonPressedEvent    XExposeEvent         XPropertyEvent
XButtonReleasedEvent   XFocusChangeEvent    XReparentEvent
XCirculateEvent        XGraphicsExposeEvent XResizeRequestEvent
XCirculateRequestEvent XGravityEvent        XSelectionClearEvent
XClientMessageEvent    XKeyEvent            XSelectionEvent
XColormapEvent         XKeymapEvent         XSelectionRequestEvent
XConfigureEvent        XKeyPressedEvent     XUnmapEvent
XConfigureRequestEvent XKeyReleasedEvent    XVisibilityEvent
XCreateWindowEvent     XMapEvent
XCrossingEvent         XMappingEvent
```

The following Xlib regular structures are defined:

```
XAtoms         XHost               XScreenSaver
XCharStruct    XHostAddress        XSetWindowAttributes
XClassHint     XIconSize           XSizeHints
XColor         XImage              XStandardColormap
XComposeStatus XInputFocus         XTextItem
XCursorFont    XKeyboardControl    XTextProperty
XFontStruct    XKeysym             XVisualInfo
XGCValues      XModifierKeymap     XWindow
XGeneral       XPixmapFormatValues XWindowChanges
XGeom          XProperty           XWMHints
XGrab          XRectangle
```

Structure Commands - Summary

AP 144 structure commands are case-sensitive and must be entered exactly as given. The available commands are:

Command	Usage and Syntax

Clear	Clear a structure instance
	
	`SHR144←'Clear' struct handle`
	`rc←SHR144`

Get	Import an APL2 vector from a structure
	
	`SHR144←'Get' struct handle`
	`(rc vals)←SHR144`

GetConst	Get a structure's constants
	
	`SHR144←'GetConst' struct`
	`(rc const)←SHR144`

GetFields	Get a structure's fields
	
	`SHR144←'GetFields' struct`
	`(rc fields)←SHR144`

GetSize	Get the byte size of a structure
	
	`SHR144←'GetSize' struct`
	`(rc size)←SHR144`

MClear	Clear multiple structure instances at once `SHR144←'MClear' struct handle start count`

© Copyright IBM Corporation 1994, 2017

Command	Usage and Syntax
	`rc←SHR144`

MFree	Free multiple structure instances at once
	`SHR144←'MFree' struct handle count`
	`rc←SHR144`

MGet	Import multiple structures into APL2
	`SHR144←'MGet' struct handle start count`
	`(rc vals)←SHR144`

MNew	Create multiple new abutting structure instances
	`SHR144←'MNew' struct count`
	`(rc handle)←SHR144`

MPut	Export multiple structures from APL2
	`SHR144←'MPut' struct handle start array ...`
	`rc←SHR144`

New	Create a new structure instance
	`SHR144←'New' struct`
	`(rc handle)←SHR144`

NewPut	Create a new structure instance and fill it with data
	`SHR144←'NewPut' struct values`
	`(rc handle)←SHR144`

Put	Copy an APL2 nested array to a structure instance
	`SHR144←'Put' struct handle array`
	`rc←SHR144`

SFree	Free a structure instance
	`SHR144←'SFree' struct handle`
	`rc←SHR144`

System Commands - Summary

AP 144 system commands are case sensitive and must be entered exactly as given. The available commands are:

Command	Usage and Syntax

)Cmds	List the available commands
	`SHR144←')Cmds' [env] ...`
	`(rc cmds)←SHR144`

)Env Get	Get the current or default list of environments
	`SHR144←')Env' 'Get' ['Default']`
	`(rc envs)←SHR144`

)Env Set	Set a new list of environments
	`SHR144←')Env' 'Set' env ...`
	`rc←SHR144`

)RC	List a return code message

© Copyright IBM Corporation 1994, 2017

Command	Usage and Syntax
	SHR144←')RC' rc_no (rc erc)←SHR144 (rc_no name msg)←erc
)Structs	List the available structures SHR144←')Structs' [env] ... (rc structs)←SHR144
)Syntax	List the syntax of a specific command SHR144←')Syntax' cmd (rc syntax)←SHR144 (env cmd intypes outtypes)←syntax
)Version	List the AP 144 version identifier SHR144←')Version' (rc version)←SHR144

Understanding the Return Codes

Each call to the command interface results in a return code being generated and passed back to the caller. A nonzero return code is an indication that the command failed to execute for some reason or another.

If a nonzero return code is returned, and the error is found in the content of the parameter list, additional return codes can be returned in the second return parameter. These parameter return codes relate one-to-one to the parameters passed. For example, the success of the first parameter (the command name) is indicated by the first parameter code. These return codes help you locate the source of the error. Some examples may clarify this. In general, the command processor tries to provide a return code for each parameter specified or required:

```
      SHR144←'XOpenDisplay'
      SHR144
 16   0 16     ⍝ 16:  Expected parameter missing
      SHR144←'XOpenDisplay' 'first' 'second'
      SHR144
 17   0 0 17   ⍝ 17:  Too many parameters specified
```

AP 144 Return Codes

The text describing each valid return code can be retrieved from AP 144 by using the AP 144 `)RC` system command described above. The return codes are:

Code	Meaning
0	Success
11	Parameter has invalid type
12	Unknown command or structure specified
13	Unknown environment specified
14	Command not implemented
15	Multiple errors have occurred
16	Expected parameter missing

© Copyright IBM Corporation 1994, 2017

Code	Meaning
17	Too many parameters specified
21	Dimension incorrect
22	Parameter must be a vector
23	Invalid length of data
24	Invalid parameter specified
25	Invalid character in parameter
26	Cannot convert data type
27	Parameter type cannot fit value
28	APL service routine error
29	Unknown structure member or constant specified
31	AP 144 internal data structure is not owned, modify is prohibited
32	Unknown class specified
33	Invalid null pointer
34	Insufficient space left in argument space to store or retrieve data
41	Parameter does not contain any data
42	AP 144 internal data structure cannot be converted to result; prototype element missing
43	Unknown return type specified
44	Error building result
51	Type code parse error
52	Requested type not found
91	Exceeded internal limitation
92	Cannot open connection to external environment
99	Fatal error during command execution

© Copyright IBM Corporation 1994, 2017

AP 145 - GUI Services Processor

Note: This processor is not provided on Unix systems.

AP 145 provides facilities for building GUI applications, printing, and communication with other applications using Dynamic Data Exchange (DDE).

For information about building GUI applications and AP 145's services, consult *APL2 Programming: Developing GUI Applications*.

For information about printing, consult Printing Functions in the GUITOOLS chapter.

For information about DDE, consult DDESHARE - High-Level DDE Access.

© Copyright IBM Corporation 1994, 2017

AP 200 - Calls to APL2

AP 200 is used to control additional APL2 interpreter sessions (known as *slave* interpreters). Its commands allow you start and stop an APL2 slave interpreter session, create and manage objects within the slave session, and execute expressions and functions in the slave session.

The slave sessions controlled by AP 200 run asynchronously in a separate process from the user's session. This allow the user a great deal of flexibility in starting multiple sessions and allowing long-running computations to run in the background without suspending the main user session.

For more information on this interface, see Calling APL2 from APL2.

- AP 200 Commands
- AP 200 Return Codes

Notes:

1. Each variable shared with AP 200 is used to control a single APL2 interpreter session. Multiple variables may be shared if multiple sessions are required.
2. Commands are passed as nested vectors. The first element of the value assigned to the variable is a character command. Additional parameters may be passed, as defined for the command.
3. The general form of the result is a two-element vector:
   - AP 200 return code
   - Result data as defined by the command, or null if the return code is non-zero or no result is defined for the command.
4. No AP invocation options are required or defined.

AP 200 Commands

The following sections describe the syntax of the AP 200 commands. The examples assume that a variable named `SV200` has already been shared with AP 200.

- START
- STOP
- PUT
- GET
- FREE
- EXECUTE
- EXTOKEN

START

Starts an APL2 interpreter session.

```
SV200 ← 'START' [options]
APRC ← ↑SV200
```


© Copyright IBM Corporation 1994, 2017

Where:

`options`
: are invocation parameters, passed as a vector of character vectors with one element for each blank-delimited item in a command line parameter list. Any APL2 invocation parameters may be passed, as defined in Invoking APL2. However, the following parameters will be ignored if specified:

```
-hostwin
-input
-lx (is always OFF)
-quiet (is always ON)
-run
-rns
-sm (is always OFF)
-svplisten
-svptrace
```

The shared variable processor options `-svplisten` and `-svptrace` can be enabled by passing the parameters to the master APL2 session or by setting environment variables APLSVPLISTEN and APLSVPTRACE before starting APL2. On Windows, the menu options in The SVP Monitor Facility or the SVPLISTEN and SVPTRACE keywords in the [Invocation Options] section of the `apl2.ini` configuration file may also be used.

To accept the default invocation settings for APL2, pass any null array for `options`, or omit `options`.

Note:

If a session is already running in association with this shared variable, it is terminated and a new session is started. To run multiple sessions simultaneously, share multiple variables with AP 200.

Example:

```
      SV200←'START' ('-ws' '20m')
      SV200
 0
```

STOP

Stops the APL2 session.

```
SV200 ← 'STOP'
APRC ← ↑SV200
```

Example:

```
      SV200←'STOP'
      SV200
 0
```


© Copyright IBM Corporation 1994, 2017

PUT

Establishes an unnamed object in the interpreter session's workspace.

```
SV200 ← 'PUT' array
(APRC atoken) ← SV200
```

Where:

`array`
: is the array to be established.

`atoken`
: is the identifier token of the array in the slave session. This token must be used to refer to the array on subsequent `GET`, `FREE` and `EXTOKEN` commands.

Example:

```
      SV200←'PUT' ('A CHARACTER ARRAY')
      SV200
0 11
```

GET

Gets the value of an unnamed array from the interpreter session's workspace.

```
SV200 ← 'GET' atoken
(APRC array) ← SV200
```

Where:

`atoken`
: is identifier token of the array (from `PUT`).

`array`
: is the value of the array

Example:

```
      SV200←'GET' 11
      SV200
 0 A CHARACTER ARRAY
```

FREE

Removes an unnamed array from the interpreter session's workspace.

© Copyright IBM Corporation 1994, 2017

```
SV200←'FREE' atoken
APRC ← ↑SV200
```

Where:

`atoken`
is identifier token of the array (from `PUT`).

Example:

```
      SV200←'FREE' 11
      SV200
 0
```

EXECUTE

Executes a function or expression in the slave session.

```
SV200 ← 'EXECUTE' [left] expression [right]
(APRC RESULT) ← SV200
(codes ind result) ← RESULT
```

Where:

`expression`
is a character scalar or vector containing either a complete APL expression or the name of a function. If a function, it can be a defined function, system function, primitive function, or the assignment arrow. If the function is the assignment arrow, the left argument is the name of the object to be assigned, and the right argument is the value to be assigned to it.

`right`
is the APL array to be used as the right argument to the specified function.

`left`
is the APL array to be used as the left argument to the specified function.

`codes`
are the error codes (`⎕ET`) from the execution of the function or expression.

`ind`
is a Boolean indicator of whether a result was created.

`result`
If `codes` is `0 0` and `ind` is `1`, the result from execution of the function or expression.
If `codes` is not `0 0` and `ind` is `1`, a character matrix containing the error message and execution stack at the time of the error.
Otherwise, a null character vector.

Notes:

© Copyright IBM Corporation 1994, 2017

APRC is an indicator of errors discovered by AP 200 only. Even if APRC is 0, it is possible that an APL error occurred during the execution of the function or expression. In that case, the error is indicated by non-zero values in codes.

The expression or function name and arguments are all passed directly to this service. The corresponding objects for these parameters are automatically created in the slave session's workspace and the result, if any, is automatically retrieved. Any objects handled automatically are deleted after the operation is complete.

Example:

```
      SV200←'EXECUTE' '⎕WA'
      SV200
 0    0 0  1 20964272
      SV200←'EXECUTE' (10 20 30) '+' (5 10 15)
 0    0 0  1  15 30 45
      SV200←'EXECUTE' (10 20 30) '+' 'ABC'
      SV200
 0    5 4  1 DOMAIN ERROR
```

EXTOKEN

Executes a function or expression in the slave session using object identifier tokens.

```
SV200 ← 'EXTOKEN' [ltoken] etoken [rtoken]
(APRC RESULT) ← SV200
(codes ind atoken) ← RESULT
```

Where:

etoken
: is the identifier token of a character scalar or vector containing either a complete APL expression or the name of a function. If a function, it can be a defined function, system function, primitive function, or the assignment arrow. If the function is the assignment arrow, the left argument is the name of the object to be assigned, and the right argument is the value to be assigned to it.

rtoken
: is the identifier token of the APL array to be used as the right argument to the specified function.

left
: is the identifier token of the APL array to be used as the left argument to the specified function.

codes
: are the error codes (⎕ET) from the execution of the function or expression.

ind
: is a Boolean indicator of whether a result was created.

result
: If codes is 0 0 and ind is 1, the identifier token of the result from execution of the function or expression.
: If codes is not 0 0 and ind is 1, the identifier token of a character matrix containing the error message and execution stack at the time of the error.
: Otherwise, a null character vector.

Notes:

© Copyright IBM Corporation 1994, 2017

APRC is an indicator of errors discovered by AP 200 only. Even if APRC is 0, it is possible that an APL error occurred during the execution of the function or expression. In that case, the error is indicated by non-zero values in codes.

The expression or function name and arguments are all passed to this service as locator tokens. This means that these objects must exist in the workspace prior to calling the service. If executing an expression, there must be a character object containing the expression. If executing a function, there must be a character object containing the function name. These objects can be created in the workspace by use of the PUT command.

Example:

```
      SV200←'PUT' (10 20 30)
      SV200
0 11
      SV200←'PUT' (5 10 15)
0 13
      SV200←'PUT' '+'
0 14
      SV200←'EXTOKEN' 11 14 13
 0   0 0  1  19
      SV200←'GET' 19
      SV200
 0   0 0  1  15 30 45
      SV200←'FREE' 19
 0
```

AP 200 Return Codes

Code	Meaning
0	Success
¯1	APL2 session not active
¯2	Shared variable rank error
¯3	Shared variable length error
¯4	Invalid command
¯5	Invalid object token
¯6	Invalid invocation parameter
¯7	Error starting APL2
¯8	No space in shared memory
¯9	No space in slave session
¯10	No space for AP 200 work areas
¯11	AP 200 internal error

© Copyright IBM Corporation 1994, 2017

AP 207 - Universal Graphics Processor

AP 207 enables you to generate fast, high-quality graphics and text, easily and quickly. It accepts an easy-to-learn command syntax that has been designed for both simplicity of use and high performance. It has built-in support for operating system window services, giving high-quality graphics combined with the flexibility of a powerful window management facility. The following basic functions are supported:

- Representation of graphic information either as line segments or as higher-level structures, such as arcs or sectors of a circle
- Generation of text characters in various sizes and orientations, using external image, vector, PostScript or TrueType fonts.
- Definition of a world coordinate system
- Definition of clipping and scissoring rectangles

AP 207 is described in the following sections:

- AP 207 Interface
- AP 207 Commands
- AP 207 Programming Techniques
- AP 207 Return Codes

AP 207 Interface

AP 207 works with a single shared variable to control each graphics window. There are no special name restrictions for the shared variables. For example:

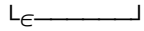
```
      207 SVOFFER 'SHR207'
2
```

offers variable `SHR207` to AP 207 and sets access control. See Using the Share-Offer Utilities for a description of the `SVOFFER` function.

Once the variable has been shared and the appropriate access control established, commands can then be passed to AP 207 in the form of a two-element vector, the first element of which is the name of the command and the second contains the parameters required by the command. For example:

```
      SHR207←'OPEN' 0
      DISPLAY ⎕←SHR207
 0  1
┌→────────┐
│ 0  ┌→┐  │
│    └~┘  │
└∊────────┘
      SHR207←'DRAW' (1 100 100)
      DISPLAY ⎕←SHR207
 0
┌→────────┐
│ 0  ┌⊖┐  │
│    └~┘  │
```


© Copyright IBM Corporation 1994, 2017



This opens an AP 207 window, and then issues command DRAW with parameters `1 100 100`.

Most commands return a two-element vector. The first element is the numeric return code for the call, and the second contains any parameters returned by the call (as an enclosed array). Wherever possible, if a parameter is given outside of the valid range of values, the closest possible available value is chosen.

Additional notes:

1. You can issue more than one command in a single call. This is explained later, in Multiple-Call Sequences.
2. Command names can be entered in any combination of uppercase or lowercase letters.
3. Each shared variable controls a single graphics window.
4. AP 207 may run as a global auxiliary processor.
5. On Unix systems, AP 207 allows generation of PostScript output through use of a special driver. Details of how to load and use the PostScript driver are discussed in Generating PostScript Output. Special notes about the PostScript driver are also included in the descriptions of the individual AP 207 commands where appropriate.

AP 207 Commands

Command	Description
ARC	Draw arc
BEGAREA	Begin filled area
BITMAP	Read/write bitmap files
BOX	Draw box
CLEAR or ERASE	Clear the graphics window
CLOSE	Close device
COLMAP	Define color mapping
COLOR	Set color
CURSOR	Sets the mouse cursor image
DRAW	Draw a line
ENDAREA	End filled area
ESCAPE	Send escape sequence to device
FONT	Select font characteristics
FONTDEF	Define font
GRDATA	Query, show, or create formatted graphics data
IMAGE	Display or read an image
LINETYPE	Set line type
LOAD	Load a device driver
MARKER	Write marker
MENU	Issue a menu command
MIX	Set mix mode

© Copyright IBM Corporation 1994, 2017

Command	Description
MOVE	Move to new point
OPEN	Open a device
PALETTE	Set color palette
PATTERN	Set area fill pattern
POINT	Activate pointer device
PRINT	Print graphics window
QUERY	Query driver parameters
QWRITE	Query position and size of string
SCISSOR	Set scissoring rectangle
SECTOR or WEDGE	Draw a sector
SETPEL	Set one or more pels on
USE	Select a driver
VIEW	View graphics
WAIT	Wait for an event
WINDOW	Define window
WRITE	Write a character string

ARC

Description:

Draws an arc, an ellipse, or a circle.

If used between BEGAREA and ENDAREA calls, each arc is filled independently.

On all operating systems except Windows, the current position is set to the end point of the arc. On Windows, ARC does not change the current position.

Syntax:

```
C←'ARC' (x y x_rad y_rad start_ang end_ang)
```

`x,y`
: give the center of the arc.

`x_rad,y_rad`
: give the radius of the arc along the x and y axes.

`start_ang,end_ang`
: give the starting and ending angles of the arc in degrees.

BEGAREA

Description:

© Copyright IBM Corporation 1994, 2017

Start collecting points to define an area to fill. The ENDAREA call ends the collection of points and fills any polygons defined since the previous BEGAREA call.

The following commands are valid between the BEGAREA and ENDAREA commands:

- ARC
- BOX
- COLOR
- DRAW
- MOVE
- PATTERN
- SECTOR
- WRITE

Other commands should not be used, as results are unpredictable.

If an area is already open when BEGAREA is called, the open area will be ended and a new area started. The COLOR, PATTERN and WRITE commands cause the area to be closed and re-opened during their processing.

Syntax:

```
C←'BEGAREA' ''
```

BITMAP

Description:

This command reads and writes Device-Independent Bitmap (DIB) files.

Note: This command is not portable across platforms. It is not implemented on Unix systems.

Syntax:

```
C←'BITMAP' 'filespec'
```

Reads the bitmap file and returns the size in pels of the bitmap as defined in the bitmap file.

```
C←'BITMAP' ('filespec' width height)
```

Writes the bitmap to the window at the current position. The bitmap is scaled to fit the width and height requested. Colors are mapped to the logical color table on a best fit basis. Returns the size of the bitmap as defined in the bitmap file.

```
C←'BITMAP' ('filespec' xleft ybottom xright ytop)
```

Writes an area in world coordinates to the filespec. The area bounded by the world coordinates must be visible on the screen as the data is read directly from the display. The resulting size and resolution of the bitmap is dependent on the current window scale. For completely repeatable results, the window size, the world

© Copyright IBM Corporation 1994, 2017

coordinates, and the view coordinates should all be the same. Returns a character array comparable to the IMAGE read command.

BOX

Description:

This command draws a rectangular box.
The current position is not altered.

Syntax:

```
C←'BOX' (xcorner ycorner)
```

Draws a rectangular box with square corners from the current position to the corner at `xcorner`, `ycorner`.

```
C←'BOX' (xcorner ycorner haxis vaxis)
```

Draws a rectangular box from the current position to the corner at `xcorner`, `ycorner` with the corners rounded by the ellipse on the horizontal full axis of `haxis` and the vertical full axis of `vaxis`. Values of `0`,`0` for `haxis`,`vaxis` give a box with square corners. Values of `xcorner`,`ycorner` for `haxis`,`vaxis` give a completely rounded box.

CLEAR or ERASE

Description:

Clears the graphics window. The window is filled with the specified color or, if none is specified, the default background color.

If an area has been opened with BEGAREA, it is closed. All other properties of the window (colors, cursor type, font, line type, marker, mix mode, pattern, position, scissoring rectangle, and window viewport) are preserved. To clear the window and reset to the default properties, use OPEN.

Syntax:

```
C←'CLEAR' 'color_name'
```

`color_name` is one of:

BACKGROUND	GREEN	DCYAN
BLUE	DGRAY (or DGREY)	MAGENTA
BLACK	DGREEN	NEUTRAL
BROWN	DMAGENTA	RED
CYAN	DRED	WHITE
DBLUE	YELLOW	GRAY (or GREY)

© Copyright IBM Corporation 1994, 2017

The closest matching color from the color map array is selected. Any color beginning with a "D" is a dark version of the specified color. Returns the color map row number used.

```
C←'CLEAR' color_no
```

`color_no` is 0 to 255 and specifies the zero origin row number of the color map array, as defined by the COLMAP call. Returns the color map row number used.

```
C←'CLEAR' (r,g,b)
```

The window is filled with the color that is the best possible match available in the current color map array. `r`, `g`, and `b` are numbers in the range 0 to 1000 and specify the desired level of red, green, and blue. Returns the color map row number used.

```
C←'CLEAR' ('')
```

The window is filled with the default background color. Returns the color map row number used.

PostScript Driver:

The CLEAR command causes any graphics from previous commands to be printed on the current page. Graphics generated after the invocation of the CLEAR command appear on a new page. The background color named (if any) is also set on the new page, except where the driver has been opened in black and white mode. In this case the background of the printed page is always white. Since the default background color is white instead of black, you must explicitly specify black as the parameter to the CLEAR command to get a black background.

CLOSE

Description:

Close the active device.

Syntax:

```
C←'CLOSE' ('')
```

Closes the currently active device.

PostScript Driver:

This command is necessary to indicate to PostScript that all graphics generated for the current page are complete and that at actual print time, the painted marks saved in PostScript memory are sent to the output device for printing. If the close command is *not* specified, few (if any) printed marks appear when the file is printed.

COLMAP

Description:

© Copyright IBM Corporation 1994, 2017

Defines the mapping between the color numbers used with the COLOR call and the actual palette entry used. The `r`, `g`, and `b` values are numbers in the range 0 to 1000 and specify the desired level of red, green and blue.

The first row of this table (index 0) defines the background color.

Syntax:

```
C←'COLMAP' ((n,3)⍴r1,g1,b1,...,rn,gn,bn)
```

Sets the color map table for entries 0 to `n-1` with the color that is the best possible match available in the current palette.

```
C←'COLMAP' ((n,4)⍴index1,r1,g1,b1,...,indexn,rn,gn,bn)
```

Sets the color map table for the entries specified in the first column with the color that is the best possible match available in the current palette.

```
C←'COLMAP' (index,r,g,b)
```

Sets the color map table entry `index` with the color that is the best possible match available in the current palette.

```
C←'COLMAP' (index 'color_name')
```

Sets the color map table entry `index` to the closest matching color available in the current palette. `color_name` is one of:

BACKGROUND	GREEN	DCYAN
BLUE	DGRAY (or DGREY)	MAGENTA
BLACK	DGREEN	NEUTRAL
BROWN	DMAGENTA	RED
CYAN	DRED	WHITE
DBLUE	YELLOW	GRAY (or GREY)

```
C←'COLMAP' index
```

Returns the current setting for the color map table entry index as a 1-by-5 matrix.

```
C←'COLMAP' ¯1
```

Resets the color map table to match the current palette colors.

```
C←'COLMAP' ('')
```

Returns the current color map table as a 256-by-5 matrix, each row of which contains the color map table row index, the corresponding row of the palette table and the r, g, b values for this row.

Note: The null parameter may not be used in a multiple-call sequence.

© Copyright IBM Corporation 1994, 2017

COLOR

Description:

Sets a new foreground color.

If issued within a BEGAREA-ENDAREA sequence, this call forces the polygon defined by the points already established to be filled with the previous color, and any new points are filled with the new color.

Syntax:

```
C←'COLOR' 'color_name'
```

`color_name` is one of:

BACKGROUND	GREEN	DCYAN
BLUE	DGRAY (or DGREY)	MAGENTA
BLACK	DGREEN	NEUTRAL
BROWN	DMAGENTA	RED
CYAN	DRED	WHITE
DBLUE	YELLOW	GRAY (or GREY)

The closest matching color from the color map is selected. Any color beginning with a "D" is a dark version of the specified color. Returns the color map row number selected.

```
C←'COLOR' color_no
```

`color_no` is 0 to 255 and specifies the zero origin row number of the color map array, as defined by the COLMAP call. Returns the color map row number selected.

```
C←'COLOR' (r,g,b)
```

Sets the active foreground color to the best possible match available in the current color map. `r`, `g`, and `b` are numbers in the range 0 to 1000 and specify the desired level of red, green, and blue. Returns the color map row number selected.

```
C←'COLOR' ('')
```

Returns the current color map row number of the active foreground color.

PostScript Driver:

If the driver is opened in black and white mode, all colors are converted to black and are not represented as colors or greyscales. In greyscale mode, colors are represented as their greyscale equivalents, like a black and white television shows color pictures. In all modes of the PostScript driver, the color BACKGROUND is white and the color NEUTRAL is black. Using BACKGROUND and NEUTRAL instead of BLACK and WHITE creates APL2 code that is usually more portable between drivers.

© Copyright IBM Corporation 1994, 2017

CURSOR

Description:

Sets the mouse cursor image

Syntax:

```
C←'CURSOR' cursor
```

Selects the mouse cursor image. `cursor` is a number from 0 to 13. A value of 0 gives the default cursor image. A value of 1 gives the cross-hair cursor image. Values from 2 to 13 give additional alternate cursor images which are operating system dependent.

```
C←'CURSOR' ('')
```

Queries the current cursor image.

PostScript Driver:

This command is not supported under the PostScript driver.

DRAW

Description:

Move or draw to a specified point.
The line is drawn with the active foreground color and current linetype.
The current position is set to the last coordinate given.

Syntax:

```
C←'DRAW' (x,y)
```

Draws a line from the current position to (`x`,`y`).

```
C←'DRAW' (md,x,y)
```

Moves to (`x`,`y`) if `md` is 0, or draws a line to (`x`,`y`) if `md` is 1.

```
C←'DRAW' ((n,2)⍴x1,y1,...,xn,yn)
```

Performs a move to (`x1`,`y1`), and then draws to each (`x`,`y`) coordinate given in subsequent rows of the matrix.

```
C←'DRAW' ((n,3)⍴md1,x1,x1,...,mdn,xn,yn)
```

Performs successive moves (if `md` is 0) or draws (if `md` is 1) to the `x` and `y` coordinates given in each row of the matrix.

© Copyright IBM Corporation 1994, 2017

ENDAREA

Description:

Signal the end of a polygon to fill. This ends the collection of points and fills any polygons defined since the previous BEGAREA call.

Syntax:

```
C←'ENDAREA' ('')
```

ESCAPE

Description:

Send a device-driver-dependent string to the driver. An application using this is not portable between systems.

Syntax:

```
C←'ESCAPE' (parameters)
```

As defined by device driver. Currently, the only driver for which escape sequences are defined is the PostScript driver.

PostScript Driver:

ESCAPE can be used to insert single new lines of text at the end of the currently open print file. These lines can be lines containing PostScript commands or lines of comments, in the PostScript language format. For example:

```
      SHR207←'ESCAPE' '%% A valid PostScript comment'
```

adds the line:

```
%% A valid PostScript comment
```

to the end of the opened print file. Line feed characters are automatically added to the end of the parameter string.

FONT

Description:

Specifies the characteristics of a font. Sets the font direction, size, horizontal alignment, vertical alignment, and various other parameters.

Syntax:

```
C←'FONT' ('fontname' size angle xjust yjust direction shear xmag noparse)
```

```
fontname
```


© Copyright IBM Corporation 1994, 2017

name of the font as defined with the FONTDEF call, or `0` to select the default font.

`size`
uppercase character height in world coordinate system units (ignored for image font for display; on printers, this is scaled depending on the print driver)

`angle`
specifies the text baseline angle (must be `0` for an image font). `angle` is not supported for Image fonts on Windows.

`xjust`
`0` (left justify), `1` (center) or `2` (right justify)

`yjust`
`0` (character baseline), `1` (center), `2` (uppercase character top), `3` (character box bottom), or `4` (character box top). `1` and `2` are not supported for Image and TrueType fonts on Windows.

`direction`
(reserved - set to `0`)

`shear`
angle (in degrees) to shear vector characters in order to produce effects such as italics (must be `0` for image font). `shear` is not supported for Image and TrueType fonts on Windows.

`xmag`
magnification to be applied to x-axis of characters to produce effects like stretched characters (must be `1` for image font). `xmag` is not supported for Image and TrueType fonts on Windows.

`noparse`
(reserved - set to `0`).

Any parameters that are not supplied default to `0`, except `xmag` defaults to `1` and `size` defaults to the size defined by the font.

Many parameters have no effect on image characters. Portable applications should use vector fonts since these can be reproduced consistently on various devices.

```
C←'FONT' ('')
```

Returns current font parameters. This has four elements in addition to those defined above:

`box_height`
the total character box height

`descender_height`
the distance from the baseline to the bottom of the character box

`max_width`
the maximum width of any character in the font

`font_type`
`0` for an image font, `1` for a vector font, `2` for a PostScript or TrueType font

Note: The null parameter may not be used in a multiple-call sequence.

PostScript Driver:

The `fontname` parameter may be set to `0` to use a Courier PostScript font in place of image fonts. As with image fonts, the `size`, `shear` and `xmag` parameters are not supported; however, vector fonts are fully supported. Portable applications should use vector fonts, because these can be reproduced consistently on various devices.

© Copyright IBM Corporation 1994, 2017

FONTDEF

Description:

Manages the AP 207 font table.

Syntax:

```
C←'FONTDEF' ('fontname' 'filename')
```

Adds a font to the AP 207 font table.

`fontname`
: is the name by which you will refer to the font in subsequent `FONT` commands. It can be any character string up to 12 characters in length.

`filename`
: is the font locator. For APL2 vector fonts, this is the name of the file which contains the font, including the .AVF extension (for example, `'GOTENG.AVF'`). For system fonts (Image, Postscript, or TrueType), it is the face name of the font (for example, `'Courier APL2'`).

System fonts must have been previously installed into the operating system font palette in order to be used. APL2 vector fonts will be loaded from the file specified. The path used to search for vector font files can be defined by passing invocation parameter `-207fl` when starting APL2, by a 207FL keyword definition in the [Invocation Options] section of the `apl2.ini` file, or in the APL207FL environment variable. If a path is not specified by any of those methods, the default path, `/usr/APL2/fonts/vector/` on Unix systems or `ibmapl2w\fonts\` on Windows, is used. On Windows, if the file is still not found, the standard operating system search order is used.

Notes:

1. The `FONTDEF` command can be used to load bitmap fonts. However, most bitmap fonts only contain display resolution bitmaps. When printing on Windows, AP 207 substitutes Courier APL2 for bitmap fonts which do not contain bitmaps matching the printer's resolution.
2. When you define a font, AP 207 discards any previous definition of `fontname`. When the same font name is defined multiple times in the same picture, the behavior of the Windows and Unix versions of AP 207 differ. On Windows, the new definition takes effect immediately. On Unix systems, the new definition does not take effect until a `FONT` command is issued. For more predictable results, select a unique name for each font.

```
C←'FONTDEF' ('fontname' ¯1)
```

Removes `fontname` from AP 207 font table. If `fontname` is the active font, the default font will be selected as the new active font. The default font cannot be removed.

```
C←'FONTDEF' ('')
```

Returns current AP 207 font table. This is returned as three character arrays:

© Copyright IBM Corporation 1994, 2017

1. Font status - a two-column matrix. The first column contains "A" if the font is active or a blank otherwise. The second column contains "V" for a vector font, "I" for an image font, "P" for a PostScript or TrueType font, or a blank if no font is defined for this row.
2. A matrix of font names (as specified on FONTDEF call).
3. A matrix of file names.

The first row of the font table describes the default font.

Note: The null parameter may not be used in a multiple-call sequence.

PostScript Driver:

Names can be defined only for vector font files. Image files are not supported.

GRDATA

Description:

Query, show, or create formatted graphics data.

Note: The GRDATA command is based in part on the work of the Independent JPEG Group. If any application using GRDATA is distributed, then the application's accompanying documentation must state that "this software is based in part on the work of the Independent JPEG Group".

Syntax:

```
C←'GRDATA' ('type' data)
```

Returns the width and height in pixels of the formatted picture stored in `data`. `type` identifies how the picture was formatted.

```
C←'GRDATA' ('type' x y width height)
```

Returns a formatted picture of the area at position `x y`. `width` and `height` specify the width and height of the area to be returned. `x`, `y`, `width`, and `height` are in world coordinates. `type` identifies how the picture should be formatted.

The size of the formatted picture corresponds to the physical size, in pixels, of the area described by the world coordinates. On Windows, user changes to the window size affect the size of the picture that is returned.

PostScript Driver:

Retrieving formatted pictures is not supported when using the PostScript driver.

```
C←'GRDATA' ('type' x y width height data)
```

Displays the formatted picture stored in `data` at position `x y`. The picture is scaled to fill an area `width` wide and `height` tall. `x`, `y`, `width`, and `height` are in world coordinates. `type` identifies how the picture was formatted. `data` is a character vector containing a formatted picture.

© Copyright IBM Corporation 1994, 2017

The following table lists the valid values of `type` The values are case insensitive. Next to each value, extensions are listed that are commonly used for files containing that type of picture.

type	File Extensions
Bitmap	BMP VGA BGA RLE DIB RL4 RL8
GemRas	IMG
GIF	GIF
Greymap	PGM
ILBM	IFF LBM
JPEG	JPE JPG JPEG
PCX	PCC PCX
PSEG	PSE PSEG PSEG38PP PSEG3820
Pixmap	PPM
Sprite	SPR SPRITE
TIFF	TIF TIFF
Targa	TGA VST AFI
XBitmap	XBM
YUV12C	VID

Example:

```
⍝ Associate a name with the FILE external function
3 11 ⎕NA 'FILE'
⍝ Read a picture from a GIF file
PICTURE←FILE 'c:\sample.gif'
⍝ Query the size of the GIF picture
C←'GRDATA' ('GIF' PICTURE)
(RC SIZE)←C
⍝ Display the GIF picture at 0 0 scaled to 649 by 479
C←'GRDATA' ('GIF' 0 0 649 479 PICTURE)
⍝ Retrieve a portion of the window formatted as a JPEG.
C←'GRDATA' ('JPEG' 0 0 100 100)
(RC NEW_PICTURE)←C
⍝ Write the picture to a JPEG file.
NEW_PICTURE FILE 'c:\sample.jpg'
```

IMAGE

Description:

Display a character image array, or read an image.

Syntax:

```
C←'IMAGE' (x1,y1,x2,y2)
```


© Copyright IBM Corporation 1994, 2017

Reads an image within the rectangle defined by the world coordinates `x1`, `y1`, `x2`, and `y2`. The coordinates identify either pair of diagonally-opposite corners. The result is a character matrix. Each character is `⎕AF` of the index of the color map row that is the closest match to the color of the corresponding pel. The shape of the matrix corresponds to the physical size, in pixels, of the rectangle defined by the world coordinates. User changes to the window size affect this shape.

Note: The read form of IMAGE may not be used in a multiple-call sequence.

```
C←'IMAGE' IMAGE
```

Displays an image. The lower-left corner of the image is placed at the current position, as set by a previous move call. `IMAGE` is a character matrix. Each character is `⎕AF` of the index of a color map row in zero index origin. You receive an error return code if you specify color indexes that are not defined in the color map.

On Unix systems, the number of rows and columns in `IMAGE` are used as the pixel height and width of the displayed image.

On Windows, the number of rows and columns in `IMAGE` are used as the world coordinate system height and width of the displayed image. To control the size of the displayed image, use WINDOW to set the world coordinate viewport. For example:

```
⍝ Create an image using a 4000 by 3000 world coordinate viewport
C←'OPEN' (0 'Image Demo' 640 480)
C←'WINDOW' (0 0 639 479 0 0 4000 3000)
C←'DRAW' (5 2⍴500 500 500 2500 3500 2500 3500 500 500 500)
C←'IMAGE' (0 0 3999 2999)
(RC IMAGE)←C
⍝ Use an adjusted viewport to fill the window with the image
C←'WINDOW' (0 0 639 479 0 0,¯1+⌽⍴IMAGE)
C←'MOVE' (0 0) 'IMAGE' IMAGE
```

PostScript Driver:

There are several driver modes that are selected with the OPEN command. Using drivers 1 and 2, the greyscale modes, color image data is rendered in its greyscale equivalent, similar to a non-color newspaper photo. Using color modes 3 and 4, color image data is printed in color. With black and white modes 5 and 6, all image data is printed in black.

The ability to read images is not defined for the PostScript driver.

LINETYPE

Description:

Set the line attributes.

Syntax:

```
C←'LINETYPE' (width,style)
```

`width`

© Copyright IBM Corporation 1994, 2017

gives the width of the line. Default: 1 pixel wide
`width` is a number between 0 and the number of widths given by the QUERY call.

`style`
gives the style of the line. Default: solid
`style` is a number between 0 and the number of styles given by the QUERY call.

A value of `0` for either attribute gives the default value. Returns the selected line type parameters.

```
C←'LINETYPE' ('')
```

Returns the current line type attributes.

Notes:

1. On Windows, a `width` value of `0` or `1` results in lines 1 pixel wide. All other values produce lines that are scaled as the window is resized. The maximum line width is given by the QUERY call.

   Scaled lines are `width÷2` pixels wide when the window is the width and height specified in the `OPEN` command. If the window has been resized, line widths are scaled to maintain the same sizes relative to the window.

   Most modern printers have such high resolution that single pixel wide lines are barely visible. Scaled width lines are usually more legible when printed.

2. On Unix systems, the line widths are defined with fixed sizes. The number of available line widths is given by the QUERY call.
3. LINETYPE is an implicit argument for WRITE of APL Vector fonts.

LOAD

Description:

Load a device driver into the auxiliary processor.

Note: This command is ignored under Windows.

Syntax:

```
C←'LOAD' device_name
```

Loads the device driver with name `device_name`. Returns the device handle (an integer value that can be used by the USE call) assigned to the driver.

```
C←'LOAD' ('')
```

Returns the device names of the available drivers, along with their device handles or a `¯1` if the driver is available but not loaded.

Currently, the only alternate driver available for Unix systems is the PostScript driver (`'PS'`).

© Copyright IBM Corporation 1994, 2017

MARKER

Description:

Write one or more markers centered at given positions.

Syntax:

```
C←'MARKER' marker
```

Selects marker type. `marker` is `1` to maximum. A value of `0` gives the default marker that is a cross. The number of available markers is given by the QUERY call. Returns the marker selected.

```
C←'MARKER' (x,y)
```

Writes current marker centered at position (`x`,`y`).

```
C←'MARKER' ((n,2)⍴x1,y1,...,xn,yn)
```

Writes current marker centered at the `x` and `y` coordinates given by each row of the matrix.

```
C←'MARKER' ((n,3)⍴md1,x1,y1,...,mdn,xn,yn)
```

Writes current marker centered at the `x` and `y` coordinates of the ends of each line segment that would be drawn by a DRAW call with the same array. (`md` indicates move or draw. A marker is drawn on all rows where `md` is `1` and on rows where `md`=0 precedes a row where `md`=1.)

```
C←'MARKER' ('')
```

Returns the current marker type.

MENU

Description:

Issues a menu command.

Note: This command is not portable across platforms. It is only implemented on Windows systems.

Syntax:

```
C←'MENU' 'OPEN'
```

Displays the Open dialog.

```
C←'MENU' 'SAVE AS'
```

Displays the Save As dialog.

```
C←'MENU' 'PRINTER SETUP'
```


© Copyright IBM Corporation 1994, 2017

Displays the Print Setup dialog.

```
C←'MENU' ('PRINTER SETUP' 'FILENAME' filename)
```

Sets or queries the output filename for the `'MENU' 'PRINT'` command.

`filename` should be a filename or `''` to query the current output filename.

Returns the current output filename.

Note: The setting from this command does not affect menubar actions and dialogs.

```
C←'MENU' ('PRINTER SETUP' 'ORIENTATION' orientation)
```

Sets or queries the printer orientation.

Valid values for `orientation` are:

`'LANDSCAPE'`	Specifies the image should be printed with its top along the long edge of the paper.
`'PORTRAIT'`	Specifies the image should be printed with its top along the short edge of the paper.
`''`	Queries the printer orientation.

Returns the current printer orientation.

```
C←'MENU' ('PRINTER SETUP' 'OUTPUT' output)
```

Sets or queries the output target for the `'MENU' 'PRINT'` command.

Valid values for `output` are:

`'FILE'`	Specifies the printer driver should write to the current output filename.
`'PRINTER'`	Specifies the output should be sent to the currently selected printer.
`''`	Queries the output target.

Returns the current output target.

Note: The setting from this command does not affect menubar actions and dialogs.

```
C←'MENU' ('PRINTER SETUP' 'PRINTERS' '')
```

Returns the names of the available printers.

```
C←'MENU' ('PRINTER SETUP' 'PRINTER' '')
```

Queries the current printer.

```
C←'MENU' ('PRINTER SETUP' 'PRINTER' printer)
```


© Copyright IBM Corporation 1994, 2017

Sets the current printer. `printer` is a character vector containing the name of a printer.

```
C←'MENU' 'PRINT'
```

Prints the contents of the graphics area.

```
C←'MENU' 'COPY'
```

Copies the contents of the graphics area to the clipboard.

```
C←'MENU' 'PASTE'
```

Pastes the contents of the clipboard into the graphics area.

MIX

Description:

Sets the color mixing mode.

Notes:

1. The colors resulting from using color mix modes other than 0 are operating system dependent.
2. On Windows, MIX does not affect text written using image, PostScript, or TrueType fonts and output sent to PostScript printers.

Syntax:

```
C←'MIX' 0
```

Pels to be written directly replace existing pels. This is the default. Returns the mixing mode selected.

```
C←'MIX' 1
```

Pels to be written are ANDed with existing pels. Returns the mixing mode selected.

```
C←'MIX' 2
```

Pels to be written are ORed with existing pels. This gives a color mixing effect. Returns the mixing mode selected.

```
C←'MIX' 3
```

Pels to be written are exclusive-ORed with existing pels. This can be used to move objects around, since writing the same object back into the same position in this mode restores the original screen, making the object disappear. Returns the mixing mode selected.

```
C←'MIX' ('')
```

Returns the current mixing mode.

© Copyright IBM Corporation 1994, 2017

PostScript Driver:

This command is not supported under the PostScript driver.

MOVE

Description:

Move to a specified point. The current position is set to the coordinate given. No line is drawn.

Syntax:

```
C←'MOVE' (x,y)
```

Moves to (x,y). Returns the new x,y position.

```
C←'MOVE' ('')
```

Returns the current x,y position.

OPEN

Description:

Open the window into a state of readiness for graphics.

An implicit CLOSE is performed if an open device exists.

Syntax:

```
C←'OPEN' (video_mode 'title' width height xpos ypos)
```

Opens a graphics window with a titlebar and sizing border.

`video_mode`
: is the mode of the video device driver and is a numeric value.
  - `0` Opens the driver in default mode, which is `1`.
    This mode is preferred since this allows AP 207 code to be portable between different implementations.
  - `1` Opens the driver in single-byte character mode.
    Text input and output, if any, uses single-byte characters.
  - `¯1` (Windows Only) Opens the driver in Unicode mode.
    Text input and output uses the Unicode character set. See Support for Double-byte Characters for more information.

`title`
: is the name that appears on the title bar of the window and in the icon for this window. If not specified, the title is "AP207".

`width` and `height`

© Copyright IBM Corporation 1994, 2017

specify the width and height of the initial window in device units (pels) and default to `640` and `480` respectively.

`xpos` and `ypos`
specify the initial x and y position of the window. If these are not given, interactive window placement is used.

```
C←'OPEN' (handle id width height xpos ypos)
```

Opens a graphics window within the client area of a window created through AP 145.

`handle`
is a window handle returned by either of the functions `CREATEDLG` or `WINDOWFROMID`.

`id`
is a control window identifier. If `id` is omitted or is zero, the graphics window is opened within the window identified by `handle`. If `id` is non-zero, the graphics window is opened within the child window whose parent is `handle` and whose identifier is `id`.

`width` and `height`
specify the width and height of the graphic window in pixels. The default size fills the client area.

`xpos` and `ypos`
specify the position of the window within the client area. The default position is the origin.

When a graphic window is opened in a Unicode dialog, AP 207 Unicode support is automatically enabled.

When a graphic window is opened in a dialog the WAIT and POINT commands do not return tab and shift-tab keystrokes. Dialog tab key processing is performed instead.

Graphic windows support the CONTEXT HELP property. The graphic window has identifier 32776 (FID_CLIENT). If a graphic window is opened within a control window, the control window's contextual help text will be displayed if no contextual help has been supplied for the graphic window.

Note: Opening a graphic window within an AP 145 window is only supported on Windows.

```
C←'OPEN' (mode 'filename')
```

Opens the PostScript driver. OPEN has several modes to handle different printers and page orientations.

`mode`
is the only required parameter and is a numeric value from 0-6.

- Mode 0 opens the driver in default mode, which is 1.
- Modes 1, 3 and 5 are landscape modes, which means that all graphics are printed with the page in "landscape" or sideways orientation.
- Modes 2, 4 and 6 cause graphics to be printed in "portrait" format.
- Modes 1 and 2 are greyscale modes, meaning that on black and white printers all color values are converted to their greyscale equivalents before printing. All images are printed as greyscales.
- Modes 3 and 4 are similar to modes 1 and 2 except that PostScript color image commands are used to represent images. Note that many non-color PostScript printers fail with PostScript interpreter errors in modes 3 and 4.

© Copyright IBM Corporation 1994, 2017

- Modes 5 and 6 are purely black and white modes where all color marks are converted to black prior to printing.

`'filename'`
: is used to specify the file used to collect PostScript commands. If not specified, a filename of `PRN.207` is used. If the file already exists, it is overwritten from the beginning.

Note: Opening the PostScript driver is only supported on Unix systems.

```
C←'OPEN' ('')
```

Returns the current video mode or window handle. Returns `0` if an OPEN has not yet been issued.

PALETTE

Description:

Sets the color palette lookup table. Each row of this table defines the color to be displayed for a particular entry in the color map table. Each entry is the intensity level of red, green, and blue, expressed as a number in the range 0 to 1000 (0 = off, 1000 = maximum intensity.)

Initially, the color map table matches the palette lookup table. The color table for displaying an image can be set using the COLMAP command. Doing this, however, limits your color selection to the 256 colors currently loaded in the palette. Alternatively, use of the PALETTE command gives much more control over color selection, enabling you to select 256 colors from millions of possible color shades on some displays.

Notes:

1. On Unix systems, the PALETTE command affects the colors shown in other windows on the same display.

Syntax:

```
C←'PALETTE' ((n,3)⍴r1,g1,b1,...,rn,gn,bn)
```

Sets the palette table for entries 0 to `n-1`.

```
C←'PALETTE' ((n,4)⍴index1,r1,g1,b1,...,indexn,rn,gn,bn)
```

Sets the palette table for the entries specified in the first column.

```
C←'PALETTE' (index,r,g,b)
```

Sets the palette table entry `index`.

```
C←'PALETTE' (index 'color_name')
```

Sets the palette table entry `index` to the closest matching color. `color_name` is one of:

© Copyright IBM Corporation 1994, 2017

BACKGROUND	GREEN	DCYAN
BLUE	DGRAY (or DGREY)	MAGENTA
BLACK	DGREEN	NEUTRAL
BROWN	DMAGENTA	RED
CYAN	DRED	WHITE
DBLUE	YELLOW	GRAY (or GREY)

```
C←'PALETTE' index
```

Returns the current setting for the palette table entry `index` as a 1-by-4 matrix.

```
C←'PALETTE' ¯1
```

Resets the palette table back to the default. The color map table is also reset.

```
C←'PALETTE' ('')
```

Returns the current palette table as an n-by-4 matrix, each row of which contains the palette row index, followed by the r, g, b values for this row.

Note: The null parameter may not be used in a multiple-call sequence.

PATTERN

Description:

Sets the active fill pattern.

If issued within a BEGAREA-ENDAREA sequence, this call forces the polygon defined by the points already established to be filled with the previous pattern and any new points are filled with the new pattern.

Syntax:

```
C←'PATTERN' pattern
```

Sets the active fill pattern. `pattern` is `1` to maximum. A value of `0` gives the default fill pattern that is a solid fill. The number of available patterns is given by the QUERY call. Returns the pattern selected.

```
C←'PATTERN' ('')
```

Returns the current fill pattern number.

PostScript Driver:

The PATTERN command is supported for black and white modes 5 and 6. In modes 5 and 6, calls to PATTERN create fill patterns using different shades of grey. In other modes, area fills use the current color setting, which is converted to greyscale on non-color printers. In all modes, using the PATTERN command to

© Copyright IBM Corporation 1994, 2017

change the fill pattern will cause a fill of any unfilled polygon. The driver mode determines if the fill is performed with the previous fill pattern, or with the current color or greyscale equivalent.

POINT

Description:

Activate the graphics pointer device.

POINT draws rubber band lines differently on different operating systems. Users should select a foreground color that is visible against the background image. Rubber band lines are drawn as follows:

Windows The current color, line width and type, with mix 0.
Unix The current color, line width and type, with mix 3.

Syntax:

```
C←'POINT' (0[,x_initial,y_initial])
```

Activates the graphics pointer device, pointing initially at (`x_initial`, `y_initial`), or the current position if an initial position is not given. Returns (`1,x_final,y_final,button_no`) when a mouse button is released, or (`2,x_final,y_final,c,s`) when a keyboard key is pressed. `c` and `s` are the character code and scan code as returned by the AP 124 calls (3 1), (3 2), and (3 3) as described for AP 124 under Read and Wait or Read and Test.

Note: When the OPEN command has enabled Unicode support, the character codes returned are `⎕UCS` character codes.

```
C←'POINT' (1[,x_start,y_start[,x_initial, y_initial]])
```

Activates the graphics pointer device, pointing initially at (`x_initial`, `y_initial`) or the current position if an initial position is not given. Draws a "rubber band" line from (`x_start`,`y_start`) (or the current position if not given) to the location of the pointer. Returns the same data as for the `0` case.

```
C←'POINT' (2,s_start,y_start[,x_end,y_end [,x_initial,y_initial]])
```

Activates the graphics pointer device, pointing initially at (`x_initial`, `y_initial`) or the current position if an initial position is not given. Draws a "rubber band" line from (`x_start`,`y_start`) to (`x_end`,`y_end`) (or the current position if not given) as two line segments via the location of the pointer. Returns the same data as for the `0` case.

```
C←'POINT' (3,x_start,y_start[,x_end,y_end [,x_initial,y_initial]])
```

Same as for `2` option, but an additional line is drawn to form a "rubber band" triangle between (`x_start`,`y_start`), (`x_end`,`y_end`) and the current pointer location. Returns the same data as for the `0` case.

```
C←'POINT' (4[,x_start,y_start[,x_initial, y_initial]])
```


© Copyright IBM Corporation 1994, 2017

Same as for `1` option, but draws a "rubber band" rectangle with one corner at (`x_start`,`y_start`) and the opposite corner at (`x_initial`,`y_initial`) or the current position if an initial position is not given. Returns the same data as for the `0` case.

```
C←'POINT' ('')
```

Immediately returns the graphics pointer position and mouse button status. Returns (`0,x_pointer,y_pointer[,button_nos]`). If the pointer lies outside of the AP 207 window, `x_pointer` and `y_pointer` are still relative to the lower-left corner; therefore, negative values are possible. `button_nos` are the numbers of any mouse buttons currently pressed.

PostScript Driver:

This command is not supported under the PostScript driver.

PRINT

Description:

Dumps the graphic window contents to the currently-selected printer (Windows) or user-supplied print program (Unix).

Syntax (Windows):

```
C←'PRINT' 'filespec'
```

Stores the window contents in a metafile. If the file already exists, it will be replaced.

```
C←'PRINT' ('')
```

Prints the window contents on the default printer.

Notes:

1. The **Printer Setup** menu item gives access to change the default printer and job properties.
2. The `'MENU' 'PRINTER SETUP'` command can be used to change the default printer and job properties. Changes made with this command override settings specified with the **Printer Setup** menu item.
3. When a filespec is supplied to the PRINT command, it produces an enhanced metafile. Use the extension .EMF for compatibility with other applications.

Syntax (Unix Systems):

```
C←'PRINT' '[filename][,P][,opt1][,opt2][,...]'
```

Requests printing of the AP 207 graphics window.

© Copyright IBM Corporation 1994, 2017

If the window contents are to be retained in a print file, `filename` should be provided as the first parameter. If no file name is specified, AP 207 passes the print routine a default file name of `PRN.207`. The file name is passed with the keyword `-fn`.

AP 207 also passes the window identifier to the called routine with the keyword `-wid`, followed by an integer value.

Each optional keyword following a comma (`,`) in the parameter list is passed by AP 207 (without validation) to the user-supplied print routine in the form `-keyword [value]`.

If `,P` is appended, the print routine is passed the `-P` keyword to request portrait-format printing.

The PRINT command returns two elements:

1. The return code from the AP 207 command
2. The return code from the called routine

When AP 207 receives the PRINT command, it looks for the name of the user-supplied print routine to be called in the value of APL2 invocation option `-prtg`, or a PRTG keyword definition in the [Invocation Options] section of the `apl2.ini` file, or environment variable APLPRTG. If a print routine name is not supplied by any of those methods, a filename of `/usr/APL2/bin/graphprt` is used. If the specified print routine is not found, return code 57 is issued.

PostScript Driver:

Under the PostScript driver, the PRINT command has only one parameter, `'filename'`, which is passed to the print shell. All other parameters are ignored. `'filename'` is the name of the file that holds the generated PostScript commands. AP 207 also passes the driver name so the print shell can take different actions depending on the active driver. In the example shell, PostScript files are queued for printing using `qprt` and are sent to `cps`, a color printer queue.

Examples:

AP 207 Command	Arguments passed to the print shell
`'PRINT' ''`	`-wid 536938136 -fn PRN.207`
`'PRINT' 'FILE1,P'`	`-wid 536938136 -fn FILE1 -P`
`'PRINT' ',D 3812'`	`-wid 536938136 -fn PRN.207 -D 3812`
`'PRINT' 'AP207.PS'`	`-wid PostScript -fn AP207.PS`

Sample Print Shell:

The following is an example of a print shell that can be called by AP 207 using the PRINT command. It uses standard Unix print commands to allow the user to click the mouse pointer in a graphics window and dump the image to a file (using `xwd`). The file is passed to the `xpr` command to format the dump file for printing, and then piped through `qprt` to a 4029 printer queue (`asc`).

© Copyright IBM Corporation 1994, 2017

Note: This example is provided to illustrate one simple technique for enabling graphic printer support through the AP 207 PRINT command; it produces very primitive hardcopy of a color graphics window - for instance, there is no greyscale.

```
#! bin/ksh
#-------------------------------------------------------------------#
#          Sample shell program to print a graphics window          #
#                or a file containing PostScript commands           #
#                   to an IBM 4029 laser printer                    #
#-------------------------------------------------------------------#
XWDparms='-bitmap -nobdrs '               # parameters for xwd
XPRparms='-rv '                           # parameters for xpr
PRTshell='qprt '                          # print program
PRTshellparms='-P asc '                   # print program parms
PRTpsq='-P cps '                          # alternate print parms
while test -n "$1"                        # loop through parms
do
  case $1 in
    -wid) shift; PRTdrive=$1;;            # optional: driver name
    -fn) shift; XWDfile=$1;;                        # output file
    -D) shift; XPRparms=$XPRparms"-device $1 ";;    # device name
    -P) shift; XPRparms=$XPRparms"-portrait ";;     # portrait
  esac
  shift
done
XWDfile=${XWDfile:=PRN.207}  # Default output file if none specified
if [ $PRTdrive = "PostScript" ] ; then    # PostScript to print?
    qprt $PRTpsq $XWDfile                 # print to a PostScript queue
else                                      # Simple graphics window?
    xwd $XWDparms -out $XWDfile           # Dump window to output file
    cat $XWDfile | xpr $XPRparms | prtshell $PRTshellparms   # and print
fi
```

QUERY

Description:

Queries driver parameters.

Syntax:

```
C←'QUERY' ('')
```

Returns: `driver_name,parms`.

On Windows, the default driver name is "Microsoft Windows".
On Unix systems, the default driver name is "X-Windows".

`parms` is a numeric matrix, each row of which is:

`[;1]`
: Video mode or window handle.

  On Unix systems, the active video mode is returned as 1. On Windows, if the window was opened with a titlebar, the active video mode is returned as 1. If the window was opened within an AP 145 window, the handle of that window is returned.

© Copyright IBM Corporation 1994, 2017

`[;2]`
Width of screen in pels

`[;3]`
Height of screen in pels

Before an OPEN call is issued, the screen width and height are given as the total physical screen size. After an OPEN call is issued, on Unix systems, the current window size is returned. On Windows, the height and width from OPEN are returned.

`[;4]`
Number of different colors displayable

`[;5]`
Number of different line styles available

`[;6]`
Number of different line widths available

`[;7]`
Number of different fill patterns available

`[;8]`
Number of different markers available

`[;9 10]`
Aspect ratio. The ratio of these two numbers (`÷/parms[;9 10]`) gives the x size of a pel divided by the y size of a pel.

Note: This command may not be used in a multiple-call sequence.

QWRITE

Description:

Returns the position and size of the character string specified.

Syntax:

```
C←'QWRITE' 'text'
```

Returns six elements specifying the x and y coordinates of the lower-left, upper-left and the lower-right corners of the text box assuming it is placed at the current x and y location. The coordinates of the remaining corner (upper-right) can be readily calculated from the three given.

SCISSOR

Description:

Defines the rectangular area within which graphics can be drawn. Any parts of a graphics object falling outside of this rectangle are clipped at the boundary.

Setting the scissor window to `(0 0 0 0)` turns off scissoring. The default is off.

Syntax:

© Copyright IBM Corporation 1994, 2017

```
C←'SCISSOR' (X1,Y1,X2,Y2)
```

Defines the lower-left and upper-right corners of the scissoring rectangle in world coordinate units. Returns the new scissoring rectangle selected.

```
C←'SCISSOR' ('')
```

Returns the current scissoring rectangle.

SECTOR or WEDGE

Description:

Draw a sector of an ellipse or a circle.
The current position is set to the coordinates of the center of the sector.
If used between a BEGAREA and an ENDAREA, each sector is filled independently.

Syntax:

```
C←'SECTOR' (x,y,x_rad,y_rad,start_ang,end_ang)
```

`x,y`
: give the center of the sector

`x_rad,y_rad`
: give the radius of the sector along the x and y axes

`start_ang,end_ang`
: give the starting and ending angles of the sector in degrees

SETPEL

Description:

Sets one or more pels to the specified color.
The current position is set to the coordinates of the last point.

Syntax:

```
C←'SETPEL' (color,x,y)
```

Sets the pel at (`x`,`y`) to the color defined by the row `color` in the color map array.

```
C←'SETPEL' ((n,3)⍴color1,x1,y1,...,colorn,xn,yn)
```

Sets the pel at each (`x`,`y`) location to the corresponding color in the color map array.

USE

Description:

Select a driver for subsequent use by AP 207.

© Copyright IBM Corporation 1994, 2017

Note: This command is ignored under Windows.

Syntax:

```
C←'USE' device_handle
```

Device 0 is the internal AP 207 window services driver. This is automatically used when AP 207 is started and need not be explicitly specified.

To use an alternate driver, the LOAD command must first be issued to load the driver and get the device handle. That handle is then used as the parameter to USE.

Currently, the only alternate driver available for Unix systems is the PostScript driver (`'PS'`).

```
C←'USE' ('')
```

Returns the name of the current driver.

VIEW

Description:

Activate or deactivate the AP 207 window.

Note: This command has no effect on Unix systems.

Syntax:

```
C←'VIEW' ('')
```

Makes the AP 207 window the active window.

```
C←'VIEW' ('ON')
```

Makes the AP 207 window the active window. The handle of the program that was active when this command was issued is saved.

```
C←'VIEW' ('OFF')
```

Restores the active status to the window saved by the previous ON command.

PostScript Driver:

This command is not supported under the PostScript driver.

WAIT

Description:

© Copyright IBM Corporation 1994, 2017

Wait for a user input or other event.

Syntax:

```
C←'WAIT' n
```

Waits for `n` seconds. If `n` is `0`, returns immediately. Returns:

`0,time,x,y`	if no event occurred during the specified interval
`1,time,x,y,button_no,bpr`	if a mouse button was either pressed or released
`2,time,x,y,c,s`	if a keyboard key was pressed

`time`
: is the time (in seconds) that the event was detected.

`x` and `y`
: give the current graphic pointer position. `x` or `y` may be negative if the cursor is outside of the AP 207 window.

`button_no`
: is the number of the mouse button that was pressed or released. 1, 2, and 3 for the left, middle, and right buttons respectively. Devices with more buttons may return other numbers.

`bpr`
: is `1` if the button has been pressed, or `0` if the button has been released.

`c` and `s`
: are the character code and scan code as returned by the AP124 calls (3 1), (3 2), and (3 3) as described for AP 124 under Read and Wait or Read and Test.

  Note: When the OPEN command has enabled Unicode support, the character codes returned are `⎕UCS` character codes.

To clear all queued events, use the expression: `→(0≠↑↑1↓SHR207,SHR207←'WAIT' 0)/⎕LC`

```
C←'WAIT' ('')
```

Waits indefinitely for an event. Returns the same data as above, except that the 0 time-out case is never returned.

PostScript Driver:

This command is not supported under the PostScript driver.

WINDOW

Description:

Define a window on the device for graphics and establish a world coordinate system to be mapped into this window. See Defining World Coordinates with Correct Aspect Ratio for details of using this call to preserve screen aspect ratios.

Syntax:

© Copyright IBM Corporation 1994, 2017

```
C←'WINDOW' (window,viewport)

window = wx1,wy1,wx2,wy2
```
Specifies the bottom left and top right of the display window box in device (pel) coordinates.
```
viewport = vx1,vy1,vx2,vy2
```
Specifies world coordinates corresponding to the bottom left and top right of the window.

```
C←'WINDOW' ('')
```

Returns the current window and viewport parameters. The window defaults to the entire screen, and the viewport defaults to the device (pel) coordinates.

Note: The null parameter may not be used in a multiple-call sequence.

PostScript Driver:

The WINDOW command can be used to adjust for different paper sizes. For the PostScript driver, the window size is based on 1043 units per centimeter or 2650 units per inch of paper. If the paper size differs from 8.5x11 inch USA paper size, it is necessary to change the default window size by changing the window parameter. For example, to adjust the window size for A4 paper:

```
SHR207←'WINDOW' (0 0 31009 21925 0 0 31009 21925)
```

WRITE

Description:

Write some text in the current setting of color, font, and position.

Note: LINETYPE is an implicit argument for WRITE of APL Vector fonts. For non-filled fonts, the current line width and style will be used to draw the characters. For filled fonts, the behavior is as if within an area: outlines are not drawn around the characters.

Syntax:

```
C←'WRITE' 'text'
```

Writes `text` at the current position.

Returns six elements specifying the x and y coordinates of the lower-left, upper-left and the lower-right corners of the text box. The coordinates of the remaining corner (upper-right) can be readily calculated from the three given.

AP 207 Programming Techniques

- Multiple-Call Sequences
- Defining World Coordinates with Correct Aspect Ratio
- Generating PostScript Output
- Image and Vector Font Support

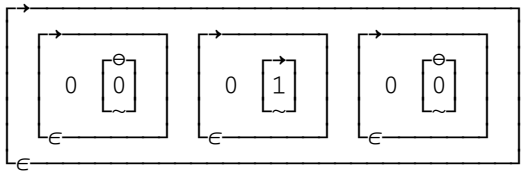
© Copyright IBM Corporation 1994, 2017

- Support for Double-byte Characters

Multiple-Call Sequences

Several commands can be issued to AP 207 in a single call. This requires the calls to be catenated into an even-length vector with alternating commands and parameters. The result is one longer than the number of commands. Its first element is the highest return code encountered in all the calls issued, followed by the return codes from each individual call. For example:

```
      SHR207←'COLOR' 'BLUE' 'ARC' (100 100 50 50 0 360)
      DISPLAY ⎕←SHR207
0       0  1    0
```



This enables you to use the expression `0≠↑↑SHR207` to establish whether or not an error occurred in any of the calls issued.

Defining World Coordinates with Correct Aspect Ratio

To get the aspect ratio for a particular mode after a window has been opened (using the OPEN command):

```
⎕IO←1
SHR207←'OPEN' ''
(RC MODE)←SHR207
MODE←↑MODE
SHR207←'QUERY' ''
(RC DRIVER PARMS)←SHR207
PARMS←⊃PARMS                    ⍝ Data for supported modes
Q←,(PARMS[;1]=''⍴MODE)⌿PARMS    ⍝ Q is data for this mode
AR_PEL←÷/Q[9 10]                ⍝ Ratio of pel x to y size
AR_SCRN←÷/Q[2 3]×Q[9 10]        ⍝ Screen aspect ratio.
```

Usually you want to define a set of world coordinates that are independent of the actual pel coordinates on the screen. This is done by using the WINDOW call to AP 207. In many cases, the world coordinates should preserve aspect ratio; that is, one unit in world coordinates in the x direction should be the same physical size on the device as one unit in world coordinates in the y direction. The following example is useful in defining a world coordinate system that preserves aspect ratio. To map the whole screen to world coordinates ranging from 0 to 1000 in x and pick the maximum y world coordinate to preserve aspect ratio:

```
WIN_PARMS←0 0,(Q[2 3]-1),0 0 1000,¯1+⌊0.5+1001÷AR_SCRN
SHR207←'WINDOW' WIN_PARMS
```

Generating PostScript Output

Note: Generation of PostScript output is supported only on Unix systems.

© Copyright IBM Corporation 1994, 2017

The PostScript driver for AP 207 interprets each AP 207 command and generates the appropriate PostScript commands. These commands are sent to a file. The PostScript commands can be executed to produce printed graphics and text by sending the file to any printer with PostScript support. APL2 programs written to use the default display driver of AP 207 need to be modified to make use of the PostScript driver.

The AP 207 `LOAD` command is needed to load the PostScript driver. This also has the effect of switching the active driver to the PostScript mode:

```
⍝ Load the PostScript driver
SHR207←'LOAD' 'PS'
```

The `OPEN` command is used to open the file that will collect the generated PostScript commands. It has several modes to handle different printers and different printed page orientations. In the example below, the driver is opened in default mode and the file `test.ps` is created (if it doesn't exist) and opened to receive generated PostScript:

```
⍝ Open the driver: default mode
SHR207←'OPEN' (0 'TEST.PS)
```

APL2 applications that use world coordinates rather than device or pel coordinates are portable to other device drivers that have different coordinate systems. Use the `WINDOW` command to map world coordinates to PostScript page coordinates. The `ASPECT` function from workspace 2 DEMO207 (see Defining World Coordinate with Correct Aspect Ratio) can be called immediately after the `OPEN` command to establish the correct world coordinates for PostScript. To use paper sizes that differ from 8.5x11 inches, change the window size using the window parameter of the `WINDOW` command. Multiply the number of inches or centimeters in paper size by 2650 points per inch or 1043 points per centimeter to derive the coordinates for the maximum values of `x` and `y` (the upper right corner coordinates).

To close the driver and ready the file for printing, use the `CLOSE` command. Without the `CLOSE` command, no graphics are printed when the file of generated PostScript commands is sent to the printer. `CLOSE` also closes the file:

```
⍝ Close driver, ready file for printing
SHR207←'CLOSE' ''
```

To execute the PostScript commands and print the AP 207 graphics, use the AP 207 `PRINT` command or any operating system command that can send a file to a PostScript printer. The `PRINT` command requires a user-supplied routine (see PRINT) and an environment variable (`APLPRTG`) to point to that routine. For example:

```
⍝ Invoke user-supplied routine
SHR207←'PRINT' 'AP207.PS'
```

To switch to a different device driver, use the `USE` command. When AP 207 is switched to a different device driver with the `USE` command, the driver should have been opened previously with the `OPEN` command. If the device driver is not open after issuing `USE`, issue the `OPEN` command to open that driver. To switch to the PostScript driver, the `LOAD` command must also have been executed previously to load the PostScript driver. For example:

© Copyright IBM Corporation 1994, 2017

```
⍝ Execute AP 207 commands under the PostScript driver
...
⍝ Switch to the default X-Window driver
SHR207←'USE' 0
RC←SHR207
⍝ If necessary, open the X-Window driver
SHR207←'OPEN' 1
(RC MODE)←SHR207
⍝ All AP 207 commands will now be passed to X-windows
```

Image and Vector Font Support

AP 207 can use either external image (bitmap) or vector font files in writing text. This support provides for rapid display of characters from any of a variety of fonts.

Image (bitmap) fonts are easy to read in small sizes since they are mapped directly to the pels on the screen. They differ in appearance on different displays and in printed copies. The minimum size may also be larger than desired.

Vector fonts can be scaled to any size and rotated, sheared, or stretched as desired. They are device-independent, although they may be somewhat distorted on low-resolution displays.

- Supplied Vector Fonts
- Vector Font File Format

Supplied Vector Fonts

The vector fonts supplied with APL2 are listed below. Most of the fonts consist of straight-line segments only. Thus the space between the vectors may become visible for characters in larger sizes. Fonts labeled as "Filled" use the AP 207 area filling routines to fill the character interiors. These characters look better in large sizes, although they are slower to draw and may not look as good in small sizes.

Font Filename	Description
GOTENG.AVF	Gothic English
GOTGER.AVF	Gothic German
GOTITA.AVF	Gothic Italian
GRESER.AVF	Greek Serif (with some mathematical symbols)
GRESIM.AVF	Greek Simplex (with some mathematical symbols)
MARKERS.AVF	Markers (correspond to image markers from MARKERS call)
MODSIM.AVF	Modern Simplex
ROMDUP.AVF	Roman Duplex
ROMDUPF.AVF	Roman Duplex Filled
ROMITA.AVF	Roman Italic
ROMITAB.AVF	Roman Italic Bold
ROMSER.AVF	Roman Serif
ROMSERB.AVF	Roman Serif Bold
ROMSIM.AVF	Roman Simplex (contains all codepage 910 characters)

© Copyright IBM Corporation 1994, 2017

Font Filename	Description
ROMSIMM.AVF	Roman Simplex Monospace
SANSER.AVF	Sans Serif
SANSERF.AVF	Sans Serif Filled
SCRIPT.AVF	Script
THKRNDF.AVF	Thick Round Filled
THKRNDO.AVF	Thick Round Outlined
THKSQUF.AVF	Thick Square Filled
THKSQUO.AVF	Thick Square Outlined

Vector Font File Format

AP 207 vector fonts are stored in files with a file extension of ".AVF" (APL Vector Font.) Each file defines 256 vector characters. Code page 910 is used to assign the characters to the code points except for fonts consisting of special characters not included in code page 910. The files consist of four parts:

Bytes	Description
0 - 31	Font Header
32 - 545	Font Index
546 - 1057	Font Width Table
1058 - end	Font Character Definition Data

Note: All two-byte and four-byte integers stored in APL2 vector fonts are stored in normal order (not byte-reversed).

Font Header

The 32 bytes of the font header are further subdivided as follows:

Bytes	Description
0 - 3	Total number of bytes in the file
4 - 5	Font type. The only type currently allowed is 1 (2-byte vector font.)
6 - 7	Flags. Bit 0 (low-order bit) is 0 if Monospace font and 1 if Proportional font. All other bits are reserved.
8 - 9	Maximum character width in the font
10 - 11	Height of capital letters
12 - 13	Recommended vertical spacing between lines (height of the character box)
14 - 15	Distance from baseline (y=0) to bottom of the character box
16 - 31	Reserved (must be 0)

Font Index

© Copyright IBM Corporation 1994, 2017

These 514 bytes consist of 257 two-byte unsigned integers that give the offset from the beginning of the font character definitions of the definition of each of the 256 characters. These offsets are in units of two bytes since each coordinate is stored in two bytes (see below). These 256 integers are followed by an integer that contains the total number of byte pairs of font character definition data. The first of the 257 integers must always be 0 since the definition of the first character is always at offset 0. If a character in the font is not defined, two successive entries in the index are the same.

Font Width Table

These 512 bytes consist of 256 two-byte signed integers containing the widths of each of the 256 characters in the font. If a character is undefined, its width must be 0. For a monospace font, all the entries are equal to the number in the maximum character width field in the file header.

Font Character Definition Data

The remaining bytes consist of the vector data for the characters in the font. Each coordinate is stored in two bytes; the first contains the x coordinate information, and the second contains the y information. The high-order bit of x contains the beam-blanking information: a vector is drawn to this point with the beam on if this bit is 1, and off if it is 0. The high-order bit of y contains information on filled areas. When this bit changes from 0 to 1, a BEGAREA call is generated before the vector is drawn. When it changes from 1 to 0, an ENDAREA call is generated before the vector is drawn.

The remaining seven bits for x and y are interpreted as unsigned integers, and 64 is subtracted to obtain a pair of integers (x and y) in the range -64 to 63. These give the relative length of the vector to be drawn in x and y. All characters start at coordinate (0,0), and the last vector must end at y=0. The character baseline (bottom of "A") is at y=0.

The following APL2 function extracts the character definition for single character `C` from the font data stored as a global character vector called `FONTDATA`, which can be read in as one long record using AP 210's Code C.

```
    ∇
[0]  Z←GETFONT C;BEAM;FDATA;FHEADER;FILL;FINDEX;XY;⎕IO
[1]   ⎕IO←1
[2]  ⍝ FONTDATA is the font data in byte format.
[3]  ⍝ It contains a 32-byte header, 514-byte index,
[4]  ⍝ 512-byte width table, and font definition data.
[5]   FHEADER←256⊥⍉16 2⍴⎕AF 32↑FONTDATA
[6]   FINDEX←256⊥⍉257 2⍴⎕AF 32↓546↑FONTDATA
[7]   FDATA←((¯1↑FINDEX),2)⍴1058↓FONTDATA
[8]   C←⎕AF ''⍴C
[9]  ⍝ Get rows of FDATA which describe character C
[10]  Z←⎕AF FDATA[FINDEX[1+C]+⍳-/FINDEX[2 1+C];]
[11]  BEAM←Z[;1]≥128
[12]  FILL←Z[;2]≥128
[13]  XY←+\¯64+128|Z
[14] ⍝ Z is a 4-column matrix:  beam off/on,
[15] ⍝ fill off/on, x coord, y coord
[16]  Z←BEAM,FILL,XY
    ∇
```

Support for Double-byte Characters

© Copyright IBM Corporation 1994, 2017

AP 207 supports four commands that use or return character information. WRITE and QWRITE are used to display character data. POINT and WAIT accept character input. These commands operate in either single-byte or Unicode mode. The parameters of the OPEN command control which mode is in effect.

When AP 207 is in single-byte mode, the WRITE and QWRITE commands support use of Multibyte DBCS characters. When a single-byte character vector is passed to an AP 207 window, if the currently selected font and the current input locale support Multibyte characters, any Multibyte data encoded in the vector is processed as DBCS characters. The WAIT and POINT commands do not support entry of DBCS characters. Character keystrokes are returned as the zero origin `⎕AV` index of the character corresponding to the key pressed.

When AP 207 is in Unicode mode, the WRITE and QWRITE commands convert all character data to Unicode before processing. The POINT and WAIT commands support entry of DBCS characters. Character keystrokes are returned as the `⎕UCS` index of the Unicode character corresponding to the key pressed.

Note: Unicode support is available only on Windows.

See Double-Byte Character Set Support for information about double-byte characters and support for them in other APL2 components.

AP 207 Return Codes

AP 207 return codes are returned as the first element of the two elements returned for each call.

Code	Meaning
0	Success
1	Unexpected driver or subsystem error
22	Buffer overflow
30	Display mode not supported by this driver
31	Invalid command
32	Shared variable domain error
33	Shared variable length error
34	Shared variable rank error
35	Invalid device handle
36	No device currently selected
37	Call not available with this driver or in this mode
38	Call not permitted in multiple-call sequence
39	Invalid parameter
40	Scissor boundary outside window
41	Attempt to place point outside of window
42	Conflicting window-viewport definition
50	Invalid video mode for this driver
51	Device was not opened
52	No fill area started
53	Insufficient space for fill point storage

© Copyright IBM Corporation 1994, 2017

Code	Meaning
54	Printer not ready
55	Incorrect escape command
56	Device not available
57	Error while loading device driver
58	Invalid device driver
60	Font table is full
61	Requested font not in font table
62	Unable to load requested font
63	Cannot read image
64	File access denied
99	Command not implemented
¯26	Insufficient space

© Copyright IBM Corporation 1994, 2017

AP 210 - File Processor

The file auxiliary processor, AP 210, is used to read from, or write to, operating system files. The access can be either sequential or random, and can use fixed-length or variable-length record conventions.

Two shared variables are required to process a file: a data variable and a control variable. The control variable must be offered first. AP 210 then matches the earliest pending corresponding data variable offered by the partner (after the control variable). This occurs the first time it is required to satisfy an AP 210 command.

The name of the data variable must always begin with the letter `'D'` or the letters `'DAT'`, and the control variable must begin with the letter `'C'` or the letters `'CTL'`. The remaining characters in both names (possibly none) must be the same, because the coupling of both variables is recognized by their name. Examples of valid pairs are: `C` and `D`, `C1` and `D1`, and `CXjj` and `DXjj`. Also accepted as valid pairs (for compatibility with APL2/370) are `CTL` and `DAT` or `CTL1` and `DAT1`. The control variable is used to select the operation to perform and to control each input/output operation. For example:

```
      210 SVOPAIR 'C210' 'D210'
2 1
```

offers two shared variables to AP 210 and sets access control. See Using the Share-Offer Utilities for a description of the `SVOPAIR` function (from the 1 UTILITY workspace).

- Control Commands
- Control Subcommands
- Establishing the AP 210 Translate Table
- AP 210 Return Codes
- Example of Use

Control Commands

Once the control variable has been shared and the appropriate access control established, the first value you assign to it should be a *character* vector, which is considered to be a command that describes the file name and specifies the function to be performed. The following commands are accepted:

`IR,filespec[,code]`	Open file for read-only
`IW,filespec[,code]`	Open file for read/write
`PR,filespec[,code]`	(Unix systems only) Open pipe file for read-only
`PW,filespec[,code]`	(Unix systems only) Open pipe file for read/write
`DL,filespec`	Delete file
`RN,filespec,filespec`	Rename file

`filespec` is the file identification, of the form:

```
      ["][path]filename["]
```


© Copyright IBM Corporation 1994, 2017

`path` is a valid path and `filename` is a valid file name. The surrounding quotes are required only if the file identification contains the comma character, to distinguish commas in the file identification from commas in the AP 210 command string.

`code` is a single letter selecting a given interpretation of the file data. Five different interpretations are supported.

Code	Interpretation of Data
A (APL2)	The records in the file contain APL2 objects, with their headers. Arrays of arbitrary rank and depth can be stored and recovered. Different records of a file can contain objects of different types (for example, characters, integers, or real numbers). An APL2 object in a record may occupy up to the actual record length (not necessarily the same number of bytes), but the header fills a part of that area.
B (Bool)	The records in the file contain strings of bits without any header (packed eight bits per byte). The equivalent APL2 object is a Boolean vector. In this case, all records must be equal to the selected record length.
C (Char)	The contents of the record is a string of characters in ASCII, with no header. All records must be equal to the selected record length with each character occupying one byte. Variable-length records are not supported.
D (ASCII)	The contents of the record is a string of characters in ASCII code, with no header. Each character occupies one byte. Variable-length records are supported.
T (Translate)	The contents of the record is a string of characters, without any header, translated according to the AP 210 translate table defined as described in Establishing the AP 210 Translate Table. All records must be equal to the selected record length except when variable-length operations are being performed.

If the code is not stated explicitly, code A is the default.

The `IR` command opens the file for read-only operations. If the operation is successful, the control variable passes into the *subcommand state*. You must then specify which data transfer operation you want to perform. (See Control Subcommands.) The `IW` command works in a similar way, but the file is opened for both read and write operations. If the file cannot be opened, the control variable remains in the command state.

The `PR` command opens a pipe file for read-only operations. The `PW` command opens a pipe file for write operations. The syntax and behavior of these commands is generally the same as the `IR` and `IW` commands, except that they are used for pipes. They can also be used to access external devices such as cassettes and floppy disk drives.

Notes:

1. The pipe commands are intended for use on Unix systems only. However, they are not disabled on other systems, and they may work to access some kinds of pipes on those systems, if the pipes are created first by other programs. The creation of new pipes using AP 210 is only supported on Unix systems.
2. If an `IR` or `IW` command executes successfully (giving a return code of `0` in the control variable), the data variable is set to the size, in bytes, of the file just opened. The size of pipe files (opened with `PR` or `PW`) is not available. The data variable is set to `¯1` for these files.

© Copyright IBM Corporation 1994, 2017

3. Subject to any further limitations of the file system in use, the largest file size accessible by AP 210 is 4 Gigabytes (4294967295). If they can exist on the file system, files larger than 4G can be opened, but the size reported by AP 210 in the data variable for these files will be 4G and attempts to access data in the file beyond the 4G limit will result in the end-of-file return code.

When the `DL` command is received, the file with the specified filespec is erased. Then the control variable returns to the command state.

When the `RN` command is received, the name of the file specified in the first parameter is changed to the name given in the second parameter. If a different path is given in the second parameter, a move is performed instead of a rename. After this command has been executed, the control variable returns to the command state.

Once a command has been received and executed, a return code is passed back to APL2 through the control variable, indicating whether or not the command was executed successfully and, if not, the reason for the failure.

Control Subcommands

Once a file has been opened for input (command `IR` or `PR`) or input/output (command `IW` or `PW`), the control variable passes into the subcommand state. It now accepts the assignment of *numeric* vectors specifying the operation to perform, with the following structure:

```
C210←op[,rn[,rs]]
```

The following are valid operations:

0	Read a fixed-length record. Record size is defaulted to 128 unless specified by the `rs` operand or by a previous subcommand. Files are considered as being divided into fixed size records. `rn` is the record number. If not given, sequential operation is assumed.
1	Write a fixed-length record. Record size is defaulted to 128 unless specified by the `rs` operand or by a previous subcommand. Files are considered as being divided into fixed size records. `rn` is the record number. If not given, sequential operation is assumed.
2	Direct read from a file. Record size is defaulted to 128 unless specified by the `rs` operand or by a previous subcommand. Files are considered as being continuous strings of data. `rn` is the starting byte. If not given, sequential operation is assumed.
3	Direct write to a file. Record size is defaulted to 128 unless specified by the `rs` operand or by a previous subcommand. Files are considered as being continuous strings of data. `rn` is the starting byte. If not given, sequential operation is assumed.
4	Read a variable-length record. This command can only be used if the file was opened with codes A, D, or T. `rn` is the record number. If not given, sequential operation is assumed. If `rn` is given but does not specify the next record in sequence, the file is scanned in search of the requested record. If `rn` specifies a record earlier in sequence than the previous record accessed, the scan restarts from the beginning of the file. Either specifying `rn` in sequence, or not specifying it at all, gives fastest execution.

© Copyright IBM Corporation 1994, 2017

	`rs` is the scan distance. For codes D and T, the LF character is searched for within the first `rs` bytes of a record. If `rs` is not given, the LF character is searched for within the first 128 bytes, unless `rs` has been specified by a previous subcommand. If LF cannot be located, the record is not read, and a return code `¯44` is issued. If reading with either code D or code T, the record returned in the data variable includes any record delimiter characters (LF or CR/LF depending on the origin of the file).
5	Write a variable-length record. This command can only be used if the file was opened with codes A, D, or T. `rn` is the record number. If not given, sequential operation is assumed. If `rn` is given but does not specify the next record in sequence, the file is scanned in search of the requested record. If `rn` specifies a record earlier in sequence than the previous record accessed, the scan restarts from the beginning of the file. Either specifying `rn` in sequence, or not specifying it at all, gives fastest execution. If `rn` is greater than the number of records currently in the file, the record is appended to the end of the file. `rs` is the scan distance. For codes D and T, the LF character is searched for within the first `rs` bytes of a record when scanning for a requested record not in sequence. If `rs` is not given, the LF character is searched for within the first 128 bytes, unless `rs` has been specified by a previous subcommand. If LF cannot be located, the record is not read, and a return code `¯44` is issued. Direct write of variable-length records is allowed, but should be done with great care. Records should be replaced by others with exactly the same length. If this is not done, the whole file, starting at the replaced record, can be damaged. If the file was opened with either code D or code T, the CR/LF delimiters are appended to the end of each record as expected for sequential files. If the record already has a LF character (`⎕TC[⎕IO+2]`) appended, no additional delimiters are added. Code A files have no need for the LF record separator, as each record includes its own length, which makes each record self-describing, allowing AP 210 to skip through the file to the requested record.
6	Operation 6 has the same syntax and behavior as operation 4, but end-of-line indicators (either CR/LF or LF) are removed from the data.

Notes:

- `rn` is always defined in zero origin (the first record in a file is record 0; the first byte in a file is byte 0). If not specifically stated, the first value of `rn` after opening a file is 0 (that is, the first record or byte position in the file). For pipe files, all operations are sequential so `rn` values must be `0`.
- Write operations are not allowed if the subcommand state was entered through the `IR` or `PR` commands.
- If the control variable is assigned an empty vector while in the subcommand state, the file is closed and the control variable reverts to the command state.
- Retracting the shared variables, either explicitly, with `⎕SVR`, or implicitly, by erasing the variables (`)ERASE`, `⎕EX`, or, if the variables have been localized, by exiting the function), clearing the workspace (`)CLEAR`), loading a new workspace (`)LOAD`) or ending the session (`)OFF`) causes AP 210 to close the appropriate files automatically.

© Copyright IBM Corporation 1994, 2017

- Once an operation has been requested, the data variable is used as a buffer, where the actual transfer of records takes place. If the operation is a read, the value of the record can be found in the data variable *after* the successful completion of the requested operation (confirmed by the return code passed through the control variable). If the desired operation is a write, the value of the record must be assigned to the data variable *before* the corresponding subcommand is assigned to the control variable.
- A pipe file, sometimes called a FIFO file, is a special file with unique characteristics. Two processes can communicate through a pipe file, with (typically) one process writing data to the pipe and the other process reading data from the pipe in the same order it was written (first in, first out).

  Once the data has been read from a pipe file, it is removed from the pipe and cannot be re-read. If the writer stops writing, the read process continues to read until the pipe is empty. The reader than hangs up in a "sleep" state on the reference of the control variable, instead of returning an EOF return code. If the pipe is empty and the writer closes it, the read process wakes up from the sleep state and returns an EOF return code. If the write process closes and there are no read processes, any m remaining data in the pipe is removed.

Establishing the AP 210 Translate Table

AP 210 can be supplied with a translate table for use with files opened with code T. Each file opened can have its own translate table. If no table has been defined, the data is not translated. The table can be defined or redefined, at any time, by:

```
C210←2 256⍴READ_TABLE,WRITE_TABLE
```

where `READ_TABLE` is applied to the data read as:

```
D210←READ_TABLE[⎕AF DATA_READ]
```

and `WRITE_TABLE` is applied to the data written as:

```
DATA_WRITTEN←WRITE_TABLE[⎕AF D210]
```

Both `READ_TABLE` and `WRITE_TABLE` must be 256-element character vectors.

AP 210 Return Codes

The following table lists the AP 210 return codes:

Code	Meaning
`0`	Success
`1`	Invalid command
`2`	File not found
`¯18`	Insufficient User Authority
`¯26`	No space available
`¯29`	Invalid APL2 object

© Copyright IBM Corporation 1994, 2017

Code	Meaning
¯31	Control variable domain error
¯32	Control variable rank error
¯33	Control variable length error
¯36	Invalid file translation code
¯37	Data variable value error
¯38	Data variable domain error
¯39	Data variable interlocked
¯40	Data variable not shared
¯41	File is not open, issue a primary command
¯44	LF not found in scan length
¯45	End of file
¯46	Incomplete record, padded with nulls
¯47	Invalid subcommand
¯48	Record only partially written (file system full)
¯49	Data variable size exceeds record size for fixed or variable replace

Note: Any other positive values are unexpected return codes from the operating system file services used by AP 210. The CHECK_ERROR function can be used to obtain more information.

Example of Use

Starting with a clear workspace, offer variables C1 and D1 to share with AP 210 using the SVOPAIR function from the 1 UTILITY workspace:

```
      )CLEAR
CLEAR WS
      )COPY 1 UTILITY SVOPAIR
SAVED ...
      210 SVOPAIR 'C1' 'D1'
2 1
```

Attempt to create a file called FILE. Records contain APL2 objects with header (default code):

```
      C1←'IW,FILE'
      C1
0
```

We are now in subcommand mode. The first record is a vector of elements from 1 to 10, so assign this to D1:

```
      D1←⍳10
```

Now issue the subcommand to write the first record in the file. The default record number is 0.

```
      C1←5
      C1
```


© Copyright IBM Corporation 1994, 2017

```
0
```

Second record is a matrix of 2 rows, 3 columns, of elements from 1 to 6:

```
      D1←2 3⍴⍳6
```

Issue subcommand to write this record sequentially to the file:

```
      C1←5
      C1
0
```

An empty vector closes the file and puts the control variable into command mode:

```
      C1←''
```

Open the same file for read-only operation:

```
      C1←'IR,FILE'
      C1
0
```

Read the second record first:

```
      C1←4 1
      C1
0
      D1
 1 2 3
 4 5 6
```

Read the first record:

```
      C1←4 0
      C1
0
      D1
1 2 3 4 5 6 7 8 9 10
```

Close the file and go into command state:

```
      C1←⍳0
```

Rename the file to NEWFILE:

```
      C1←'RN,FILE,NEWFILE'
      C1
0
```

Delete the file:

© Copyright IBM Corporation 1994, 2017

```
      C1←'DL,NEWFILE'
      C1
0
```

Finally, retract the shared variables:

```
      ⎕SVR 2 2⍴'C1D1'
2 2
```


© Copyright IBM Corporation 1994, 2017

AP 211 - APL2 Object Library Processor

AP 211 provides a facility for storing APL2 arrays in an object library.

AP 211 uses a single shared variable of any name to control access to a library.

The following description of the commands accepted by AP 211 assumes that a variable called `SHR211` has been shared with AP 211. For example:

```
      211 SVOFFER 'SHR211'
2
```

offers the shared variable to AP 211 and sets access control. See Using the Share-Offer Utilities for a description of the `SVOFFER` function.

- AP 211 Commands
- AP 211 Return Codes
- Example of Use

AP 211 Commands

AP 211 accepts the following commands:

- CREATE
- DROP
- USE
- RELEASE
- SET
- GET
- RENAME
- ERASE
- LIST

CREATE

This command creates an AP 211 object library and must be issued before APL2 arrays can be assigned to a given library file.

```
      SHR211←'CREATE' filespec [rec_size]
      return_code←SHR211
```

`filespec` is the file identification, of the form:

```
      [path]filename
```


© Copyright IBM Corporation 1994, 2017

rec_size specifies the record size used to store APL2 objects in the file. APL2 objects stored in the library use one or more records depending on size. Objects smaller than the record size still use a full record and any excess space is wasted. Therefore, you should carefully select an optimum size for your APL2 objects to reduce the amount of wasted space in the file.

The number of blocks that are used is given by: ⌈object_size÷rec_size

where object_size can be obtained by: ↑4 ⎕AT object_name

AP 211 uses a default record size of 1024 if none is specified. The record size must be in the range 128 to 32704 bytes. If the record size is not an exact multiple of 64 bytes, it is rounded up to the next multiple of 64 bytes.

An error is given if the file already exists. It is the responsibility of the application developer to check that the name is not already in use and to erase the file if necessary.

DROP

This command deletes an entire APL2 object library from disk.

```
SHR211←'DROP' filespec
return_code←SHR211
```

filespec is the file identification, of the form:

```
[path]filename
```

USE

This command opens an already-existing AP 211 library file. It must be issued before using the SET, GET, ERASE, and LIST commands.

```
                                  ←-----Access Control------→
SHR211←'USE' filespec [user_id] | ['PRIVATE'|'UPDATE'|'READ']
(return_code rec_size)←SHR211
```

filespec is the file identification, of the form:

```
[path]filename
```

user_id is a scalar integer, which can be used to implement an audit trail of updates, particularly where the file is shared between many users. The default value is ↑⎕AI.

Access Control Parameters:

PRIVATE

© Copyright IBM Corporation 1994, 2017

opens the file for exclusive READ/WRITE access.
A file that is already open cannot be opened as PRIVATE, and once opened PRIVATE, cannot be opened by anyone else until a RELEASE is issued or the shared variable is retracted.

UPDATE
opens the file for shared READ/WRITE access.

READ
opens the file for shared READ/ONLY access.

READ or UPDATE should be specified when accessing AP 211 files that require simultaneous multiuser READ or READ/WRITE access. When running in this mode, AP 211 enables an internal locking procedure, which ensures file integrity, but impacts performance for file updates. If you do not require shared access, opening the file in PRIVATE mode provides a significant performance advantage for file writes.

If the USE command is issued without any of the optional Access Control parameters, the file is opened with the maximum access authority permitted, for exclusive use if possible or shared access if the file is already opened.

rec_size is the record size of the file (from the CREATE).

RELEASE

Releases the object library that was allocated to a shared variable. This command is issued implicitly when retracting or erasing the shared variable, or if a subsequent USE or CREATE are specified to the same shared variable.

```
SHR211←'RELEASE'
return_code←SHR211
```

SET

Allows you to associate a name in the object library with an APL2 array. The command requires a three-element vector:

```
SHR211←'SET' name APL2_object
return_code←SHR211
```

The maximum permitted length of name is 31 characters.

If the name is already in use, the old definition is deleted, and the new definition added to the object library. The space taken by the old definition is freed for later use.

GET

This returns the array (if any) associated with a given name.

```
SHR211←'GET' name
(return_code APL2_object)←SHR211
```


© Copyright IBM Corporation 1994, 2017

The response to this command is a two-element vector: the first item is the return code, the second is the `APL2_object` array, if found.

RENAME

Renames an object stored in an AP 211 file.

```
      SHR211←'RENAME' oldname newname
      return_code←SHR211
```

ERASE

This command allows you to remove an APL2 array from an object library, and makes its storage available for other updates. Note, however, that the overall size of the file remains unchanged.

```
      SHR211←'ERASE' name
      return_code←SHR211
```

LIST

This command allows you to list the contents of the current library:

```
      SHR211←'LIST' 'NAMES'
      ρ□←SHR211
Object1
Object2
Object3
3 31
      SHR211←'LIST' 'ATTS'
      ρ□←SHR211
1 1001 1991 1 2 12 30 14 12
2 1001 1991 1 2 12 30 14 12
1 1001 1991 1 2 12 30 14 12
3 9
```

The information returned is:

- The number of records used for this object
- User ID number (as specified on USE command) of user who last updated this object
- The date and time the object was updated (in `□TS` format)
  Note: The time is listed in Greenwich Mean Time.

Each row in this list corresponds to the equivalent row in the list of object names.

AP 211 Return Codes

The following table lists the AP 211 return codes.

Code	Meaning
0	Success

© Copyright IBM Corporation 1994, 2017

Code	Meaning
2	File not found
¯1	Old format file (access restricted to read only)
¯2	Rank error
¯3	Length error
¯4	Type error
¯7	Invalid command
¯8	Invalid block size
¯9	Invalid library file
¯10	No library file accessed
¯11	Name has no value
¯12	Invalid name specified
¯13	Error encountered during set operation
¯14	Invalid file name
¯15	Invalid access mode
¯16	Invalid user ID
¯17	Filename already in use
¯18	Insufficient user authority
¯19	Name already exists
¯20	Unsupported data format
¯21	Temporary interlock
¯22	Unexpected SVP return code
¯24	Media full
¯26	No space available

Note: Any other positive values are unexpected return codes from the operating system file services used by AP 211.

Example of Use

First share a variable with AP 211:

```
      211 SVOFFER 'SHR211'
2
```

Then create a new file called FILE:

```
      SHR211←'CREATE' 'FILE'
      SHR211
0
```

An empty file has now been created. Before we can go any further, the file must be opened for access:

© Copyright IBM Corporation 1994, 2017

```
      SHR211←'USE' 'FILE'
      SHR211
0 1024
```

The `1024` returned by this call shows that the default record size was used when the file was created. Now we can place data into the file:

```
      SHR211←'SET' 'ABC' (⍳10)
      SHR211
0

      SHR211←'SET' 'DEF' (3 5⍴'AP211IS   EASY!')
      SHR211
0
```

This has placed two objects into the file. We can determine the names of objects stored on the file with:

```
      SHR211←'LIST' 'NAMES'
      SHR211
ABC
DEF
```

At any time, we can retrieve any objects stored in the file:

```
      SHR211←'GET' 'DEF'
      SHR211
 0   AP211
     IS
     EASY!
```

Note that a length two result is given by the `GET` call. Normally we want to separate the return code (the first item) and the data array (the second item) into two variables. Vector assignment provides a simple way to do this:

```
      SHR211←'GET' 'DEF'
      (RC ARR)←SHR211
      RC
0
      ARR
AP211
IS
EASY!
```

When we have finished with the file, we can release it:

```
      SHR211←'RELEASE'
      SHR211
0
```

And finally, if we have no further need for the file, we can delete it:

```
      SHR211←'DROP' 'FILE'
      SHR211
0
```


© Copyright IBM Corporation 1994, 2017

AP 227 - ODBC Processor

AP 227 allows you to use the Structured Query Language (SQL) to access databases and programs that support the Open Database Connectivity (ODBC) protocol.

AP 227 has the same commands and return code structure as AP 127, the DB2 processor.

Supplied workspace SQL is a companion to both AP 127 and AP 227. To use the workspace with AP 227, set variable `SQL_AP` to `227` before using the functions in the workspace. See APL2 Programming: Using Structured Query Language for complete information about AP 127, AP 227 and the SQL workspace.

- AP 227 Commands
- AP 227 Return Codes

AP 227 Commands

Operation Code and Syntax	Workspace Function
`'CALL'` *name* [*values*]	`CALL`
`'CLOSE'` *name*	`CLOSE`
`'COMMIT'` [`'RELEASE'`]	`COMMIT`
`'CONNECT'` *database-identifier*	`CONNECT`
`'DECLARE'` *name* [`'HOLD'`|`'NOHOLD'`]	`DECLARE`
`'DESCRIBE'` *name* [*type*]	`DESC`
`'EXEC'` *stmt*	`EXEC`
`'FETCH'` *name* [*options..*]	`FETCH`
`'GETOPT'`	`GETOPT`
`'ISOL'` [*setting*]	`ISOL`
`'MSG'` *rcode*	`MESSAGE`
`'NAMES'`	`NAMES`
`'ODBC'` *type*	`ODBC`
`'ODBCOPEN'` *name type*	`ODBCOPEN`
`'OPEN'` *name* [*values*]	`OPEN`
`'PREP'` *name stmt*	`PREP`
`'PURGE'` *name*	`PURGE`
`'PUT'` *name values*	`PUT`
`'ROLLBACK'` [`'RELEASE'`]	`ROLLBACK`
`'SETOPT'` *options..*	`SETOPT`
`'SQLCA'`	`SQLCA`
`'SQLSTATE'`	`SQLSTATE`
`'SSID'` [*subsystem*]	`SSID`

© Copyright IBM Corporation 1994, 2017

Operation Code and Syntax	Workspace Function
`'STATE'` *name*	`STATE`
`'STMT'` *name*	`STMT`
`'TRACE'` [*(module level)..*]	`TRACE`
`'XMLSIZE'` [*size*]	`XMLSIZE`

AP 227 Return Codes

Return Code Vector	Meaning
`0 0 0 0 0`	Normal return. All operations completed. Result table retrieved by a FETCH request is complete.
`0 0 1 0 0`	Normal return, but a result table may not have been completely retrieved.
`1 0 0 1` *msgn*	Error in auxiliary processor. *msgn* is the number of the auxiliary processor error message.
`1 0 0 2` *msgn*	Error detected in the database system. *msgn* gives the SQL return code (SQLCODE).
`1 0 0 3` *msgn*	Error detected in an SQL workspace function. *msgn* gives the message number.
`0 1 0` *n msgn*	Warning message. For example, FETCH has no more rows to retrieve, a DELETE statement deletes nothing, or the value-list is longer than the highest vector index.
`1 1 0` *n msgn*	Transaction backout. All changes made to tables since the last COMMIT or ROLLBACK have been discarded. Application must restore processing to point of last COMMIT or ROLLBACK. All locks are released and all cursors closed.

© Copyright IBM Corporation 1994, 2017

AP 488 - GPIB Support Processor

Note: This processor is not provided on Unix systems.

A GPIB (General Purpose Interface Bus) adapter provides an interface between a personal computer and the IEEE-488 General Purpose Interface Bus (GPIB), allowing control of multiple devices or instruments (such as plotters, multimeters, and disk drives).

Auxiliary processor 488 provides an interface between APL2 and the National Instruments GPIB Driver.

The following hardware is supported:

- National Instruments GPIB Adapter Card (for computers with PC bus)
- National Instruments MC-GPIB Adapter Card (for computers with Micro Channel bus)

The following software is required:

- National Instruments GPIB Driver (GPIB-32.DLL on Windows). This DLL must be accessible to be loaded when AP 488 is loaded.

A GPIB adapter can perform as a controller, a talker, or a listener with compatible devices. The GPIB adapter also provides capabilities for data transfer between workstations, and the connection of several computers for sharing of instruments or peripheral I/O devices.

Shared Variable Protocols

Two shared variables may be required to process GPIB calls: a control variable and a data variable. The control variable is used to select the operation to perform, and the data variable is used to send or receive additional data as required by the command.

The control variable must be offered first. For some GPIB calls the control variable is all that is needed. For other calls, the data variable is also required. At the time a data variable is required, AP 488 matches the earliest pending corresponding data variable offered by the partner (after the control variable).

The name of the data variable must always begin with the letter `'D'` or the letters `'DAT'`, and the control variable must begin with the letter `'C'` or the letters `'CTL'`. The remaining characters in both names (possibly none) must be the same, because the coupling of both variables is recognized by their name. Examples of valid pairs are: `C` and `D`, `C1` and `D1`, and `CXjj` and `DXjj`. Also accepted as valid pairs are `CTL` and `DAT` or `CTL1` and `DAT1`. For example:

```
      488 SVOPAIR 'C488' 'D488'
2 1
```

offers two shared variables to AP 488 and sets access control. See Using the Share-Offer Utilities for a description of the `SVOPAIR` function (from the 1 UTILITY workspace).

Note:

© Copyright IBM Corporation 1994, 2017

Although AP 488 runs asynchronously from the APL2 interpreter, the interface between AP 488 and the GPIB device is a synchronous call. If a GPIB request does not return to AP 488, AP 488 will hang. You can free up your APL2 session by interrupting the shared variable reference, but AP 488 will continue to wait. To cancel the hung GPIB request, you will need to kill AP 488. This can be done from the *SVP Monitor* window, in the **Processors** dialog of the **Info** menu item.

General Information

The general format of AP 488 processor calls is:

```
[D488←optional data ]
C488←command code,handle [,optional parms ]
VALUE←C488
[DATA←D488]
```

All commands return `VALUE`, which is unlikely to be zero. In most cases, the integer representation of `IBSTA` (the IEEE-488 sixteen bit integer status word) is returned. Two commands (19 - `IBRPP`, and 25 - `IBRSP`) return integer poll responses if no error has occurred.

If AP 488 has detected an error in the commands or data passed to it, and cannot call the GPIB interface routines, `VALUE` will be null (`⍳0`) and an error code will be returned in `DATA`. The error codes returned by AP 488 are defined in AP 488 Return Codes

`C488` accepts only integer values and `D488` accepts only character vectors. This is true for all commands.

In the command syntax descriptions for AP 488:

`DEVICE`
: refers to a device connected to the GPIB, and the device-level commands that may be sent to it.

`ADAPTER`
: refers to the GPIB adapter board, and the board-level commands that may be sent to it.

`EITHER`
: refers to either an adapter or a device.

IBSTA Status Word Layout

The status variable is a sixteen bit integer variable. The bits have the following meanings:

Bit	Value	Name	Meaning
0	0000000000000001	DCAS	Device in Device Clear state
1	0000000000000010	DTAS	Device in Device Trigger state
2	0000000000000100	LACS	Device is Listener
3	0000000000001000	TACS	Device is Talker
4	0000000000010000	ATN	Attention is asserted
5	0000000000100000	CIC	Device is Controller-In-Charge
6	0000000001000000	REM	Device is in Remote State
7	0000000010000000	LOK	Device is in Lockout State

© Copyright IBM Corporation 1994, 2017

Bit	Value	Name	Meaning
8	0000000100000000	CMPL	I/O Completed
9	0000001000000000		Reserved
10	0000010000000000		Reserved
11	0000100000000000	RQS	Device Requires Service
12	0001000000000000	SRQI	SRQ Detected
13	0010000000000000	END	Device Detected END or EOS
14	0100000000000000	TIMO	Time Limit Exceeded
15	1000000000000000	ERR	GPIB Error

When ERR is true, the sixteen bit integer status word is returned to APL2 as a negative value, which indicates that an error or abnormal condition has occurred. Any positive return code is good.

- `IBSTA` bits are set.
- `IBERR` numeric return code is available through command 20 (`IBSTAT`).

The sixteen bits may be easily decoded with `(16⍴2)⊤IBSTA`.

IBERR Error Number

Code	Meaning
0	Driver or operating system error
1	Function requires GPIB adapter to be Controller-In-Charge
2	Write function detected no listeners
3	Interface adapter not addressed correctly
4	Invalid argument to function call
5	Function requires GPIB adapter to be System Active Controller
6	I/O operation aborted
7	Nonexistent interface adapter
8	Reserved
9	Reserved
10	Asynchronous operation not complete
11	No capability for operation
12	Unable to access file
13	Reserved
14	Bus command error during device call
15	Serial Poll status byte lost
16	SRQ remains asserted.

AP 488 Commands

© Copyright IBM Corporation 1994, 2017

Forty-four commands are defined to auxiliary processor 488, numbered 0 through 43. Each numeric command code has a matching APL2 function in the AP488 workspace, as follows:

Code	Function	Description
0	IBWAIT	Wait for Selected Event
1	IBONL	Online or Offline
2	IBRSC	Request or Release System Control
3	IBSIC	Send Interface Clear
4	IBSRE	Set or Clear Remote Enable Line
5	IBLOC	Go to local
6	IBRSV	Request Service
7	IBPPC	Parallel Poll Configure
8	IBPAD	Change Primary Address
9	IBSAD	Change or Disable Secondary Address
10	IBIST	Individual Status Bit
11	IBDMA	Enable or disable DMA
12	IBEOS	Change or Disable EOS Method
13	IBTMO	Change or Disable Timeout Limit
14	IBEOT	Enable or Disable END Message
15	IBGTS	Active Controller Go To Standby
16	IBCAC	Become Active Controller
17	IBRDF	Read Data Into File
18	IBFIND	Open Device or Adapter File Handle
19	IBRPP	Conduct Parallel Poll
20	IBSTAT	Return IBSTA, IBERR, IBCNT
21	IBSTOP	Stop asynchronous Operation
22	IBCLR	Clear a specific device
23	IBTRG	Trigger selected device
24	IBPCT	Pass control to another GPIB device with Controller capability
25	IBRSP	Conduct a serial poll
26	IBBNA	Change the access board of a device
27	IBSIZE	Set data buffer size
28	IBRD	Read Data
29	IBRDA	Read Data Asynchronously
30	IBWRT	Write Data
31	IBWRTA	Write Data Asynchronously
32	IBCMD	Send GPIB commands
33	IBCMDA	Send GPIB commands asynchronously
34	IBWRTF	Write Data From File
35	IBXTRC	Reserved

© Copyright IBM Corporation 1994, 2017

Code	Function	Description
36	IBTRAP	Reserved
37	IBPOKE	Set Device Driver Parameters
38	IBDIAG	Get Diagnostic Data
39	IBASK	Get Software Configuration Parameters
40	IBCONFIG	Change Software Configuration Parameters
41	IBDEV	Open and Initialize Device
42	IBLINES	Get Status of Control Lines
43	IBLN	Check for Device

0 (IBWAIT) - Wait for Selected Event

```
C488←0,EITHER,MASK
IBSTA←C488
```

Causes the system to wait for any of the events specified in the `MASK` integer. The mask layout is exactly the same as that of the `IBSTA` status word.

1 (IBONL) - Online or Offline

```
C488←1,EITHER,FLAG
IBSTA←C488
```

The inverse of `IBFIND`.

If `FLAG` is zero, the unit descriptor is closed. If `FLAG` is not zero, it does nothing (except waste time).

2 (IBRSC) - Request or Release System Control

```
C488←2,ADAPTER,FLAG
IBSTA←C488
```

Enables or disables system controller functions (Remote Enable, Interface Clear). The IEEE-488 specification does not specifically permit schemes where system control may be passed back and forth between instruments, but it does not forbid them either. This command would be used in such a scheme.

If `FLAG` is zero, system control is released. If `FLAG` is not zero, system control is requested. If no error occurs, the previous System Controller state is returned in `IBERR`.

3 (IBSIC) - Send Interface Clear

```
C488←3,ADAPTER
IBSTA←C488
```

Causes the adapter board to assert the *Interface Clear* signal if the adapter is currently supporting system controller functions.

© Copyright IBM Corporation 1994, 2017

4 (IBSRE) - Set or Clear Remote Enable Line

```
C488←4,ADAPTER,FLAG
IBSTA←C488
```

Asserts the *Remote Enable* signal if `FLAG` is non-zero, else it unasserts it.

Note: Most instruments won't go into remote until they have been addressed.

5 (IBLOC) - Go to Local

```
C488←5,EITHER
IBSTA←C488
```

Temporarily removes *Local Lockout* from the specified instrument. Normally, this will re-enable front panel controls. The next time the instrument is accessed via the interface, *Local Lockout* will resume. There is no way to turn off this condition permanently, except by changing the device configuration.

6 (IBRSV) - Request Service

```
C488←6,ADAPTER,STATUS
IBSTA←C488
```

Sets the byte that is returned when the system controller performs a serial poll.

This command is used when the adapter is configured as an instrument, not the system controller.

If `STATUS` has the `0x40` bit on, this command will also assert `SRQ`.

7 (IBPPC) - Parallel Poll Configure

```
C488←7,EITHER,CONFIG
IBSTA←C488
```

Configures an instrument or an adapter to respond to a parallel poll. Be certain that the device to be configured has a GPIB address in the range of zero through seven.

For information on the `CONFIG` integer, see the GPIB Function Reference Manual.

8 (IBPAD) - Change Primary Address

```
C488←8,EITHER,ADDRESS
IBSTA←C488
```

Permits you to override the primary address of the device or adapter that was specified by `IBCONF`. This change remains in effect until `IBONL` is called, or your program ends.

9 (IBSAD) - Change or Disable Secondary Address

© Copyright IBM Corporation 1994, 2017

```
C488←9,EITHER,ADDRESS
IBSTA←C488
```

Permits you to override the secondary address of the device or adapter that was specified by IBCONF. This change remains in effect until IBONL is called, or your program ends.

10 (IBIST) - Individual Status Bit

```
C488←10,ADAPTER,FLAG
IBSTA←C488
```

Sets the parallel poll response bit to true if FLAG is not zero, else the routine sets the response bit to false. The adapter must have been previously configured to respond to a parallel poll.

11 (IBDMA) - Enable or disable DMA

```
C488←11,ADAPTER,FLAG
IBSTA←C488
```

Permits you to temporarily disable and re-enable DMA transfers.

If FLAG is zero, DMA is disabled and the adapter will use programmed I/O exclusively. If FLAG is not zero, then the adapter will use DMA.

12 (IBEOS) - Change or Disable EOS Method

```
C488←12,EITHER,FLAGWORD
IBSTA←C488
```

Changes or disables the *End-Of-String* method.

FLAGWORD specifies both the EOS character, and what to do when it is detected during a read or write. If FLAGWORD is 0, the EOS configuration is disabled. If it is non-zero, the low byte is the EOS character and the high byte is the configuration bit.

For further information on the FLAGWORD integer, see the GPIB Function Reference Manual.

13 (IBTMO) - Change or Disable Timeout Limit

```
C488←13,EITHER,FLAGWORD
IBSTA←C488
```

Changes the amount of time that the interface will wait before reporting a timeout error. Limits range from 10 microseconds through one thousand seconds, or forever.

FLAGWORD ranges from zero through seventeen and is detailed in the IBTMO function description.

14 (IBEOT) - Enable or Disable END Message

© Copyright IBM Corporation 1994, 2017

```
C488←14,EITHER,FLAG
IBSTA←C488
```

Specifies whether or not EOI is set concurrently with the last byte of the data.

If FLAG is zero, then EOI is not sent. If FLAG is non-zero, then EOI is sent.

EITHER may be either an adapter or a device. If the handle refers to an adapter, then the value specified by FLAG overrides the specification on all devices attached to the adapter card.

15 (IBGTS) - Active Controller Go To Standby

```
C488←15,ADAPTER,FLAG
IBSTA←C488
```

Unasserts the ATN line and goes to the standby state.

This command is very useful for transferring data between two instruments without bothering to read it into APL2 first. It is normally used in conjunction with IBCMD.

If FLAG is non-zero, then the adapter monitors the data transfer and goes into a *Not Ready For Data* state when the END message is detected. This permits synchronous resumption of control via IBCAC. If FLAG is zero, then no monitoring is performed.

16 (IBCAC) - Become Active Controller

```
C488←16,ADAPTER,FLAG
IBSTA←C488
```

Resumes control of the GPIB system.

If FLAG is zero, then control is forced immediately (and possibly asynchronously with respect to data transfer). If FLAG is non-zero, then control is resumed synchronously with respect to data transfer.

17 (IBRDF) - Read Data Into File

```
D488←'FULL_FILE_SPECIFICATION'  (Including Path)
C488←17,EITHER
IBSTA←C488
```

Reads data from the GPIB and writes it to an operating system file.

Any data already in the file is over-written.

The transfer will end when either the END or the EOS message is detected.

18 (IBFIND) - Open Device or Adapter File Handle

© Copyright IBM Corporation 1994, 2017

```
D488←'UNIT_NAME'
C488←18,0
RESULT←C488
```

Performs an *open* on the specified device. You supply the name of the instrument or adapter board in `D488`, and the command returns a unit descriptor.

`RESULT` is the file handle or unit descriptor if the integer that is returned is positive, or `IBSTA` if the integer is negative.

19 (IBRPP) - Conduct Parallel Poll

```
C488←19,EITHER,0
RESULT←C488
```

Causes the adapter to perform a parallel poll.

`RESULT` is the response byte from the poll (if the value is positive) or `IBSTA` if the value is negative (an error was detected).

`EITHER` may be either an adapter or a device. If the handle refers to a device, the software will actually perform a parallel poll on the adapter board that owns the device.

20 (IBSTAT) - Return IBSTA, IBERR, IBCNT

```
C488←20
(IBSTA IBERR IBCNT)←C488
```

Retrieves the current values of `IBSTA`, `IBERR`, and `IBCNT`. No actual GPIB activity results, this command only returns three integers from the auxiliary processor.

21 (IBSTOP) - Stop Asynchronous Operation

```
C488←21,EITHER
IBSTA←C488
```

Causes any asynchronous operations currently in progress to be aborted.

22 (IBCLR) - Clear Device with Selected Device Clear

```
C488←22,DEVICE
IBSTA←C488
```

Clears (or attempts to clear) the internal device dependent functions of the specified instrument. The routine actually sends *Selected Device Clear* (`SDC`) to the device. Not all instruments support the `SDC` message.

23 (IBTRG) - Trigger Device

© Copyright IBM Corporation 1994, 2017

```
C488←23,DEVICE
IBSTA←C488
```

Sends the GET (*Group execute trigger*) message to the device specified by DEVICE.

Many devices do not support this function.

24 (IBPCT) - Pass Control

```
C488←24,DEVICE
IBSTA←C488
```

Passes *Controller-In-Charge* authority to the specified device.

Be *certain* that the device specified can support controller functions.

25 (IBRSP) - Conduct Serial Poll

```
C488←25,DEVICE,0
RESULT←C488
```

Performs a serial poll of the specified device.

RESULT is the response byte from the poll (if the value is positive) or IBSTA if the value is negative (an error was detected).

26 (IBBNA) - Change Adapter Name

```
D488←'GPIBx'
C488←26,DEVICE
IBSTA←C488
```

Changes the adapter used to access the instrument specified by DEVICE.

D488 is assigned a five character name that consists of *GPIB* followed by the character zero through three.

27 (IBSIZE) - Set Data Buffer Size

```
C488←27,BUFFSIZE
IBSTA←C488
```

Sets the default maximum read buffer size in the auxiliary processor. No I/O is performed.

If no IBSIZE call is ever issued, the default size used is 1024.

Maximum read buffer size can also be passed as an optional parameter to IBRD and IBRDA. In that case, the default set by IBSIZE is temporarily overridden during the IBRD or IBRDA call.

© Copyright IBM Corporation 1994, 2017

28 (IBRD) - Read Data

```
C488←28,EITHER[,MAX_SIZE]
IBSTA←C488
DATA←D488
```

Reads data from the specified device or adapter and returns it to APL2 through `D488`.

If `MAX_SIZE` is not specified, the last size specified with `IBSIZE` will be used. If a size has never been specified with `IBSIZE` a default size of 1024 will be used.

29 (IBRDA) - Read Data Asynchronously

```
C488←29,EITHER[,MAX_SIZE]
IBSTA←C488
DATA←D488
```

Reads data asynchronously from the specified device or adapter and returns it to APL2 through `D488`. This call may be used in place of `IBRD` when the application program needs to continue execution while the GPIB I/O operation is in progress.

If `MAX_SIZE` is not specified, the last sized specified with `IBSIZE` will be used. If a size has never been specified with `IBSIZE` a default size of 1024 will be used.

30 (IBWRT) - Write Data

```
D488←DATA
C488←30,EITHER
IBSTA←C488
```

Writes data from a character vector to the instrument that is specified by `EITHER`.

31 (IBWRTA) - Write Data Asynchronously

```
D488←DATA
C488←31,EITHER
IBSTA←C488
```

Writes data asynchronously from a character vector to the instrument that is specified by `EITHER`. This call may be used in place of `IBWRT` when the application program needs to continue execution while the GPIB I/O operation is in progress.

32 (IBCMD) - Send GPIB Commands

```
D488←COMMANDS
C488←32,ADAPTER
IBSTA←C488
```

Sends GPIB commands out through the adapter. You may send any valid sequence of IEEE-488 commands.

© Copyright IBM Corporation 1994, 2017

See the IBCMD workspace function description for a list of the GPIB board-level commands.

33 (IBCMDA) - Send GPIB Commands Asynchronously

```
D488←COMMANDS
C488←33,ADAPTER
IBSTA←C488
```

Sends GPIB commands asynchronously out through the adapter. You may send any valid sequence of IEEE-488 commands. This call may be used in place of IBCMD when the application program needs to continue execution while the GPIB commands are being processed.

34 (IBWRTF) - Write Data From File

```
D488←'FULL_FILE_SPECIFICATION'  (Including Path)
C488←34,EITHER
IBSTA←C488
```

Reads data from an operating system file, and sends it to the specified device (or adapter) as one long record. No translation is done.

37 (IBPOKE) - Set Device Driver Parameters

```
C488←37,EITHER,SUB_FUNCTION,VALUE
IBSTA←C488
```

Sets the device driver parameter identified by SUB_FUNCTION to VALUE.

38 (IBDIAG) - Get Diagnostic Data

```
C488←38,EITHER,DATA_SIZE
IBSTA←C488
DIAG_DATA←D488
```

Returns a data block of length DATA_SIZE from the device driver's unit descriptor. The maximum DATA_SIZE for an adapter is 99 bytes and for a device is 37 bytes.

39 (IBASK) - Get Software Configuration Parameters

```
C488←39,EITHER,OPTION
IBSTA←C488
VALUE←D488
```

Return information about software configuration parameters.

OPTION is a numeric code for a configuration parameter. VALUE is the current value for that parameter.

For a complete list of parameters, see the GPIB Function Reference Manual.

© Copyright IBM Corporation 1994, 2017

40 (IBCONFIG) - Change Software Configuration Parameters

```
C488←40,EITHER,OPTION,VALUE
IBSTA←C488
```

Change a software configuration parameter for a device.

`OPTION` is a numeric code for a configuration parameter. `VALUE` is the value the parameter is to be set to.

For a complete list of parameters, see the GPIB Function Reference Manual.

41 (IBDEV) - Open and Initialize Device

```
C488←41,BDINDEX,PAD,SAD,TMO,EOI,EOS
RESULT←C488
```

Open and initialize a device descriptor. The arguments are as follows:

`BDINDEX`
: Index of the access board for the device

`PAD`
: The primary GPIB address of the device

`SAD`
: The secondary GPIB address of the device

`TMO`
: The I/O timeout value

`EOI`
: EOI mode of the device

`EOS`
: EOS character and modes

`RESULT` is the unit descriptor if the integer that is returned is positive, or `IBSTA` if the integer is negative.

For more information on the valid values for the parameters to this call, see the GPIB Function Reference Manual.

42 (IBLINES) - Get Status of Control Lines

```
C488←42,EITHER
IBSTA←C488
STATUS←D488
```

Returns the status of the eight GPIB control lines.

`STATUS` is the numeric representation of the 8-bit mask of state information.

43 (IBLN) - Check for Device

```
C488←43,EITHER,PAD,SAD
```


© Copyright IBM Corporation 1994, 2017

```
IBSTA←C488
STATUS←D488
```

Check for the presence of a device on the bus.

`PAD` is the primary GPIB address of the device. `SAD` is the secondary GPIB address of the device. Use `0` for `SAD` to test only a primary address. Use `¯1` for `SAD` to test all secondary addresses.

`STATUS` is non-zero if a Listener is detected, `0` if not detected.

AP 488 Return Codes

The following table lists the codes returned in the AP 488 data variable when null is returned in the control variable.

Code	Meaning
`¯1`	Unexpected SVP error
`¯2`	Data variable not shared
`¯3`	Data variable interlocked
`¯4`	Data variable has no value
`¯5`	LENGTH ERROR in data variable
`¯6`	RANK ERROR in data variable
`¯7`	DOMAIN ERROR in data variable
`¯8`	Invalid command in control variable
`¯9`	LENGTH ERROR in control variable
`¯10`	RANK ERROR in control variable
`¯11`	DOMAIN ERROR in control variable
`¯12`	Insufficient storage to allocate read/write buffer

© Copyright IBM Corporation 1994, 2017

Supplied Workspaces

IBM APL2 supplies a number of workspaces to perform various tasks, and many of these are common to other supported APL2 environments. These workspaces contain functions that you can call from your programs. The functions listed can also be used as examples of how to program with auxiliary processors they utilize. For example:

- FILE uses the file processors (AP 210 and AP 211)
- The host workspaces (AIX, LINUX, SOLARIS, and WINDOWS) use the host command processor (AP 100)
- DEMO124 and AP124 use the text display processor (AP 124)
- DEMO145, DDESHARE, and GUITOOLS use the GUI services processor (AP 145)
- DEMO207 and GRAPHPAK use the Universal Graphics processor (AP 207)
- SQL uses the DB2 processor (AP 127) or the ODBC processor (AP 227)

Each public workspace contains variables with information about the workspace. These variables are: `ABSTRACT`, `DESCRIBE`, and `HOW`.

The workspaces distributed with APL2 are assigned to either library 1 or library 2, as is shown in following table:

Name	Library	Systems	Function
AIX	2	AIX	Tools to access AIX functions and commands
AP124	2	All	Aids in building text display applications
AP144	2	All Unix	Aids in building X Window system applications
AP488	2	Windows	Aids in using the GPIB support auxiliary processor
DDESHARE	2	Windows	Dynamic Data Exchange shared variable utilities
DEMO124	2	All	Demonstrations of some of the capabilities of the text display auxiliary processor (AP 124)
DEMO144	2	All Unix	Demonstrations of some of the capabilities of the X Window system auxiliary processor (AP 144)
DEMO145	2	Windows	Demonstrations of some of the capabilities of the GUI services auxiliary processor (AP 145)
DEMO207	2	All	Demonstrations of some of the capabilities of the universal graphics auxiliary processor (AP 207)
DEMOJAVA	2	All	Demonstration and utility functions for Associated Processor 14, the Calls to Java processor
DISPLAY	1	All	Display of array data structure
EDIT	1	All	Compatibility editors
EXAMPLES	1	All	Usage examples of APL2
FILE	2	All	Tools for accessing operating system files

© Copyright IBM Corporation 1994, 2017

Name	Library	Systems	Function
GRAPHPAK	2	All	Business and analytic graphics
GUITOOLS	2	Windows	A toolkit for writing Graphical User Interface (GUI) applications
GUIVARS	2	Windows	A collection of variables defining constants useful in programming GUI applications
IDIOMS	1	All	Catalog of common APL2 phrases
LINUX	2	Linux	Tools to access Linux functions and commands
MATHFNS	1	All	Mathematical functions
MIGRATE	2	All	Conversion of applications migrated from other systems
NETTOOLS	2	All	Network tools including an HTTP 1.1 web server
PRINT	2	All Unix	Printing utilities
PRINTWS	2	All	Printing workspace contents
SOLARIS	2	Solaris	Tools to access Solaris functions and commands
SQL	2	All	Tools for using SQL auxiliary processors
TCL	2	All	Demonstration and utility functions that use the TCL external function interface to Tcl.
TIME	1	All	Monitor performance
UTILITY	1	All	Manipulations and services beyond those provided by primitive operations
WINDOWS	2	Windows	Tools to access Windows functions and commands
WSCOMP	1	All	Comparing the contents of workspaces.

© Copyright IBM Corporation 1994, 2017

Host Workspaces - AIX, LINUX, SOLARIS, WINDOWS

The host workspaces contain functions for performing operating system dependent tasks.

- Operating System Commands
- Listing the APL2 Libraries
- Error Code Translation
- Character Set Translation

Operating System Commands

The set of functions described in this section perform some useful operating system tasks, such as renaming files and creating directories. They are common to all the host workspaces. Note, however, that while these functions have the same names in each host workspace, their contents are slightly different from operating system to operating system. When porting an application from one system to another, new copies of these functions should be obtained from the workspace on the new system to ensure correct operation.

The host workspaces use AP 100 to invoke operating system commands.

Except where noted, these are monadic functions. All functions return either `0` or data if successful. They return a numeric error code if an error occurs. The functions available are:

`Z←DIR path`
: Returns a directory list. This is equivalent to the `dir` command on Windows, and the `ls` command on Unix systems. The right argument, `path`, is the path leading to the directory whose contents are to be displayed.

`Z←ERASE filename`
: Erases a file. This is equivalent to the `delete` command on Windows, and the `rm` command on Unix systems. `filename` is the name of the file to be erased, and can include a path definition.

`Z←HOST cmd`
: Issues command `cmd` to the host operating system through AP 100. When `HOST ''` is executed (`cmd` is null), it returns the name of the currently running operating system.

`Z←MKDIR path`
: Creates a new subdirectory.

`Z←cmd PIPE data`
: Provides a means to pipe APL2 data to an operating system command and returns any output generated by the command.

  The right argument, `data`, must be either a simple character vector, a character matrix, or a vector of character vectors specifying the input to be passed to the command as *stdin* (standard input). If this is an empty vector, no input is passed to the command. The left argument, `cmd`, must be a simple character vector specifying the command to be executed. The result is a vector of vectors containing the output *stdout* (standard output) generated by executing the command. Some examples of the use of the `PIPE` function:

  To get names of APL workspaces and transfer files, sorting by date:

```
⊃'dir /b/od *.atf *.apl' PIPE ''
```


© Copyright IBM Corporation 1994, 2017

To run batch scripts through the interpreter:

```
JOB1←')LOAD 1 DISPLAY' 'DISPLAY ''APL2'' 123'
JOB2←')LOAD 2 WINDOWS' '∇PIPE[⎕]∇'
OUTPUT←(⊂'apl2 -sm piped -quiet on')PIPE¨JOB1 JOB2
```

```
Z←RENAME oldnew
Z←old RENAME new
```

Renames a file. When called monadically, the right argument, `oldnew`, has the form `'oldname newname'` where `oldname` is the name of the file to be renamed (and may include a path definition), and `newname` is the new name by which the file is to be known. `RENAME` can also be used dyadically, with the old name as the left argument and the new name as the right argument.

```
Z←RMDIR path
```

Removes the subdirectory pointed to by `path`.

Listing the APL2 Libraries

```
Z←LIBS
```

A Processor 11 external function which returns a character matrix giving the definition of each valid library number for this APL2 session. For more information, see LIBS - Get APL2 Library Definitions.

Error Code Translation

```
Z←CHECK_ERROR num
```

A Processor 11 external function which converts the message number specified by the right argument to a text error message. The message number can be a number returned by one of the other functions in this workspace, or a return code from an external function or auxiliary processor that is not defined for that function or processor. For more information, see CHECK_ERROR - Get System Error Text.

Character Set Translation

The WINDOWS workspace includes two functions for translating between the APL2 and the Windows Character Set:

```
R←APL2_TO_WINDOWS 'APL2'
R←WINDOWS_TO_APL2 'Windows'
```

The arguments are simple character arrays. `'Windows'` is data that was created using a Windows font or application. `'APL2'` is data that was created using an APL2 font or application. `R` is the translated data.

Note: Because the APL2 and Windows character sets contain different characters, these functions are not complete inverses.

© Copyright IBM Corporation 1994, 2017

AP124 Workspace

This workspace contains a set of cover functions to assist in the use of the text display auxiliary processor, AP 124, in an application. A screen definition facility makes it possible to define a screen as a set of fields, each one with its own name or number, where information can be sent or retrieved by means of appropriate functions.

- Fundamentals
- Building a Menu
- Primary Functions
- Additional Functions
- Example Driver Function
- AP124 Internal Operation and Global Variables

Fundamentals

The window you are using on your display should be regarded as a character array for the purposes of the following discussion.

The size of this array defaults to 25 by 80, but the window can be resized using normal desktop facilities. This area can be subdivided into smaller rectangular sections that are more convenient to your data processing needs. These rectangular areas are called fields. The text display auxiliary processor, AP 124, works only in terms of fields. Functions in the AP124 workspace perform actions to a field or a group of fields.

Every field has a type, which determines how it can be acted upon. There are three basic types of fields:

Input/output
: Allowing input from the keyboard to be typed and displayed

Numeric input only/any output
: Allowing only numbers to be input from the keyboard, but allowing any characters to be displayed

Output only
: Capable of receiving data from an APL2 function but not from the keyboard.

Fields also have attributes that define how the information contained in the fields is to be displayed. An attribute is expressed as an integer value, depending on the actual display to be used, and defines the color of the field.

Building a Menu

As an example of the use of the AP124 workspace, a sample screen is defined, containing some information and requesting data from a user. The AP124 workspace includes functions enabling you to define menus quickly, and making it easy to maintain them.

Assume the menu you wish to define is for an overtime payment system. You should collect and process the following data:

- Employee serial number
- Employee name
- Number of overtime hours worked each day (Monday to Sunday)

© Copyright IBM Corporation 1994, 2017

- Per-day overtime rate. This is fixed at the moment, but can be adjusted at a later date.

Now we have to step though some simple decisions to design the screen. The following is a representation of the desired menu:

```
  0000000001111111111222222222233333333334444444444555555555566666666667777777778
  1234567890123456789012345678901234567890123456789012345678901234567890123456789
01                             APL2 Overtime System
02--------------------------------------------------------------------------------
03
04
05                     Input the following data, press Enter:
06
07                        Employee Number   XXXXXXX
08                        Employee Name     XXXXXXXXXXXXXXX
09
10                           Mon  Tue  Wed  Thu  Fri  Sat  Sun
11                     Hours X.XX X.XX X.XX X.XX X.XX X.XX X.XX
12                     Rate  1.25 1.25 1.25 1.25 1.25 1.50 2.00
13
14
15
16
17                                /
18
19
20--------------------------------------------------------------------------------
21                   The following options can also be used:
22
23                       F1 - Help             F3 - Exit
24
25
```

The row and column numbers at the top and left of the diagram are included just for clarity.

A name is needed for the screen. Use the name OVERTIME.

Now that everything is planned and ready, we will look at the AP124 workspace functions we need to build the menu.

- FSDEF - Define fields
- FSSHOW - Display a panel

FSDEF - Define fields

```
FSDEF 'Menu_Name'
```

the first time it is used, and:

```
Field_Def FSDEF 'Field_Data'
```

once per field to be defined.

© Copyright IBM Corporation 1994, 2017

`Menu_Name` defines the name by which this screen is to be known. This name should begin with an uppercase or lowercase letter, the delta (∆) character or the delta underbar (⍙) character; it can continue with any of these, plus the digits 0-9, overbar (¯) or underbar (_).

`Field_Def` is a vector containing four, five, or six elements with the following information:

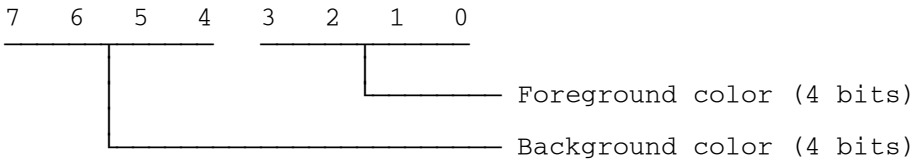
1. Start row of the field
2. Start column of the field
3. Field height
4. Field width

The fifth and sixth elements of the vector are optional, and are defined as:

5. Field type:

`0`	Input/output/selectable
`1`	Numeric input only/any output/selectable
`2`	Output only (the default)
`3`	Output only/selectable

6. Field Attribute: an integer between 0 and 255. The default field attribute is 1, which normally gives blue characters on a black background. The following diagram shows the meanings of the bits of the display attribute byte on color display adapters:



   The combinations of colors available are:

Code	**Bits**	**Color**
0	0 0 0 0	Black
1	0 0 0 1	Blue
2	0 0 1 0	Green
3	0 0 1 1	Cyan
4	0 1 0 0	Red
5	0 1 0 1	Magenta
6	0 1 1 0	Yellow
7	0 1 1 1	Gray
8	1 0 0 0	Light Gray
9	1 0 0 1	Light Blue
10	1 0 1 0	Light Green

© Copyright IBM Corporation 1994, 2017

Code	Bits	Color
11	1 0 1 1	Light Cyan
12	1 1 0 0	Light Red
13	1 1 0 1	Light Magenta
14	1 1 1 0	Brown
15	1 1 1 1	White

If the field width is given as 0, the width is automatically derived from the value of `Field_Data`. This is the data to be written on the field at initial menu usage. It must be a character vector.

Alternatively, the following field definition format can be used to assign a name to the field being defined:

```
Field_Def FSDEF 'Field_Name' FSDEF 'Field_Data'
```

`Field_Def` and `Field_Data` are the same as above. `Field_Name` is the name you wish to assign to this field. This name should consist of upper or lowercase letters, digits, delta (∆), delta underbar (⍙), overbar (¯) or underbar (_).

You can also use `FSDEF` to define a group of fields so that they can be referred to collectively. This is especially useful for reading or writing a set of fields or for changing the attribute of a complete set of fields, or for switching the type of a set of fields from input/output to output only, or vice versa. This is done in the following way:

```
'Group_Name' FSDEF Number_of_Fields
```

`Number_of_Fields` is the number of fields defined immediately before the execution of this line that are to be included in the group. You can define nonconsecutive fields as a group of fields by issuing this call with the same group name after each individual field or after each consecutive group of fields. Groups must be exclusive. Each field can be defined as being in only one group.

If workspace is at a premium, any repeated items in the screen definition variables can be converted to aliases of the first unique occurrence of each item by using `FSDEF` with an empty character right argument:

```
FSDEF ''
```

You can use this after each (or the last) menu is defined, and after you have copied menus into the active workspace, using the) `IN`,) `PIN`,) `COPY`, or) `PCOPY` system commands.

FSSHOW - Display a panel

```
FSSHOW 'Menu_Name'
```

`FSSHOW` can be used to display a menu after it has been completely defined.

The following example is a function written to define our sample menu and display it on the screen.

© Copyright IBM Corporation 1994, 2017

```
    ∇
[0]  DEFINE;A
[1]   FSDEF 'OVERTIME'
[2]   1 30 1 20 2 7 FSDEF 'APL2 Overtime System'
[3]   2 1 1 80 2 7 FSDEF 80⍴'-'
[4]   A←'Input the following data, press Enter:'
[5]   5 21 1 0 2 7 FSDEF 'PROMPT' FSDEF A
[6]   7 24 1 16 2 7 FSDEF 'Employee Number'
[7]   8 24 1 16 2 7 FSDEF 'Employee Name'
[8]   7 42 1 7 0 7 FSDEF 'Emp_No' FSDEF ''
[9]   8 42 1 15 0 7 FSDEF 'Emp_Name' FSDEF ''
[10]  A←'Mon  Tue  Wed  Thu  Fri  Sat  Sun'
[11]  10 27 1 0 2 7 FSDEF A
[12]  11 21 1 0 2 7 FSDEF 'Hours'
[13]  12 21 1 0 2 7 FSDEF 'Rate'
[14]  11 27 1 4 0 7 FSDEF 'Hours_Mon' FSDEF ''
[15]  11 32 1 4 0 7 FSDEF 'Hours_Tue' FSDEF ''
[16]  11 37 1 4 0 7 FSDEF 'Hours_Wed' FSDEF ''
[17]  11 42 1 4 0 7 FSDEF 'Hours_Thu' FSDEF ''
[18]  11 47 1 4 0 7 FSDEF 'Hours_Fri' FSDEF ''
[19]  11 52 1 4 0 7 FSDEF 'Hours_Sat' FSDEF ''
[20]  11 57 1 4 0 7 FSDEF 'Hours_Sun' FSDEF ''
[21]  'Hours' FSDEF 7
[22]  12 27 1 4 0 7 FSDEF 'Rate_Mon' FSDEF '1.25'
[23]  12 32 1 4 0 7 FSDEF 'Rate_Tue' FSDEF '1.25'
[24]  12 37 1 4 0 7 FSDEF 'Rate_Wed' FSDEF '1.25'
[25]  12 42 1 4 0 7 FSDEF 'Rate_Thu' FSDEF '1.25'
[26]  12 47 1 4 0 7 FSDEF 'Rate_Fri' FSDEF '1.25'
[27]  12 52 1 4 0 7 FSDEF 'Rate_Sat' FSDEF '1.50'
[28]  12 57 1 4 0 7 FSDEF 'Rate_Sun' FSDEF '2.00'
[29]  'Rates' FSDEF 7
[30]  18 21 1 40 2 7 FSDEF 'Msg_Area' FSDEF ''
[31]  20 1 1 80 2 7 FSDEF 80⍴'-'
[32]  A←'The following options can also be used:'
[33]  21 10 1 0 2 7 FSDEF A
[34]  A←'F1 - Help             F3 - Exit'
[35]  23 23 1 0 2 7 FSDEF A
[36]  FSDEF ''
[37]  FSSHOW 'OVERTIME'
    ∇
```

Notice that some of the fields have been named. These are the ones the example operates with. Also observe that an extra field, called `Msg_Area`, has been defined, where the program can output any errors found in input validation, or any other system message.

Ordinarily a function is written like this for each panel in the system. Each of these panel definition functions then needs to be executed only once in order to create the global variables that are used by other functions in this workspace. (They would have to be re-executed if you change the panels by editing the appropriate panel definition function.)

Primary Functions

Look at the next stage of combining the screen with the function that drives the input-output operation (a "driver"). The following cover functions can be used in the driver:

- FSUSE - Initialize a panel
- FSSETCURSOR - Position the cursor
- FSWRITE - Put data into fields
- FSWAIT - Wait for a user response
- FSREAD/FSREADV - Read field contents

© Copyright IBM Corporation 1994, 2017

FSUSE - Initialize a panel

```
Z←FSUSE 'Menu_Name'
```

This function initializes the menu named `Menu_Name`. This is the basic call used to allow you to start using a menu. It shares variables with AP124, loads the indicated predefined menu, and leaves you ready to use it. The result is 0 if successful, 1 if the function failed.

FSSETCURSOR - Position the cursor

```
Z←Cursor_Offset FSSETCURSOR Field
```

This call sets the cursor in a specific field or at any position on the screen. The left argument can be omitted, and the cursor offset is defaulted to the first position in that field. `Field` can be either a character vector containing the name you selected for the field during definition of the menu, or an integer field number. The result is 0 if successful, 1 if the function failed.

FSWRITE - Put data into fields

```
Z←Data FSWRITE Field
```

This function writes `Data` to the field or fields identified by `Field`. `Field` can be either a character scalar or vector to identify a single field, or a character matrix of names or a vector of names to identify several fields. `Field` can also be a numeric scalar or a vector of field numbers. If one field is being written, `Data` should be a character scalar, a vector, or a one-row matrix. If more than one field is being written, `Data` must be a character matrix with the corresponding number of rows or a vector of character arrays with the corresponding number of elements. The result is 0 if successful, 1 if the function failed.

FSWAIT - Wait for a user response

```
Z←FSWAIT
```

This function displays the active menu, and waits for user input. When a user presses certain keys, control returns to APL2, and the result, `Z`, of function `FSWAIT` is the following:

`Z[1]`	Return code: 0 if successful, otherwise 1.
`Z[2 3]`	Key pressed to complete the call, as defined in Special Key Code Processing.
`Z[4]`	Field number where cursor was located at return to APL2, or zero if it was outside all the fields.
`Z[5 6]`	Cursor offset (row/column) into that field. If field was zero, then offset is from the top-left corner of the screen.

© Copyright IBM Corporation 1994, 2017

`Z[7...]` List of fields updated during this request.

The list of updated fields that is returned enables you to optimize the panel processing time. In general, it is necessary to read and validate only those fields, rather than to read back and check *all* of the fields, which, for a very large screen, could be a very long process.

FSREAD/FSREADV - Read field contents

```
Z←FSREAD Field
```

This function reads data from the field or fields identified by `Field`. `Field` can be either a character scalar or vector to identify a single field, or a character matrix of names or a vector of names to identify several fields. `Field` can also be a numeric scalar or a vector of field numbers.

If the result, `Z`, is numeric, it is a return code indicating that the operation has failed. Otherwise, `Z` is a character matrix containing the requested field data.

```
Z←FSREADV Field
```

This function reads data from a field or a group of fields. If the result is numeric, it is a return code indicating that the operation failed. Otherwise, the result is a nested vector of character or numeric arrays containing the requested field or group field data.

Additional Functions

The following functions assist in the use of the screen.

```
Z←FSAPLOFF
Z←FSAPLON
```

`FSAPLOFF` and `FSAPLON` turn the keyboard from APL2 to National mode (`FSAPLOFF`), and vice versa (`FSAPLON`). The result is 0 if successful, 1 if the function failed.

```
Z←FSBEEP
```

`FSBEEP` sets the beep flag. A beep sounds at the next read and wait call (`FSWAIT`). The result is 0 if successful, 1 if the function failed.

```
Z←FSCLEAR
```


© Copyright IBM Corporation 1994, 2017

Clears the display. The result is 0 if successful, 1 if the function failed.

```
Z←FSCLOSE
```

FSCLOSE retracts the AP124 shared variables and expunges all global variables associated with the AP124 full-screen functions (except the 'fsf',Name panel definition variables). It should be used at the end of your session. The result is 0 if successful, 1 if the function failed.

```
Z←FSCOPY
```

Returns the current screen contents as a matrix.

```
Z←FSFIELD Field_Name
Z←FSFIELD Field_Number
```

This function translates a field name to the corresponding field number, or vice versa.

```
Z←FSFORMAT
```

Returns, in Z, the active format array. If this is used immediately after an FSOPEN, it returns a format array of one field completely covering the screen. You can use this to determine how many rows and columns are available for display on the screen.

```
Z←FSINKEY
```

Waits for the user to press any single key, or reports on any key pressed since the most recent FSWAIT, FSINKEY, or FSSCAN. The result is a 6-element vector:

- Z[1] Return code: 0 if successful, otherwise 1.
- Z[2 3] Key pressed to complete the call, as defined in Extended Key Code Processing.
- Z[4] Field number where cursor was located at return to APL2, or zero if it was outside all the fields.
- Z[5 6] Cursor offset (row/column) into that field. If field was zero, then offset is from the top-left corner of the screen.

```
Z←Data FSIWRITE Field
```


© Copyright IBM Corporation 1994, 2017

Immediate write of `Data` to the selected fields on the display. It is used in the same way as the `FSWRITE` function, described in FSWRITE - Put data into fields. The result is 0 if successful, 1 if the function failed.

```
Z←FSMENUON
Z←FSMENUOFF
```

`FSMENUOFF` hides the menu in the AP 124 window, and `FSMENUON` shows the menu. The result is 0 if successful, 1 if the function failed. These functions have no effect on AIX and Sun.

Note: The menubar cannot be turned on or off until after at least one field has been formatted (`FSUSE` or `FSSHOW` functions).

```
Z←FSMODE 'mode'
```

Clears the screen and establishes display mode indicated (APL2/PC compatibility). The result is 0 if successful, 1 if the function failed.

```
Z←FSOPEN
```

Shares variables with AP124. This function is used internally by `FSUSE`. The result is 0 if successful, 1 if the function failed.

```
Z←FSSCAN
```

Scan for a key pressed. The result is a 6-element vector like that returned by `FSINKEY`, except that if no key had been pressed, `Z[23]` will be `¯1 ¯1`.

```
Z←FSSCREEN
```

Return the contents of the screen.

```
Z←A FSSETFI Field
```

Changes the attribute of a field or a group of fields to `A`. The result is 0 if successful, 1 if the function failed.

© Copyright IBM Corporation 1994, 2017

```
Z←T FSSETFT Field
```

Changes the type of a field or a group of fields to `T`. The result is 0 if successful, 1 if the function failed.

```
Z←FSSTATUS
```

Returns the status of the session:

`Z[1]` Return code of the AP124 status call (normally `0`)
`Z[2]` Keyboard mode (`0` = national mode, `1` = APL2 mode)
`Z[3]` Reserved; always 0
`Z[4]` Reserved; always 1
`Z[5]` Beep request pending (`1` = beep pending)
`Z[6]` Reserved; always 0
`Z[7]` Cursor mode (`0` = normal, `1` = field)

```
Z←FSTITLE Title
```

Sets the title bar on the AP 124 window to the character string named in `Title`. The result is 0 if successful, 1 if the function failed.

Note: The title bar cannot be set until after at least one field has been formatted (`FSUSE` or `FSSHOW` functions).

Example Driver Function

The following is an example of a driver function that uses the menu defined above:

```
[0]  DRIVER;R;A
[1]   →(FSOPEN)/E124
[2]   →(FSUSE 'OVERTIME')/E124
[3]   →(FSSETCURSOR 'Hours_Mon')/E124
[4]  ASK:→(↑R←FSWAIT)/E124
[5]   →(1=R[2])/FK
[6]  ⍝ Examine fields changed
[7]  ⍝ These are listed by 6↓R
[8]   A←FSREAD 'Emp_No' 'Emp_Name' 'Hours' 'Rates'
[9]   ....
[..] FK:....
[..] E124:'Full-screen error in driver'
```

AP124 Internal Operation and Global Variables

The `FSOPEN` function shares variables called `Cfs` and `Dfs` with AP124. It also creates a two-element numeric variable, called `fss`, containing the number of rows and columns that can be displayed.

© Copyright IBM Corporation 1994, 2017

The FSDEF function creates a screen definition variable with the name 'fsf',Name, where Name is the menu name. The screen definition is stored as a three- or four-column nested matrix with one row for each field defined. The columns are:

1. Field format (six-element numeric vector)
2. Field contents (character vector)
3. Field name (character vector, or empty vector for no name)
4. Group name that this field belongs to (character vector, or empty vector if not defined as a group member). This column is not present if no groups are defined.

The FSUSE function calls FSOPEN if the AP124 variables are not already shared, and then copies the group definition 'fsf',Name to variable fsf. It also adds an extra column to fsf containing the field numbers for each group definition.

© Copyright IBM Corporation 1994, 2017

AP144 Workspace

Note: This workspace is provided only on Unix systems.

AP 144 is an interface between APL2 and the X Window System. It enables a very large subset of the X Window System Xlib calls and data structures to be used from the APL2 environment, and in so doing enables APL2 to use a true windowing environment. Several sample APL2 programs using the interface are provided in the DEMO144 workspace.

Note: Using these functions rather than the AP 144 interface directly allows operation with the APL2/370 X Window System interface provided with the *Transmission Control Program/Internet Protocol Version 2 for VM* Licensed Program (Program Number 5375-FAL).

The AP144 workspace contains cover functions for the X Window System auxiliary processor (AP 144). For a tutorial on the use of the X Window System interface, see Using the X Window System Interface.

To call the X Window System from APL2, use the following APL2 function:

```
[[rc] [data] ←] [c] XWIN command [parm] [parm] ...
```

`command`
: The name of the X Window System call to be invoked, specified as an APL2 character vector.

`[parm]`
: Most of the X Window System calls require that additional input parameters be specified. These are given after the name of the call itself, in the order in which they are listed in the X Window System documentation.

`[c]`
: An optional vector of one or two elements controlling the behavior of the `XWIN` function. The two elements are independent of each other.

  The first or only element of `c` (`↑c`) determines what is returned by the function:

  - `0` Do not return a result.
  - `1` Return an integer return code `rc`.
  - `2` Return any result data produced by the call, if any. If no data is produced, nothing is returned by `XWIN`.
  - `3` Return a one- or two-element vector. The first element always contains the return code. The second element, if present, contains the result data generated by the call.

  The second element of `c` (`c[⎕IO+1]`) determines what happens if the call results in an error:

  - `0` Ignore any errors
  - `1` Display error messages describing the cause of any errors
  - `2` Halt execution
  - `3` Display error message and halt execution

© Copyright IBM Corporation 1994, 2017

If only the first element is given, the second is assumed to be equal to `0`. `⎕` can be omitted entirely, in which case the behavior is equivalent to a setting of `2 3`.

The different control options cater for many different possibilities. If called with a setting of `3 0`, `XWIN` returns a numeric error code as part of the function result. It is then up to the calling program to check this code and take appropriate action on a nonzero return code.

If, on the other hand, `XWIN` is called with the control setting of `2 3` (which is the default if no left argument is given), then `XWIN` provides a default error-handler to check the return codes as they are returned from each call to the X Window System. If an error is encountered, `XWIN` suspends operation in the function so that the programmer can correct the problem interactively.

The AP144 workspace also contains a couple of useful APL2 functions that extend the use of the structure commands:

```
(rc nl) ← prefix axGetFF struct
```

`axGetFF` creates a series of variables in the workspace that can be used when working with a particular structure class. The variables are created from the structure field and constant information retrieved by using the `'GetFields' struct` and `'GetConst' struct` commands.

The field information is used to create a set of index variables that can help you index a particular element in the data structure. These variables are all prefixed with a common, user-definable constant, since many structure fields use the same names.

Normally, this function would be run only if the C library is changed and the constants need to be reinitialized.

An APL2 variable is created for each constant, and the constant's value is assigned to that variable.

`struct`
: The name of the structure. The structure must already be defined to the X Window System interface.

`prefix`
: A prefix that is added to all the structure field names returned. This enables you to load the fields from multiple structures without any conflict of field names.

`rc`
: The function return code.

`nl`
: A list of the APL2 variables defined given as a vector of character vectors. This can be used to expunge all the defined variables once they are no longer needed by executing `⎕EX⍤¯1 nl`. Any **.** characters found in the C field names are replaced by the APL2 overbar (¯) to create a syntactically valid identifier.

Note: The field indexes created are sensitive to the `⎕IO` setting in place when the `axGetFF` is called; that is, the variable describing the first field in the structure has a content of either 0 or 1, depending on the setting of `⎕IO`.

Example:

© Copyright IBM Corporation 1994, 2017

```
      (RC NL)←'SH_' axGetFF 'XSizeHints'
      RC
0
      2↑NL
 SH_flags SH_x
      SH_flags
0
      SH_x
1
```

```
(rc values) ← names axGetFF1 struct
```

`axGetFF1` extracts the values of selected field indexes or constants defined in a C structure.

`struct`
: The name of the structure. The structure must already have been defined to the X Window System interface.

`names`
: One or more names of fields or constants defined in the structure. Only the actual C field name is considered in the structure definition, not any type of information.

`rc`
: The function return codes: `0` for value set, `29` if there is no field or constant by that name, or `32` if an unknown structure is specified.

`values`
: If a name matches a field name, the field index is returned. If it matches the name of a constant, the value of that constant is returned. Field names take precedence over constants, if there is a name overlap.

  All names must find a match, or the function does not complete successfully.

If any of the given names refer to field names, the values returned are dependent on the current `⎕IO` setting, as described above for `axGetFF`.

Examples:

```
      'flags' axGetFF1 'XSizeHints'
 0  0
      'x' axGetFF1 'XSizeHints'
 0  1
      'unknown' axGetFF1 'XSizeHints'
 29  29
```


© Copyright IBM Corporation 1994, 2017

AP488 Workspace

Note: This workspace is not provided on Unix systems.

The AP488 workspace provides a set of cover functions to make it easier to use AP 488 - GPIB Support Processor.

AP 488 is an interface to the device driver software that is supplied with the programming support for the National Instruments GPIB/IEEE-488 hardware. Most, but not all, of the functions that are available in the GPIB driver software package have been implemented.

If you have not already done so, please read the introductory chapters of the documentation supplied with the appropriate GPIB driver software before continuing with AP488. It is easy to become frustrated unless you are familiar with the basic concepts of the IEEE-488 standard.

Some hints to avoid trouble:

- Few instruments can support tri-state timing (the default with the National Instruments support packages). Unless you are certain that all instruments on your interface adapter can support this option, you should select the open collector interface instead.
- Local Lockout is good in a production environment, but not when you are setting up the hardware.
- Automatic Serial Polling is not always a good idea. Some instruments cannot respond to a serial poll.
- Do not try to read from, or write to a file that has the same name as an instrument. For example, if you have an instrument named `DVM`, do not try to access a file named `DVM.DAT`. This access will be interrupted and send the data to your instrument with sometimes amusing but always unpredictable results.
- Read your instrument manual *carefully*. Some instruments are *very sensitive* to the format of data that is sent to them. For example, one instrument may *require* line feed terminator on every message, while another may totally lock up if it receives one. *Data format is instrument specific and **not** part of the IEEE-488 standard.*

- Description of AP488 Functions
- Example of AP488 Usage

Description of AP488 Functions

All functions in this workspace return one or more values. For those functions that do not return a data value, the `IBSTA` status word is returned.

Before any of these functions may be used, you must execute the `SHARE_488` function to establish the appropriate shared variables. Similarly, when the usage of the AP488 functions is complete, you may execute `RETRACT_488` to retract and expunge the shared variables.

The following variables are reserved:

- `C488` - The AP488 control variable
- `D488` - The AP488 character vector data variable

© Copyright IBM Corporation 1994, 2017

- IBSTA - The GPIB status variable
- IBERR - The GPIB error number variable
- IBCNT - The GPIB auxiliary count variable.

In all functions, the word *device* refers to the integer that is returned by the IBFIND or IBDEV function. This always refers to an instrument. The word *adapter* is also an integer returned by IBFIND, but in this case it refers to the GPIB adapter board itself, not an instrument. The word *either* means either an instrument or an adapter board.

Function	Description
IBASK	Get Software Configuration Parameters
IBBNA	Change the access board of a device
IBCAC	Become Active Controller
IBCLR	Clear a specific device
IBCMD	Send GPIB commands
IBCMDA	Send GPIB commands asynchronously
IBCONFIG	Change Software Configuration Parameters
IBDEV	Open and Initialize Device
IBDIAG	Get Diagnostic Data
IBDMA	Enable or disable DMA
IBEOS	Change or Disable EOS Method
IBEOT	Enable or Disable END Message
IBFIND	Open Device or Adapter File Handle
IBGTS	Active Controller Go To Standby
IBIST	Individual Status Bit
IBLINES	Get Status of Control Lines
IBLN	Check for Device
IBLOC	Go to local
IBONL	Online or Offline
IBPAD	Change Primary Address
IBPCT	Pass control to another GPIB device with Controller capability
IBPOKE	Set Device Driver Parameters
IBPPC	Parallel Poll Configure
IBRD	Read Data
IBRDA	Read Data Asynchronously
IBRDF	Read Data Into File
IBRPP	Conduct Parallel Poll
IBRSC	Request or Release System Control
IBRSP	Conduct a serial poll
IBRSV	Request Service
IBSAD	Change or Disable Secondary Address

© Copyright IBM Corporation 1994, 2017

Function	Description
IBSIC	Send Interface Clear
IBSIZE	Set data buffer size
IBSRE	Set or Clear Remote Enable Line
IBSTAT	Return IBSTA, IBERR, IBCNT
IBSTOP	Stop asynchronous Operation
IBTMO	Change or Disable Timeout Limit
IBTRG	Trigger selected device
IBWAIT	Wait for Selected Event
IBWRT	Write Data
IBWRTA	Write Data Asynchronously
IBWRTF	Write Data From File
CHK_488	Check Return Code

IBASK - Get Software Configuration Parameters

```
VALUE←option IBASK either
```

Returns information about software configuration parameters.

`option` is a numeric code for a configuration parameter. `VALUE` is the current value for that parameter.

For a complete list of parameters, see the GPIB Function Reference Manual.

IBBNA - Change Adapter Name

```
Z←device IBBNA adaptername
```

Changes the adapter used to access the specified device.

`adaptername` is a string containing `GPIBx` where `x` is a number from zero through three.

This change is temporary and disappears after you leave APL2.

IBCAC - Become Active Controller

```
Z←flag IBCAC adapter
```

`flag` is zero to take control immediately (possibly asynchronously), and non-zero to force synchronous assumption of control with respect to data transfer.

© Copyright IBM Corporation 1994, 2017

`adapter` refers to a GPIBx adapter handle obtained from `IBFIND`.

IBCLR - Clear Device with Selected Device Clear

```
Z←IBCLR device
```

Sends the listen address(es) of the specified device followed by *Selected Device Clear* (`SDC`), unlisten and untalk.

Usually clears a device to some specified initial state.

Not all devices respond to `SDC`.

IBCMD - Send GPIB Commands

```
Z←gpib_commands IBCMD adapter
```

Sends the data in `gpib_commands` out through the specified adapter with `ATN` true.

It is up to you to ensure that the string contains valid GPIB commands.

The following is a table of IEEE-488 addresses with their character equivalents (in ASCII).

Listen Address	Talk Address	Device Number	Notes
0x20 (sp) 0x21 ! 0x22 " 0x23 # 0x24 $ 0x25 % 0x26 @ 0x27 '	0x40 @ 0x41 A 0x42 B 0x43 C 0x44 D 0x45 E 0x46 F 0x47 G	00 01 02 03 04 05 06 07	These addresses may be used with Parallel Polling.
0x28 (0x29) 0x2A * 0x2B + 0x2C , 0x2D - 0x2E . 0x2F / 0x30 0 0x31 1 0x32 2 0x33 3 0x34 4 0x35 5 0x36 6 0x37 7 0x38 8	0x48 H 0x49 I 0x4A J 0x4B K 0x4C L 0x4D M 0x4E N 0x4F O 0x50 P 0x51 Q 0x52 R 0x53 S 0x54 T 0x55 U 0x56 V 0x57 W 0x58 X	08 09 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24	Primary Listen Address = Device number + 32 Primary Talk Address = Device number + 64 Secondary addresses extend from 0x60 through 0x7E and are always device dependent.

© Copyright IBM Corporation 1994, 2017

Listen Address	Talk Address	Device Number	Notes
0x39 9 0x3A : 0x3B ; 0x3C < 0x3D = 0x3E >	0x59 Y 0x5A Z 0x5B [0x5C \ 0x5D] 0x5E ^	25 26 27 28 29 30	
0x3F ?	0x5F _		Unlisten/Untalk

IBCMDA - Send GPIB Commands Asynchronously

```
Z←gpib_commands IBCMDA adapter
```

Asynchronously sends the data in `gpib_commands` out through the specified adapter with `ATN` true.

May be used in place of `IBCMD` when the application program needs to continue execution while the GPIB commands are being processed.

It is up to you to ensure that the string contains valid GPIB commands.

See IBCMD for a table of IEEE-488 addresses with their character equivalents (in ASCII).

IBCONFIG - Change Software Configuration Parameters

```
Z←(option value) IBCONFIG either
```

Changes a software configuration parameter for a device.

`option` is a numeric code for a configuration parameter. `value` is the value the parameter is to be set to.

For a complete list of parameters, see the GPIB Function Reference Manual.

IBDEV - Open and Initialize Device

```
DEVICE←IBDEV info
```

Opens and initializes a device descriptor.

`info` is a 6-element integer vector containing:

1. Index of the access board for the device

© Copyright IBM Corporation 1994, 2017

2. The primary GPIB address of the device
3. The secondary GPIB address of the device
4. The I/O timeout value
5. `EOI` mode of the device
6. `EOS` character and modes

`DEVICE` is an integer handle that must be used in all subsequent device calls.

IBDIAG - Get Diagnostic Data

```
INFO←size IBDIAG either
```

Returns a data block of length `size` from the device driver's unit descriptor. The maximum `size` for an adapter is 99 bytes and for a device is 37 bytes.

IBDMA - Enable or disable DMA

```
Z←flag IBDMA adapter
```

Enables or disables DMA on the specified adapter, provided that DMA was not disabled when you configured your device driver.

If `flag` is zero, programmed I/O is used (temporarily). If `flag` is non-zero, DMA is reactivated.

IBEOS - Change or Disable EOS Method

```
Z←flag IBEOS either
```

Changes the way the `EOS` termination byte is handled by `IBRD` and `IBWRT`. See the GPIB Function Reference Manual for a full description.

IBEOT - Enable or Disable END Message

```
Z←flag IBEOT either
```

If `flag` is zero, the END message (`EOI`) is not sent concurrently with the last byte of an `IBWRT`. If `flag` is non-zero, then it is.

This can be very useful when you're making adapter-level writes.

IBFIND - Open Device or Adapter File Handle

© Copyright IBM Corporation 1994, 2017

```
EITHER←IBFIND unit_name
```

This function is the first thing that has to be done before an instrument or controller may be accessed.

`EITHER` is an integer handle that must be used in all subsequent device or board level calls.

If the integer returned is negative, then an error has occurred. See `IBSTA` and `IBERR` for a complete description of the error.

IBGTS - Active Controller Go To Standby

```
Z←flag IBGTS adapter
```

If `flag` is zero, disables the controller function. If `flag` is non-zero, then the controller is disabled, but monitors the bus waiting for an END message. When the END message is detected, the adapter enters the NRFD holdoff state.

This function is normally used in board level I/O calls.

IBIST - Individual Status Bit

```
Z←flag IBIST adapter
```

Although an adapter is specified, this function is used when the PC is NOT the active controller but rather a device being controlled elsewhere.

If zero, `flag` sets the parallel poll status bit false. If non-zero, it sets this bit true. The actual state of the bit (0 or 1) is specified by the external controller when it sends the parallel poll configure message.

IBLINES - Get Status of Control Lines

```
STATUS←IBLINES either
```

Returns the status of the eight GPIB control lines.

`STATUS` is the numeric representation of the 8-bit mask of state information.

IBLN - Check for Device

© Copyright IBM Corporation 1994, 2017

```
STATUS←(pad sad)IBLN either
```

Check for the presence of a device on the bus.

`pad` is the primary GPIB address of the device. `sad` is the secondary GPIB address of the device. Use `0` for `sad` to test only a primary address. Use `¯1` for `sad` to test all secondary addresses.

`STATUS` is non-zero if a Listener is detected, 0 if not detected.

IBLOC - Go to Local

```
Z←IBLOC either
```

Sends unlisten, listen address(es) of the specified device, "Go to Local" (`GTL`), unlisten and untalk. This temporarily overrides the "Local Lockout" state. (Local Lockout is useful in a production environment, but less so when you are setting up the hardware.)

IBONL - Online or Offline

```
Z←flag IBONL either
```

If `flag` is zero, the device or adapter is placed in an offline state (essentially a close function). The device descriptor is no longer valid and cannot be used to place the device back online!

If `flag` is non-zero, this function does nothing. `IBFIND` is the inverse of this function.

IBPAD - Change Primary Address

```
Z←address IBPAD either
```

Specifies a new primary address for subsequent GPIB activity.

`address` may range from zero through thirty (be sure not to conflict with anything already on the bus).

This function is primarily useful when you are adding a device to a system temporarily and don't want to configure it in permanently.

See also `IBSAD`, `IBEOS` and `IBEOT`.

IBPCT - Pass Control

© Copyright IBM Corporation 1994, 2017

```
Z←IBPCT device
```

Passes control of the GPIB bus to another controller. The adapter enters controller idle state at the end of this function (CIDS).

Be sure that the device you specify can act as a controller.

IBPOKE - Set Device Driver Parameters

```
Z←(sub_function value) IBPOKE either
```

Sets the device driver parameter identified by `sub_function` to `value`.

IBPPC - Parallel Poll Configure

```
Z←config IBPPC either
```

Configures an instrument or an adapter to respond to a parallel poll.

See the GPIB Function Reference Manual for a description of the `config` word.

IBRD - Read Data

```
DATA←[max_size] IBRD either
```

Reads data from the specified device or adapter until:

- `EOS` is detected (if active)
- `END` is detected (always)
- Buffer is full (always).

If `max_size` is not specified, the last size specified with `IBSIZE` will be used. If a size has never been specified with `IBSIZE` a default size of 1024 will be used.

IBRDA - Read Data Asynchronously

```
DATA←[max_size] IBRDA either
```

Reads data asynchronously from the specified adapter or device until:

© Copyright IBM Corporation 1994, 2017

- EOS is detected (if active)
- END is detected (always)
- Buffer is full (always).

If max_size is not specified, the last size specified with IBSIZE will be used. If a size has never been specified with IBSIZE a default size of 1024 will be used.

This function may be used in place of IBRD when the application program needs to continue execution while the GPIB I/O operation is in progress.

IBRDF - Read Data Into File

```
Z←filename IBRDF either
```

Performs a read from the specified device or adapter and sends the output to an operating system file rather than back to APL2.

filename is a character vector that may include a full drive, path, filename and extension specification.

The file is opened for output, not append, so only one record may be placed in each unique file.

IBRPP - Conduct Parallel Poll

```
RESPONSE←IBRPP either
```

Returns a parallel poll byte.

If you specify a device, it is mapped into the correct adapter instead.

IBRSC - Request or Release System Control

```
Z←flag IBRSC adapter
```

If flag is zero, all system control functions are disallowed until an IBONL followed by an IBFIND occurs.

IBRSP - Conduct Serial Poll

```
RESPONSE←IBRSP device
```

Returns the serial poll byte from the specified instrument as an integer.

© Copyright IBM Corporation 1994, 2017

If the integer is negative, an error has occurred. Analyze IBSTA, IBERR and IBCNT to find the exact problem.

IBRSV - Request Service

```
Z←flag IBRSV adapter
```

Puts flag into the serial poll response register of the specified controller.

If bit 6 (0x40) is true, then service is requested as well.

Normally used when the adapter is not the system controller.

IBSAD - Change or Disable Secondary Address

```
Z←address IBSAD either
```

Specifies a new secondary address for subsequent GPIB activity.

address may range from 0x60 through 0x7E normally. If address is zero or 0x7F then the secondary address is disabled.

This is a temporary function.

See also IBPAD, IBEOS and IBEOT.

IBSIC - Send Interface Clear

```
Z←IBSIC adapter
```

Sends the interface clear message for 100 microseconds.

The adapter must be the system controller.

IBSIZE - Set Data Buffer Size

```
Z←IBSIZE integer
```

Specifies the maximum data size for IBRD and IBRDA.

If no IBSIZE call is ever issued, the default size used is 1024.

© Copyright IBM Corporation 1994, 2017

Maximum data size can also be passed as an optional left argument to IBRD and IBRDA. In that case, the default set by IBSIZE is temporarily overridden during the IBRD or IBRDA call.

IBSRE - Set or Clear Remote Enable Line

```
Z←flag IBSRE adapter
```

If flag is zero, the remote enable line of the specified adapter is turned off; if flag is non-zero, it is turned on.

IBSTAT - Return IBSTA, IBERR, IBCNT

```
Z←IBSTAT
```

Refreshes workspace variables IBSTA, IBERR and IBCNT to their current values.

IBSTOP - Stop Asynchronous Operation

```
Z←IBSTOP either
```

Any asynchronous operations currently in progress are aborted.

IBTMO - Change or Disable Timeout Limit

```
Z←time IBTMO either
```

Changes the timeout limit on the specified device or adapter.

time may range from zero through seventeen. The value should be chosen so as to be sufficiently large to allow all expected operations to complete. Too small a value may cause data to be lost on commands like IBRD, IBWRT or IBCMD.

The following is a list of the available timeout control codes:

Mnemonic	Code	Time Out
TNONE	0	10 us
T10US	1	30 us
T30US	2	100 us
T100US	3	300 us

© Copyright IBM Corporation 1994, 2017

Mnemonic	Code	Time Out
T300US	4	1 ms
T1MS	5	3 ms
T3MS	6	10 ms
T10MS	7	30 ms
T30MS	8	100 ms
T100MS	9	300 ms
T300MS	10	1 s
T1S	11	3 s
T3S	12	10 s
T10S	13	30 s
T30S	14	100 s
T100S	15	300 s
T300S	16	1000 s
T1000S	17	Infinite

IBTRG - Trigger Device

```
Z←IBTRG device
```

Sends a "Group Execute Trigger" message (`GET`) to the specified device.

Not all instruments respond to `GET`.

IBWAIT - Wait for Selected Event

```
Z←mask IBWAIT either
```

Permits waiting for a specified event or events to occur.

`mask` is as defined by the `IBSTA` status word.

IBWRT - Write Data

```
Z←data IBWRT either
```

Sends data to the specified adapter or device until:

- `EOS` is detected (if active)

© Copyright IBM Corporation 1994, 2017

- Buffer is empty (always).

IBWRTA - Write Data Asynchronously

```
Z←data IBWRTA either
```

Sends data asynchronously to the specified adapter or device until:

- `EOS` is detected (if active)
- Buffer is empty (always).

This function may be used in place of `IBWRT` when the application program needs to continue execution while the GPIB I/O operation is in progress.

IBWRTF - Write Data From File

```
Z←filename IBWRTF either
```

Writes all of the data from the specified operating system file to the device or adapter.

A full drive, path, filename and extension specification may be given for `filename`.

No translation is done; ALL characters are sent. This includes any CR/LF's in the file and the Ctrl-Z that is placed in the file by many editors.

`EOI` is sent concurrent with the last byte of data.

CHK_488 - Check Return Code

This function may be used to validate the return codes generated by the functions in the AP488 workspace. It takes a right argument of the value returned by the functions, and a left argument listing the valid return codes that should be accepted.

Example of AP488 Usage

This example assumes:

- An adapter card with name GPIB0 at address zero
- A digital volt meter (DVM) with name DEV08 at address eight.

```
⍝  First establish the shared variables
SHARE_488
⍝  Now get the adapter card device handle
HANDLE ← IBFIND 'GPIB0'
⍝  Set Remote Enable Line
```


© Copyright IBM Corporation 1994, 2017

```
      ⍝  This puts device into remote mode
      ⍝  Suppress return code of 256
      256 CHK_488 1 IBSRE HANDLE
      ⍝  Set Interface Clear
      304 CHK_488 IBSIC HANDLE
      ⍝  Send command to GPIB:
      ⍝  Unlisten (ASCII "?"), PC Talk address, DVM Listen address.
      ⍝  Suppress both 376 and 312 return codes
      376 312 CHK_488 '?@(' IBCMD HANDLE
      ⍝  Now send command string to the DVM to initialize it such
      ⍝  that it will send data.  This is device dependent!
      372 296 CHK_488 'Command string' IBWRT HANDLE
      ⍝  DVM now ready to send out data.
      ⍝  Send commands to GPIB:
      ⍝  Unlisten, DVM Talk address, PC Listen address
      372 CHK_488 '?H ' IBCMD HANDLE
      ⍝  Now listening to DVM
      ⍝  Get the handle for the DVM
      HANDLEDVM ← IBFIND 'DEV08'
      ⍝  Read in the data
      VOLTS ← IBRD HANDLEDVM
      VOLTS
+1.234567E+02(cr,lf)
```


© Copyright IBM Corporation 1994, 2017

DDESHARE Workspace

Note: This workspace is not provided on Unix systems.

Dynamic Data Exchange (DDE) is used to share data between applications. Many programs such as spreadsheets, word processors, graphic display tools, databases, and language products support DDE. The DDESHARE workspace contains tools and sample applications for sharing data with these applications.

The following sections describe these facilities:

- The DDE Protocol
- Easy DDE Functions
- Objects and Utility Functions
- Sample Functions
- DDE Limitations

The DDE Protocol

DDE is a protocol for communication between programs. The tools in DDESHARE conceal most of the details of the protocol so that in many cases you can simply share variables with other applications. However, to make effective use of these shared variables, some background knowledge of the protocol and its terminology is required.

The DDE protocol defines two types of programs: servers and clients. DDESHARE provides tools to help you create both servers and clients.

DDE servers can provide two types of services:

1. Executing commands
2. Providing shared access to data

DDE clients can request two corresponding types of services:

1. Request that commands be executed
2. Request access to share data

Before a client can issue a request to a server, the client must *link* to the server. In order to establish a link, a client needs three pieces of information:

1. The name of the server application. For example, `Excel` and `APL2 DDE Server` are server names.
2. The name of a topic the server supports. Topic names are usually file names. For example, Excel uses spreadsheet file names as topic names.
3. The name of an item within the topic. For example, Excel item names refer to a cell or group of cells. The item name is not needed if the client is only going to issue commands.

When working with DDE applications, there are two common techniques for supplying this information to clients so they can link to servers:

© Copyright IBM Corporation 1994, 2017

1. **Copy** data to the clipboard from a *source* document managed by a server and **Paste Link** or **Paste Special** the data from the clipboard into a *destination* document managed by a client.

   When you **Copy** the data, the server puts the link information clients need on the clipboard. When you **Paste Special**, the client retrieves this information and uses it to link to the server.

2. Tell a client explicitly what server you want to link to and what data within the server you want to access.

DDESHARE provides tools to help you establish links using both of these techniques.

Easy DDE Functions

In many cases you simply need to share data with another application. For example, you want to be able to share data with or issue commands to a non-APL application such as a spreadsheet or graphics display tool. In these cases, the non-APL application can be the DDE server and your APL2 application can be the client. Three functions are provided in DDESHARE which make it easy to establish links with servers:

- PASTE_SPECIAL
- APL2_DDE_CONNECT
- APL2_XLTABLE_CONNECT

Some DDE applications, particularly word processors, do not operate as servers; they only operate as clients. When communicating with these applications, your application needs to be a DDE server so that the non-APL application can behave as a DDE client. The following function can be used to easily create a server for use with applications that only operate as clients:

- COPY_LINK

Many DDE applications use the tab character to delimit data. For example, some spreadsheets use tab to delimit columns. However, this use of tab is not part of the DDE text protocol and many applications such as word processors use tab for other purposes. So, APL2's support for DDE text does not automatically delimit data using tabs. The following functions are supplied for working with tab delimited text:

- DDE_MATRIX_TO_TEXT
- DDE_TEXT_TO_MATRIX
- DDE_TEXT_TO_MATRIX_NUM

These functions are not necessary when APL2 clients share data with Excel and other servers which support the **XLTABLE** data format. APL2 provides support that automatically converts between the APL2 and XLTABLE formats.

PASTE_SPECIAL

```
HDATA←'property' PASTE_SPECIAL 'variable_name'
```


© Copyright IBM Corporation 1994, 2017

After copying data to the clipboard from a server, use this function to share a variable with the data. The function retrieves the server's link information, creates a DDE DATA object, shares a variable with the object, and returns the object's handle.

Arguments:

variable_name
: Name of the variable to be shared with the DDE DATA object

property
: Name of the property to be shared. *property* defaults to `'DATA'`.
`'XLTABLE DATA'` can be used when sharing with Excel and other servers which support the XLTABLE data format.
For further information see Data Objects.

Result:

Handle of the object. If link information is not available on the clipboard, or the link cannot be established, zero is returned. The handle can be passed to `DESTROYOBJ` to destroy the object.

APL2_DDE_CONNECT

```
HANDLE←[link_info] APL2_DDE_CONNECT variable_name [class_name]
```

This function displays the active servers and topics available on the system and prompts the user to make a selection and enter an item; it then establishes the link. The function creates a DDE object, shares a variable with the object, and returns the object's handle. When it completes, you can simply specify and reference the variable to set and query the value of the server's item or issue commands to the server.

Arguments:

variable_name
: Name of the variable to be shared with the DDE object

class_name
: Class of object to create; the class of object determines the type of link that will be established. Use `'DDE DATA'` to share data. Use `'DDE COMMAND'` to issue commands. If you only supply a variable name, the class defaults to `'DDE DATA'`.

  If you use `'DDE DATA'`, specifying and referencing the shared variable sets and queries the server's value for the item.

  If you use `'DDE COMMAND'`, specifying the shared variable sends the server a command and referencing it retrieves the return code.

link_info
: Server, topic, and item with which to establish link. Item is not required if `DDE COMMAND` class is specified. If this argument is supplied, the function does not prompt the user.

© Copyright IBM Corporation 1994, 2017

Item names are server and country dependent. Some spreadsheets in the United States use the following syntax:

RxCy
where x and y are a cell's row and column number (index origin 1)
Rx1Cy1:Rx2Cy2
where x1 and y1 are the row and column numbers of the upper left cell in a rectangular group of cells and x2 and y2 are the row and column numbers of the lower right cell in the group of cells.

Consult the documentation for the application to which you want to link to determine the syntax used for item names. Alternatively, use `PASTE_SPECIAL`.

Result:

Handle of the DDE object. If the link cannot be established, zero is returned. The handle can be passed to `DESTROYOBJ` to destroy the object.

Here is an example of using `APL2_DDE_CONNECT`:

```
HDATA←'Excel' '[Book1]Sheet1' 'R1C1' APL2_DDE_CONNECT 'VAR'
□ES(HDATA=0)/'Link to Excel failed'
...
DESTROYOBJ HDATA
```

APL2_XLTABLE_CONNECT

```
HANDLE←[link_info] APL2_XLTABLE_CONNECT variable_name
```

This function displays the active servers and topics available on the system and prompts the user to make a selection and enter an item; it then establishes the link using the XLTABLE data format. The function creates a DDE object, shares a variable with the object, and returns the object's handle. When it completes, you can simply specify and reference the variable to set and query the value of the server's item.

Note: Not all servers support the XLTABLE format. Excel is the primary server which supports it.

Arguments:

variable_name
Name of the variable to be shared with the DDE object

link_info
Server, topic, and item with which to establish link. If this argument is supplied, the function does not prompt the user.

Item names are server and country dependent. Some spreadsheets in the United States use the following syntax:

RxCy

© Copyright IBM Corporation 1994, 2017

where x and y are a cell's row and column number (index origin 1)
Rx1Cy1:Rx2Cy2
where x1 and y1 are the row and column numbers of the upper left cell in a rectangular group of cells and x2 and y2 are the row and column numbers of the lower right cell in the group of cells.

Consult the documentation for the application to which you want to link to determine the syntax used for item names. Alternatively, use PASTE_SPECIAL.

Result:

Handle of the DDE object. If the link cannot be established, zero is returned. The handle can be passed to DESTROYOBJ to destroy the object.

Here is an example of using APL2_XLTABLE_CONNECT:

```
HDATA←'Excel' '[Book1]Sheet1' 'R1C1' APL2_XLTABLE_CONNECT 'VAR'
□ES(HDATA=0)/'Link to Excel failed'
...
DESTROYOBJ HDATA
```

COPY_LINK

```
HANDLES←value COPY_LINK 'server name' 'topic name' 'item name'
'variable_name'
```

This function creates a hierarchy of DDE server, topic, and item objects, shares a variable with the item, and returns the objects' handles. It also turns on the STATE LINK property of the item so that all assignments to the item will be copied to the clipboard. This enables client's Paste Special processing to work. When it completes, you can simply specify and reference the variable to set and query the value of the item.

Arguments:

server name
DDE server application name
topic name
Topic name
item name
Item name
variable_name
Name of the variable to be shared with the DDE item
value
Optional initial value to assign to item.

Note: If a left argument is not supplied, COPY_LINK does not assign an initial value to the shared variable. Client application's Paste Special facilities will not work until you have assigned a value.

Result:

© Copyright IBM Corporation 1994, 2017

Handles of the DDE server, topic, and item objects. If any of the objects cannot be created, zero is returned as their handle. The handles can be passed to DESTROYOBJ to destroy the objects.

Here is an example of using COPY_LINK:

```
(HSERVER HTOPIC HITEM)←COPY_LINK 'App' 'Topic' 'Item' 'VAR'
□ES(HITEM=0)/'Create of item failed'
...
DESTROYOBJ HSERVER
```

Objects and Utility Functions

You have seen that the functions PASTE_SPECIAL, APL2_DDE_CONNECT and APL2_XLTABLE_CONNECT can be used to establish links between APL2 variables and server applications. The rest of this chapter describes the underlying facilities that these functions use to establish links. For interactive work sharing data between APL2 and other applications, you generally do not need to work with these facilities. If you build DDE applications, you should be familiar with this material.

APL2's shared variable support for DDE is based on objects creating using the CREATEOBJ function. DDE objects are special invisible windows which manage the details of the DDE protocol. You can use the function SHAREWINDOW to share variables with DDE objects. These variables are then specified and referenced to set and query the objects' contents and behavior.

For example, to create an object linked with a cell in an Excel spreadsheet and share a variable with the object's data in XLTABLE format, you could do this:

```
HDATA←CREATEOBJ 'DDE DATA' 'Excel' '[Book1]Sheet1' 'R1C1'
'XLTABLE DATA' SHAREWINDOW HDATA 'CELL1'
```

To change or query the contents of the Excel cell, you simply specify or reference the variable CELL1.

Like windows which signal events when users interact with them, DDE objects signal events when DDE events occur. To respond to these events, you specify APL expressions to variables shared with objects' EVENTS property and then use the EXECUTEDLG operator to process the events.

The following sections describe using DDE objects in greater detail:

- Client Applications
- Server Applications

Detailed information on CREATEOBJ can be found in the GUITOOLS workspace and the online help for AP145. Use the function HELP145 to display the online help for AP 145.

Detailed information on using SHAREWINDOW to share variables with window properties and using EXECUTEDLG to respond to events can be found in AP 145 - GUI Services Processor.

Client Applications

© Copyright IBM Corporation 1994, 2017

Objects used in client applications issue requests to read and write items and execute commands and retrieve command return codes when you reference and specify variables. They signal events when new values are available and if the server terminates.

Two different classes of objects are used in DDE client applications:

1. Data Objects
2. Command Objects

Data Objects

Data objects share data with server applications.

Data objects support the following properties:

EVENTS
: The EVENTS property is used to specify APL expressions that should be executed when server events occur. Two events are supported:
'New value'
When the 'New value' event occurs, the server has changed the value of the item; referencing a variable shared with the DATA property will return the new value.
'Server close'
When the 'Server close' event occurs, no further uses of the object should be attempted. Using the handle returned from `CREATEOBJ`, the object should be destroyed and any variables shared with its properties retracted.

DATA
: The DATA property is used to read and write shared data. Referencing a variable shared with the DATA property will always return the server's latest value. Specifying a value will cause a write request to be sent to the server; if the server rejects the write request, a message box will be displayed. The format of the data that can be written is server dependent; most servers accept character vectors and vectors of character vectors. Many servers use tab characters to delimit data. Only one variable can be shared with the DATA or XLTABLE DATA property of a data object.

XLTABLE DATA
: The XLTABLE DATA property is used to read and write data shared with Excel and other servers which support the XLTABLE data format. The XLTABLE DATA property is used like the DATA property except that the format of the data is different.

  The XLTABLE DATA property supports APL2 matrices. Each element of the matrix can be a character scalar, a character vector, a real numeric scalar, or a length 1 integer vector. Length 1 integer vectors are error codes. The following variables in public workspace DDESHARE define the supported error codes:

```
XLERROR_DIVZERO←7
XLERROR_NA     ←42
XLERROR_NAME   ←29
XLERROR_NULL   ←0
XLERROR_NUM    ←36
XLERROR_REF    ←23
XLERROR_VALUE  ←15
```


© Copyright IBM Corporation 1994, 2017

When using the XLTABLE DATA property, Microsoft date and time data is returned to APL2 as floating point numbers. The functions `MS2TS` and `TS2MS` can be used to convert data between APL2's `⎕TS` and Microsoft's date and time formats.

To create a data object, supply server, topic and item names:

```
HDATA←CREATEOBJ 'DDE DATA' 'Server name' 'Topic Name' 'Item name'
'EVENTS' SHAREWINDOW HDATA 'DATA_EVENTS'
'DATA'   SHAREWINDOW HDATA 'DATA'
DATA_EVENTS←2 2⍴'Server close' '→0' 'New value' 'PROCESS_NEW_VALUE'
```

When a data object is destroyed, AP145 retracts any variables shared with the objects' DATA property.

Command Objects

Command objects issue commands to server applications.

Command objects support the following properties:

EVENTS
: The EVENTS property is used to specify APL expressions that should be executed when server events occur. One event is supported:
'Server close'
When the 'Server close' event occurs, no further uses of the object should be attempted. Using the handle returned from `CREATEOBJ`, the object should be destroyed and any variables shared with its properties retracted.

DATA
: The DATA property is used to specify the command to be executed and reference the server's return code. Commands must be character vectors. Return codes are integer scalars from 0 to 255. Only one variable can be shared with the DATA property of a command object.

To create a command object, supply server and topic names:

```
HCOMMAND←CREATEOBJ 'DDE COMMAND' 'Server name' 'Topic Name'
'EVENTS' SHAREWINDOW HCOMMAND 'COMMAND_EVENTS'
'DATA'   SHAREWINDOW HCOMMAND 'COMMAND'
COMMAND_EVENTS←1 2⍴'Server close' '→0'
```

When a command object is destroyed, AP145 retracts any variables shared with the objects' DATA property.

Server Applications

Generally if you only need to share data with other applications, you can use only APL2 client objects. However, if your application needs to be able to execute commands sent by other applications, then you will need to write a server application.

Objects used in server applications automatically manage client links, terminations, and requests to read data. When you modify items within your application, the objects automatically notify your clients that the items have changed.

© Copyright IBM Corporation 1994, 2017

When clients issue requests to write data or have commands executed, events will be signaled that your application should respond to.

Three different classes of objects are used in DDE server applications:

- Server Objects
- Topic Objects
- Item Objects

Server, topic, and item objects must be created in a hierarchy. The handle of a server object must be supplied when creating a topic object, and the handle of a topic object must be supplied when creating an item object.

Server Objects

Server objects manage topic objects.

Server objects support no properties and signal no events.

To create a server object, supply a server name:

```
HSERVER←CREATEOBJ 'DDE SERVER' 'Server name'
```

When a server object is destroyed, all topics within the server are destroyed.

Topic Objects

Topic objects process client requests to execute commands and write data, and manage item objects.

Topic objects support the following properties:

EVENTS
: The EVENTS property is used to specify APL expressions that should be executed when client events occur. Two events are supported:
'Execute value'
When the 'Execute value' event occurs, a variable shared with the DDE EXECUTE property can be referenced to retrieve the command to be executed. If no APL expression is supplied to handle this event, the server will return a *Not Processed* signal.
'Write value'
When the 'Write value' event occurs, a variable shared with the DDE WRITE property can be referenced to retrieve the item name and value to be written. If no APL expression is supplied to handle this event, the server will return a *Not Processed* signal.

DDE EXECUTE
: A variable shared with the DDE EXECUTE property is referenced when an 'Execute value' event occurs to retrieve the command to be executed. Commands are character vectors.

  After the command has been executed, the variable is then specified with a return code. Return codes should be 2 element integer vectors. The first element should be 0 or 1. The second element should be from 0 to 255. If the first element is 1, then the response sent to the client is a 'Not Processed' signal. If

© Copyright IBM Corporation 1994, 2017

the first element is 0, then a successful response is sent to the client and the second element is the server application return code.

Once a variable shared with the DDE EXECUTE property is referenced in response to an 'Execute value' event, the server will respond to all further requests to execute commands with a 'Busy' signal until a return code is specified.

Only one variable can be shared with the DDE EXECUTE property of a topic object.

DDE WRITE

A variable shared with the DDE WRITE property is referenced when an 'Write value' event occurs to retrieve the the item name and value to be written. References return a two element array. The first element is the name of an item a client has requested be written. The second element is the value to be written.

After the write request has been processed, the variable is then specified with a return code. Return codes should be Boolean scalars. If the value is 1, then the response sent to the client is a 'Not Processed' signal. If the value is 0, a successful response is sent to the client.

Once a variable shared with the DDE WRITE property is referenced in response to a 'Write value' event, the server will respond to all further requests to write data with a 'Busy' signal until a return code is specified. If the item being written is modified during processing of the 'Write value' event, all clients will be notified that the item has changed except for the client which sent the write request.

Only one variable can be shared with the DDE WRITE property of a topic object.

Although APL2 high level DDE client facilities do not support requests to write items which do not already exist at the server, the DDE protocol does allow this. Your code to process 'Write value' events should be prepared to handle requests to write items which do not exist.

If no variable is shared with the DDE WRITE property, the topic will automatically accept clients' write requests and replace the value of the appropriate item. A 'New Value' event will be signaled for the item.

To create a topic object, supply a server object handle and a topic name:

```
HTOPIC←CREATEOBJ 'DDE TOPIC' HSERVER 'Topic name'
'EVENTS'      SHAREWINDOW HTOPIC 'TOPIC_EVENTS'
'DDE EXECUTE' SHAREWINDOW HTOPIC 'TOPIC_EXECUTE'
'DDE WRITE'   SHAREWINDOW HTOPIC 'TOPIC_WRITE'
EV_EXEC←'Execute value' 'PROCESS_EXECUTE'
EV_WRITE←'Write value' 'PROCESS_WRITE'
TOPIC_EVENTS←⊃EV_EXEC EV_WRITE
```

When a topic object is destroyed, all clients linked to issue commands with the item are informed that the server is terminating, all items within the topic are destroyed, and AP145 retracts any variables shared with the topic object's DDE WRITE or DDE EXECUTE properties.

Item Objects

Item objects share data with clients.

© Copyright IBM Corporation 1994, 2017

Item objects support the following properties:

DATA
: The DATA property is used to specify and reference the value of the item. Any value can be specified to a variable shared with the DATA property of a item, but non-APL clients will only be able to process character vectors and vectors of character vectors.

STATE LINK
: The STATE LINK property is used to control whether values specified to the item's DATA property are copied to the clipboard. If the link property is set to 0, no data is copied to the clipboard. If the link property is set to 1, each time a variable shared with the data property is specified, the value will be copied to the clipboard and appropriate link information will also be be placed there. This enables client's Paste Special processing to work.

  Note: Setting the STATE LINK property to 1 impacts the server's performance and space requirements.

EVENTS
: The EVENTS property is used to specify APL expressions that should be executed when client events occur. One event is supported:
  'New value'
  The 'New value' event occurs when a client has written a new value for the item.

  Note: The 'New value' event is only signaled if no variable is shared with the DDE WRITE property of the DDE TOPIC object.

To create a item object, supply a topic object handle and an item name :

```
HITEM←CREATEOBJ 'DDE ITEM' HTOPIC 'Item name'
'DATA'       SHAREWINDOW HITEM 'ITEM'
'STATE LINK' SHAREWINDOW HITEM 'ITEM_LINK'
```

When an item object is destroyed, all clients sharing the item are informed that the server is terminating.

Sample Functions

Three sample functions are provided in DDESHARE which demonstrate how to build DDE server and client applications. Their comments and coding style illustrate how to integrate the use of DDE within APL2 applications. All three functions are niladic and return no result. Each function should be run from a separate APL2 session.

APL2_DDE_SERVER

The function is a sample DDE server. It displays shared data in a dialog window. It accepts all requests to write the sample item. It also supports execution of commands as APL expressions. It displays received commands and their result in a dialog window.

`APL2_DDE_SERVER` supports the following link information:

Server APL2 DDE Server

© Copyright IBM Corporation 1994, 2017

Topic Sample topic
Item Sample item

APL2_DDE_COMMAND

The function is a sample DDE client which issues commands to the APL2 DDE sample server. It prompts for commands in a dialog window.

APL2_DDE_CLIENT

The function is a sample DDE client which shares data with the APL2 DDE sample server. It displays the shared data in a dialog window.

DDE Limitations

The Microsoft implementations of DDE use 16 bits to store the size of DDE objects. If you try to pass too large an array to a server, APL2 will display a message box stating that the DDE server application responded to an assignment with a not processed signal. To avoid this problem, you will need to understand how data is passed through DDE connections.

The DDE protocol specifies that clients and servers must negotiate what format will be used to pass data in DDE connections. APL2 automatically performs these negotiations and converts between DDE formats when necessary. APL2 provides support for three formats:

The CF_TEXT format is used to pass data between APL2 and non-APL applications other than Excel. The CF_TEXT format is defined as a vector of single byte characters with carriage return line feed sequences delimiting paragraphs within the vector. The vector is terminated by a hex zero character.

The XLTABLE format is used to pass data matrixes to and from Excel. The following table lists the amounts of storage required for each type of data:

Boolean	2 bytes
Other numbers	8 bytes
Characters	1 byte for vector length and 1 byte per character
Error	2 bytes

If the array is homogeneous, the data is preceded by 8 bytes which provide the size and type of the array.

If the array is non-homogeneous, each element is preceded by 4 bytes which provide the type of the element, and the first type bytes are preceded by 4 bytes which provide the size of the array.

For further information about the XLTABLE format, please consult the Microsoft Developer's Network (MSDN) web site. Search for XLTABLE or the Fast Table Format.

The APL2 *CDR* format is used to pass data between APL2 applications. CDR format is described in Common Data Representation. The first number returned by the `4 ⎕AT` system function can be used to determine the number of bytes required to convert an array to a CDR.

© Copyright IBM Corporation 1994, 2017

DEMO124 Workspace

This workspace is designed to give the user a sample of the capabilities of the AP 124 text display processor.

The demonstration provides an online reference to the various calls to AP 124, and shows some sample applications.

The demonstration can be run by entering:

```
)LOAD 2 DEMO124
DEMO
```


© Copyright IBM Corporation 1994, 2017

DEMO144 Workspace

Note: This workspace is provided only on Unix systems.

The DEMO144 workspace contains some sample programs that make use of the AP 144 X Window System interface auxiliary processor.

Chaos - Example of Drawing Lines and Points

```
Chaos
```

The `Chaos` function is an example of how to draw lines and points by using X Window System calls.

The program redraws the image whenever an Expose event or a MousePress event is detected. To terminate the function, position the mouse pointer in the `Chaos` window and press the "q" key.

The function displays an example of chaotic behavior. The algorithm implemented in the function is very simple:

1. Draw a triangle.
2. Select one of the three corners as a starting point.
3. Select one of the three corners at random, move half the distance from the current location toward this corner, and draw a point. Keep repeating this step ad infinitum.

The expectation is that the surface is randomly covered by the points given a long enough timespan - but instead, a strange figure is created.

In fact, no matter what the starting point is (either inside or outside the triangle), the same figure ultimately results. With the `Chaos` function, you can try this out. You can set the starting point by simply moving the mouse pointer to any location within the window and pressing a mouse button. This is then the starting point.

Towers of Hanoi - A Dynamic X Window System Example

```
Hanoi 'parms'
```

The purpose of `Hanoi` is to show that an APL2 application can drive an interactive graphics program displayed on an X Window System workstation.

This function is a graphical version of the classic example of recursive programming. The underlying goal of the task is to move a series of differently-sized blocks from an originating tower to a target tower, using a temporary tower as needed. Only one block can be moved at a time, and a block must not be placed on top of another block that is smaller than itself. The parameters are specified in a C-like syntax:

© Copyright IBM Corporation 1994, 2017

-a	Solve the problem automatically.
-d host:disp	Workstation display address.
-i	Solve the problem interactively.
-n blocks	Number of blocks in the problem. Up to eight blocks can be specified; five blocks is the default.
-r	Reverse background and foreground.
-s speed	The block-move speed (when auto-solve is used). The speed parameter must be in the range 1-1000 and defaults to 250.

Each of these options must be specified with the leading -, if used. The default settings are equivalent to the call:

```
Hanoi '-a -n 5 -s 250'
```

Initially, the towers are drawn, and the specified number of blocks placed on the leftmost tower. The other two towers are empty.

If the `-a` auto-solve option is being used, the blocks start to move from tower to tower, progressing toward the final solution, which is to place the blocks on the right-most tower in the original stacking order. When this has been achieved, the program pauses briefly, and then reverses to the original state.

The alternative to the auto-solve option is the interactive mode given by the `-i` option. The user has the responsibility of solving the problem. You do so by using the mouse to grasp a block, moving it to another tower, and dropping it on the tower. To grasp a block, move the mouse-pointer to the block and press (and hold) any mouse button; to place it on a new tower, move the mouse-pointer (and the block) to the new tower, and release the button. Only top blocks can be grabbed, and the function disallows cheating (for example, larger blocks cannot be placed on smaller blocks). Any invalid move results in a beep.

In either case, moving the mouse pointer to the `Hanoi` window and pressing the "q" key closes the window and terminates the function. Any other key is ignored.

```
⍝ Autosolve problem at a reasonably fast pace
Hanoi '-a -s 100'
⍝ Manual solve with 3 blocks; reverse colors
Hanoi '-i -n 3 -r'
```

The function was inspired by the xhanoi.c sample X Window System program created by Douglas Earl from The University of Michigan.

HelloWorld - The Standard X Window System Sample Program

```
HelloWorld
```


© Copyright IBM Corporation 1994, 2017

This is the "standard" sample program that comes with all windows-based environments. It shows how much work is required to display a simple message (Hello, World.) on the display and illustrates the basic layout of such a program.

There are no parameters for this function, and it has no explicit result.

`HelloWorld` creates a new window, displays it on the screen, and writes the message "Hello, World." to the window. It then waits for events to occur. It responds to the following events:

MousePress
: Moving the mouse pointer to the `HelloWorld` window and pressing any of the mouse buttons writes the word "Hi!" at the position of the mouse pointer.

KeyPress
: Moving the mouse pointer to the `HelloWorld` window and pressing any alphanumeric key causes the key character to be written in the window at the position of the mouse pointer. Pressing the "q" key closes the window and terminates the function.

Expose
: Redraws the window using the initial layout and message. Any text written to the window as a result of either of the two actions mentioned above is lost.

A full tutorial on the operation of this function is given in The HelloWorld Function.

Xfonts - Another Sample X Window System Program

```
Xfonts
```

This sample program creates a window on the default X Window System screen, and prints a line of text using a variety of fonts.

Expose, Keyboard, and Mouse-button events are enabled. If the window is Exposed (for example, by being resized, moved, or uncovered), it redraws itself. If it receives a Keyboard event, it types the character pressed on a line following the last font displayed.

The sample program can be terminated by pressing any of the mouse buttons.

There are no parameters for this function, and no explicit results of calling the function. Upon termination, it writes a message to the screen:

```
      Xfonts
Bang!!!
```

Xsamp1 - A Sample X Window System Program

```
Xsamp1
```


© Copyright IBM Corporation 1994, 2017

This sample program creates a window on the default X Window System screen, displays a window for 10 seconds, and then erases the window again.

There are no parameters for this function, and no explicit results of calling the function. The function, however, writes a couple of lines to the screen through APL2 □ output.

A window appears on the X Window System display, remains there for approximately ten seconds, and is closed automatically.

```
      Xsamp1
Hold for 10 seconds ...
...Okay, back again
```


© Copyright IBM Corporation 1994, 2017

DEMO145 Workspace

Note: This workspace is not provided on Unix systems.

This workspace contains functions that demonstrate the use of AP 145, the GUI services processor.

The demonstrations illustrate how to use the dialog functions from the GUITOOLS workspace to process dialogs, and how to use the APL2 print object cover functions from GUITOOLS.

The main demonstration can be run by entering:

```
)LOAD 2 DEMO145
DEMO
```

The `PRINT` function prints a character matrix. It displays a print selection dialog and then a print cancel dialog as the matrix is printed.

This workspace also contains a `HELP145` function which displays the online help for AP 145.

© Copyright IBM Corporation 1994, 2017

DEMO207 Workspace

This workspace is designed to give the user a sample of the capabilities of the Universal Graphics auxiliary processor, AP 207.

The demonstration provides an online view of text characters in various sizes and orientations. It displays various geometric shapes that can be used in business graphics along with the possible colors and symbols available. An image is also displayed as part of the demonstration.

The demonstration can be run by entering:

```
)LOAD 2 DEMO207
DEMO
```


© Copyright IBM Corporation 1994, 2017

DEMOJAVA Workspace

This workspace contains demonstration and utility functions for Associated Processor 14, the Calls to Java processor.

The workspace contains the following demonstration functions:

DEMO_JAVA	Demonstrate calling Java from APL2
DEMO_HASH	Demonstrate using the Java Hashtable class
DEMO_SWT	Demonstrate using the Eclipse Standard Widget Toolkit
DEMO_THREAD	Demonstrates running a slave APL2 interpreter in a separate thread

The workspace contains the following utility functions:

APL2OBJECT_TO_ARRAY	Extract the array value of an Apl2object
APL2XML	Convert an APL array to XML
FORMAT_OBJECT	Format an object for printing
JAR_READ	Read a file from a JAR file
JAVA_PRINT	Print data
JAVA_PROPERTY	Set or get the value of a Java system property
JAVAMAIL_SEND	Send an email using the JavaMail API
P14_ASSOC	Associate a name with Processor 14
P14_CALL	Call a Java method
P14_GET	Reference a Java field
P14_MAKE	Instantiate an instance of a class
P14_SET	Specify a Java field
XML2APL	Convert XML to an APL array

For further information, consult the workspace variables DESCRIBE, HOW, and HOW_XML.

© Copyright IBM Corporation 1994, 2017

DISPLAY Workspace

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

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

Z is a character matrix representing the array X.

DISPLAY and DISPLAYG use box characters. DISPLAYG is identical to DISPLAY and is included for compatibility with the DISPLAY workspace distributed with APL2 on mainframe systems. DISPLAYC uses characters that display on all implementations. This is functionally equivalent to the DISPLAY function in the APL2 mainframe DISPLAY workspace.

The following characters are used to convey shape information:

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

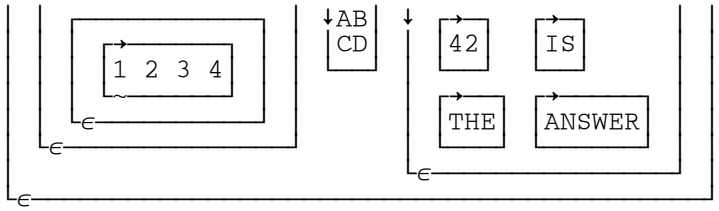
The following characters are used to convey type information:

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

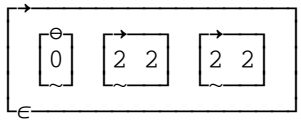
To use each of the DISPLAY functions:

```
      )PCOPY 1 DISPLAY DISPLAY DISPLAYC DISPLAYG
SAVED ...
      (DISPLAY ⍳5) (DISPLAYC ⍳5) (DISPLAYG ⍳5)
┌→─────────┐ .→--------. ┌→─────────┐
|1 2 3 4 5| |1 2 3 4 5| |1 2 3 4 5|
└~─────────┘ '~---------' └~─────────┘
      X←⊂⍳5
       (DISPLAY X) (DISPLAYC X) (DISPLAYG X)
┌──────────┐ .----------. ┌──────────┐
| ┌→─────┐ | | .→--------. | | ┌→─────┐ |
| |1 2 3 4 5| | | |1 2 3 4 5| | | |1 2 3 4 5| |
| └~─────┘ | | | '~---------' | | └~─────┘ |
└∈─────────┘ '∈----------' └∈─────────┘
      X←(⊂⊂⍳4)(2 2⍴'ABCD')(2 2⍴'42' 'IS' 'THE' 'ANSWER')
      DISPLAY X
┌→──────────────────────────────────────────┐
| ┌──────────┐ ┌→┐ ┌→──────────┐           |
```

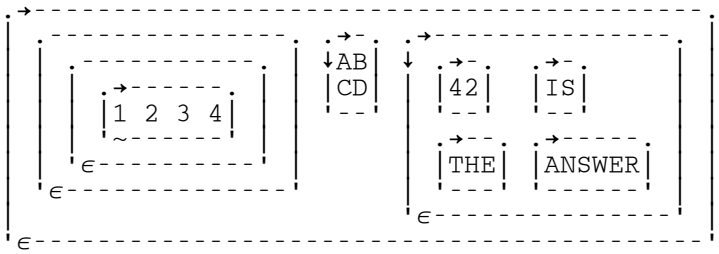

© Copyright IBM Corporation 1994, 2017



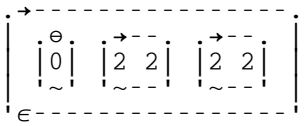

```
      ρX
3
      ρ¨X
   2 2  2 2
      DISPLAY ρ¨X
```




```
      DISPLAYC X
```




```
      ρX
3
      ρ¨X
   2 2  2 2
      DISPLAYC ρ¨X
```



© Copyright IBM Corporation 1994, 2017

EDIT Workspace

This workspace contains two alternative editors.

- EDIT, a simple fullscreen editor.
- EDITOR_2, a fullscreen editor compatible with the mainframe editor `)EDITOR 2`

EDIT

This is a simple, limited-function APL2 full-screen editor for editing defined functions and operators. It uses the text display auxiliary processor, AP 124.

This editor can be used to create new defined functions or operators and to modify existing ones.

To edit an APL2 function or operator with the editor, copy the EDIT workspace into your active workspace with the command:

```
)PCOPY 1 EDIT EDIT
```

If the name of the function you want to create or edit is `FN1`, enter:

```
EDIT 'FN1'
```

After you invoke the editor, a window pops up and the function is displayed, or as much of it as can fit into the available screen area. One line is reserved for definitions of function keys.

You can now move the cursor, using the four arrow keys on the numeric keypad, change any character in the lines displayed, insert characters (with the Ins key), delete characters (with the Del key), delete to the end of a line (with the Ctrl-End key combination), delete to the beginning of a line (with the Ctrl-Home key combination), and move the cursor to the beginning of the next line (by pressing the Tab key). Also, you can use the function keys as indicated in the lowest line of the screen. The full purpose of these, and other special keys, is described below:

F3	QUIT	Cancels function definition. No changes are kept. The function remains as it was at the beginning of the edit session. If you have made changes, you are prompted to confirm the intention to quit.
F4	FILE	Ends function definition. All modifications to the function are kept and the new definition of the function is established in the active workspace. If this process fails, the bottom line of the display is updated with an error message to indicate the line number found to be in error.
F5	TOP	Displays the first or top page of the function.
F6	BOT	Displays the last or bottom portion of the function, with room to add additional lines.
F7	LINE	Clears the screen and displays only the line pointed to by the current cursor position. You can use this to edit lines longer than the screen width. The maximum line length this method allows is 800 characters.

© Copyright IBM Corporation 1994, 2017

F8	INS	Inserts a new line after the current cursor position.
Shift-F8	DEL	Deletes the line pointed to by the cursor.
F9	EXEC	Executes the line pointed to by the cursor. The line is executed under the control of `⎕EA`, so that any error that occurs does not suspend the execution of `EDIT`.
F10	COPY	Copies a line. Move the cursor to the line you want to be copied, and press F10. The F10 definition on the bottom line of the screen is now highlighted. The system is now in "copy" state. Then move the cursor to the line after which the indicated line is to be copied (possibly on another page). Finally, press F10 again to cause the copy to take place. The highlighting of the F10 definition is then removed, and the system is no longer in "copy" state.

All other function keys are ignored.

Other Special Keys:

Tab	Moves the cursor to the beginning of the next line.
Shift-Tab	Moves the cursor to the beginning of the preceding line.
PageDown	Displays the next page.
PageUp	Displays the preceding page.
End	Moves the cursor to the end of the current line (unless that line is wider than the screen).
Home	Moves the cursor to the start of the current line.
Enter	Moves the cursor down one line at a time. If the cursor is in the last displayed line when this key is pressed, the whole function is scrolled up one line.
Esc	Provides a limited "UNDO" facility. All changes made since the last press of the Enter key or a function key are removed, and the changed lines revert to their original state. Esc does *not* interrupt execution of `EDIT`.

Locked functions or operators cannot be edited with this function.

If you use `EDIT` to edit an APL2 function that is already suspended in the active workspace, you create a new version of the function. The suspended version remains unaltered, but disappears once execution is complete, or the state indicator is cleared.

You can also use this editor to copy a function to a new name, leaving the old version intact. Just invoke `EDIT` for the original function, change the name in the header line, and press FILE (F4).

EDITOR_2

`EDITOR_2` is functionally equivalent to Editor 2 (available through `)EDITOR 2`) on APL2 mainframe systems. Editor 2 is fully described in *APL2 Programming: Language Reference*.

To use this function, copy it into the active workspace, and then invoke it with a right argument of the name of the object to be edited:

```
)PCOPY 1 EDIT EDITOR_2
EDITOR_2 'object_name'
```


© Copyright IBM Corporation 1994, 2017

This is equivalent to entering the mainframe APL2 commands:

```
)EDITOR 2
∇object_name
```


© Copyright IBM Corporation 1994, 2017

EXAMPLES Workspace

This section describes the EXAMPLES workspace.

- Introduction
- Mathematical Calculations
- Miscellaneous Utility Functions
- The Group GPAPL2

Introduction

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

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

Mathematical Calculations

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

ASSOC - Associativity

```
Z←ASSOC M
```

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

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


© Copyright IBM Corporation 1994, 2017

```
1 2 3 4 4
1 2 3 4 5
      ASSOC MULTTABLE
1
      MULTTABLE[3;3]←1
      MULTTABLE
1 1 1 1 1
1 2 2 2 2
1 2 1 3 3
1 2 3 4 4
1 2 3 4 5
      ASSOC MULTTABLE
0
```

BIN - Binomial coefficients

```
Z←BIN N
```

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

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

COMB/FC/LFC - Combinations

```
Z←COMB N
Z←FC N
Z←LFC N
```

The function `COMB` uses recursive definition to produce a `2!N` by 2 matrix of all possible pairs of elements from `⍳N`.

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

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

© Copyright IBM Corporation 1994, 2017

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

The function LFC uses FC to generate letter pairs.

```
      LFC 5
AB
AC
AD
AE
BC
BD
BE
CD
CE
DE
```

GCD - Greatest common divisor

```
Z←L GCD R
```

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

```
      30 GCD 40
10
      GCD/ 30 40
10
      GCD/ 30 40 45
5
      GCD/ 30 40 39 45
1
      GCD/ 30 42 39 45
3
```

HILB - Hilbert matrix

```
Z←HILB N
```

The function HILB produces a Hilbert matrix of order N.

PALL/PER/PERM - Permutations

© Copyright IBM Corporation 1994, 2017

```
Z←PALL N
Z←PER N
Z←PERM N
```

The function PALL produces the matrix of all permutations of order N. Its subfunction PERM produces the B-th permutation of order N.

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

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

PO/POL/POLY/POLYB - Polynomials

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

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

```
      ¯1 0 1 PO ¯2 ¯1 0 1 2
3 0 ¯1 0 3
      ¯1 0 1 POL 2
3
      ¯1 0 1 POLY 1
0
      ¯1 0 1 POLYB ¯1
0
```

To find the zeros of polynomials, see the POLYZ function from the MATHFNS workspace, described in Roots of Polynomials.

© Copyright IBM Corporation 1994, 2017

TRUTH - Truth tables

```
Z←TRUTH N
```

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

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

ZERO - Roots of a function

```
Z←TOL (F ZERO) R
```

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

```
      ⎕FX 'Z←SIN X' 'Z←1○X'
SIN
      .1 SIN ZERO ¯1 1
0
      .1 SIN ZERO 1 4
3.0625
      .001 SIN ZERO 1 4
3.141601563
```

Miscellaneous Utility Functions

- PACK/UNPACK - Illustration of Base and Representation
- DEC2HEX/HEX2DEC/HEX - Hexadecimal arithmetic and conversions
- SORTLIST - Sort with collating sequence
- TIME - Provide CPU time used

PACK/UNPACK - Illustration of Base and Representation

```
Z←PACK X
Z←UNPACK X
```


© Copyright IBM Corporation 1994, 2017

The functions PACK and UNPACK illustrate the use of the ⊤ and ⊥ functions in transforming between a four-number encoding of SERIAL NUMBER (1 to 9999), MONTH, DAY, and YEAR, and a single number encoding of the same data.

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

DEC2HEX/HEX2DEC/HEX - Hexadecimal arithmetic and conversions

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

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

DEC2HEX Converts decimal to hexadecimal
HEX2DEC Converts hexadecimal to decimal
+ HEX Performs hexadecimal addition
- HEX Performs hexadecimal subtraction
... and so on

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

SORTLIST - Sort with collating sequence

```
Z←SORTLIST R
```

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

TIME - Provide CPU time used

```
Z←TIME
```


© Copyright IBM Corporation 1994, 2017

The function TIME yields the amount (in minutes, seconds, and milliseconds) of CPU time used since its previous execution. It is useful in measuring the execution times of other functions. The global variable TIMER is assigned the value of the cumulative CPU time at each execution of the function TIME.

The Group GPAPL2

The group GPAPL2 consists of various functions and operators designed to show some of the capabilities of APL2.

- Workspace Information Functions
- Miscellaneous Functions
- Operators to Conform Arguments
- Operators for Debugging
- Operators to Handle Depth
- Operators for Program Control
- Miscellaneous Operators

Workspace Information Functions

```
EXAMPLE R
```

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

```
EXAMPLES
```

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

Miscellaneous Functions

```
Z←L IOTAU R
```

This is the Find Index function from an early experimental version of APL2. R and L may be any arrays. Z is an integer matrix containing the starting positions (in row major order) where pattern R begins in the array L.

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


© Copyright IBM Corporation 1994, 2017

```
BAABC
ABAAB
      L IOTAU 'BA'
3 1
4 2
      L IOTAU 2 1⍴'BA'
1 2
1 5
2 3
3 1
3 4
```

```
Z←L REPLICATE R
Z←L EXPAND R
```

These functions are identical to their primitive counterparts, Replicate and Expand, respectively represented by "/" and "\", except that the primitive versions are operators, so you cannot apply operators to them. The defined REPLICATE and EXPAND really *are* functions, so you can apply operators to them.

```
      (1 0 1)(0 3) REPLICATE⍤1 'ABC' 'DE'
 AC EEE
      REPLICATE/ 5 '*'
 *****
      (1 0 1)(0 1 0) EXPAND⍤1 (2 4) 6
 2 0 4  0 6 0
```

```
Z←L ENLISTA R
```

Returns (in Z) the array L with each item of the enlist of L (∈L) replaced by the corresponding item from R. By using this, you can replace the selective specification (∈L)←R with L←L ENLISTA R.

```
Z←EXPUNGE NL
```

Expunges the objects listed in NL. NL may be a character matrix with one name per row, or a character vector with names separated by blanks. If a name is enclosed in parentheses, it is assumed to be the name of a character array (rank not greater than 2) containing a list of objects to be expunged. Unlike the)ERASE system command, this expunges the local copy of an object that is localized in a pendent or suspended function or operator.

```
Z←REP R
```

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

```
Z←TYPE R
```


© Copyright IBM Corporation 1994, 2017

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

```
Z←UNIQUE R
```

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

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

Operators to Conform Arguments

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

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

```
      (4 4⍴'WE  THEYUS  OURS') ∧.(=PAD) ⍉ 2 3⍴'WE OUR'
1 0
0 0
0 0
0 0
      (4 4⍴'WE  THEYUS  OURS') ∧.(=TRUNC) ⍉ 2 3⍴'WE OUR'
1 0
0 0
0 0
0 1
      (2 3 4⍴⍳24) +PAD CR 5 6⍴100×⍳30
 101  202  303  404  500  600
 705  806  907 1008 1100 1200
1309 1410 1511 1612 1700 1800
1900 2000 2100 2200 2300 2400
2500 2600 2700 2800 2900 3000
  13   14   15   16    0    0
  17   18   19   20    0    0
  21   22   23   24    0    0
   0    0    0    0    0    0
   0    0    0    0    0    0
```

Operators for Debugging

```
Z←L (F TRACE) R
Z←(F TRACE) R
```


© Copyright IBM Corporation 1994, 2017

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

```
      +TRACE\ 1 4 9       ⍝ Expression as entered
1 4                       ⍝ TRACE output
5                         ⍝    &colon.
4 9                       ⍝    &colon.
13                        ⍝    &colon.
1 13                      ⍝    &colon.
14                        ⍝ TRACE output
1 5 14                    ⍝ Final result
      2 +TRACE\ 1 2 3 4   ⍝ Expression as entered
1 2                       ⍝ TRACE output
3                         ⍝    &colon.
2 3                       ⍝    &colon.
5                         ⍝    &colon.
3 4                       ⍝    &colon.
7                         ⍝ TRACE output
3 5 7                     ⍝ Final result
```

```
Z←L (F TRAP) R
Z←(F TRAP) R
```

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

```
      2 ÷TRAP 0
 DOMAIN ERROR
       L F R
        ^
      ⍴⊃2 ÷TRAP 0
3 12
```

Operators to Handle Depth

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

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

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


© Copyright IBM Corporation 1994, 2017

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

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

```
      1 (2 3)⍴PL ⍳6
 1   1 2 3
     4 5 6
      3 ⍴PR 1,⊂2,⊂3 4
 1 1 1   2 2 2   3 4 3
      (⍴PR 'A' 'BC' ('DEF' 'HIJK')) ⍴ PL '□'
 □ □□   □□□ □□□□
```

Operators for Program Control

```
Z←C (F ELSE G) R
```

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

```
Z←(F IF C) R
```

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

Miscellaneous Operators

```
Z←L (F AND G) R
Z←(F AND G) R
```

Applies two functions to the same argument(s).

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

```
Z←L (F COMMUTE) R
```


© Copyright IBM Corporation 1994, 2017

Switches the arguments of the function to which it is applied.

```
      0.5 ⋆COMMUTE 9
3
```

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

Applies outer product to all levels of the arrays `L` and `R`.

```
      (10 20)(30 40 50) ∘.FAROUT (1 2)(3 4 5)
 11 12   13 14 15
 21 22   23 24 25
 31 32   33 34 35
 41 42   43 44 45
 51 52   53 54 55
```

```
Z←L (F NOP) R
Z←(F NOP) R
```

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

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

```
Z←(F POWER N) R
```

Applies `F` monadically `N` times.

```
      (⍴ POWER 2) 2 3 4⍴⍳24
3
```


© Copyright IBM Corporation 1994, 2017

FILE Workspace

The FILE workspace has been designed to help you work with operating system files.

There are four groups of functions in the FILE workspace:

- AP 210 Group - functions for accessing files with AP 210
- AP 211 Group - `REBUILD211` and `TRYLOAD`
- Delta Group - `ΔFM` and `ΔFV`
- Transfer Group - `IN`, `PIN`, and `OUT`

AP 210 Group

The functions in this group aid in manipulating operating system files. They allow either sequential or random access, using fixed or variable-length records. They use the file auxiliary processor, AP 210.

This group of functions enables you to *create* a file, *write* into it, and *read* from it. To write into a file, you `WOPEN` an old or new file, and `WRITE` data into it. You then `CLOSE` the file to save it on disk. If you only want to read data from an old file, without writing any more data into it, you `OPEN` the file and `READ` records, either randomly or sequentially.

To use the functions described here, you can copy them from the FILE workspace into your active workspace by entering:

```
)PCOPY 2 FILE (GPAP210)
```

- Terminology
- OPEN - Open a file for read/only
- WOPEN - Open a file for read/write
- CLOSE - Close a file
- EBCDIC - Set up an EBCDIC translation table
- SIZE - Return file size in bytes
- READ - Read a fixed-length record
- READD - Read designated bytes
- READV - Read a variable-length record
- READVS - Read a record, stripping EOL
- WRITE - Write a fixed-length record
- WRITED - Write designated bytes
- WRITEV - Write a variable-length record
- APPENDFILEV - Append to a variable-length record file
- COMPARE - Compare two files
- DELETE - Delete a file
- RENAME - Rename a file
- READFILEV - Read a file of variable-length records
- TYPE - Display the contents of a file
- WRITEFILEV - Write a file of variable-length records
- Auxiliary Functions and Variables

© Copyright IBM Corporation 1994, 2017

- Example of Use

Terminology

The following common terms and definitions are used in the descriptions of the syntax of the functions in this group:

sequential access
: to a file occurs when the first record of the file (record 0) is accessed by the first read or write command immediately after the `OPEN` or `WOPEN`; the second record (record 1) is accessed on the next command, and so on.
The `READ`, `READD`, `READV`, `READVS`, `WRITE`, `WRITED`, and `WRITEV` functions work from the same access point, meaning that the access point is advanced sequentially to the next record each time any of these commands is issued.

Brackets `[ ]`
: indicate that a parameter is optional.

`file_no`
: is a positive integer that you define for future reference to a file when you open it.

`filespec`
: must be in the following syntax: `["][path]filename["]`
The surrounding quotes are required only if the file identification contains the comma character, to distinguish commas in the file identification from commas in the AP 210 command string.

`code`
: can be any of the following characters:

A	(APL) The records in the file are APL2 objects and their headers in APL2 internal form. Arrays of arbitrary rank and depth can be stored and recovered. Different records of a file can contain objects of different types (for example, characters, integers, or real numbers). The number of bytes occupied by the object includes APL2 header information as well as the actual data.
B	(Bool) The records in the file contain strings of bits with no header (packed eight bits per byte). The equivalent APL2 object is a Boolean vector. In this case, all records must be equal to the selected record length.
C	(Char) The contents of the record is a string of characters in ASCII, with no header. All records must be equal to the selected record length, with each character occupying one byte. Variable-length records are not supported.
D	(ASCII) The contents of the record is a string of characters in ASCII code, with no header. Each character occupies one byte. Variable-length records are supported.
T	(Translate) The contents of the record is a string of characters, with no header, translated according to the AP 210 translate table defined as described in Establishing the AP 210 Translate Table. Each character occupies one byte. Variable-length records are supported.

Note: Errors encountered during the execution of these functions can cause a message containing an AP 210 return code to be displayed. The meanings of these return codes are listed in AP 210 Return Codes.

OPEN - Open a file for read/only

```
[file_no] OPEN 'filespec[,code]'
```


© Copyright IBM Corporation 1994, 2017

This function opens the specified file for read-only access.

If no file by the name indicated in `filespec` exists, an error results; see AP 210 Return Codes for a list of all possible return codes.
If `code` is omitted, A (APL) is assumed.
If `file_no` is omitted, 1 is assumed.

`OPEN` creates three global variables, with the following names:

`'CZ',⍕:file_no` Control shared variable
`'DZ',⍕:file_no` Data shared variable
`'EZ',⍕:file_no` File size in bytes

Issuing `OPEN` on a `file_no` without having closed a file previously opened using that same number causes the open file to be closed, and a new open to be issued according to the new request.

WOPEN - Open a file for read/write

```
[file_no] WOPEN 'filespec[,code]'
```

This function opens the specified file for read/write access.

If no file by the name indicated in `filespec` exists, a new file is created.
If `code` is omitted, A (APL) is assumed.
If `file_no` is omitted, 1 is assumed.

`WOPEN` creates three global variables, with the following names:

`'CZ',⍕file_no` Control shared variable
`'DZ',⍕file_no` Data shared variable
`'EZ',⍕file_no` File size in bytes

Issuing `WOPEN` on a `file_no` without having closed a file previously opened using that same number causes the open file to be closed, and a new open to be issued according to the new request.

CLOSE - Close a file

```
CLOSE file_no
```

This function closes a file that has been opened with `OPEN` or with `WOPEN`. The previously assigned `file_no` is now available for reuse.

© Copyright IBM Corporation 1994, 2017

If you OPEN or WOPEN a file_no without having closed a file previously opened using that same number, the open file is closed, and a new open issued according to the new request.

EBCDIC - Set up an EBCDIC translation table

```
EBCDIC file_no
```

The EBCDIC function defines an APL2-EBCDIC translation table to AP 210. This can be used after opening an AP 210 file with code T.

file_no must match the number you specified on OPEN or WOPEN.

SIZE - Return file size in bytes

```
Z←SIZE file_no
```

This function returns the size of a file when it was opened.

SIZE can only be used after the file has been successfully been opened with OPEN or WOPEN.

file_no must match the number you specified on OPEN or WOPEN.

READ - Read a fixed-length record

```
Z←READ file_no [record_no [record_size]]
```

This function reads fixed-length records from a data file that was opened using OPEN or WOPEN.

file_no must match the number that you specified on OPEN or WOPEN.

Random access is designated by specifying a particular record number in record_no.

If record_no is not specified, the default is sequential access to the file.

record_size indicates the number of bytes in each record. If record_size is not specified, the default is the record_size specified on the previous read or write operation. If record_size is not specified on the first read or write operation, the default is 128 bytes.

READD - Read designated bytes

```
Z←READD file_no [byte_no [record_size]]
```


© Copyright IBM Corporation 1994, 2017

This function reads a fixed number of bytes from a data file that was opened using `OPEN` or `WOPEN`.

`file_no` must match the number that you specified on `OPEN` or `WOPEN`.

Random access is designated by specifying a particular `byte_no` position in the file (`0≤byte_no`).

If `byte_no` is not specified, the default is sequential access to the file.

`record_size` indicates the number of bytes to be read. If `record_size` is not specified, the default is the `record_size` specified on the previous read or write operation. If `record_size` is not specified on the first read or write operation, the default is 128 bytes.

READV - Read a variable-length record

```
Z←READV file_no [record_no [scan_length]]
```

This function reads a variable-length record from a data file that was opened using `OPEN` or `WOPEN`. This function can be used only if the file was opened with codes A, D, or T.

`file_no` must match the number that you specified on `OPEN` or `WOPEN`.

Random access is designated by specifying a particular record number in `record_no`.

If `record_no` is not specified, the default is sequential access to the file.

For files opened with code D or T, `scan_length` defines the maximum distance for which AP 210 will scan the file for the end-of-record delimiter (linefeed). It should be set to the maximum record length that you expect to read. If `scan_length` is not specified, the default is the `scan_length` specified on the previous read or write operation. If `scan_length` is not specified on the first read or write operation, the default is 128 bytes.

For files opened with code A, `scan_length` is ignored.

If the file was opened with codes D or T, the record returned by `READV` includes the end-of-record delimiter(s). On PC systems, this is normally two characters - carriage return and linefeed. On Unix systems, this is normally a single linefeed.

READVS - Read a record, stripping EOL

```
Z←READVS file_no [record_no [scan_length]]
```

This function is the same as READV, except that the end-of-record delimiters (carriage return and/or linefeed) are not returned with the data.

WRITE - Write a fixed-length record

© Copyright IBM Corporation 1994, 2017

```
[file_no [record_no [record_size]]] WRITE data
```

This function writes a fixed-length record to a data file that was opened using WOPEN.

When the WRITE function is issued, it writes over any existing data at the specified record. If the size of data is greater than the record size, an error is reported. If the size of data is less than the record size, it is padded with X'00'.

file_no must match the number that you specified on WOPEN. If no number is given, 1 is assumed.

Random access is designated by specifying a particular record. in record_no.

If record_no is not specified, the default is sequential access to the file.

record_size indicates the number of bytes in each record. If record_size is not specified, the default is the record_size specified on the previous read or write operation. If record_size is not specified on the first read or write operation, the default is 128 bytes.

WRITED - Write designated bytes

```
[file_no [byte_no [record_size]]] WRITED data
```

This function writes a fixed number of bytes to a data file that was opened using WOPEN.

When the WRITED function is issued, it writes over any existing data at the specified record. If the size of data is greater than the record size, an error is reported. If the size of data is less than the record size, it is padded with X'00.

file_no must match the number that you specified on WOPEN. If no number is given, 1 is assumed.

Random access is designated by specifying a particular byte_no position in the file (0≤byte_no).

If byte_no is not specified, the default is sequential access to the file.

record_size indicates the number of bytes to be written. If record_size is not specified, the default is the record_size specified on the previous read or write operation. If record_size is not specified on the first read or write operation, the default is 128 bytes.

WRITEV - Write a variable-length record

```
[file_no [record_no [scan_length]]] WRITEV data
```


© Copyright IBM Corporation 1994, 2017

This function writes a variable-length record to a data file that was opened using WOPEN. This function can be used only if the file was opened with codes A, D, or T.

file_no must match the number that you specified on WOPEN. If no number is given, 1 is assumed.

Random access is designated by specifying a particular record number in record_no. If record_no is greater than the number of records currently in the file, the record is appended to the end of the file.

If record_no is not specified, the default is sequential access to the file.

Note: In general, random access when writing variable-length records is not recommended. If a record is replaced with a new record that does not contain exactly the same number of bytes as the one it is replacing, partial records will be created when the new data either does not cover all the old data, or when it covers part of the next record in the file. All data after the replaced record is likely to be corrupted.

For files opened with code D or T, scan_length defines the maximum distance for which AP 210 will scan for the end of record delimiter (linefeed) when searching for a record during random access. It should be set to the maximum record length that you expect to use. If scan_length is not specified, the default is the scan_length specified on the previous read or write operation. If scan_length is not specified on the first read or write operation, the default is 128 bytes.

For files opened with code A, scan_length is ignored.

If the file was opened with code D or code T, carriage return and linefeed characters are appended to the end of each record as expected by the operating system for sequential files. If the record already has a linefeed character (⎕TC[⎕IO+2]) at the end, however, no additional end-of-record delimiters are added.

APPENDFILEV - Append to a variable-length record file

```
'filespec' APPENDFILEV vector_of_charvectors
```

This function accepts a vector of character vectors as its right argument and a file name as its left argument; it then appends each vector in vector_of_charvectors as code D (ASCII) variable-length records to the file filespec. To write a single character vector as one record, enclose (⊂) the right argument.

COMPARE - Compare two files

```
[record_size] COMPARE filespec_matrix
```

This function compares two files. The right argument is a two-row character matrix, each row containing the filespec of one of the files to be compared, followed by a comma, followed by the code with which the file is to be read. The optional left argument, record_size, specifies the record length with which the files are to be read. If this left argument is not specified, the files are assumed to contain variable-length records.

© Copyright IBM Corporation 1994, 2017

If the files are identical, the COMPARE function gives no output. Otherwise, it lists the pairs of corresponding records that differ. It also indicates which of the files is shorter, if applicable.

Example:

```
      80 COMPARE ⊃'FILE1,D' 'FILE2,D'
```

This example compares files, FILE1 and FILE2, both of which are read as 80-byte fixed-length record ASCII files.

DELETE - Delete a file

```
DELETE 'filespec'
```

This function deletes a file.

RENAME - Rename a file

```
'new_filespec' RENAME 'old_filespec'
```

This function changes the name of the file specified in the right argument to the name and extension specified in the left argument. If a different subdirectory is specified, a move is performed instead of a rename.

READFILEV - Read a file of variable-length records

```
Z←READFILEV 'filespec[,code]'
```

This function reads a code A (APL), code D (ASCII), or code T (Translate) variable-length record file and returns each record as an element of a vector of arrays.

If code is not specified, the file is assumed to be an ASCII (code D) file. End-of-record delimiters are removed.

Note: The ΔFV function, described in Delta Group, provides a similar facility and is independent of other functions in this workspace.

TYPE - Display the contents of a file

```
[record_size [n]] TYPE 'filespec[,code]'
```


© Copyright IBM Corporation 1994, 2017

This function displays the contents of a file.

The file identified by `filespec` is displayed at the terminal.

If the left argument is given, `record_size` specifies the record length to be used, and `n` specifies the number of characters of each record to be displayed. If `n` is not specified, the full `record_size` is displayed. If `code` is not specified, code A (APL) is used.

If the left argument is not given, the file is assumed to contain ASCII variable-length records (code D). Any `code` specified is ignored.

WRITEFILEV - Write a file of variable-length records

```
'filespec[,code]' WRITEFILEV vector_of_arrays
```

This function accepts a vector of arrays as its right argument and a file name as its left argument; it then writes each vector in `vector_of_arrays` as a variable-length record in the file `filespec`.

If specified, `code` must be A (APL), code D (ASCII), or code T (Translate). If `code` is not specified, the file is written as an ASCII (code D) file.

For code D or code T, `vector_of_arrays` must be a vector of character vectors. To write a single array as one record, enclose (⊂) the right argument.

Note: The `ΔFV` function, described in Delta Group, provides a similar facility and is independent of other functions in this workspace.

Auxiliary Functions and Variables

```
ok_list ΔCHK return_code
```

Translates return codes. `ok_list` is optional and, if included, contains a specific list of return codes (in addition to 0) which should be treated as non-error conditions. `return_code` is the code to be checked. If an error condition is encountered a message is printed and the application is terminated.

```
ΔSH file_no
```

Shares a pair of variables with AP 210. Uses `file_no` as a variable suffix.

```
EOF
```


© Copyright IBM Corporation 1994, 2017

The EOF variable contains the data value to be returned by the READ, READD, READV, and READVS functions when reading has gone beyond end of file. As shipped, this variable contains the null value (⍳0). The user may modify this value as desired for their application.

ole

The ole variable contains error message text. It is used by the function ΔCHK.

Example of Use

The following is an example of how to use some of the file-handling functions contained in the FILE workspace:

```
)LOAD 2 FILE
```

First create a new file. The records contain strings of ASCII character data. The file number is set to 1:

```
1 WOPEN 'NAMEFILE,D'
```

The first record (record 0) is a variable-length record written to file number 1. Line feed characters are added automatically as record separators:

```
1 0 WRITEV 'Jane Doe, Megabyte System, 555-5555'
```

Write a second record to file 1:

```
1 1 WRITEV 'Jim Diaz, Success Inc., 555-9999'
```

Close the file:

```
CLOSE 1
```

Open the same file for read-only operations, with the same file number:

```
OPEN 'NAMEFILE,D'
```

Read the second record first with the linefeed character stripped off:

```
      READVS 1 1
Jim Diaz, Success Inc., 555-9999
```

Now request the first record, again removing the linefeed separator:

```
      READVS 1 0
```


© Copyright IBM Corporation 1994, 2017

```
Jane Doe, Megabyte Systems, 555-5555
```

Close the file:

```
      CLOSE 1
```

Delete the file:

```
      DELETE 'NAMEFILE'
```

AP 211 Group

This group contains the following two functions:

`REBUILD211` Rebuild an AP 211 file
`TRYLOAD` Load a workspace saved under TryAPL2

```
[record_size] REBUILD211 filespec
```

Rebuild the AP 211 file indicated by `filespec`. The optional left argument, `record_size`, specifies the block size to be used in the file. If no left argument is supplied, the existing record size of `filespec` is used.

The resulting file has data stored in contiguous records and uses the minimum space on the disk.

```
TRYLOAD 'filespec'
```

This function loads a workspace saved under TryAPL2. `filespec` is the name of the file to be loaded. The extension `.TRY` is added to the name.

Delta Group

This group contains the following two functions:

`∆FM` Read or write a file with fixed-length records
`∆FV` Read or write a file with variable-length records

These two functions emulate the `∆FM` and `∆FV` external functions of Processor 10. They can be used to read and write files that can be interchanged with other non-APL2 applications.

Note: These functions can handle either the CR/LF or LF end-of-record indicators. They automatically write the correct indicator for the running system, and can read files with either type of indicator.

© Copyright IBM Corporation 1994, 2017

```
Z←[array] ΔFM 'filespec'
```

`R←ΔFM 'filespec'`
: Reads the file `filespec` and returns a character matrix. Records are padded on the right with blanks to the length of the longest record.

  If the operation is not successful, `R` is a numeric return code from AP 210.

`R←array ΔFM 'filespec'`
: Writes the data from the matrix or vector of vectors `array` to the file `filespec`. If `array` is a character matrix, all records are padded with blanks to the full width of the array.

  `R` is `0` if the operation is successful, or a numeric return code from AP 210 if not successful.

  If `array` is empty, the file specified by the right argument is erased, if it exists, and a return code of `0` is returned. If the file does not exist, a return code of `2` is given.

```
Z←[array] ΔFV 'filespec'
```

`R←ΔFV 'filespec'`
: Reads the file `filespec` and returns a vector of character vectors. Trailing blanks in any records are deleted.

  If the operation is not successful, `R` is a numeric return code from AP 210.

`R←array ΔFV 'filespec'`
: Writes the data from the matrix or vector of vectors `array` to the file `filespec`. A file with variable-length records is created. Trailing blanks in each record are deleted.

  `R` is `0` if the operation is successful, or a numeric return code from AP 210 if not successful.

  If `array` is empty, the file specified by the right argument is erased, if it exists, and a return code of `0` is returned. If the file does not exist, a return code of `2` is given.

Transfer Group

This group contains the following three functions:

- IN - Simulated)IN
- OUT - Simulated)OUT
- PIN - Simulated)PIN

IN - Simulated)IN

```
Z←[namelist] IN 'filespec'
```


© Copyright IBM Corporation 1994, 2017

```
Z← IN 'filespec names'
```

This function imitates the)IN command under the control of AP 210. It can be called from another APL2 function, thus effectively providing a powerful IN facility.

You can call this function in two different ways:

1. To copy a whole file into your active workspace, call the IN function in the following way:

   ```
   IN 'filespec'
   ```

   where filespec is the name of the file you want to copy. This can include an extension. If no extension is specified, a default extension of .atf is initially used.

   The result is 1 if the file exists or 0 if it does not.

   Example:

   ```
   IN 'MYFILE'
   ```

   This line copies the whole APL2 transfer file, MYFILE.atf, into your active workspace.

2. To copy only part of a file (for example, some particular operators, functions, or variables) into your active workspace, call the IN function in either of the following ways:

   ```
   namelist IN 'filespec'
   ```

   or,

   ```
   IN 'filespec names'
   ```

   In namelist, you give the names of the operators, functions, and variables (APL2 objects) you want to copy. If there is more than one object, each name must be given as a row of a character matrix, as an element of a vector of character vectors, or in a character vector with each name separated by one or more blanks. Alternatively, the names can follow the filespec, separated from one another by one or more blanks. If a name is enclosed in parentheses, it is assumed to be the name of a character array (rank not greater than 2) containing a list of objects to be copied. Only the indicated objects are copied into the active workspace. The function returns a logical vector result - a 1 for each object copied and 0 for each object not copied.

© Copyright IBM Corporation 1994, 2017

Example:

```
(2 3⍴'FUNVAR') IN 'MYFILE'
```

The left argument of the IN function in this example is a 2-by-3 character matrix, in which the first row is FUN and the second is VAR. This line copies into your active workspace the objects (which can be operators, functions, or variables), FUN and VAR, from MYFILE.atf.

OUT - Simulated)OUT

```
Z←[namelist] OUT 'filespec'
Z← OUT 'filespec names'
```

This function emulates the)OUT command under the control of AP 210, and can be called from another APL2 function, thus effectively providing a powerful OUT facility. You can call this function in two different ways:

1. To copy your entire active workspace (all operators, functions, and variables) into an .atf file (that is, a transfer file), call the OUT function in the following way:

   ```
   OUT 'filespec'
   ```

   where filespec is the name of the transfer file. This can include an extension. If no extension is specified, a default extension of .atf will be used.

   The result is 1 if the operation is successful, or 0 otherwise.

   Example:

   ```
   OUT 'MYFILE'
   ```

   This line copies all operators, functions, and variables of your active workspace into the file MYFILE.atf.

2. To copy only part of your workspace (some particular operators, functions or variables) into a file, call the OUT function in either of the following ways:

   ```
   namelist OUT 'filespec'
   ```

   or,

© Copyright IBM Corporation 1994, 2017

```
OUT 'filespec names'
```

In `namelist`, you give the names of the operators, functions, and variables (APL2 objects) you want to copy. If there is more than one object, each name must be given as a row of a character matrix, as an element of a vector of character vectors, or in a character vector with each name separated by one or more blanks. Alternatively, the names can follow the filespec, separated from one another by one or more blanks. If a name is enclosed in parentheses, it is assumed to be the name of a character array (rank not greater than two) containing a list of objects to be copied. Only the indicated objects are included in the file. The function returns a logical vector result - 1 for each object copied and 0 for each object not copied.

Example:

```
(2 3⍴'FUNVAR') OUT 'MYFILE'
```

The left argument of the `OUT` function in the preceding example is a 2-by-3 character matrix, in which the first row is `FUN` and the second is `VAR`. This line creates a transfer file called MYFILE.atf and writes into it the objects `FUN` and `VAR` in the transfer form.

PIN - Simulated)PIN

```
Z←[namelist] PIN 'filespec'
Z← PIN 'filespec names'
```

This function imitates the `)PIN` command under the control of AP 210. It performs a *protected* `IN`, which works like `IN` except that an object is copied only if the object name in the active workspace has no current value.

© Copyright IBM Corporation 1994, 2017

GRAPHPAK Workspace

The GRAPHPAK workspace distributed with the workstation APL2 products uses the AP 207 Universal Graphics auxiliary processor to display results.

GRAPHPAK is a powerful general-purpose graphical library. For a full description of GRAPHPAK, refer to *APL2 GRAPHPAK: User's Guide and Reference*.

Users should be aware of the following differences between the workstation AP 207 version of GRAPHPAK, and GRAPHPAK as implemented for APL2 on mainframes:

1. Area fill may not fill some areas (such as a torus) in exactly the same way as the mainframe or APL2/PC versions of GRAPHPAK. In general, when multiple polygons are used, the fill is compatible, but when images include arcs or sectors, there may be discernible differences.
2. The workstation GRAPHPAK generates "stroked" characters as its default.
3. The `SMFIELD` function is not implemented.
4. The workstation GRAPHPAK permits the use of a *mouse* in addition to certain function keys when using the `READ` function discussed in *APL2 GRAPHPAK: User's Guide and Reference*. When the left argument of `READ` is a null, and the intent is to do freehand drawings within the graphics window, the mouse buttons offer an easy alternative to the use of the cursor. Each button produces a different color. To draw freehand, place the mouse pointer where you wish to start and press a mouse button, then release it and move to the next position, and again press one of the mouse buttons. Alternatively, function keys may be used to draw the line, in a variety of colors and patterns. Repeat this procedure until you are finished, then press Enter.

The GRAPHPAK workspace includes seven groups that form a set of functions implemented compatibly across all APL2 platforms.

The seven groups are:

GPBASE Contains the fundamental drawing and writing functions. This group contains the base code to use the AP 207 Universal Graphics auxiliary processor.
GPCHT Contains functions for drawing charts.
GPCONT Contains functions for drawing contour maps.
GPDEMO Contains functions illustrating various aspects of GRAPHPAK.
GPFIT Contains functions for curve fitting.
GPGEOM Contains descriptive geometry functions.
GPPLOT Contains plotting functions.

To run the demonstration contained in the GRAPHPAK workspace, enter:

```
)LOAD 2 GRAPHPAK
DEMO
```

GPBASE

© Copyright IBM Corporation 1994, 2017

This group contains the fundamental drawing and writing functions, and is required by all other GRAPHPAK workspaces. It contains the following functions:

COLOR	Changes the current color
DRAW	Draws lines between points
ERASE	Clears the screen
FILL	Fills a polygon
FIXVP	Sets the viewport
GRFIELD	Sets the graphics field
INTO	Used in coordinate transformations
MODE	Changes the current line mode (not functional, included for compatibility only)
READ	Reads coordinates or character data from the screen
SMFIELD	Sets the session manager field
STYLE	Changes the current fill or line style
USE	Permanently changes the current attributes (color and width)
USING	Temporarily changes the current attributes (color and width)
VIEW	Displays the current contents of the graphics field
VIEWPORT	Returns the coordinates of the corners of the current clipping viewport
WIDTH	Changes the current line mode
WRITE	Writes text on the current graphics field
XFM	Used in coordinate transformations

GPCHT

This group contains functions for drawing charts. It requires GPBASE and GPPLOT, and the following functions are available:

CHART	Draws a bar or column chart
FREQ	Plots a frequency chart
HCHART	Plots a hierarchical chart
PIECHART	Draws a pie chart
PIELABEL	Labels a pie chart
SAXES	Plots all three default axes on a surface or skyscraper chart
SAXISX	Plots the X axis of a surface or skyscraper chart with definable labels and annotation
SAXISY	Plots the Y axis of a surface or skyscraper chart with definable labels and annotation
SAXISZ	Plots the Z axis of a surface or skyscraper chart with definable labels and annotation
SLABEL	Writes the default labels on the axes of a surface or skyscraper chart.
SLBLX	Places definable labels on the X axis of a surface or skyscraper chart
SLBLY	Places definable labels on the Y axis of a surface or skyscraper chart
SLBLZ	Places definable labels on the Z axis of a surface or skyscraper chart
SS	Plots a skyscraper chart

© Copyright IBM Corporation 1994, 2017

STEP	Plots a step chart
STITLE	Adds a title to a surface or skyscraper chart
SURFACE	Plots a surface chart
SXFM	Maps three-dimensional coordinates into a screen window
WITH	Formats data for use with PIECHART

GPCONT

This group contains functions for drawing contour maps. It requires GPBASE and GPPLOT, and contains the following functions:

BY	Used to structure the input to CONTOUR
CONTOUR	Draws a contour map
OF	Used to structure the input to CONTOUR

GPDEMO

This group contains functions illustrating many aspects of GRAPHPAK. The demonstration can be started with the DEMO function, and it cycles through the complete set of demonstrations.

The individual demonstration functions contained in this workspace are:

APPLE	Shows the use of DRAW to create a picture
ATTRIBUTES	Shows the various line type and fill options available
BLI	Shows the use of DRAW to create a picture
CAYUGAPLOT	Shows the use of PLOT to create line graphs
DEMO	Cycles through all the demo functions in the workspace
FLAG	Shows the use of DRAW to create a picture
FSTAR	Shows the use of DRAW to create a picture
HEALTH	Shows the use of CHART to produce a column chart
IBMF	Draws the IBM logo
MILERUN	Shows the use of CHART to produce a bar chart
NHIST	Shows the use of CHART to produce a step chart
PIES	Shows the use of CHART to produce a pie chart
REVB	Shows the use of CHART to produce a simple bar chart
REVC	Shows the use of CHART to produce a simple column chart
REVENUES	Shows the use of PLOT to create line graphs
SKYSCRAPER	Shows the use of SS to produce a skyscraper chart
SPIRAL	Shows the use of SKETCH, THREEVIEWS, and PERSPECTIVE to create pictures
WAVEGUIDE	Shows the use of SURFACE to produce a surface chart
WGCONT	Shows the use of SURFACE to produce a contour chart

© Copyright IBM Corporation 1994, 2017

GPFIT

This group contains functions for curve fitting. It requires GPBASE and GPPLOT, and contains the following functions:

AVG	Prepares data for FIT to plot the average Y value
CLEAR	Clears the display screen
FIT	Draws a line graph through data points prepared by AVG, SL, POLY, EXP, LOG, POWER, LOGLOG, or SPLINE
EXP	Prepares data for FIT to plot the best log-linear fit on linear axes
FITFUN	Executes the last function plotted
LOG	Prepares data for FIT to plot the best log-linear fit on log-linear axes
LOGLOG	Prepares data for FIT to plot the best log-log fit on log-log axes
POLY	Prepares data for FIT to plot the best polynomial of specified degree
POWER	Prepares data for FIT to plot the best log-log fit on linear axes
SCRATCH	Erases points from a data array
SL	Prepares data from which FIT can plot the best straight-line fit
SPLINE	Prepares data for FIT to use in plotting a cubic spline through specified points

GPGEOM

This group contains descriptive geometry functions. It requires GPBASE and GPPLOT, and contains the following functions:

ISOMETRIC	Restructures data so that SKETCH produces an isometric projection
MAGNIFY	Transforms a data array so that the object represented is magnified in size
OBLIQUE	Restructures data so that SKETCH produces an oblique projection
PERSPECTIVE	Restructures data so that SKETCH produces a perspective drawing
RETICLE	Draws an outline of the clipping viewport and a pair of axes, showing the extent of the window into problem space
ROTATE	Transforms a data array so that the object represented is rotated
SCALE	Scales all elements of a data array so that the largest lies within set bounds
SKETCH	Produces an orthogonal projection
STEREO	Produces a stereo pair of images of an object
THREEVIEWS	Produces three views of an object, projected into the planes of the axes
TRANSLATE	Transforms a data array so that the object represented is moved

GPPLOT

This group contains plotting functions. It is required by GPFIT, GPCONT, and GPCHT, and requires GPBASE. It contains the following functions:

AND	Aids in formatting input for PLOT or SPLOT

© Copyright IBM Corporation 1994, 2017

ANNX	Draws an annotated horizontal axis with definable label positions
ANNY	Draws an annotated vertical axis with definable label positions
AXES	Draws default axes
AXIS	Draws definable axes
HOR	Draws an annotated horizontal axis with default label positions
LABEL	Produces the default labels for the X and Y axes
LBLX	Produces definable labels for the X axis
LBLY	Produces definable labels for the Y axis
PLOT	Plots a line graph
RESTORE	Restores all attribute and plotting variables to their default values
SPLOT	Similar to PLOT, but allows more user control over plotting characteristics
TITLE	Adds a title to the graph
VER	Draws an annotated vertical axis with default label positions
VS	Aids in formatting input for PLOT or SPLOT.

© Copyright IBM Corporation 1994, 2017

GUITOOLS Workspace

Note: This workspace is not provided on Unix systems.

GUITOOLS contains tools for building GUI applications. The general classes of functions are:

- Dialog Processing Functions
- Utility Functions
- Printing Functions

Dialog Processing Functions

The dialog processing tools support processing dialogs in applications. For detailed information, consult *APL2 Programming: Developing GUI Applications* and the `HOW_DLGPROCESS` variable in the GUITOOLS workspace.

`ALIGN`	Align controls
`BITWISE`	Apply function between bit representations of integers
`CALLCOM`	Call Component Object Model interface from AP 145
`CALLAPI`	Call an API
`CENTER_CHILD`	Center a child window within its parent's client area
`CENTER_WINDOW`	Center one window within another
`CHECK_EVENTS`	Process events during long running operations
`CONTEXTHELP`	Show context help
`CREATECTL`	Create a control window
`CREATEDLG`	Create a dialog
`CREATEMENU`	Create a menu
`CREATEOBJ`	Create an instance of an object class.
`DEFAULTPROC`	Send a message to a window's default procedure
`DESTROYDLG`	Destroys a dialog
`DESTROYOBJ`	Destroys an object
`EXECUTEDLG`	Execute a dialog and it's event handlers
`EXECUTEDLGX`	Execute a dialog and monitor other shared variable events
`FILEDLG`	Display a file dialog
`FILEDLGM`	Display a file dialog that allows multiple selections
`FOLDERDLG`	Display a browse for folder dialog
`FONTDLG`	Display a font dialog
`FREEAPI`	Free an API
`GETCHILDREN`	Get the handles of a parent window's children
`GETPARENT`	Get the handle of a child window's parent.
`GET_CELLSIZE`	Get the size of a grid control cell

© Copyright IBM Corporation 1994, 2017

GET_PROFILE	Get and list profile values from the registry or an INI file
GET_PROPERTY	Get a window property
GET_RANGE	Get a window range property
GUIRETRACT	Retract SV145
GUISHARE	Share SV145
HANDLE_DESKTOP	Retrieve the actual handle of the desktop
IDFROMWINDOW	Retrieve a window's identifier
ISWINDOW	Determine whether a window handle is valid
LOADAPI	Load an API
MOVEWINDOW	Move a window
MSGBOX	Display a message box
POPUPMENU	Display a popup menu
POSTMSG	Post a Message
RESIZE	Resize controls
SENDMSG	Send a message
SET_CELLSIZE	Set the size of a grid control cell
SET_PROFILE	Set and delete profile values in the registry or an INI file
SET_PROPERTY	Set a window property
SET_RANGE	Set a window range property
SHAREWINDOW	Share a variable with a window property
SHOW	Show or hide a window
SIZETOTEXT	Resize a control to fit its text
SPACE	Space controls
STARTWAIT	Start AP 145 holding application messages
UNICREATEDLG	Create a Unicode dialog
UNICREATEMENU	Create a Unicode menu
UNIFILEDLG	Display a Unicode file dialog
UNIFILEDLGM	Display a Unicode file dialog that allows multiple selections
UNIFOLDERDLG	Display a Unicode browse for folder dialog
UNIMSGBOX	Display a Unicode message box
UNIPOPUPMENU	Display a Unicode popup menu
WAIT_EVENT	Wait for an event
WINDOWFROMID	Retrieve a child window's handle

Utility Functions

The following functions are tools for performing common high level user interface operations. For detailed information, consult the HOW_UTILITY variable in the GUITOOLS workspace.

APLEDIT	Use Apledit to edit a function or operator
APLEXECPGM	Execute a system command

© Copyright IBM Corporation 1994, 2017

EDIT	Allows the user to edit an array.
HELP145	Access the on-line documentation for AP 145
PROMPT	Displays a prompt and provides an entry field
QUERYSYSCOLOR	Query the RGB value associated with a system color index
SELECT_1	Prompts the user to select one item from a matrix
SELECT_SOME	Prompts the user to make selections from a matrix
UNIEDIT	Allows the user to edit an array containing Unicode characters.
UNIFILE	Read a Multibyte or Unicode file or write a Unicode file as a character vector
UNI∆FM	Read a Multibyte or Unicode file or write a Unicode file as a character matrix
UNI∆FV	Read a Multibyte or Unicode file or write a Unicode file as a vector of character vectors

Printing Functions

The printing functions allow applications to produce printed documents.

To create a typical document, the printing functions are used as follows:

1. Create a logical printer with CREATE_PRINTER.
2. Select a printer by using PRINT_PROPERTY to set the PRINTER property to the name of the desired printer. Use the PRINTERS property to retrieve the names of the available printers. Alternatively, display a printer selection dialog with SELECT_PRINTER.
3. Set other printer properties such as orientation and margins with PRINT_PROPERTY. Alternatively, display a page customization dialog with PAGE_SETUP.
4. Start the document with OPEN_DOCUMENT.
5. Set desired attributes for the document with PRINT_PROPERTY and the SET_ functions.
6. Add text to the document with PRINT_SENTENCE.
7. Begin new lines and pages with NEWLINE and NEWPAGE.
8. Repeat steps 5, 6, and 7 as needed to create the entire document.
9. Close and send the document to the printer with CLOSE_DOCUMENT.
10. Destroy the logical printer with DESTROY_PRINTER.

For detailed information, consult the HOW_PRINT variable in the GUITOOLS workspace.

CREATE_PRINTER	Creates a logical printer
CREATE_UNICODE_PRINTER	(Windows only) Creates a Unicode logical printer
DESTROY_PRINTER	Destroys a logical printer
SELECT_PRINTER	Displays the print queue selection dialog
PAGE_SETUP	Displays the page customization dialog
PRINTER_PROPERTIES	Display printer properties dialog
PRINT_PROPERTY	Queries and sets printer properties
OPEN_DOCUMENT	Starts a print job
CANCEL_DOCUMENT	Cancels a print job
CLOSE_DOCUMENT	Completes a print job
PRINT_SENTENCE	Adds a sentence to a print job

© Copyright IBM Corporation 1994, 2017

NEWLINE	Starts a new line
NEWPAGE	Starts a new page
QUERY_PAGENUMBER	Returns the current page number
QUERY_LENGTH	Returns the length, in points, of a string
SET_COLOR	Sets the color of the body of the document
SET_FONT	Sets the font for a section of the document
SET_INDENT	Sets the indentation amount
SET_MARGIN	Sets the inside margin (used when output is duplex)
SET_WORDBREAK	Enables or disables wordbreak
SET_LINESPACE	Enables or disables interline spacing
SET_HEADING	Sets the running heading
SET_FOOTING	Sets the running footing
SET_PAGENUMBERS	Enables of disables page numbering

© Copyright IBM Corporation 1994, 2017

GUIVARS Workspace

Note: This workspace is not provided on Unix systems.

The GUIVARS workspace contains variables that define a set of common constants used in GUI programs.

Some AP 145 services require parameters that contain specific numeric or character values. These values are used to indicate what operation the service is to perform.

For example, when attempting to retrieve the handle of a standard control like a titlebar, you need to use the standard titlebar identifier. This identifier is called FID_TITLEBAR. The variable `FID_TITLEBAR` can be found in the GUIVARS workspace.

© Copyright IBM Corporation 1994, 2017

IDIOMS Workspace

APL2 is a very powerful and concise language. Although experienced APL2 programmers can produce working solutions to complex problems in a very short time, learning the APL2 language can take years. The novice is usually entranced with the power of APL2, but may have a hard time thinking in vector notation. APL2 algorithms are not always obvious!

In order to speed up the learning process of APL2, IDIOMS was developed. With over 600 distinct APL2 phrases, sorted into 24 general categories, IDIOMS represents a fairly complete list of solutions to common application problems and lets you take advantage of algorithms that many others have developed.

Since this list is in soft copy, you can access it directly from your workspace and dynamically insert the idioms into your own code.

- Executing the IDIOMS Function
- Categories
- Naming Conventions

Executing the IDIOMS Function

Copy the `IDIOMS` function into your active workspace and then execute it:

```
)PCOPY 1 IDIOMS IDIOMS
IDIOMS
```

You can run this program from your own workspace, because it does not overwrite any programs you use. The `IDIOMS` function and all the subroutines it uses are stored in an APL2 namespace, `APL2IDI.ans`.

Once `IDIOMS` is running, a full-screen interface gives you control over all the facilities available. The following keys can be used from the main screen. Use the F1 (Help) key to get assistance on the Window or Group screen.

Key	Function	Description
F1	Help	Display a series of screens that give assistance for the current screen.
F2	Window	Display a window of previous searches. Move the cursor to the desired item and press Enter to select a search, which appears on the main screen. Press F3 to return to the main screen.
F3	Return	Return to the APL2 session, passing any idioms selected with F9 to the workspace as an explicit result.
F4	Function	Save the APL2 idiom identified by the cursor as a function called `IDIOM_LIST`. If `IDIOM_LIST` already exists, the selected idiom is appended as a new line to the end of the function. This function can be a prototype for a new program using the selected expressions to accomplish the desired task.
F5	Local Search	Search the displayed group of idioms for the search argument. This can be used to narrow the search to a particular group of idioms.
F6	Group	Display a list of each group of idioms. Place the cursor beside the desired group and

© Copyright IBM Corporation 1994, 2017

Key	Function	Description
		press F6 again to select that group. Press F3 to return to the main screen without selecting a group.
F7	PageUp	Scroll one screen toward the top.
F8	PageDown	Scroll one screen toward the bottom.
F9	Result	Append the idiom identified by the cursor to the result of the `IDIOMS` function. Using the session manager, you can place these lines in your function by entering function definition mode and then inserting a line number `[n]` at the beginning of each desired idiom.
F10	Environment	Cycle environment for which appropriate idioms are displayed. Note that idioms selected from a different environment than you are running may not run in your environment.
Shift-F7	Home	Scroll to the top.
Shift-F8	End	Scroll to the bottom.
PageUp	PageUp	Scroll one screen toward the top.
PageDown	PageDown	Scroll one screen toward the bottom.
Home	Home	Scroll to the top.
End	End	Scroll to the bottom.
Enter	Search	Search through all the idioms for the search argument.

The idioms are available in either index origin. The `IDIOMS` program lets you select the index origin preferred for the code you are writing. To change the origin, overtype the displayed value with the desired index origin and press Enter.

To trace the searches that you generate, `IDIOMS` calls itself recursively. You can do multiple searches, without losing the results of any intermediate search.

A history of searches is saved from session to session in an external file named `IDIOMS.asf`. The file is created automatically, if it does not already exist, in the user's default working directory.

Categories

To facilitate fast selection of idioms, they are grouped into categories, as follows:

Category	Type of Algorithms
1	Assignment
2	Boolean Selection
3	Boolean Tests General
4	Boolean Tests Numeric
5	Computational
6	Conversion
7	Date and Time
8	External Name Routine

© Copyright IBM Corporation 1994, 2017

Category	Type of Algorithms
9	Financial
10	Formatting
11	Function
12	Manipulating Characters
13	Manipulating Numbers
14	Numeric Range
15	Numerical Geometry
16	Selecting Positions
17	Sorting
18	Statistics Descriptive
19	Statistics Distribution
20	Structural
21	Text Arrangement
22	Text Selection and Change
23	Trigonometry
24	Vectorizing

Naming Conventions

A consistent naming convention is used throughout the list. The names used are:

Rank		Type		Usage	
A	(Array)	B	(Boolean)	G	(Graded or grouped)
M	(Matrix)	C	(Character)	L	(Lengths)
O	(One-item vector)	F	(Floating point)	P	(Positions)
S	(Scalar or one-item vector)	I	(Integer)	U	(Unique)
		N	(Numeric)	X	(Extension)
V	(Vector)	Z	(Complex)	Y	(Extension)

These are combined in various ways to identify the type of object. For example:

Name	Contents
A,AX,AY	General arrays
IM	Integer matrix
BM	Boolean matrix
N,NX,NY	Numeric vectors
BS	Boolean scalar
PAV	Position array of vectors
CA	Character array
PS	Position scalar
C,CX,CY	Character vectors

© Copyright IBM Corporation 1994, 2017

Name	Contents
UM	Unique matrix
GAF	Graded array of floating point
VM	Vector of matrices
GI	Graded integer vector
VV	Vector of vectors
GM	Graded matrix
V,X,Y	General vectors

© Copyright IBM Corporation 1994, 2017

MATHFNS Workspace

The functions in this workspace are as follows:

Matrix Inverse and Matrix Divide

`DOMINO` Computes matrix inverse or matrix divide
`DOMINOF` Computes matrix inverse or matrix divide (fast)

Eigenvalues and Eigenvectors

`EIGEN` Computes eigenvalues and eigenvectors
`EIGENVALUES` Computes eigenvalues

Factorial and Binomial

`FACTORIAL` Computes factorials and binomials

Fast Fourier Transform

`FFT` Computes fast Fourier transform
`IFFT` Computes inverse fast Fourier transform

Formatting Complex Numbers

`FMTPD` Formats in polar form with angular measure in degrees
`FMTPR` Formats in polar form with angular measure in radians

Roots of Polynomials

`POLYZ` Computes the zeros of polynomials

Matrix Inverse and Matrix Divide

```
Z← DOMINO R ⍝ Matrix inverse
Z←L DOMINO R ⍝ Matrix divide
Z← DOMINOF R ⍝ Matrix inverse (fast)
Z←L DOMINOF R ⍝ Matrix divide (fast)
```

These two functions provide a direct replacement for the matrix inverse and matrix divide (⌹) primitive when applied to arguments containing complex numbers. `DOMINO` is a direct APL2 implementation of the algorithm used by the primitive ⌹ function of the mainframe APL2 interpreter. By applying the primitive ⌹ function to combinations of the real and imaginary parts of its arguments, `DOMINOF` produces results equivalent to those

© Copyright IBM Corporation 1994, 2017

given by the DOMINO function. This is usually faster than DOMINO, but the intermediate applications of the primitive ⌹ function may fail for some arguments.

Eigenvalues and Eigenvectors

```
Z← EIGEN R
Z←L EIGEN R
```

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

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

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

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

These statements obey the identity:

```
      X×[2]V ↔ R+.×V
```

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

```
      Z←⍉ EIGEN ⍉R

      X←Z[;1]

      V←0 1↓Z
```

They obey the identity:

```
      X×[1]V ↔ V+.×R
```

In the dyadic case, L is also a simple square matrix of real numbers, and ⍴L equals ⍴R. Z is a simple real or complex matrix of shape 1 0+⍴R containing the solution to the generalized eigenvalue problem for L and R:

```
      Z←L EIGEN R
      X←Z[1;]
```


© Copyright IBM Corporation 1994, 2017

```
V←1 0↓Z
```

They obey the identity:

```
(L+.×X)×[2]V ↔ R+.×V
```

The eigenvalues and eigenvectors are computed by using the "QZ Algorithm". (See "An Algorithm for Generalized Matrix Eigenvalue Problems" by C.B. Moler and G.W. Stewart, *SIAM Journal of Numerical Analysis*, 10: 241 - 256, 1973.) The numerical accuracy of the result depends upon the "condition" of the matrix of eigenvectors. In particular, accuracy may be degraded if there are repeated eigenvalues.

```
Z← EIGENVALUES R
Z←L EIGENVALUES R
```

This function takes the same arguments as `EIGEN` but returns only the eigenvalues. It is faster than the `EIGEN` function for the same inputs, and can be used whenever the full result is not required.

Factorial and Binomial

```
Z← FACTORIAL R ⍝ Factorial
Z←L FACTORIAL R ⍝ Binomial
```

This function provides a direct replacement for the primitive factorial or binomial (`!`) function when applied to arguments containing complex numbers. `FACTORIAL` is a direct APL2 implementation of the algorithm used by the primitive `!` function of the mainframe APL2 interpreter.

Fast Fourier Transform

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

This function computes the discrete Fourier transform of a set of numbers.

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

The result of the `FFT` function corresponds to that of the discrete Fourier transform as defined by SC23-0526, *Engineering and Scientific Subroutine Library, Guide and Reference*:

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

© Copyright IBM Corporation 1994, 2017

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

This function computes the inverse Fourier transform of a set of numbers.

```
      R ↔ IFFT FFT R
```

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

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

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

For example:

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

Formatting Complex Numbers

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

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

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

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

Roots of Polynomials

```
Z←POLYZ R ⍝ POLYnomial Zeros
```


© Copyright IBM Corporation 1994, 2017

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

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

```
      POLYZ ¯2 1
0.5
      POLYZ 2 0J1
0J¯0.5
      POLYZ 1 ¯2 1
1 1
      POLYZ 1 0 1
0J1 0J¯1
      POLYZ 1 ¯6 11 ¯6
1 2 3
      POLYZ 1 ¯20 154 ¯584 1153 ¯1124 420
1 1.999999978 2.000000022 3 5 7
```

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

POLYZ uses subroutines POLYZC and POLYZF.

© Copyright IBM Corporation 1994, 2017

MIGRATE Workspace

The MIGRATE workspace contains functions useful in migrating workspaces from other APL2 platforms.

ATFUSTOLC - Remove Underbarred Characters

```
ATFUSTOLC 'filename'
```

Takes the name of a transfer form file that has been downloaded in binary from a mainframe APL2 system. and changes *all* the underbarred letters to the equivalent lowercase characters.

This differs from using `CASE(1)` or `CASE(2)` on the host before creating and downloading a transfer file, in that this converts underbarred letters even in literal constants, function or operator comments, and arrays.

VSCOPY - Convert VS APL Workspaces

```
VSCOPY 'wsname [object_list]'
```

`VSCOPY` is a function that makes it possible for objects from a VS APL workspace (saved with the VS APL `)SAVE` command as a CMS file or a TSO data set, and then downloaded in binary to a workstation system) to be copied directly into the active workspace of APL2.

`wsname`
: is the workspace name. If no extension is given, a default extension of `.VWS` is appended. If the name contains one or more dots or is enclosed in quotes, it is used exactly as given.

`object_list`
: is the list of names of objects to be copied. This may be omitted, in which case *all* objects in the workspace are copied.

Groups are copied in as character arrays, with each name represented as a row in the array. The names of all groups copied are added into the variable `grps`.

Object names may be prefixed by an asterisk (`*`):

`*function_name`
: causes the function to be left in its canonical form

`*group_name`
: causes the group definition, but not its contents to be copied

`*variable_name`
: causes the variable to be left untranslated

© Copyright IBM Corporation 1994, 2017

NETTOOLS Workspace

The NETTOOLS workspace contains tools for writing network enabled applications. The workspace includes the following groups of tools:

GPDESC	Descriptive variables and functions
GPDATE	Date formatting and parsing tools
GPENCODE	Data encoding and decoding tools
GPHTTP	HTTP 1.1 Web Server and Client Tools

The workspace contains documentation that is designed to be viewed with a web browser. To view the documentation, start the APL2 web server. The server prints out the network name and IP address of the machine.

```
      )LOAD 2 NETTOOLS
      HTTP_SERVER
Host name: AplMachine
Host IP address: 9.112.25.240
```

Next, start a web browser and type in the name or IP address of the machine. Use either of the following formats:

```
      http://AplMachine
      http://9.112.25.240
```

On systems where the default listening port number specified in `CONFIGURE_HTTP_SERVER` is not 80, that port number must be specified explicitly:

```
      http://AplMachine:8000
      http://9.112.25.240:8000
```

To shut down the server, use the browser to connect to the server's administration port. Use either of the following methods:

```
      http://AplMachine:8888
      http://9.112.25.240:8888
```

You will be prompted for a user name and a password. Type the user name "IBM" and the password "APL2". The APL2 HTTP Server Administration page will be displayed. Push the Stop button.

© Copyright IBM Corporation 1994, 2017

PRINT Workspace

Note: This workspace is provided only on Unix systems.

This workspace sends text to an attached printer or to a file. The three main functions in the workspace are `PRINTER`, `PRINT`, and `NEWPAGE`.

The PRINT and PRINTWS facilities share common code. Be sure to read PRINT and PRINTWS: Common Features before reading about the PRINT workspace functions.

- PRINT and PRINTWS: Common Features
- The PRINTER Function: Using PRINTER ON and PRINTER OFF
- Filling In the Full-Screen Prompts
- The PRINT and NEWPAGE Functions
- A Typical Use of PRINT
- Modifying Applications to Use PRINT
- Additional Functions
- Additional Details for Programmers

PRINT and PRINTWS: Common Features

Both the PRINT and PRINTWS workspaces send data to an attached printer or to a file. PRINT is designed to send your own reports, while PRINTWS displays images of the functions, operators and variables in the workspace. These two facilities share some common code, by means of a function file. When each facility is invoked, they bring in subfunctions from a file (`PRINTWS.aof`). These functions are brought in as local functions, and are not seen in the workspace.

Both facilities bring up a set of full-screen panels to prompt for your printer configuration and printing parameters. They can direct your output to the screen, to a printer, or to an AIX file. The prompts on the first menu differ slightly between the two facilities; for details on your entries on these full-screen panels, refer to Filling in the Full-Screen Prompts in the PRINT or PRINTWS documentation.

Here is the action of each of the function keys on the main panel:

F1 **HELP.** Online help is available on all of the menus by pressing F1. For context-sensitive help, position the cursor to the field of interest and press F1. To view more extensive help text, press F1 again.

F2 **SELECT A PRINTER.** Both PRINT and PRINTWS support a variety of printers. Different printers have different capabilities with regard to fonts (character sets) and other parameters. F2 takes you to a second menu panel that lets you position the cursor and press Enter to select the characteristics of the printer that you wish to use. For each of the printers listed, PRINT and PRINTWS either downloads an APL font to the printer or selects an APL font cartridge. If your printer does not have provision for either of these methods of support, you can select the menu item marked *Other: APL chars generated as graphics*. This causes the printer to use its default character set for alphanumeric characters, and, as the selection implies, it generates each of the APL symbols that you print as graphics images. This is less desirable than using an APL font, because it causes slow printing and because the APL characters may be lower-resolution than the text. However, it provides a means for printing APL characters on printers that would not otherwise support them.

F3 **CANCEL.** If you want to exit from the full-screen panels without starting a report, press F3. F12 can be

© Copyright IBM Corporation 1994, 2017

used as an alternate key.

F5 **AIX SMIT.** When running under AIX, the F5 key within both PRINT and PRINTWS gives access to the System Management Interface Tool (SMIT). The command `smit spooler` is issued through AP 100, so only the portions of SMIT related to printer support are available. This provides an easy menu-driven means for selecting information regarding, among other things, the printer queue names and queue status.

F9 **START.** This key begins your report, and in the PRINT facility, takes you out of the full-screen panels.

All of the values that you enter on each of the full-screen panels are retained in a profile in your home directory. The next time that you use either facility, only *changes* need to be entered. Often, your only entry is a single function key.

The profile is called `.PRINT.prf` in the PRINT facility, or `.PRINTWS.prf` in the PRINTWS facility. In this way, titles and parameters are kept separate between the two facilities.

The PRINTER Function: Using PRINTER ON and PRINTER OFF

`Z←PRINTER ON`	Starts printing session; brings up full-screen panels
`Z←PRINTER OFF`	Ends printing session; returns control to the screen
`Z←PRINTER-ON`	Same as `PRINTER ON`, but suppresses the panels
`Z←PRINTER-OFF`	Same as `PRINTER OFF`
`Z←PRINTER 0`	Same as `PRINTER OFF`
`Z←PRINTER 1`	Same as `PRINTER ON`
`Z←PRINTER ⍳1`	Same as `PRINTER-ON`
`Z←PRINTER 2`	Sends data to a file; brings up full-screen panels
`Z←PRINTER ⍳2`	Same, but suppresses full-screen panels
`Z←PRINTER 2 '@output.fns'`	Explicitly names the output file
`Z←PRINTER ⍳2 ⍙_FileName`	Explicitly names the output file

`ON` and `OFF` are two small utility functions that return a `1` and a `0`, respectively. Calling `PRINTER ON` brings up a set of full-screen panels to prompt for your printer configuration and printing parameters. It can direct your output to the screen, to a printer, or to an AIX file. For details on your entries on these full-screen panels, refer to Filling in the Full-Screen Prompts. A negative number as the argument bypasses the panels, so to suppress the full-screen panels and use the values previously entered on those panels, use `PRINTER-ON` (or `PRINTER ⍳1`).

`PRINTER ON` presents you with a full-screen menu panel, requesting the information discussed below. Online help is available by pressing F1. For context-sensitive help, position the cursor to the field of interest and press F1. After you have filled in the full-screen prompts, press F9 to exit from the panels and begin your printing (or F3 to cancel).

`PRINTER OFF` ends the printing session, and switches control back to the screen.

The result from `PRINTER ON` is a three-element vector:

© Copyright IBM Corporation 1994, 2017

Panel Selection	Three-Element Result
F3 (exit) pressed	`⍳1 'PRINTER CANCELLED: ' 'reason'`
`S` (for *Screen*)	`0 'PRINTER OFF: ' 'reason'`
`P` (for *Printer*)	`1 'PRINTER ON: ' 'printer info'`
`F` (for *File*)	`2 'WRITING TO FILE: ' 'file name'`

Note: The output is spooled to a file before printing. It is `PRINTER OFF` that sends this file to the printer (and also controls the downloading of fonts and the font selection). Therefore, "`PRINTER OFF`" *must* be run in order to get printed output. If this step is skipped, no output is generated.

Filling In the Full-Screen Prompts

Title
: You can enter an optional title, up to 32 characters long. This entry is not used by the functions provided in PRINT, but it is available for use by a customized "`HEADING`" function.

Width of Printed Report
: Indicate the maximum number of printed characters you wish to have on a line of output, so that the program can break output lines at the proper places.

Lines Per Page
: Indicate the number of printed lines per page that your printer can handle. This tells the program when to skip to a new page. If the value is set to `0`, no paging occurs. This number includes the page headings, if you use that facility.

Margin Control
: The size of the left margin is the amount of blank indentation that this program should provide. You may want to position the text close to the margin (to get more onto the page) or leave some extra space (for three-hole-punched pages, for instance).

Duplexing
: Indicate whether the printed pages should be duplexed (printed on both sides) or not, for printers that support this operation. This entry is not used by the functions provided in PRINT, but it is available for use by a customized "`HEADING`" function. See the description of the "`⍙_Duplex`" variable.

Output
- `P` sends the output to a file during processing, and then routes the file to a printer when the report is finished.
- `S` displays the output on your screen. This may be useful when you are capturing the session log for inclusion in another report.
- `F` sends the output to a file, and does not spool it further.

File Name
: This field lets you specify the name of the file to which the output is directed.

The PRINT and NEWPAGE Functions

`PRINT` can be used to print any APL2 object or result, of any rank or type from your APL2 program. It can be used directly or it can be called from any other APL2 user-defined function, thus giving the program control of the printer.

© Copyright IBM Corporation 1994, 2017

```
PRINT object
```

The PRINT function formats the array object and sends the output to the attached printer. The current page position is maintained by global variables named "⎕_PageNo" (page number) and "⎕_LineNo" (line number).

The following examples show what is printed for various entries:

Entry	Printed
PRINT 2+2	4
PRINT 'ABCabc'	ABCabc
PRINT ⍳10	1 2 3 4 5 6 7 8 9 10
PRINT 2 3⍴'ABCDEF'	ABC DEF

(A variable can also be printed)

```
X←'IS A VARIABLE'
PRINT 'X ',X      X IS A VARIABLE
```

When the Printer or File options are in use, the PRINT function displays the page numbers of the material being printed, as feedback on the progress of the printing. (These page numbers are produced by the HEADING function, to be described later. HEADING is called from within PRINT.)

```
NEWPAGE
```

Causes following output (if any) to be printed starting at the top of a fresh page.

A Typical Use of PRINT

A typical use of the PRINT facility would be to call PRINTER ON to start the printing session, then use PRINT as many times as needed to send all of the data to the printer, with calls to NEWPAGE wherever you specifically want material to start at the top of a page, and finally a call to PRINTER OFF to end the session.

Modifying Applications to Use PRINT

You can easily incorporate the PRINT facility into existing applications, so that reports, for instance, can be displayed at your screen or sent to a printer or file.

As an example, assume that you have a simple reporting function, like this:

```
    ∇
[0]  REPORT;YEAR;REGION;TABLE;SALES
[1]  LOOP:
```


© Copyright IBM Corporation 1994, 2017

```
 [2]   YEAR←PROMPT 'Enter the year desired:  '
 [3]   →(0=⍴YEAR)/END
 [4]   REGION←PROMPT 'Enter the region:  '
 [5]   OPEN 'YESALES'
 [6]   TABLE←GET REGION
 [7]   SALES←(TABLE[;1]=YEAR)⌿TABLE
 [8]   'Sales Report for Region ',⍕REGION
 [9]   '¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯'
 [10]  'Year' 'Salesman' 'Customer' 'Calls' 'Cost' 'Net'
 [11]  '¯¯¯¯' '¯¯¯¯¯¯¯¯' '¯¯¯¯¯¯¯¯' '¯¯¯¯¯' '¯¯¯¯' '¯¯¯'
 [12]  ⎕←SALES
 [13]  ''
 [14]  →LOOP
 [15] END:
 [16]  'DONE.'
    ∇ 03/17/1991 12.11.25 (GMT-7)
```

Using the functions supplied in the PRINT facility, you can place a call to the PRINT function on each of the lines that cause printing to occur, and include a call to PRINTER ON and PRINTER OFF, like this:

```
    ∇
 [0]  REPORT;YEAR;REGION;TABLE;SALES;R
>[1]   R←PRINTER ON
>[2]   →(¯1=↑R)/0
 [3]  LOOP:
 [4]   YEAR←PROMPT 'Enter the year desired:  '
 [5]   →(0=⍴YEAR)/0
 [6]   REGION←PROMPT 'Enter the region:  '
 [7]   OPEN 'YESALES'
 [8]   TABLE←GET REGION
 [9]   SALES←(TABLE[;1]=YEAR)⌿TABLE
>[10]  PRINT 'Sales Report for Region ',⍕REGION
>[11]  PRINT '¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯¯'
>[12]  PRINT 'Year' 'Salesman' 'Customer' 'Calls' 'Cost' 'Net'
>[13]  PRINT '¯¯¯¯' '¯¯¯¯¯¯¯¯' '¯¯¯¯¯¯¯¯' '¯¯¯¯¯' '¯¯¯¯' '¯¯¯'
>[14]  PRINT SALES
>[15]  PRINT ''
>[16]  NEWPAGE
 [17]  →LOOP
>[18] END:R←PRINTER OFF
 [19]  'DONE.'
    ∇ 05/02/1993 14.13.19 (GMT-7)
```

Prompting still occurs at the screen, and the finished report is directed to the point desired: screen, printer, or file. If the "screen" option is selected, this function operates essentially the same way that it did before the modifications.

Additional Functions

```
Z←HEADING
```

This function is called by the PRINT function; it is not meant to be called directly. In its supplied form, it returns a 0-by-0 matrix, so that no page heading is generated. For application building, you can edit this function to insert whatever code you wish into it; the explicit result of this function is used as the heading for each page. The ⍙_Title and ⍙_PageNo variables could be used here (see the next page). Also, the ⍙_Duplex variable could be checked to see if alternation of headings was requested, so that, for instance, page numbers could be made to always print at the outside (nonbinding edge) of a page. For example,

© Copyright IBM Corporation 1994, 2017

```
Z←(¯1 1ρ⍙_Width)[1+0=2|⍙_PageNbr]↑'Page ',⍕⍙_PageNbr
```

prints the current page number at the left side of even-numbered pages and at the right side of odd-numbered pages.

Page numbers (showing how far along your printer output has gotten) are generated from within this function. If you want to suppress the numbers or capture them, this function can be altered.

HEADING is an optional function; if it does not exist in your workspace, there are simply no headings and no feedback of page numbers as printer output is being generated.

```
Z←FN name
```

Returns the canonical representation of the function name with line numbers added to the left-hand side and any comments or labels exdented. This can be used to print the definition of a function F with PRINT FN 'F' or PRINT WRAP FN 'F' (see below).

```
Z←L WRAP R
```

This function "wraps" the lines in the matrix R so that they are no greater than the length specified by the integer value in L. Z is always of rank 2. If L is omitted, the maximum width of the lines is controlled by ⍙_Width.

By using the FN and WRAP functions, you can create function listings similar to those produced by the PRINTWS facility, but with more flexibility. For instance, you may wish to display a small function in a large typesize with no page headings, for creating viewfoils. Or you may want to show the definition of a function along with examples of its use.

Additional Details for Programmers

The responses that you enter on either of the full-screen panels cause several global variables to be set, which are used to control the printing. These variables could be referenced by your application code. They are:

PRINTGP	Names of all of the functions and variables used by the PRINT facility; useful for expunging objects
⍙_Duplex	(1=duplex); could be used by a HEADING function that you define yourself
⍙_Indent	Number of characters of indent at left margin
⍙_PageLng	Page length (the number of lines that can be printed on the page, using the current font). If ⍙_PageLng is specified as 0 or as an empty vector, no page breaks are generated.
⍙_PageNo	Page number
⍙_PtrFlag	Screen/Printer/File flag (0, 1 or 2, respectively)
⍙_PtrInfo	Information regarding printer and fonts

© Copyright IBM Corporation 1994, 2017

`⎕_Title`	Title for this report
`⎕_Width`	Line-length of output, in characters

The following names are also created, and exist in your workspace after you use PRINT. However, these variables are intended for use only by the PRINT facility itself. They are *not* intended to be accessed by other application code:

`⎕_C210` and `⎕_D210`	Are the shared variables
`⎕_Facility`	Identifies the PRINT or PRINTWS facility
`⎕_FileName`	File name for output (option "`F`" only)
`⎕_LineNo`	Current line number being printed
`⎕_PageText`	Page of data being printed
`⎕_RecNo`	Record number for output

`⎕_PRINTER_ON`, `⎕_PRINTER_OFF`, and `⎕_ERR` also need to be in your workspace; they are subfunctions called by the `PRINTER` function.

© Copyright IBM Corporation 1994, 2017

PRINTWS Workspace

The PRINTWS workspace provides a means for printing the contents of the active workspace. It produces a listing of all the variables, functions, and operators in the workspace.

The workspace contains a function named PRINTWS. It is available on all systems. To use it:

```
)LOAD your-workspace
)PCOPY 2 PRINTWS PRINTWS
PRINTWS
```

You are prompted to select a printer and to set the print job properties.

- Limitations of PRINTWS
- Using PRINTWS on Windows
- Using PRINTWS on Unix Systems

On Windows the workspace also contains an external function named PRINTWSG. Unlike PRINTWS, PRINTWSG prompts for formatting options. To use it:

```
)LOAD your-workspace
)PCOPY 2 PRINTWS PRINTWSG
PRINTWSG
```

Output generated by the PRINTWSG function does not include a display of its own code. PRINTWSG has no other limitations regarding name conflicts.

Limitations of PRINTWS

The PRINTWS function has the following limitations:

1. Objects with names that conflict with the local names used by PRINTWS will not be printed. In general, all local names begin with the characters ∆_.

   On Windows, the name AP145 is also localized.

2. Output generated by the PRINTWS function does not include a display of its own code.

Using PRINTWS on Windows

Local variables control some of the properties of the printed output which users may want to customize. These variables are set at the top of the PRINTWS function. They are:

Variable	Contents
∆_Header	Text for the header
∆_Width	Width in characters of the print output
∆_Margin	Width of left margin in points
∆_HeaderFont	Font for header information

© Copyright IBM Corporation 1994, 2017

Variable	Contents
Δ_FooterFont	Font for footing information
Δ_BodyFont	Font for functions, variables and operators
Δ_JobName	Document name for the print job
Δ_PMTOOLS	Location of printing utilities
AP145	GUI interface AP number

PRINTWS creates and uses an APL print object with AP 145. Cover functions for using print objects are provided in the GUITOOLS Workspace.

Using PRINTWS on Unix Systems

The PRINT and PRINTWS workspaces share common code. For more information, see PRINT and PRINTWS: Common Features.

PRINTWS presents you with a fullscreen menu panel, requesting the information discussed below. Online help is available by pressing F1. For context-sensitive help, position the cursor on the field of interest and press F1. After you have filled in the fullscreen prompts, press F9 to exit from the panels and begin printing, or F3 to cancel.

Title

You can enter an optional title, up to 32 characters long. This title is printed a tthe top of each page of printed output.

Pagination

This field lets you control the pagination of the printed reports. Possible values are:

- 1 One function or variable per page
- 2 Several per page, as space permits
- 3 Continuous forms, no page breaks

Width of Printed Report

Indicate the maximum number of printed characters you wish to have on a line of output, so that the program can break output lines at the proper places.

Lines Per Page

Indicate the maximum number of printed lines per page that your printer can handle. This tells the program to skip to a new page. The number should include the page headings, which are four lines long.

Margin Control

The size of the left margin is the amount of blank indentation that this program should provide. You may want to position the text close to the margin (to get more onto the page) or leave some extra space (for three-hole-punched pages, for instance).

Duplexing

Indicate whether the printed pages should be duplexed (printed on both sides) or not, for printers that support this operation. This controls the page headings: duplexed output has page numbers alternated on the left and right sides of the pages, so that they are all away from the binding edge; non-duplexed pages have all of the page numbers on the right-hand side.

Select - Function and Operators

Functions and operators are time-stamped automatically by the system whenever they are edited.

- Y displays all of the functions and operators in the workspace (except the PRINTWS function).

© Copyright IBM Corporation 1994, 2017

`S` displays just the selected functions and operators (those that have changed since a given date; you are prompted for this date).

`N` indicates that no functions or operators are to be shown.

Functions Since a Given Date

If you want to print only those functions and operators that have changed since a specified date (option `S`), indicate that date here in a `"yyyy mm dd"` format. The number of functions and operators to be displayed is shown at the left side of the screen.

Select - Variables

Do you want to have variables included in the printed report?

`Y` displays all of the variables in the workspace.

`N` indicates that no variables are to be shown.

Output

`P` sends the output to a file during processing, and then routes the file to a printer when the report is finished.

`S` displays the output on your screen. This may be useful when you are capturing the session log for inclusion in another report.

`F` sends the output to a file, and does not spool it further.

File Name

This field lets you specify the name of the file to which the output is directed.

© Copyright IBM Corporation 1994, 2017

SQL Workspace

The SQL workspace enables you to execute SQL statements and retrieve information from databases using AP 127 (the DB2 processor) or AP 227 (the ODBC processor).

The following table lists the syntax of the functions and operator provided with the SQL workspace, along with their corresponding auxiliary processor operations. For additional information about the SQL workspace, AP 127 and AP 227, see APL2 Programming: Using Structured Query Language

Function Name and Syntax	**Auxiliary Processor Operation**
`CALL` *name* [*values*]	`'CALL'`
`CHART` *data*	
`CLOSE` *name*	`'CLOSE'`
`COMMIT`	`'COMMIT'`
`CONNECT` *database-identifier*	`'CONNECT'`
format `DATE` *ts*	
`DECLARE` *name* [`'HOLD'`|`'NOHOLD'`]	`'DECLARE'`
`DESC` *name* [*type*]	`'DESCRIBE'`
`EVAL` *data*	
`EVALSIM` *data*	
`EXEC` *stmt*	`'EXEC'`
`FETCH` *name* [*options..*]	`'FETCH'`
`GETOPT`	`'GETOPT'`
`IN`	
`ISOL` *setting*	`'ISOL'`
`MESSAGE` *rcode*	`'MSG'`
`NAMES`	`'NAMES'`
`ODBCOPEN` *name type*	`'ODBCOPEN'`
`ODBC` *type*	`'ODBC'`
`OFFER`	
`OPEN` *name* [*values*]	`'OPEN'`
`PREP` *name stmt*	`'PREP'`
`PROCEDURE` *stmt* [*values*]	
`PURGE` *name*	`'PURGE'`
`PUT` *name values*	`'PUT'`
`QUE` *stack*	
`QUERY` *stmt* [*values*]	
`RESUME` *stack*	

© Copyright IBM Corporation 1994, 2017

Function Name and Syntax	Auxiliary Processor Operation
ROLLBACK	'ROLLBACK'
SETOPT *options..*	'SETOPT'
SHOW *result*	
SQL *stmt* [*values*]	
SQLCA	'SQLCA'
SQLHELP *keyword*	
SQLSTATE	'SQLSTATE'
SSID *subsystem*	'SSID'
STATE *name*	'STATE'
STMT *name*	'STMT'
format TIME *ts*	
TIMESTAMP *ts*	
TRACE *(module level)..*	'TRACE'
(*f* UNTIL) *stack*	
XMLSIZE *size*	'XMLSIZE'

© Copyright IBM Corporation 1994, 2017

TCL Workspace

Tool Command Language (Tcl) is a popular scripting language that is available for many operating systems. Tcl provides a wide variety of routines including tools for string parsing and file IO. There are also numerous extensions available including packages for building graphical user interfaces (GUI), database access, object-oriented programming, and network access.

The TCL external function executes one or more Tcl commands.

The TCL workspace contains functions that demonstrate using Tcl from APL2. Consult the HOW variable for further information.

© Copyright IBM Corporation 1994, 2017

TIME Workspace

The TIME workspace contains the an association to the TIME external function, to assist with the performance monitoring and tuning of APL2 code.

The performance monitoring facility provides the capability to measure a running application and determine the CPU time used by each defined program, each line within each defined program, or both.

The facility works by associating with each line of each defined program, a pair of counters to record the number of times the line is executed and the total CPU time consumed by the line.

Typically, timing information can be obtained for an application as follows:

```
)LOAD workspace
)PCOPY 1 TIME TIME ⍝ To gain access to the facility
TIME 0             ⍝ To enable and zero counters
⍝ (Run application here)
TIME 1             ⍝ To see times for program run
⍝ (Analyze timing information here)
TIME 2             ⍝ To see times for each line
⍝ (Analyze timing information here)
)CLEAR             ⍝ When time analysis is complete
```

Use of the timing facility requires space in the workspace for the counters and also increases running time by some small amount. Thus, in general you should not) SAVE after doing a time analysis.

TIME 0
: Enable timing and create counters for all lines in all unlocked programs. The counters are set to zero.

TIME 1
: Fetch times for all programs for which timing has been gathered and return a matrix containing for each function or operator actually executed, the number of times the function or operator was executed, the total CPU time used by the function or operator, the percentage of the CPU time used by the function or operator, and the name of the program. If column headings are desired, the following may be used:

```
HD1←'COUNT' 'TIME' '%' 'PROGRAM'
HD1,[⎕IO] TIME 1
```

Three heading vectors - HD1, HD2, and HD3 - are already defined in the workspace, with the values shown here.

The figure given for the number of times each program is executed is based on the assumption that this count is the same as that for the first executable (noncomment) line of the program.

TIME 2
: Fetch times for all lines of all programs for which timing has been accumulated. The result is a matrix of the same format as TIME 1 except the line from the function or operator is also listed. If column headings are desired, the following may be used:

```
HD2←'COUNT' 'TIME' '%' 'PROGRAM' 'STMT'
HD2,[⎕IO] TIME 2
```


© Copyright IBM Corporation 1994, 2017

Each of the above uses of TIME also accepts an optional left argument that must be a character list of names. If provided, the left argument limits the scope of TIME to the programs named in the left argument.

Additional uses of the TIME function are:

TIME 3
: Fetch times for all lines of all programs even if no timing has been accumulated. The result is a matrix of the same columnar format as TIME 2 and is sequenced by line within each function or operator. A left argument can be used to obtain the timing for the specified programs. If column headings are desired, the following can be used:

```
HD3←'COUNT' 'TIME' '%' 'PROGRAM' 'STMT'
HD3,[⎕IO] TIME 3
```

TIME 4
: Fetch timing state. The result is a Boolean: 0 - timing off, 1 - timing on.

TIME ¯1
: Enable timing. If timing has been disabled, timing is resumed. A left argument is not allowed for TIME ¯1.

TIME ¯2
: Disable timing. Stops the accumulation of timing data. A left argument is not allowed for TIME ¯2.

TIME ¯3
: Deletes the space used by the counters. A name list left argument is allowed and can be used to delete the timing data for selected functions and operators.

Use of the timing facility increases space utilization and execution time. Reported timings are approximate and should only be used for relative comparisons, not absolute times.

When a program is erased, its counters are deleted. When a program is created or changed, its counters are not preserved.

Performance Analysis Using the TIME Function

The theory behind performance tuning through the TIME facility is that very often a small amount of code within an application consumes the majority of CPU cycles. By quickly identifying these "hot spots", the programmer can focus his attention on optimizations that provide the greatest effect in reducing overall application CPU time. The following APL2 session demonstrates the use of TIME:

```
      )LOAD COSTEST
SAVED 1993-09-19 14.13.20 COSTEST
      )PCOPY 1 TIME TIME   ⍝ Fetch TIME function
SAVED 1993-12-02 12.00.00 TIME
      TIME 0               ⍝ Zero time counters
      ESTIMATE 10          ⍝ Run the application
COMPLETED... SEE ¨COST_REPORT¨
      ⎕←T←TIME 1           ⍝ Fetch time summary
500 31.19    72.83469164 PRODCOST
500   7.4    17.28043341 CHARGE
 10 3.627    8.469747566 CALC
500 0.208   0.4857202905 EVAL
 10 0.196    0.457697966 PC
 10 0.103   0.2405249515 STORE
  1  0.05   0.1167596852 ESTIMATE
  1 0.044    0.102748523 TIME
```


© Copyright IBM Corporation 1994, 2017

```
 10 0.004 0.009340774817 FINDMAX
 10 0.001 0.002335193704 GETNEXT
  1     0              0 CLOSE
  1     0              0 OPEN
      +/T[;2]              ⍝ Compute total CPU time
42.823
      'PRODCOST' TIME 2    ⍝ Fetch time detail
500 31.028  99.48060276 PRODCOST[2] PCOST←SCHG∘.×⍳WEEKS
500  0.162 0.5193972427 PRODCOST[1] SCHG←CHARGE N
```

The `TIME 1` report identifies the function `PRODCOST` as the major CPU consumer, attributable to about 73% of the total CPU time (31.19 of the total CPU 42.823 seconds for this small sample run). Further analysis of the `PRODCOST` function with the `TIME 2` report shows that the mathematical calculation performed in line 2 is the best target for potential performance improvement.

© Copyright IBM Corporation 1994, 2017

UTILITY Workspace

This section describes the UTILITY workspace.

- Introduction
- GPDATACV: Converting Between External and Internal Representations
- GPMISC: Miscellaneous Utility Functions
- GPSTRIP: Removing Comments
- GPSVP: Controlling Communication through SVP
- GPTEXT: Manipulating Text
- GPTRACE: Setting and Removing Trace and Stop Vectors
- GPXLATE: Translating from One Character Representation to Another

Introduction

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

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

- Functional
- Instructional

The functional use is relatively straightforward:

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

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

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

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

GPDATACV: Converting Between External and Internal Representations

- LI/LO - Boolean (Logical)
- II/IO - System/370 Integer
- PCII/PCIO - IBM PC Integer
- FI/FO - System/370 Floating Point
- IEEEFI/IEEEFO - IEEE Floating Point
- PCFI/PCFO - IBM PC Floating Point

© Copyright IBM Corporation 1994, 2017

- PDI/PDO - Packed Decimal

LI/LO - Boolean (Logical)

```
Z←LI R ⍝ Logical In
```

R is a simple character array whose last axis contains IBM PC or System/370 logical data; that is, a string of bits.

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

```
⍴Z ←→ (¯1↓⍴R),8×¯1↑1,⍴R
```

```
Z←LO R ⍝ Logical Out
```

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

Z is a character array whose last axis contains the IBM PC or System/370 representation of the logical data in the last axis of R. The rank of Z is the same as the rank of R, but the length of the last axis of Z is one-eighth of the last axis of R.

```
⍴Z ←→ (¯1↓⍴R),(¯1↑⍴R)÷8
```

II/IO - System/370 Integer

```
Z←II R ⍝ Integers In
```

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

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

```
⍴Z ←→ ¯1↓⍴R
```

```
Z←L IO R ⍝ Integers Out
```

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

© Copyright IBM Corporation 1994, 2017

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

```
ρZ ↔ (ρR),L
```

PCII/PCIO - IBM PC Integer

```
Z←PCII R ⍝ PC Integers In
```

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

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

```
ρZ ↔ ¯1↓ρR
```

```
Z←L PCIO R ⍝ PC Integers Out
```

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

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

```
ρZ ↔ (ρR),L
```

FI/FO - System/370 Floating Point

```
Z←FI R ⍝ Floating In
```

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

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

```
ρZ ↔ ¯1↓ρR
```

```
Z←FO R ⍝ Floating Out
```


© Copyright IBM Corporation 1994, 2017

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

```
ρZ ←→ (ρR),8
```

IEEEFI/IEEEFO - IEEE Floating Point

```
Z←IEEEFI R ⍝ IEEE Floating In
```

R is a simple character array, the last axis of which must have a length of 4 or 8. The last axis thus represents either single- or double-precision IEEE floating-point numbers.

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

```
ρZ ←→ ¯1↓ρR
```

```
Z←L IEEEFO R ⍝ IEEE Floating Out
```

R is a simple numeric array. L is an integer scalar with a value of 4 or 8. A value of 4 for L gives single-precision floating point representation, and a value of 8 gives double precision. Z is a character array whose last axis has length L, and which contains the IEEE single or double-precision floating-point representations of the numbers in R. The rank of Z is one greater than the rank of R.

```
ρZ ←→ (ρR),L
```

PCFI/PCFO - IBM PC Floating Point

```
Z←PCFI R ⍝ PC Floating In
```

R is a simple character array, the last axis of which must have a length of 4 or 8. The last axis thus represents either single- or double- precision IBM PC (reversed) IEEE floating-point numbers.

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

```
ρZ ←→ ¯1↓ρR
```

```
Z←L PCFO R ⍝ PC Floating Out
```


© Copyright IBM Corporation 1994, 2017

R is a simple numeric array. L is an integer scalar with a value of 4 or 8. A value of 4 for L gives single precision floating point representation, and a value of 8 gives double precision. Z is a character array whose last axis has length L, and which contains the IBM PC (reversed) IEEE single- or double-precision floating-point representations of the numbers in R. The rank of Z is one greater than the rank of R.

```
ρZ ←→ (ρR),L
```

PDI/PDO - Packed Decimal

```
Z←PDI R ⍝ Packed Decimal In
```

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

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

```
ρZ ←→ ¯1↓ρR
```

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

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

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

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

```
ρZ ←→ (ρR),L
```

GPMISC: Miscellaneous Utility Functions

- ANNOTATE - Add comments to lines
- ASSIGN - Specify values for a set of names
- CASE - Gives case attribute of active workspace
- CODECOUNT - Count lines in all workspace programs
- CONCEAL - Make a function non-suspendable
- DATETIME - Format date and time
- EXPAND - Function version of \
- FNHEADS - List headers for a set of functions
- FRAME - Put a border around a character matrix
- HEXDUMP - Produce character+hex display of data
- LINECOUNT - Count lines in a list of programs

© Copyright IBM Corporation 1994, 2017

- LIST - Convert an arbitrary array to a vector
- MASKCONV - Splits numbers into n-bit fields
- MESH - Mesh two or more vectors as prescribed by a mask
- NAMEREFS - Find all names in a defined program
- NAMES - Find all names in a string
- NHEAD - Produce character representations of index vectors
- REPLICATE - Function version of /
- REVEAL - Make a function suspendable
- TYPE - Determine if array is alphabetic or numeric
- UNIQUE - Remove duplicates
- WSID - Return active workspace name

ANNOTATE - Add comments to lines

```
Z←L ANNOTATE R
```

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

ASSIGN - Specify values for a set of names

```
L ASSIGN R
```

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

CASE - Gives case attribute of active workspace

```
Z←CASE
```

`Z` is the case attribute of the active workspace. In workstation APL2 systems, `Z` always has a value of 2 to indicate that lowercase characters are used in object names.

CODECOUNT - Count lines in all workspace programs

```
Z←CODECOUNT
```

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

© Copyright IBM Corporation 1994, 2017

CONCEAL - Make a function non-suspendable

```
CONCEAL R
```

Make the function named by `R` non-suspendable.

DATETIME - Format date and time

```
Z←DATETIME
```

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

```
      DATETIME
06/27/85    10:00:42
```

**EXPAND - Function version of **

```
Z←L EXPAND R
```

`R` is any array. `L` is a Boolean vector. `Z` is `L\R`. See Miscellaneous Functions for a discussion of this function.

FNHEADS - List headers for a set of functions

```
Z←FNHEADS R ⍝ FuNction HEADerS
```

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

FRAME - Put a border around a character matrix

```
Z←FRAME R
```

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

HEXDUMP - Produce character+hex display of data

```
Z←HEXDUMP R
```


© Copyright IBM Corporation 1994, 2017

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

LINECOUNT - Count lines in a list of programs

```
Z←LINECOUNT R
```

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

LIST - Convert an arbitrary array to a vector

```
Z←LIST R
```

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

MASKCONV - Splits numbers into n-bit fields

```
Z←L MASKCONV R ⍝ MASK CONVert
```

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

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

MESH - Mesh two or more vectors as prescribed by a mask

```
Z←L MESH R
```

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

© Copyright IBM Corporation 1994, 2017

```
                                        00122233333333'
                                        ↓↓↓↓↓↓↓↓↓↓↓↓↓↓
      0 2 2 1 3 3 3 3 3 2 3 0 3 3 MESH 'HE IS WORDSMAN'
HIS WORDS MEAN
↑↑↑↑↑↑↑↑↑↑↑↑↑↑
02213333323033
```

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

NAMEREFS - Find all names in a defined program

```
Z←NAMEREFS R
```

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

NAMES - Find all names in a string

```
Z←NAMES R
```

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

NHEAD - Produce character representations of index vectors

```
Z←L NHEAD R ⍝ Numeric HEADers
```

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

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

REPLICATE - Function version of /

```
Z←L REPLICATE R
```

`R` is any array. `L` is a vector of integers. `Z` is `L/R`. See Miscellaneous Functions for a discussion of this function.

© Copyright IBM Corporation 1994, 2017

REVEAL - Make a function suspendable

```
REVEAL R
```

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

TYPE - Determine if array is alphabetic or numeric

```
Z←TYPE R
```

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

UNIQUE - Remove duplicates

```
Z←UNIQUE R
```

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

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

WSID - Return active workspace name

```
Z←WSID
```

`Z` is a character vector containing the active workspace name. In a clear workspace, a null character vector is returned.

This function is designed to work on all APL2 systems. It contains logic to determine the operating system, and use appropriate code for that operating system.

```
      WSID
1 UTILITY
```

GPSTRIP: Removing Comments

© Copyright IBM Corporation 1994, 2017

```
DECOMMENT
```

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

```
STRIP R
```

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

```
Z←L WORDS R
```

Splits a character vector into nested pieces, and is equivalent to the mainframe APL2 external function DAN (Delete And Nest). R is a character vector. L is a scalar or vector of delimiter characters. Z is a character vector, each of whose elements is a vector of the elements of R lying between occurrences of the delimiters L in R. Consecutive occurrences of delimiters in R are ignored. For example:

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

GPSVP: Controlling Communication through SVP

```
APSERVER R
```

A general AP server for implementing auxiliary processors using a client-server protocol over a single shared variable interface. For more information about APSERVER, see Writing Auxiliary Processors Using APSERVER.

© Copyright IBM Corporation 1994, 2017

```
Z←L ID R
```

Converts enclosed character processor IDs to large integers and vice versa. Typically used with the SVP profile (apl2svp.prf) in support of cross-system SVP shares for cooperative processing.

```
Z←L SVOFFER R
```

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

R is a character scalar, vector, matrix, or vector of vectors containing the name or names of the shared variables to be offered to an auxiliary processor. Surrogate names for shared variables can also be used. L is a numeric scalar or vector containing the processor ID (the number) of the AP. Z is the degree of coupling for the shared variable; a 2 indicates that the corresponding variable is fully shared with the AP.

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

```
Z←L SVOPAIR R
```

Offers shared variables, named in right argument, to SVP processors identified by numbers in the left argument. This function is used for auxiliary processors that support a two-variable interface, where the control variable begins with "C" or "CTL," and the data variable begins with "D" or "DAT" (that is, AP 124, AP 210).

```
Z←L SVRETRACT R
```

Retracts shared variable(s), named in the right argument, from processor identified by the number in the left argument. Waits for the partner to retract the variable so that subsequent offers of the same variable name will be considered as new offers rather than re-offers.

This function can be useful for cross-system shared variables and variables shared between two APL2 interpreter sessions on the same system. It is not needed when sharing with IBM-supplied auxiliary processors on the same system, as those auxiliary processors contain internal logic to correctly control the retract and re-offer process when the share is local.

R is a character scalar, vector, matrix, or vector of vectors containing the name or names of the shared variables to be retracted. L is a numeric scalar or vector containing the processor ID (the number) of the processor with which the variables are shared. Z is the degree of coupling before retract.

```
      100 SVRETRACT 'CMD'
2
```


© Copyright IBM Corporation 1994, 2017

```
      12345 SVRETRACT 2 2⍴'V1' 'V2'
2 2
      12345 SVRETRACT 'V1' 'V2'
2 2
```

```
Z←L ∆SVO R
```

∆SVO extension to support enclosed character vectors as processor IDs. Typically used with the SVP profile (apl2svp.prf) in support of cross-system SVP shares for cooperative processing.

```
Z←L ∆SVQ R
```

∆SVQ extension to support enclosed character vectors as processor IDs. Typically used with the SVP profile (apl2svp.prf) in support of cross-system SVP shares for cooperative processing.

GPTEXT: Manipulating Text

Note: Many "text" functions also work on other kinds of data.

- DOUBLE - Replace selected characters with pairs
- FIND - Search for text in all workspace programs
- GATHER - Collect parsed and delimited fields
- GVCAT - Catenate rows to arrays of any rank
- HCAT - Catenate matrixes by columns
- INBLANKS - Separate characters by blanks
- LADJ - Left adjust
- LINEFOLD - Fold and indent line as specified
- MAT - Make a matrix out of any array
- MATFOLD - Fold and indent matrix lines as specified
- NOQUOTES - Remove quoted substrings
- OBLANKS - Remove outer blanks
- QREPLACE - Replace ? occurrences by character strings
- RADJ - Right adjust
- RCNUM - Produce numerical headings for rows and columns
- REPLACE - Replace substrings in character strings
- RTBLANKS - Remove trailing blanks
- VCAT - Catenate matrixes by rows
- XBLANKS - Remove all excess blanks

DOUBLE - Replace selected characters with pairs

```
Z←L DOUBLE R
```

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

© Copyright IBM Corporation 1994, 2017

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

FIND - Search for text in all workspace programs

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

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

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

GATHER - Collect parsed and delimited fields

```
Z←L GATHER R
```

`L` is a scalar or one or two element vector, for example `'()'`. `R` is any array. `GATHER` searches the rows of `R` for a sequence "enclosed" within the first and second elements of `L` and ravels them into a vector. If `L` is a scalar or one element vector then it is used as the trailing delimiter also. A blank is inserted at each point where the resulting vector crosses a row boundary in `R`.

GVCAT - Catenate rows to arrays of any rank

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

`L` and `R` `Z` is the result of catenating `L` to `R` along the first coordinate of the array of higher rank.

HCAT - Catenate matrixes by columns

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

`HCAT` catenates columns: given two matrixes, it places them side-by-side. `L` and `R` should not be of rank greater than `2`. `Z` is always of rank `2`.

INBLANKS - Separate characters by blanks

```
Z←L INBLANKS R
```


© Copyright IBM Corporation 1994, 2017

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

LADJ - Left adjust

```
Z←LADJ R ⍝ Left ADJust
```

R can be any array. Z is that array with non-blank characters shifted to the left as far as possible.

LINEFOLD - Fold and indent line as specified

```
Z←L LINEFOLD R
```

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

MAT - Make a matrix out of any array

```
Z←MAT R ⍝ MATrix
```

Z is an array of rank 2 containing all the elements of R.

MATFOLD - Fold and indent matrix lines as specified

```
Z←L MATFOLD R ⍝ MATrix FOLD
```

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

NOQUOTES - Remove quoted substrings

```
Z←NOQUOTES R
```

R is a character vector. Z is the same vector with all quoted substrings removed.

OBLANKS - Remove outer blanks

© Copyright IBM Corporation 1994, 2017

```
Z←OBLANKS R ⍝ Outer BLANKS
```

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

QREPLACE - Replace ? occurrences by character strings

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

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

RADJ - Right adjust

```
Z←RADJ R ⍝ Right ADJust
```

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

RCNUM - Produce numerical headings for rows and columns

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

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

REPLACE - Replace substrings in character strings

```
Z←L REPLACE R
```

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

REPLACEV is a subfunction of REPLACE.

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


© Copyright IBM Corporation 1994, 2017

```
TEXT_
```

RTBLANKS - Remove trailing blanks

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

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

VCAT - Catenate matrixes by rows

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

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

XBLANKS - Remove all excess blanks

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

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

GPTRACE: Setting and Removing Trace and Stop Vectors

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

```
STOPALL
```

Creates stop vectors for all the lines of all the functions and operators in the active workspace.

```
STOPOFF
```

Cancels all the stop vectors in the active workspace.

```
STOPONE
```


© Copyright IBM Corporation 1994, 2017

Creates stop vectors for the first statements of all the functions and operators in the active workspace.

```
TRACEALL
```

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

```
TRACEBR R
```

Creates a trace vector for every branch statement of the function or operator named in `R`. `R` is a single name.

```
TRACELIST R
```

Creates trace vectors for all the statements in the functions and operators named in `R`. `R` is a simple scalar, vector, or matrix, or a vector of vectors.

```
TRACEOFF
```

Cancels all the trace vectors in the active workspace.

```
TRACEONE
```

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

GPXLATE: Translating from One Character Representation to Another

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

The third function, `TRANSLATE`, is a general-purpose translate function that requires a translate table as its left argument.

The following example demonstrates the use of the uppercase and lowercase translate functions:

```
      CV←'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
      UCTRANS CV
ABCDEFGHIJKLMNOPQRSTUVWXYZABCDEFGHIJKLMNOPQRSTUVWXYZ
      LCTRANS CV
abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz
```


© Copyright IBM Corporation 1994, 2017

Here is how the lowercase translate table was constructed:

```
(1)  ⎕IO←0
(2)  LOWERINDICES←⎕AF 'abcdefghijklmnopqrstuvwxyz'
(3)  UPPERINDICES←⎕AF 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
(4)  LCTt←⎕AV
(5)  LCTt[UPPERINDICES]←⎕AF LOWERINDICES
(6)  LOWERINDICES
     97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112
     113 114 115 116 117 118 119 120 121 122
(7)  UPPERINDICES
     65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84
     85 86 87 88 89 90
(8)  ⎕AF LCTt[UPPERINDICES]
     97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112
     113 114 115 116 117 118 119 120 121 122
(9)  ⎕CR 'LCTRANS'
     A←LCTRANS B
     ⍝ A is B with uppercase letters translated to lowercase letters.
     A←LCTt[⎕IO+⎕AF B]
```

Notes for the example:

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

(2) The indexes of the lowercase letters in `⎕AV` are determined by the use of `⎕AF`.

(3) Similarly for uppercase.

(4) The lowercase translate table is initialized as `⎕AV`.

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

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

(9) Is a canonical representation of the `LCTRANS` function, showing how it does lowercase translation.

© Copyright IBM Corporation 1994, 2017

WSCOMP Workspace

WSCOMP compares the contents of two workspaces or transfer files.

WSCOMP contains one external function: WSCOMP. Use this function to compare workspaces.

To use WSCOMP,

```
)LOAD your-workspace
)COPY 1 WSCOMP WSCOMP
WSCOMP
```

WSCOMP prompts for the following information:

- Two workspace names. Use the same syntax as you would for the system commands)IN or)COPY. Library numbers are accepted. If specifying a complete path and file name, you must enclose it in quotes.
- Whether the workspaces named are transfer files. This option controls whether)IN or)COPY is used to access the workspace.

On UNIX, the initial prompt will also ask you what name classes you want to compare.

A report is produced showing the results of the comparison.

On Unix systems, the report takes the form of four variables containing lists of names:

WSC_ONLY_1
: Objects in workspace 1 but not in workspace 2.

WSC_ONLY_2
: Objects in workspace 2 but not in workspace 1.

WSC_DIFF
: Objects with different definitions in the two workspaces.

WSC_SAME
: Objects with the same definition in both workspaces.

The variable WSCOMP_RESULTS is also set; it contains all four lists.

On Windows, a dialog is presented with the four lists of names. Additional features include the ability to open an objects' definitions and dynamically modify the lists to include or remove name classes.

© Copyright IBM Corporation 1994, 2017

APL2 Programming Interface (Calls to APL2)

The APL2 Programming Interface (APL2PI) enables APL2 to be called from other languages and programs.

APL2PI supports the following capabilities:

- Starting and stopping the APL2 interpreter
- Creation and management of workspace objects
- Execution of APL2 expressions and functions

APL2PI is useful for accessing APL2's array processing facilities from scalar programming languages. Using this technique can combine the performance benefits of compilation for scalar operations with the conciseness, productivity and performance of APL2 for array processing.

Using APL2PI, multiple APL2 interpreters can be run in the same process. This is useful for providing complete namescope isolation and, with some languages' multi-threading capabilities, asynchronous APL2 processing.

An APL2 interpreter running under APL2PI has the same restrictions as the APL2 Runtime Library. The development environment (session manager), interactive input, and library system commands are not supported. For more detail, see Restrictions of the Runtime Library.

Before using the APL2 Programming Interface, the APL2 system environment must be established. See the following section, Establishing the APL2 Environment for more information.

Support is included with APL2 for using APL2PI from the following languages and programs:

- Calling APL2 from APL2
  A Processor 11 external function or an auxiliary processor may be used to control additional APL2 sessions.
- Calling APL2 from C
  An include file contains the C prototype and constant definitions for the interface.
- Calling APL2 from Java
  A set of Java classes for APL2 are provided.
- Calling APL2 from Visual Basic
  A set of routines are provided for use as Visual Basic external procedures.

© Copyright IBM Corporation 1994, 2017

Establishing the APL2 Environment

Before using the APL2 Programming Interface, several environment variables must be set that enable APL2PI to locate APL2's components.

Establishing the APL2 Environment on Unix Systems

The APL2 Environment is established by running a shell script called `apl2env`. The `apl2env` script sets the environment variables required by the APL2 Programming Interface.

The `apl2env` script resides in the default installation directory. The installation of APL2 defines a symbolic link, `/usr/bin/apl2env`, that points to the `apl2env` shell script.

If you will be using a script to start the program that uses the APL2 Programming Interface, you may copy the appropriate commands from `apl2env` or you may call `apl2env` from your script.

Note:

The `apl2env` script must be run with . in order for its settings to affect the calling shell's environment. For example:

```
. apl2env
```

Establishing the APL2 Environment on Windows Systems

On Windows, the install program typically sets the appropriate environment variables.

If you choose not to have the APL2 install program modify your environmental settings, you need to make the following changes for the APL2 Programming Interface to run correctly. The examples assume a top-level directory of `C:\Program Files\IBMAPL2W\`.

1. Add the following directory to PATH: `C:\Program Files\IBMAPL2W\bin`
2. Set variable APL207FL to: `C:\Program Files\IBMAPL2W\fonts`
3. Set variable APLP11 to: `C:\Program Files\IBMAPL2W\bin\aplnm011.nam`
4. If you will using the Java interface, add the following file to CLASSPATH: `C:\Program Files\IBMAPL2W\bin\apl2.jar`

You can add environment settings in the **System** notebook of the **Control Panel**.

If you will be using a .BAT file to start the program that uses the APL2 Programming Interface, you may also set the environment variables in your .BAT file.

As an alternative to environment variables, the APL207FL and APLP11 values may be supplied in the [Invocation Options] section of the `apl2.ini` file, using keywords 207FL and P11.

© Copyright IBM Corporation 1994, 2017

For more information on how to set environment variables or create the `apl2.ini` file, see the section on customizing APL2 in the Installing and Customizing APL2 section for your operating system.

© Copyright IBM Corporation 1994, 2017

Calling APL2 from APL2

The interfaces to call APL2 from APL2 are provided in two different forms. External function `APL2PIA` provides a synchronous interface. Auxiliary processor AP 200 provides an asynchronous interface.

To use the external function, associate a name with `APL2PIA` and call it as if it were an APL function in your workspace.

```
      3 11 ⎕NA 'APL2PIA'
1
```

The `APL2PIA` function's right argument is a nested array containing a character command (service name) followed by additional parameters, as defined by the service. Its optional left argument is a Boolean indicator of how errors should be handled. If omitted or `0`, errors will be reported as regular APL2 errors and execution will be suspended on error. If `1`, errors will be reported as part of the result.

To use the auxiliary processor, share a variable with AP 200. Specify requests by assigning to the variable and retrieve results by referencing it.

```
      )COPY 1 UTILITY SVOFFER
      200 SVOFFER 'SV200'
2
```

AP 200 always returns a 2-item array where the first item is the AP return code and the second item is the service-defined result. The second item is null if the return code is non-zero or no result is defined for the service. For a complete list of return codes defined for AP 200, see AP 200 Return Codes.

The following sections discuss the services defined for calling APL2 from APL2.

- Controlling the APL2 Interpreter
- Creating and Managing Workspace Objects
- Executing Expressions and Functions.

The syntax diagrams at the beginning of each service's description show the `APL2PIA` external function in the top half of the box, with both forms of error handling, and AP 200 in the bottom half of the box. The examples all use `APL2PIA` with default error handling. For examples using AP 200, see AP 200 Commands.

Controlling the APL2 Interpreter

Before beginning to call APL2 from APL2, a separate APL2 interpreter session (known as a *slave* interpreter) must be started, and when finished with APL operations in the slave interpreter, it should be terminated. Once the slave interpreter is started, the APL2 environment there is available for APL operations. The slave interpreter runs under the same restrictions as the APL2 Runtime Library. The development environment (session manager), interactive input, and library system commands are not supported. For more detail, see Restrictions of the Runtime Library.

The following set of services is provided for controlling the slave interpreter:

© Copyright IBM Corporation 1994, 2017

- START - Starting the Interpreter
- STOP - Stopping the Interpreter

Multiple instances of APL2 may be started. When using the APL2PIA function, each initialization request returns a unique identifier for that instance. The identifier must be used on all subsequent requests for that instance. When using AP 200, each shared variable controls one instance of the interpreter.

START - Starting the Interpreter

```
itoken ← APL2PIA 'START' [ options ]
(codes ind itoken) ← 1 APL2PIA 'START' [ options ]
```

```
SV200 ← 'START' [ options ]
rc ← ↑SV200
```

Parameters:

`options` APL2 invocation parameters, passed as a vector of character vectors with one element for each blank-delimited item in a command line parameter list. Any APL2 invocation parameters may be passed, as defined in Invoking APL2. However, the following parameters will be ignored if specified:

```
-hostwin
-input
-lx (is always OFF)
-quiet (is always ON)
-run
-rns
-sm (is always OFF)
-svplisten
-svptrace
```

The shared variable processor options `-svplisten` and `-svptrace` can be enabled by passing the parameters to the master APL2 session or by setting environment variables APLSVPLISTEN and APLSVPTRACE before starting APL2. On Windows, the menu options in The SVP Monitor Facility or the SVPLISTEN and SVPTRACE keywords in the [Invocation Options] section of the `apl2.ini` configuration file may also be used.

To accept the default invocation settings for APL2, omit `options` or pass any null array for `options`.

Results:

`itoken` Interpreter instance identifier. This identifier must be passed on all subsequent calls to `APL2PIA` for this instance.

`codes` Error codes (`⎕ET`)

`ind` Boolean indicator of whether a result or error message text was created. If `1`, the third item of

© Copyright IBM Corporation 1994, 2017

	the result contains the result or error message text. If `0`, the third item of the result is empty.
`rc`	AP 200 return code.

Examples:

```
SESS1 ← APL2PIA 'START' ('-ws' '50m' '-id' '12000')
SESS2 ← APL2PIA 'START' ''
```

STOP - Stopping the Interpreter

```
APL2PIA 'STOP' itoken
(codes ind msg) ← 1 APL2PIA 'STOP' itoken
```

```
SV200 ← 'STOP'
rc ← ↑SV200
```

Parameters:

`itoken`	The interpreter instance identifier as returned by `START`.

Results:

`codes`	Error codes (`⎕ET`)
`ind`	Boolean indicator of whether a result or error message text was created. If `1`, the third item of the result contains the result or error message text. If `0`, the third item of the result is empty.
`msg`	Error message text or null.
`rc`	AP 200 return code.

Example:

```
APL2PIA 'STOP' SESS2
```

Creating and Managing Workspace Objects

Named objects can be created in the slave interpreter's workspace by executing APL expressions in that workspace which establish objects using assignment, `⎕FX`, `⎕TF` or `⎕NA`. Once the objects are established, they can then be referred to by name when executing other expressions.

Unnamed objects are also created in the slave interpreter's workspace, during the processing of APL expressions and functions via the `EXECUTE` and `EXTOKEN` services.

In many circumstances, the creation and management of unnamed objects can be left to be automatically handled by the interface code (`APL2PIA` function or AP 200). APL2 arrays can be passed directly to the

© Copyright IBM Corporation 1994, 2017

EXECUTE service for use as APL expressions, function names, and arguments to functions. These arrays will be moved to the slave interpreter's workspace as needed and any results of the functions and expressions will be automatically retrieved.

When the unnamed arrays are handled automatically, however, they only reside in the slave workspace temporarily, and there is a cost to establishing these objects on each call to the EXECUTE service. In some circumstances, particularly when certain arrays will be used repeatedly, the user may want to manage the unnamed arrays directly, to obtain the best performance possible. For these circumstances, a set of services are provided to manage unnamed arrays. The EXTOKEN service executes expressions and functions using this type of array.

The following set of services is provided for managing APL2 objects in the workspace:

- PUT - Establish an Array in the Workspace
- GET - Obtain the Value of an Array from the Workspace
- FREE - Remove an Array from the Workspace

PUT - Establish an Array in the Workspace

```
atoken ← APL2PIA 'PUT' itoken array
(codes ind atoken) ← 1 APL2PIA 'PUT' itoken array
```

```
SV200 ← 'PUT' array
(rc atoken) ← SV200
```

Parameters:

itoken	The interpreter instance identifier as returned by START.
array	The array to be established.

Results:

atoken	The locator token of the array. This token must be used on subsequent calls to the EXTOKEN, GET and FREE services to refer to the array.
codes	Error codes (⎕ET)
ind	Boolean indicator of whether a result or error message text was created. If 1, the third item of the result contains the result or error message text. If 0, the third item of the result is empty.
rc	AP 200 return code.

Examples:

```
LEFTID←APL2PIA 'PUT' SESS1 'LEFT ARGUMENT DATA'
RIGHTID←APL2PIA 'PUT' SESS1 'RIGHT ARGUMENT DATA'
CATID←APL2PIA 'PUT' SESS1 ','
```


© Copyright IBM Corporation 1994, 2017

```
      SAMPID←APL2PIA 'PUT' SESS1 'SAMP'
```

GET - Obtain the Value of an Array from the Workspace

```
array ← APL2PIA 'GET' itoken atoken
(codes ind array) ← 1 APL2PIA 'GET' itoken atoken
```

```
SV200 ← 'GET' atoken
(rc array) ← SV200
```

Parameters:

itoken	The interpreter instance identifier as returned by START.
atoken	The locator token of the array to be referenced.

Results:

array	The value of the array
codes	Error codes (⎕ET)
ind	Boolean indicator of whether a result or error message text was created. If 1, the third item of the result contains the result or error message text. If 0, the third item of the result is empty.
rc	AP 200 return code.

Example:

```
      APL2PIA 'GET' SESS1 LEFTID
LEFT ARGUMENT DATA
```

FREE - Remove an Array from the Workspace

```
APL2PIA 'FREE' itoken atoken
(codes ind msg) ← 1 APL2PIA 'FREE' itoken atoken
```

```
SV200 ← 'FREE' atoken
rc ← ↑SV200
```

Parameters:

itoken	The interpreter instance identifier as returned by START.
atoken	The locator token of the array to be removed.

© Copyright IBM Corporation 1994, 2017

Results:

`codes`	Error codes (`⎕ET`)
`ind`	Boolean indicator of whether a result or error message text was created. If `1`, the third item of the result contains the result or error message text. If `0`, the third item of the result is empty.
`msg`	Error message text or null.
`rc`	AP 200 return code.

Example:

```
      APL2PIA 'FREE' SESS1 LEFTID
```

Executing Expressions and Functions

The `EXECUTE` and `EXTOKEN` services allow execution of expressions and functions in the APL2 session. The two services are identical except that `EXECUTE` is passed APL arrays directly for its last three arguments, and `EXTOKEN` is passed identifier tokens for arrays already established by `PUT`.

- EXECUTE - Execute Expression or Function
- EXTOKEN - Execute Using Array Tokens

Note:

Functions processed under control of these services operate in the same manner as those processed under control of `⎕EC`, and exhibit the following behavior:

- System commands are not allowed.
- Assignment expressions return a value.
- Requests for quad input are handled the same as quad input under `⎕EC`.
- Errors generated during processing do not cause suspension of the function being processed. Errors are percolated back to the caller.
- Branch escape (`→`) causes the service to return.
- Stop control vectors (`S∆`) are ignored.
- An attention signal does not cause suspension; an interrupt signal causes the service to halt and return control to the caller.

EXECUTE - Execute Expression or Function

```
[ result ← ] APL2PIA 'EXECUTE' itoken [left] expression [right]
(codes ind result) ← 1 APL2PIA 'EXECUTE' itoken [left] expression [right]
```

```
SV200 ← 'EXECUTE' [left] expression [right]
(rc result) ← SV200
(codes ind result) ← result
```


© Copyright IBM Corporation 1994, 2017

Parameters:

`itoken`	The interpreter instance identifier as returned by `START`.
`expression`	A character scalar or vector containing either a complete APL expression or the name of a function. If a function, it can be a defined function, system function, primitive function, or the assignment arrow. If the function is the assignment arrow, the left argument is the name of the object to be assigned, and the right argument is the value to be assigned to it.
`right`	The APL array to be used as the right argument to the specified function.
`left`	The APL array to be used as the left argument to the specified function.

Note that the expression or function name and arguments are all passed directly to this service. The corresponding objects for these parameters are automatically created in the slave workspace and the result, if any, is automatically retrieved. Any objects handled automatically are deleted after the operation is complete.

Results:

`codes`	Error codes (`⎕ET`)
`ind`	Boolean indicator of whether a result was created.
`result`	If `codes` is `0 0` and `ind` is `1`, the result from execution of the function or expression. If `codes` is not `0 0` and `ind` is `1`, a character matrix containing the error message and execution stack at the time of the error. Otherwise, a null character vector.
`rc`	AP 200 return code.

Examples:

```
      APL2PIA 'EXECUTE' SESS1 (1.1 2.2 3.3) '+' 100
101.1 102.2 103.3
      APL2PIA 'EXECUTE' SESS1 '⎕FX'('Z←L SAMP R' 'Z←L R')
SAMP
      APL2PIA 'EXECUTE' SESS1 (1.1 2.2 3.3) 'SAMP' ('ABC')
 1.1 2.2 3.3  ABC
```

EXTOKEN - Execute Using Array Tokens

```
[ rtoken ← ] APL2PIA 'EXTOKEN' itoken [ltoken] etoken [rtoken]
(codes ind rtoken) ← 1 APL2PIA 'EXTOKEN' itoken [ltoken] etoken [rtoken]
```

```
SV200 ← 'EXTOKEN' [ltoken] etoken [rtoken]
(rc result) ← SV200
(codes ind rtoken) ← result
```

Parameters:

© Copyright IBM Corporation 1994, 2017

`itoken`	The interpreter instance identifier as returned by `START`.
`etoken`	The identifier token of a character scalar or vector in the slave workspace containing either a complete APL expression or the name of a function. If a function, it can be a defined function, system function, primitive function, or the assignment arrow. If the function is the assignment arrow, the left argument is the name of the object to be assigned, and the right argument is the value to be assigned to it.
`rtoken`	The identifier token of an APL array in the slave workspace to be used as the right argument to the specified function.
`ltoken`	The identifier token of an APL array in the slave workspace to be used as the left argument to the specified function.

Note that the expression or function name and arguments are all passed to this service as locator tokens. This means that these objects must exist in the workspace prior to calling the service. If executing an expression, there must be a character object containing the expression. If executing a function, there must be a character object containing the function name. These objects can be created in the workspace by use of the `PUT` service. See Creating and Managing Workspace Objects for more information.

Results:

`codes`	Error codes (`⎕ET`)
`ind`	Boolean indicator of whether a result was created.
`rtoken`	If `codes` is `0 0` and `ind` is `1`, the locater token of the result from execution of the function or expression. If `codes` is not `0 0` and `ind` is `1`, the locator token of a character matrix containing the error message and execution stack at the time of the error. The `GET` service can be used to retrieve the value of the array. Otherwise, a null character vector.
`rc`	AP 200 return code.

Example:

```
      APL2PIA 'EXTOKEN' SESS1 LEFTID CATID RIGHTID
11
      APL2PIA 'GET' SESS1 11
LEFT ARGUMENT DATA RIGHT ARGUMENT DATA
```


© Copyright IBM Corporation 1994, 2017

Calling APL2 from C

The APL2 Programming Interface for C is a routine entry point named `apl2pi` in the APL2 shared library (on Windows, `apl2lib.dll`; on Unix systems, `libapl2.so`.)

```
int _System apl2pi(CALL *);
```

The prototype for this routine, and associated structure and constant definitions, are found in the include file `aplfun.h` in the `include` subdirectory of the APL2 main directory. On Windows, the library file `apl2lib.lib` required to link the routine to your C program is found in the `lib` subdirectory.

The `apl2pi` routine takes a single argument, a pointer to a `CALL` structure. A set of services are defined for use by callers of APL2. Before calling the `apl2pi` routine, the fields of this structure must be filled in appropriately for the type of service being requested. Results are returned in `CALL` structure fields, as defined for the service.

Note: The `CALL` structure used for this interface is the same structure used for external routines called by APL2. Some fields of the structure are not used when calling `apl2pi`. All unused fields should be initialized to 0 to avoid confusion between the two interfaces.

Before beginning to use APL2 the interpreter must be started, and when finished with APL2 operations the interpreter should be terminated. Once the interpreter is started, the APL2 environment is available for APL operations. The interpreter runs under the same restrictions as the APL2 Runtime Library. The development environment (session manager), interactive input, and library system commands are not supported. For more detail, see Restrictions of the Runtime Library.

The following sections discuss the services defined for `apl2pi`:

- Controlling the APL2 Interpreter
- Creating and Managing Workspace Objects
- Executing Expressions and Functions.
- Miscellaneous Services

The following section discusses handling exceptions in C and APL2.

- Handling Exceptions under C

A sample program which demonstrates how to call APL2 from C is shipped as `callapl2.c`, in the `samples` (Windows) or `examples/qna` (Unix) subdirectory of the APL2 product directory..

Controlling the APL2 Interpreter

The following set of services is provided for controlling the interpreter:

- APL2PI_INIT - Starting the Interpreter
- APL2PI_TERM - Stopping the Interpreter

© Copyright IBM Corporation 1994, 2017

Multiple instances of APL2 may be started. Each initialization request returns a unique identifier for that instance. The identifier must be used on all subsequent requests for that instance.

APL2PI_INIT - Starting the Interpreter

Parameters:

`->request` `0 (APL2PI_INIT)`

`->parm1` A caller anchor word. This can be any value as defined by the caller to uniquely identify itself. The value will be saved in the interpreter environment and passed in field `->parm2` on every call to an external name. An associated processor or external routine can examine this word to identify the caller of the APL2 environment. If no anchor is needed, this word must be set to 0.

`->parm2` The APL2 invocation parameter count (corresponds to the standard C *argc*), or 0 to accept the APL2 default invocation option values for the runtime environment.

`->parm3` The APL2 invocation parameter array, in standard C *argv* format, or NULL to accept the APL2 default invocation option values for the runtime environment. Any APL2 invocation parameters may be passed, as defined in Invoking APL2. However, the following parameters will be ignored if specified:

```
-hostwin
-input
-lx (is always OFF)
-quiet (is always ON)
-run
-rns
-sm (is always OFF)
-svplisten
-svptrace
```

The shared variable processor options `-svplisten` and `-svptrace` can be enabled by setting environment variables APLSVPLISTEN and APLSVPTRACE before starting APL2. On Windows, the menu options in The SVP Monitor Facility or the SVPLISTEN and SVPTRACE keywords in the [Invocation Options] section of the `apl2.ini` configuration file may also be used.

Results:

`->request` return code - one of:

```
0 (MSG_OK) - if the request was successful.
2 (MSG_SYSTEMLIMIT) - if APL2 could not be initialized.
```

The same code will also be returned as the result of `apl2pi`.

`->reason` reason code, if `->request` is `MSG_SYSTEMLIMIT`:

```
5 (ET_INTERFACENA) - if the APL2 message tables could not be found,
the APL2 shared variable processor would not start, or there was
insufficient storage available to allocate the required work areas.
Additional messages indicating the source of the problem may be printed
in the program console window.
6 (ET_INTERFACEQUOTA) - if the APL2 interpreter is time-limited and has
expired.
```

`->token` the interpreter instance identifier. This identifier must be passed in the `->token` field

© Copyright IBM Corporation 1994, 2017

on all subsequent calls to `apl2pi` for this instance.

`->reloc_count` the workspace relocation counter. This value can be tested after each call to `apl2pi` to determine when workspace garbage collection has taken place.

Example:

```
static char *argv[2] = {"-ws","20m"};
long APL2_inst1;
memset(call, 0x00, sizeof(CALL));
call->request = APL2PI_INIT;
call->parm1 = 0;
call->parm2 = 2;
call->parm3 = (long)argv;
if (MSG_OK != apl2pi(call) {
   printf("APL2 initialization failed: %i %i\n",call->request,call->reason);
   return(1);
}
APL2_inst1 = call->token;
```

APL2PI_TERM - Stopping the Interpreter

Parameters:

`->request` `255 (APL2PI_TERM)`

`->token` The interpreter instance identifier as returned by `APL2PI_INIT`.

Results:

`->request` `0 (MSG_OK)`

Example:

```
call->request = APL2PI_TERM;
call->token = APL2_inst1;
apl2pi(call);
```

Creating and Managing Workspace Objects

The following set of services is provided for managing APL2 objects in the workspace:

- ARRAYSPACE - Allocate an Array in the Workspace
- ARRAYRESIZE - Change the Size of an Allocated Array
- ARRAYREF - Reference an Array in the Workspace
- ARRAYCONVERT - Convert an Array to a New Type
- FREESPACE - Remove a Reference to an Array
- TOKEN_TO_ADDRESS - Obtain the Address of an Array
- LCDR_TO_ARRAY - Convert a Linear CDR to an Array
- ARRAY_TO_LCDR - Convert an Array to a Linear CDR

Locating APL2 Objects

© Copyright IBM Corporation 1994, 2017

Each APL2 object has a unique identifier, called the *locator token* or just *token*. The token is the means by which the interpreter can always locate the object, regardless of whether it has moved within the workspace.

At any given time, each APL2 object resides at a specific *address* in the user machine. A caller must use the address to actually access the contents of the object. The address may change when the interpreter storage management routines move objects within the workspace. The caller needs to be aware of when that happens, and obtain new addresses for objects when necessary.

When the caller uses the ARRAYSPACE, ARRAYCONVERT or LCDR_TO_ARRAY services to allocate storage, or the ARRAYRESIZE service to lengthen an array, objects in the workspace can move during the time the interpreter has control to allocate the storage. Also, during execution of APL2 expressions (EXECUTE_APL), allocation of temporary objects and garbage collection can affect object addresses.

Upon return from these services, the caller can test to see whether it is necessary to refresh object addresses. The `->reloc_count` field in the `CALL` block can be used to do this. Before calling `apl2pi`, save the value from `->reloc_count`. If the value has changed when the service returns, objects in the workspace have moved, and all addresses of objects stored locally within the caller's program must be refreshed.

To obtain new addresses for workspace objects, use the TOKEN_TO_ADDRESS service.

Processing APL2 Objects

Each APL2 workspace object contains information that describes its data type, shape, size, and origin. This information is called its header, and is located at the beginning of the object. The header consists of the following fields (defined in C include file `aplobj.h`):

```
typedef struct aplobj {
  unsigned long  ptr      ;
  unsigned long  nb       ;
  unsigned long  nelm     ;
  unsigned char  type     ;
  unsigned char  rank     ;
  unsigned char  fill[2]  ;
  unsigned long  dim[1]   ;
} APLOBJ ;
```

`ptr` The locator token of the object.

`nb` The number of bytes in this APL2 object. If the datatype of this object is that of a nested array the byte count includes only this object - it does not include the subitems. (Note: this is different from the APL2 objects passed to auxiliary processors.) The length of each object must be rounded to an even multiple of 16.

`nelm` The number of elements in this APL2 object.

`type` APL2 object type:

```
0  (BOOLEAN)    Boolean             1 bit per item
1  (INTEGER)    Integer             4 bytes per item
2  (FLOAT)      Real                8 bytes per item
3  (COMPLEX)    Complex             16 bytes per item
4  (CHARACTER)  ASCII Character     1 byte per item
5  (CHARLONG)   Extended Character  4 bytes per item
6  (APV)        Progression Vector  8 bytes
7  (NESTED)     General Array       4 bytes per item
```

`rank` Rank of object (0-64).

© Copyright IBM Corporation 1994, 2017

`fill` Unused; should be set to 0.

`dim[]` Length of each dimension (number of elements in dim = rank)

Immediately following the header for each object is the data associated with the object. The length of the data for each type is shown above. If the object has more than one dimension, its elements are stored in row order (as if the APL2 primitive Ravel had been applied to the variable).

Immediately following the data are enough fill bytes to make the length of the object an even multiple of 16.

If the array is a general array, the data consists of a set of locator tokens - one for each subitem of the array. If the general array is null, the data consists of one locator token - for an array which is the prototype for the general array. To obtain the addresses for the sub-items of a nested array, use the TOKEN_TO_ADDRESS service.

Building APL2 Objects

There are two ways that a routine can create an APL2 object. One method is to allocate the array (and its subitems if general) directly in the workspace with the ARRAYSPACE service and build the object there. The other is to build the object in local storage within your program, and when the array is complete, use the LCDR_TO_ARRAY service to place the array into the workspace.

In general, building the array in the workspace is more efficient. The array contents do not have to be copied into the workspace, they will be placed directly there. However, there are several reasons that you may want to build the object in local storage first.

1. You may be in a situation (such as reading data from a pipe or port) where you do not know in advance how large the array will be. Allocating an array much larger than necessary could result in WS FULL.
2. You may be in situation (such as fetching data from a database) where you need to rearrange or compress the data after obtaining it to make the APL2 object.
3. When you have a nested array, each item and subitem must be allocated separately with a call to ARRAYSPACE. If any of the calls fails (usually because of WS FULL) the items which were allocated successfully must be freed, and depending on your routine, additional cleanup may be necessary. When you build the array locally, there is a single call to LCDR_TO_ARRAY.
4. There is an additional degree of safety in this method. When you build an array directly in the workspace, no validation of your object is done. If you make an error, you can potentially corrupt the workspace. When you build the array locally and use LCDR_TO_ARRAY, a complete validation of your object is performed before copying it to the workspace. If there are errors in the object, control will be returned to your routine, which can then perform any desired cleanup before returning an error.

When you build a result array in local storage to pass to the LCDR_TO_ARRAY service, the contents of some of the fields on the `APLOBJ` structure are different from that of workspace objects. The object format used is the same format as is used by auxiliary processors, called *Common Data Representation* (CDR). The differences between the workspace object format and CDR format are:

1. The `->ptr` field of a workspace object contains its locator token. The `->ptr` field of a CDR contains a platform identifier.

© Copyright IBM Corporation 1994, 2017

2. For general arrays, the data section of a workspace object contains the locator tokens of the subarrays, which are separate objects. The data section of a general array CDR contains the offsets to the subarrays, which must follow the general array CDR in left-list order.
3. For general arrays, the `->nb` field of a workspace object is the number of bytes in the general array only. The `->nb` field of a general array CDR is the total number of bytes in the general array and all of its subitems.

For complete information on building a CDR, see Common Data Representation.

ARRAYSPACE - Allocate an Array in the Workspace

Parameters:

`->request` `21 (ARRAYSPACE)`

`->token` The interpreter instance identifier as returned by `APL2PI_INIT`.

`->parm1` number of elements

`->parm2` rank

`->parm3` type, one of:

```
0  (BOOLEAN)
1  (INTEGER)
2  (FLOAT)
3  (COMPLEX)
4  (CHARACTER)
5  (CHARLONG)
6  (APV)
7  (NESTED)
```

Results:

`->request` return code - one of:

```
0 (MSG_OK) - if the request was successful.
2 (MSG_SYSTEMERROR) - if an invalid type is passed.
3 (MSG_WSFULL) - if there is no room in the workspace for this object.
4 (MSG_SYSTEMLIMIT) - if rank or number of elements is out of range.
```

`->reason` reason code, if `->request` is `MSG_SYSTEMLIMIT`:

```
8 (ET_ARRAYRANK) - if the rank requested was greater than 64.
9 (ET_ARRAYSIZE) - if the number of elements and type caused the
number of bytes required for this array to exceed the largest possible
integer value.
```

`->parm1` the locator token of the array

`->parm2` the address of the start of the array (descriptor section)

`->parm3` the address of the start of the data section

Notes:

1. This service allocates an array in the workspace. It should be freed with FREESPACE when no longer needed.
2. The array descriptor fields are filled in for you, except where the array is rank 2 or higher. In that case, the `dim[]` fields are not filled in, and must be filled in before using the array in any further operations.

© Copyright IBM Corporation 1994, 2017

Example:

```
call->request = ARRAYSPACE;
call->token = APL2_inst1;
call->parm1 = 2;
call->parm2 = 1;
call->parm3 = INTEGER;
if (MSG_OK == apl2pi(call)) {
   longdata = (LONG *)call->parm3;
   longdata[0] = 0;
   longdata[1] = 75;
}
```

ARRAYRESIZE - Change the Size of an Allocated Array

Parameters:

`->request` `22 (ARRAYRESIZE)`

`->token` The interpreter instance identifier as returned by `APL2PI_INIT`.

`->parm1` the locator token of the array

`->parm2` the new number of elements

`->parm3` the new rank

Results:

`->request` return code - one of:

```
0 (MSG_OK) - if the request was successful.
2 (MSG_SYSTEMERROR) - if the locator token passed is invalid.
3 (MSG_WSFULL) - if there is no room in the workspace for this object.
4 (MSG_SYSTEMLIMIT) - if rank or number of elements is out of range.
```

`->reason` reason code, if `->request` is `MSG_SYSTEMLIMIT`:

```
8 (ET_ARRAYRANK) - if the rank requested was greater than 64.
9 (ET_ARRAYSIZE) - if the number of elements and type caused the
number of bytes required for this array to exceed the largest possible
integer value.
```

`->parm2` the address of the start of the array (descriptor section)

`->parm3` the address of the start of the data section

Notes:

1. If this service is used to shorten an array, MSG_WSFULL and garbage collection will not happen. The address of the start of the array will not move. The address of the data section will move only if the rank is changed. Any data already in the array will not be moved. If rank is changed the user is responsible for moving data accordingly.
2. If this service is used to lengthen an array, WS_FULL and garbage collection are possible, and the addresses of the start of the array and data section will move. The contents of the old array are copied to the new array, but adjustments are not made for any change in rank.
3. If the rank of the array is 2 or higher, the `dim[]` fields must be filled in before using the array in any further operations.

© Copyright IBM Corporation 1994, 2017

Example:

```
call->request = ARRAYRESIZE;
call->token = APL2_inst1;
call->parm1 = array_token;
call->parm2 = 3;
call->parm3 = 1;
if (MSG_OK == apl2pi(call)) {
   longdata = (LONG *)call->parm3;
   longdata[2] = 99;
}
```

ARRAYREF - Reference an Array in the Workspace

Parameters:

`->request`	`23 (ARRAYREF)`
`->token`	The interpreter instance identifier as returned by `APL2PI_INIT`.
`->parm1`	the locator token of the array

Results:

`->request`	return code - one of:

```
0 (MSG_OK) - if the request was successful.
2 (MSG_SYSTEMERROR) - if the locator token passed is invalid.
```

Notes:

1. This service would be used, for example, if an array is part of a nested array, and will be used more than once when building the nested array.
2. The FREESPACE service should be used to remove the reference when it is no longer needed.

Example:

```
call->request = ARRAYREF;
call->token = APL2_inst1;
call->parm1 = array_token;
if (MSG_OK != apl2pi(call)) {
   return 1;
}
```

ARRAYCONVERT - Convert an Array to a New Type

Parameters:

`->request`	`24 (ARRAYCONVERT)`
`->token`	The interpreter instance identifier as returned by `APL2PI_INIT`.
`->parm1`	the locator token of the array
`->parm2`	the new type, one of:

© Copyright IBM Corporation 1994, 2017

```
0  (BOOLEAN)
1  (INTEGER)
2  (FLOAT)
3  (COMPLEX)
4  (CHARACTER)
5  (CHARLONG)
6  (APV)
7  (NESTED)
```

Results:

`->request` return code - one of:

```
0 (MSG_OK) - if the request was successful.
2 (MSG_SYSTEMERROR) - if the locator token passed is invalid
or an invalid type is passed.
3 (MSG_WSFULL) - if there is no room in the workspace for this object.
4 (MSG_SYSTEMLIMIT) - if rank or number of elements is out of range.
11 (MSG_DOMAINERROR) - if the array cannot be converted to the new type.
```

`->reason` reason code, if `->request` is `MSG_SYSTEMLIMIT`:

```
8 (ET_ARRAYRANK) - if the rank requested was greater than 64.
9 (ET_ARRAYSIZE) - if the number of elements and type caused the
number of bytes required for this array to exceed the largest possible
integer value.
```

`->parm1` the locator token of the new array

`->parm2` the address of the start of the array (descriptor section)

`->parm3` the address of the start of the data section

Notes:

1. Not all type conversions are possible. For example, numeric arrays cannot be converted to character, integer arrays can only be converted to Boolean if all the values are 0 or 1, and most nested arrays cannot be converted to a non-nested type. This service makes validity checks to ensure that the data will be representable in the new type, and returns MSG_DOMAINERROR if the conversion is not possible.
2. This service allocates a new array. WS_FULL and garbage collection are possible. The old array is not deleted. It is still present, and it, along with any other workspace objects, can move during this call.
3. The array should be freed with FREESPACE when no longer needed.

Example:

```
call->request = ARRAYCONVERT;
call->token = APL2_inst1;
call->parm1 = array_token;
call->parm2 = INTEGER;
if (MSG_OK == apl2pi(call)) {
   longdata = (LONG *)call->parm3;
}
```

FREESPACE - Remove a Reference to an Array

Parameters:

```
->request  2 (FREESPACE)
```


© Copyright IBM Corporation 1994, 2017

->token The interpreter instance identifier as returned by APL2PI_INIT.

->parm1 the locator token of the array

Results:

->request return code - one of:

```
0 (MSG_OK) - if the request was successful.
2 (MSG_SYSTEMERROR) - if the locator token passed is invalid.
```

Notes:

1. This service removes a reference to an array. If the reference removed is the only remaining reference to that array, the array is then deleted, and the storage it occupied is freed for other uses.
2. This service is used to free arrays allocated with ARRAYSPACE, ARRAYCONVERT and LCDR_TO_ARRAY, and to remove references to arrays added by ARRAYREF.

Example:

```
call->request = FREESPACE;
call->token = APL2_inst1;
call->parm1 = array_token;
if (MSG_OK == apl2pi(call)) {
   array_token = 0;
}
```

TOKEN_TO_ADDRESS - Obtain the Address of an Array

Parameters:

->request 11 (TOKEN_TO_ADDRESS)

->token The interpreter instance identifier as returned by APL2PI_INIT.

->parm1 the locator token of the array

Results:

->request return code - one of:

```
0 (MSG_OK) - if the request was successful.
2 (MSG_SYSTEMERROR) - if the locator token passed is invalid.
```

->parm2 the address of the start of the array (descriptor section)

->parm3 the address of the start of the data section

Notes:

1. This service is used to obtain addresses for sub-items of nested arrays, and to re-address arrays after garbage collection has taken place.

Example:

© Copyright IBM Corporation 1994, 2017

```
call->request = TOKEN_TO_ADDRESS;
call->token = APL2_inst1;
call->parm1 = local_token;
if (MSG_OK == apl2pi(call)) {
   local_desc = call->parm2;
   local_data = call->parm3;
}
```

LCDR_TO_ARRAY - Convert a Linear CDR to an Array

Parameters:

`->request`	`18 (LCDR_TO_ARRAY)`
`->token`	The interpreter instance identifier as returned by `APL2PI_INIT`.
`->parm2`	the address of the buffer containing the CDR

Results:

`->request`	return code - one of:
	`0 (MSG_OK) - if the request was successful.`
	`2 (MSG_SYSTEMERROR) - if the CDR in the buffer is invalid.`
	`3 (MSG_WSFULL) - if there is no room in the workspace for this object.`

`->parm1`	the locator token of the allocated array

Notes:

1. This service allocates an array in the workspace. The array should be freed with FREESPACE when it is no longer needed.

Example:

```
call->request = LCDR_TO_ARRAY;
call->token = APL2_inst1;
call->parm2 = buffer;
if (MSG_OK == apl2pi(call)) {
   array_token = call->parm1;
}
```

ARRAY_TO_LCDR - Convert an Array to a Linear CDR

Parameters:

`->request`	`19 (ARRAY_TO_LCDR)`
`->token`	The interpreter instance identifier as returned by `APL2PI_INIT`.
`->parm1`	the locator token of the array to be converted
`->parm2`	the address of a buffer to receive the CDR. The first four bytes of the buffer must contain an integer which is the total length of the buffer.

Results:

© Copyright IBM Corporation 1994, 2017

`->request` return code:

`0 (MSG_OK) - the request was successful.`

`->parm2` the actual length of the CDR.
If the buffer was at least this length, the CDR has been built in the buffer.

Notes:

1. If you want to find out how large a buffer is needed to contain the CDR, call this service with the address of a 4-byte area containing the integer 4. On return, `->parm2` will contain the required buffer size.

Example:

```
call->request = ARRAY_TO_LCDR;
call->token = APL2_inst1;
call->parm2 = buffer;
(*(long *)buffer) = buffersize;
apl2pi(call);
if (call->parm2 > buffersize) {
   buffersize = call->parm2;
   buffer = malloc(buffersize);
   call->request = ARRAY_TO_LCDR;
   call->parm2 = buffer;
   (*(long *)buffer) = buffersize;
   apl2pi(call);
}
```

Executing Expressions and Functions

The `EXECUTE_APL` and `EXECUTE_APL_ESTACK` services allow execution of expressions and functions in the APL2 session. `EXECUTE_APL` and `EXECUTE_APL_ESTACK` take the same arguments and behave in the same manner. The difference between the two is the type of object returned in `->parm1` when an error occurs.

Parameters:

`->request` `5 (EXECUTE_APL)` or
`38 (EXECUTE_APL_ESTACK)`

`->token` The interpreter instance identifier as returned by `APL2PI_INIT`.

`->parm1` the locator token of the expression or function name. If a function, it can be a defined function, system function, primitive function, or the assignment arrow. If the function is the assignment arrow, the left argument is the name of the object to be assigned, and the right argument is the value to be assigned to it.

`->parm2` the locator token of the right argument, or 0 if none

`->parm3` the locator token of the left argument, or 0 if none

Note that the expression or function name and arguments are all passed to this service as locator tokens. This means that all required objects must exist in the workspace prior to calling the service. If executing an expression, there must be a character object containing the expression. If executing a function, there must be a

© Copyright IBM Corporation 1994, 2017

character object containing the function name as well as the function itself. Objects can be created in the workspace by several methods:

1. Use of the ARRAYSPACE or LCDR_TO_ARRAY services.
2. Executing expressions which access objects in APL2 namespaces using ⎕NA.
3. Executing expressions which establish objects using assignment, ⎕FX or ⎕TF.
4. Executing an assignment directly with this service.

See Creating and Managing Workspace Objects for more information.

Results:

`->request`	one of the message codes defined in Message Codes.
`->reason`	if applicable, one of the message type codes defined in Message Codes.
`->parm1`	If `->request` is `MSG_OK`: the locator token of the result, or 0 if none If `->request` is `MSG_APLERROR`: the locator token of the error message (`EXECUTE_APL`) or error stack (`EXECUTE_APL_ESTACK`)
`->parm2`	If `->request` is `MSG_OK`: the address of the start of the result (descriptor section) If `->request` is `MSG_APLERROR`: the APL event class (first element of ⎕ET)
`->parm3`	If `->request` is `MSG_OK`: the address of the start of the data section If `->request` is `MSG_APLERROR`: the APL event type (second element of ⎕ET)

Notes:

1. Functions processed under control of this interface operate in the same manner as those processed under control of ⎕EC, and exhibit the following behavior:
   - System commands are not allowed.
   - Assignment expressions return a value.
   - Requests for quad input are handled the same as quad input under ⎕EC.
   - Errors generated during processing do not cause suspension of the function being processed. Error codes are returned via the `CALL` structure.
   - Branch escape (→) causes EXECUTE_APL to return.
   - Stop control vectors (S∆) are ignored.
   - An attention signal does not cause suspension; an interrupt signal causes EXECUTE_APL to halt and return control to the caller.
2. If a result or error message is returned in `->parm1`, a new array has been allocated. The array should be freed with FREESPACE when no longer needed.
3. WS_FULL and garbage collection are possible during the course of APL operations, regardless of whether a result is returned. Any previously allocated arrays may move if garbage collection occurs.

Example:

```
/* Allocate an array for the expression */
call->request = ARRAYSPACE;
call->token = APL2_inst1;
call->parm1 = 3;
call->parm2 = 1;
call->parm3 = CHARACTER;
```


© Copyright IBM Corporation 1994, 2017

```
if (MSG_OK == apl2pi(call)) {
   exptoken = call->parm1;
   memcpy((char *)call->parm3,"⎕TS",3);
   /* Execute the expression */
   call->request = EXECUTE_APL;
   call->token = APL2_inst1;
   call->parm1 = exptoken;
   call->parm2 = 0;
   call->parm3 = 0;
   apl2pi(call);
   restoken = call->parm1;
   /* Free the expression array */
   call->request = FREESPACE;
   call->token = APL2_inst1;
   call->parm1 = exptoken;
   apl2pi(call);
}
```

Miscellaneous Services

The following additional services are available through the `apl2pi` interface:

- QUAD_CT - Obtain the Current Comparison Tolerance
- QUAD_IO - Obtain the Current Index Origin

QUAD_CT - Obtain the Current Comparison Tolerance

Parameters:

`->request` `40 (QUAD_CT)`

`->token` The interpreter instance identifier as returned by `APL2PI_INIT`.

Results:

`->request` return code - one of:

```
0 (MSG_OK) - if the request was successful.
2 (MSG_QUADERROR) - if ⎕CT is missing or invalid.
```

`->reason` reason code, if `->request` is `MSG_QUADERROR`:

```
3 (ET_CT)
```

`->parm1` the address of a double containing the current value of `⎕CT`

Example:

```
call->request = QUAD_CT;
if (MSG_OK == (call->service)(call)) {
   local_ct = *(double *)call->parm1;
}
```

QUAD_IO - Obtain the Current Index Origin

Parameters:

© Copyright IBM Corporation 1994, 2017

`->request` `15 (QUAD_IO)`

`->token` The interpreter instance identifier as returned by `APL2PI_INIT`.

Results:

`->request` return code - one of:

```
0 (MSG_OK) - if the request was successful.
2 (MSG_QUADERROR) - if □IO is missing or invalid.
```

`->reason` reason code, if `->request` is `MSG_QUADERROR`:

```
2 (ET_IO)
```

`->parm1` the current value of `□IO`

Example:

```
call->request = QUAD_IO;
if (MSG_OK == apl2pi(call)) {
   local_io = call->parm1;
}
```

Handling Exceptions under C

Each time you call the APL2 Programming Interface, APL2 uses the C runtime library signal facility to save the caller's handler for the SIGSEGV signal and then registers a new handler. On Unix systems, APL2 also saves the caller's handlers and registers new handlers for the following additional signals:

- SIGCHLD - Child process termination
- SIGHUP - Hang up
- SIGINT - Ctrl+C
- SIGQUIT - Dump request
- SIGTERM - Process termination

Before returning from each call to the APL2 Programming Interface, APL2 restores the caller's signal handlers.

When exceptions occur during a call to APL2, the C runtime library's signal facility normally catches the exceptions and passes control to APL2's signal handlers.

On Windows however, Microsoft Visual Studio's C and C++ __try and __except structured exception handling mechanism and C++ try and catch exception handling mechanism catch exceptions before they are passed to the signal facility. Therefore, if one of these exceptions occurs inside a __try or try block, the __except or catch block will receive control.

If you call the APL2 Programming Interface from within a __try or try block and an access violation occurs, APL2's signal handler will not receive control and the APL2 interpreter will be left in an unpredictable state.

If your __except or catch block receives control during a call to the APL2 Programming Interface, you should not attempt any further calls to APL2 other than APL2PI_TERM.

© Copyright IBM Corporation 1994, 2017

Calling APL2 from Java

APL2 applications can be called from Java. The APL2-Java interface supports the following features:

- Starting and Stopping APL2 Interpreters
- Creating and Deleting Workspace Objects
- Assigning, Associating, and Expunging Names
- Executing Expressions and Functions
- Using APL Characters in Java Programs
- Querying Workspace Object Attributes
- Retrieving Workspace Object Values
- Handling APL2 Errors
- Handling Exceptions under Java

An APL2 interpreter called from Java has the same restrictions as the APL2 Runtime Library. Primarily, the session manager and workspace system commands are not supported. For a complete description, see Restrictions of the Runtime Library.

This section concludes with several samples that illustrate how to use the APL2-Java interface:

- Calling APL2 from Java
- Calling APL2 from APL2
- Working with CDRs

Finally, the APL2-Java interface can be used to call APL2 from WebSphere. This topic is described in detail in *APL2 Programming: Using APL2 with WebSphere*.

Installing Java

The APL2-Java interface requires Java 2 Version 1.4 or later. However, APL2 does not include Java. You may have to install Java before you can use the APL2-Java interface.

Checking if Java is Installed

You can check whether an appropriate version of Java is already installed by entering the following command:

```
java -version
```

Notes:

> If more than one version of Java is installed, this command will tell you only about the one that is first in the current search order.
>
> Other installed products may include Java. You should not use a copy of Java that is included in another product. Copies of Java installed by other products may be customized for use by those products. You should install a stand-alone copy of Java for use with APL2.

Downloading Java

© Copyright IBM Corporation 1994, 2017

If you need to install Java, you can download Java installation programs for AIX, Linux, and Windows from the IBM developerWorks Java technology site, http://www.ibm.com/developerworks/java and for Linux, Solaris, and Windows from the Sun Developer Network, http://java.sun.com.

Several editions of Java are available including the Standard (SE), Enterprise (EE), and Micro Editions (ME). The IBM APL Products and Services group recommends the Standard Edition.

Within each edition, two Java packages are available for download:

- Java runtime environment (JRE)

  The JRE includes the files that are necessary to run Java programs.

- Java Development Kit (JDK)

  The JDK includes the files that are necessary to develop Java programs. The JDK also includes the JRE; if you install the JDK, you do not need to also install the JRE.

If you intend to develop Java programs, install the JDK. If you are only going to use Java programs developed by others, including running the APL2 sample programs and calling Java from APL2, you can install just the JRE.

Follow the instructions that come with the Java installation package that you download.

Adding the Java Virtual Machine to the Search Order

Once you have installed Java, you may still need to modify the system's search order. The JDK and JRE installation packages typically update the system's search order so you can run Java programs, (like the `java -version` command.) However, the installation packages do not always update the search order so other programs like APL2 can locate the Java Virtual Machine (VM) library. You may have to do this yourself.

Consult the instructions for your Java installation package to find out the name of the directory that contains the VM library.

Add the name of the directory containing the VM library to the system's search order. Add the name to the `LD_LIBRARY_PATH` environment variable on UNIX systems and the `PATH` environment variable on Windows

You can verify you have located the correct directory by searching for the VM library. The VM library has these names:

AIX	`libjvm.a`
Linux	`libjvm.so`
Solaris	`libjvm.so`
Windows	`jvm.dll`

Installing the APL2-Java Interface Classes

© Copyright IBM Corporation 1994, 2017

The APL2-Java interface includes several Java classes which manage the interface between Java and APL2. The APL2-Java interface classes are shipped in the `apl2.jar` file in the APL2 product's `bin` directory. The APL2 installation process (Windows) or the invocation script (Unix) may make them available to Java automatically by adding the jar file to the `CLASSPATH` environment variable. If you choose not to have the install program or invocation script modify your environment settings, you need to add `apl2.jar` to `CLASSPATH`. For example on Windows:

```
SET CLASSPATH=%CLASSPATH%;C:\Program Files\ibmapl2w\bin\apl2.jar
```

or on Unix:

```
export CLASSPATH=$CLASSPATH:/usr/APL2/bin/apl2.jar
```

In addition to the APL2-Java interface classes, the `apl2.jar` file contains two sample classes named Apl2demo and Apl2swt which are used by the `DEMO_JAVA` and `DEMO_SWT` functions in the library 2 DEMOJAVA workspace. The Apl2demo class and Apl2swt.class were compiled from the `Apl2demo.java` and `Apl2swt.java` files in the APL2 `samples` directory.

The APL2-Java Interface Classes

The APL2-Java interface includes the following classes for using APL2 from Java:

Apl2interp	Make APL2 interpreter requests
Apl2object	Manage APL2 workspace objects
Apl2exception	Indicate and detect APL2 errors
Apl2cdr	Convert between workspace and CDR formats

Starting and Stopping APL2 Interpreters

To start an APL2 interpreter, instantiate a new Apl2interp object:

```
Apl2interp Slave = new Apl2interp("-ws","20m");
```

The Apl2interp constructor parameter list is an array of character strings. The array is a list of pairs of values; each pair is an invocation option name followed by an invocation option value. If no parameters are supplied, the interpreter starts with the default invocation option values.

Any APL2 invocation parameters may be passed, as defined in Invoking APL2. However, these parameters will be ignored if specified:

- -hostwin
- -input
- -lx (is always OFF)
- -quiet (is always ON)
- -run
- -rns
- -sm (is always OFF)
- -svplisten

© Copyright IBM Corporation 1994, 2017

- -svptrace

The shared variable processor options -svplisten and -svptrace can be enabled by setting environment variables APLSVPLISTEN and APLSVPTRACE before starting APL2. On Windows, the menu options in The SVP Monitor Facility or the SVPLISTEN and SVPTRACE keywords in the [Invocation Options] section of the `apl2.ini` configuration file may also be used.

To stop an interpreter, call the Apl2interp class's Stop method:

```
Slave.Stop();
```

Once an interpreter has been stopped, any attempt to use the interpreter or any objects in the interpreter's workspace results in an error.

Creating and Deleting Workspace Objects

The Apl2object class is used to work with objects within interpreters' workspaces. To create a workspace object, instantiate a new Apl2object object:

```
Apl2object Array = new Apl2object(Slave,new int[] {1,2,3,4,5,6});
```

The Apl2object's constructors take two parameters: an Apl2interp object and an arbitrary Java value. The constructor copies the value into the APL2 workspace and returns an instance that can be used to refer to the workspace object.

To delete a workspace object, use the Free method:

```
Array.Free();
```

When a workspace object is created from a Java object, the Apl2object constructor normally converts the Java object's contents to an APL2 array. If you need to pass the object itself to APL2, use the global reference form of the Apl2object constructor.

Use global references like objects instantiated through associated processor 14. However, use the Apl2object's Free method to delete global references; do not use DeleteLocalRef.

This example demonstrates how to use the global reference form of the Apl2object constructor:

```
   Apl2interp Interp    = new Apl2interp() ;
   Apl2object GlobalRef = new Apl2object(Interp,Object,true) ;
   Interp.Execute("FUNCTION",GlobalRef) ;
   GlobalRef.Free() ;
   Interp.Stop() ;
[0]   FUNCTION GlobalRef;Class;ClassName
[1]   □ES(1≠3 14 □NA 'DeleteLocalRef')/1 5
[2]   Class←P14_CALL GlobalRef '()Ljava/lang/Class;' 'getClass'
[3]   ClassName←P14_CALL Class '()Ljava/lang/String;' 'getName'
[4]   DeleteLocalRef Class
```

Assigning, Associating, and Expunging Names

Use the Apl2interp class's Assign method to assign a name to an object:

© Copyright IBM Corporation 1994, 2017

```
Slave.Assign("NAME",Array);
```

The Assign method takes two parameters.

1. A String or an Apl2object instance containing the name to assign.
2. An Apl2object to which the name should be assigned.

Use the Apl2interp class's Associate method to associate a name:

```
Slave.Associate(3,11,"DISPLAY");
Slave.Associate("GRAPHPAK",11,"PLOT");
```

The Associate method takes three parameters.

1. Either an integer name class or a namespace locator string
2. An associated processor number
3. A string containing the name to be associated

Use the Apl2interp class's Expunge method to expunge a name:

```
Slave.Expunge("DISPLAY");
Slave.Expunge("PLOT");
```

The Expunge method's parameter is a string containing the name of the object to be expunged.

Executing Expressions and Functions

Use the Apl2interp class's Execute method to execute functions:

```
Apl2object Matrix = Slave.Execute("DISPLAY",Array);
```

The Execute method takes 1, 2, or 3 parameters.

- If 1 parameter is supplied, it may be either a string or an Apl2object that refers to a character vector in the workspace. The string or character vector is executed.
- If 2 parameters are supplied, the first may be either a string or an Apl2object that refers to a character vector in the workspace. The string or character vector is the name of a monadic function. The second parameter is an Apl2object instance. The function is executed and the second parameter is passed as the function's argument.
- If 3 parameters are supplied, the first parameter is an Apl2object instance. The second may be either a string or an Apl2object that refers to a character vector in the workspace. The string or character vector is the name of a dyadic function. The third is an Apl2object instance. The function is executed; the first parameter is passed as the function's left argument and the third parameter is passed as the right argument.

If execution of the function or expression returns a result, the Execute method returns an Apl2object. If execution of the function or expression does not return a result, the Execute method returns null.

Functions processed under control of the Execute method operate in the same manner as those processed under control of □EC, and exhibit the following behavior:

© Copyright IBM Corporation 1994, 2017

- Assignment expressions return a value.
- Requests for quad input are handled the same as quad input under ⎕EC.
- Errors generated during execution cause Execute to throw an Apl2exception.
- Branch escape (→) causes Execute to throw an Apl2exception.
- Stop control vectors (S∆) are ignored.
- Attention and interrupt signals do not occur.

Using APL Characters in Java Programs

The Apl2interp class includes static final character and string fields corresponding to the APL2 primitive symbols, system functions, and system variables.

Primitive Symbols

- Assignment
- Branch
- Ceiling
- Circle
- Comment
- Decode
- Del
- DelTilde
- Delta
- DeltaBar
- Diamond
- Divide
- Drop
- Each
- Enclose
- Encode
- Enlist
- Execute
- Find
- Floor
- Format
- GradeDown
- GradeUp
- GreaterEqual
- IBeam
- Index
- Interval
- Jot
- LessEqual
- Match
- MatrixInverse
- Multiply
- Nand
- NaturalLog
- Nor
- Not
- NotEqual
- Or
- Pick
- Quad
- QuoteQuad
- Reverse
- Rotate
- Shape
- SlashBar
- SlopeBar
- Take
- Transpose

System Variables

- QAI
- QAV
- QCT
- QEM
- QET
- QFC
- QIO
- QLC
- QLX
- QNLT
- QPP
- QPR
- QPW
- QRL
- QSVE
- QTC
- QTS
- QTZ
- QUL
- QWA

System Functions

- QAF
- QAT
- QCR
- QDL
- QEA
- QEC
- QES
- QEX
- QFX
- QIB
- QNA
- QNC
- QNL
- QPK
- QTF
- QSVC
- QSVO
- QSVQ
- QSVR
- QSVS
- QUCS

These fields are helpful for using APL2 characters in Java programs. For example,

```
Apl2object Definition = new Apl2object(Slave,new String[] {
  'Z' + Apl2interp.Assignment + "FOO",
  'Z' + Apl2interp.Interval + "10"}) ;
Apl2object Name = Slave.Execute(Apl2interp.QFX,Definition);
```

The corresponding APL2 code looks like this:

```
⎕FX 'Z←FOO' 'Z←⍳10'
```


© Copyright IBM Corporation 1994, 2017

Querying Workspace Object Attributes

Use the Apl2object class's Type, Rank, and Shape methods to query a workspace object's structure:

```
int Type = Matrix.type();
int Rank = Matrix.rank();
int[] Shape = Matrix.shape();
```

The Apl2object class includes the following static final int fields that correspond to the values returned by the Type method.

- `CDWRTB` - Boolean
- `CDWRTI` - Integer
- `CDWRTR` - Real
- `CDWRTJ` - Complex
- `CDWRTC` - Character
- `CDWRTD` - Character long (dbcs)
- `CDWRTA` - Integer Progression vector
- `CDWRTG` - General array

Retrieving Workspace Object Values

The Apl2object class includes the following methods for retrieving values from the workspace.

- `booleanValue`
- `byteValue`
- `charValue`
- `doubleValue`
- `floatValue`
- `intValue`
- `longValue`
- `shortValue`
- `stringValue`
- `booleanarrayValue`
- `bytearrayValue`
- `chararrayValue`
- `doublearrayValue`
- `floatarrayValue`
- `intarrayValue`
- `longarrayValue`
- `shortarrayValue`
- `stringarrayValue`

These methods return a Java value or array of values from an Apl2object. For example:

```
int[] IntArray = Array.intarrayValue();
```

If the APL2 workspace object cannot be converted to the specified Java type, an Apl2exception is thrown.

Handling APL2 Errors

© Copyright IBM Corporation 1994, 2017

When an error occurs in APL2, a Java Apl2exception is thrown. The Apl2exception class's Type and Code fields contain the elements of ⎕ET. The Stack field contains the APL2 stack at the time of the error as an array of strings.

The Apl2exception class includes the following static final fields that correspond to the values placed in the Type field:

- TYPE_DEFAULTS
- TYPE_RESOURCE
- TYPE_SYNTAX
- TYPE_VALUE
- TYPE_IMPLICIT
- TYPE_EXPLICIT

The following static final fields correspond to the values placed in the Code field:

- CODE_DEFAULTS_NOERROR
- CODE_DEFAULTS_UNCLASSIFIED

- CODE_RESOURCE_INTERRUPT
- CODE_RESOURCE_SYSTEMERROR
- CODE_RESOURCE_WSFULL
- CODE_RESOURCE_SYMBOLTABLE
- CODE_RESOURCE_INTERFACENA
- CODE_RESOURCE_INTERFACEQUOTA
- CODE_RESOURCE_INTERFACECAP
- CODE_RESOURCE_ARRAYRANK
- CODE_RESOURCE_ARRAYSIZE
- CODE_RESOURCE_DEPTH
- CODE_RESOURCE_PROMPTLENGTH
- CODE_RESOURCE_VALUEUNREP
- CODE_RESOURCE_RESTRICTION

- CODE_SYNTAX_NOARRAY
- CODE_SYNTAX_ILLFORMED
- CODE_SYNTAX_NAMECLASS
- CODE_SYNTAX_INVALIDOP
- CODE_SYNTAX_COMPATABILITY

- CODE_VALUE_NOVALUE
- CODE_VALUE_NORESULT

- CODE_IMPLICIT_PP
- CODE_IMPLICIT_IO
- CODE_IMPLICIT_CT
- CODE_IMPLICIT_FC
- CODE_IMPLICIT_RL
- CODE_IMPLICIT_PR

- CODE_EXPLICIT_VALENCE
- CODE_EXPLICIT_RANK
- CODE_EXPLICIT_LENGTH
- CODE_EXPLICIT_DOMAIN
- CODE_EXPLICIT_INDEX
- CODE_EXPLICIT_AXIS

Handling Exceptions under Java

When you start Java, Java registers handlers for a variety of signals. These handlers have priority over handlers registered by the APL2 Programming Interface.

If you call the APL2 Programming Interface from Java and an exception occurs for which Java has registered a handler, APL2 will not handle the exception and will not return. Java's handler will receive control and the APL2 interpreter will be left in an unpredictable state.

If your Java code catches an exception during a call to the APL2 Programming Interface, you should not attempt any further calls to APL2 other than APL2PI_TERM.

© Copyright IBM Corporation 1994, 2017

Calling APL2 from Java

Here is a complete Java program that calls APL2 and handles errors:

```
//* Import the apl2 package of classes
import com.ibm.apl2.*;
public class Sample
{
   public static void main(String[] args) {
      try {
         //* Create a slave interpreter
         Apl2interp Slave = new Apl2interp(new String[] {"-ws","1m"}) ;
         //* Reference QuadTS
         Apl2object Time = Slave.Execute(Apl2interp.QTS) ;
         //* Convert it to a Java integer array
         int[] TimeArray = Time.intarrayValue() ;
         //* Free the workspace object that resulted from the reference
         Time.Free();
         //* Stop the interpreter
         Slave.Stop();
         //* Print the time
         System.out.println("The time is " + TimeArray[4] + ':' + TimeArray[5] + '.') ;
      }
      catch (Apl2exception Exception) {
         System.out.println("Apl2exception caught.");
         System.out.println("Exception message: " + Exception.getMessage());
         System.out.println("Exception cause: " + Exception.getCause());
         System.out.println("Event: " + Exception.Type + " " + Exception.Code) ;
      }
   return ;
   }
}
```

Complete reference information about the APL2 Java classes can be found in the `javahtml` subdirectory of the APL2 documentation directory.

Calling APL2 from APL2

The APL2 Java interface can also be used as a means to control slave APL2 interpreters from APL2. Slave interpreters provide a convenient means to run utility applications with complete workspace isolation.

Using APL2's Processor 14, an Apl2interp constructor is used to start an APL2 interpreter. The class's other methods are used to associate and assign names and execute expressions in the slave interpreter. The following example illustrates how to use Java to drive a slave interpreter from APL2:

```
⍝ Associate a name with the Apl2interp class's constructor
□ES(1≠('com/ibm/apl2/Apl2interp' '()V')14 □NA 'Constructor <init>')/'Apl2interp
constructor not available'
⍝ Start a slave interpreter
Slave←Constructor
⍝ Associate a name with the interpreter's Execute method
□ES(1≠(Slave '(Ljava/lang/String;)Lcom/ibm/apl2/Apl2object;')14 □NA 'Execute')/'Execute
not available'
⍝ Execute an expression in the slave interpreter.  This returns an Apl2object
OBJECT←Execute,⊂'□TS'
⍝ Convert the Apl2object to an array
ARRAY←APL2OBJECT_TO_ARRAY OBJECT
⍝ Associate a name with the Apl2object's Free method
□ES(1≠(OBJECT '()V') 14 □NA 'Free')/'Free unavailable'
⍝ Delete the object from the slave interpreter's workspace
```


© Copyright IBM Corporation 1994, 2017

```
Free
⍝ Associate a name with the Processor 14 built-in DeleteLocalRef function
⎕ES(1≠3 14 ⎕NA 'DeleteLocalRef')/'DeleteLocalRef unavailable'
⍝ Delete the reference to the object
DeleteLocalRef OBJECT
⍝ Stop the slave interpreter
⎕ES(1≠(Slave '()V')14 ⎕NA 'Stop')/'Stop not available'
Stop
⍝ Delete the associations with Apl2interp instance methods
0 0⍴⎕EX⊃'Execute' 'stop'
⍝ Delete the Apl2interp object
Delete Slave
```

Working with CDRs

The Apl2interp and Apl2object classes are not serializable. That is, standard Java facilities cannot be used to flatten them to simple byte representations that can be written to IO streams. Apl2interp objects cannot be serialized because an APL2 interpreter is a dynamic environment containing a workspace and an execution state. Apl2object objects cannot be serialized because internally they contain references to Apl2interp objects and arrays in the interpreters' workspaces. However, sometimes it is desirable to serialize APL2 data and write it to a stream. For example, it might be desirable to write and read APL2 data to a file, or send it across a network or copy it between interpreters. The Apl2cdr class provides this capability. Unlike the Apl2interp and Apl2object classes, the Apl2cdr class is serializable.

The Apl2cdr class stores an APL2 array in Canonical Data Representation (CDR) format. The CDR is stored in the Apl2cdr object in a Java byte array. Apl2cdr objects can be constructed either using a CDR returned by the `ATR` external function or by using the Apl2object class's cdrValue method.

The Apl2cdr class has one method named `bytearrayValue`. The bytearrayValue method produces a byte array containing a CDR. The CDR can be used to create an array using the `RTA` external function.

The Apl2object class includes a constructor that takes an Apl2cdr object as a parameter. This constructor can be used to create an Apl2object from an Apl2cdr object.

Here is a Java program that uses the Apl2cdr class to save an APL2 array in a file:

```
//* Import the apl2 and java io class packages
import com.ibm.apl2.*;
import java.io.*;
public class DemoSerialize
{
   public static void main(String[] args)
      throws FileNotFoundException, IOException, ClassNotFoundException {
      try {
          //* Start a slave interpreter, create an object, and a CDR
          Apl2interp Slave = new Apl2interp() ;
          Apl2object Array = new Apl2object(Slave,3.14159) ;
          Apl2cdr Cdr = Array.cdrValue() ;
          Array.Free() ;
          //* Write the Apl2cdr object to a file
          String FileName = new String("Apl2DemoSerialize") ;
          FileOutputStream fos = new FileOutputStream(FileName) ;
          ObjectOutputStream oos = new ObjectOutputStream(fos) ;
          oos.writeObject(Cdr) ;
          oos.flush() ;
          //* Read the Apl2cdr object from the file
          FileInputStream fis = new FileInputStream(FileName) ;
          ObjectInputStream ois = new ObjectInputStream(fis) ;
```


© Copyright IBM Corporation 1994, 2017

```
            Apl2cdr NewCdr = (Apl2cdr)ois.readObject() ;
            //* Delete the file
            File f = new File(FileName) ;
            f.delete() ;
            //* Create a new array in the workspace from the CDR
            Apl2object NewArray = new Apl2object(Slave,NewCdr) ;
            NewArray.Free() ;
            Slave.Stop() ;
            }
         catch (Apl2exception Exception) {
            System.out.println("Apl2exception caught.");
            System.out.println("Exception message: " + Exception.getMessage());
            System.out.println("Exception cause: " + Exception.getCause());
            System.out.println("Event: " + Exception.Type + " " + Exception.Code) ;
         }
      return ;
      }
   }
```


© Copyright IBM Corporation 1994, 2017

Calling APL2 from Visual Basic

APL2 applications can be called from Visual Basic.

The Visual Basic to APL2 interface supports the following features:

- Starting and Stopping APL2 Interpreters
- Using Workspace Objects
- Assigning, Associating, and Expunging Names
- Executing Expressions and Functions
- Using APL Characters in Visual Basic Programs
- Handling APL2 Errors
- Data Conversion Between Visual Basic and APL2

An APL2 interpreter called from Visual Basic has the same restrictions as the APL2 Runtime Library. Primarily, the session manager and workspace system commands are not supported. For a complete description, see Restrictions of the Runtime Library.

This section concludes with an example that illustrates how to use the Visual Basic to APL2 interface:

- Example Visual Basic Program

Note: The Visual Basic to APL2 interface is only available on Windows.

Starting and Stopping APL2 Interpreters

To start an APL2 interpreter, call the `StartApl2` function:

```
Declare Function StartApl2 Lib "apl2vb" (Optional ByRef Options As Variant) As Long
Dim Token As Long
Token = StartApl2(Array("-ws","2m"))
```

`StartApl2`'s optional argument is an array of strings containing invocation options.

Any APL2 invocation parameters may be passed, as defined in Invoking APL2. However, these parameters will be ignored if specified:

- -hostwin
- -input
- -lx (is always OFF)
- -quiet (is always ON)
- -run
- -rns
- -sm (is always OFF)
- -svplisten
- -svptrace

© Copyright IBM Corporation 1994, 2017

The shared variable processor options -svplisten and -svptrace can be enabled by setting environment variables APLSVPLISTEN and APLSVPTRACE before starting APL2, setting values for the SVPLISTEN and SVPTRACE keywords in the [Invocation Options] section of the `apl2.ini` configuration file, or using the menu options in The SVP Monitor Facility.

`StartApl2`'s result is an APL2 interpreter token or zero if an error occurred.

To stop an interpreter, call the `StopApl2` function:

```
Declare Function StopApl2 Lib "apl2vb" (ByVal Token As Long) As Long
StopApl2(Token)
```

`StopApl2`'s argument is an APL2 interpreter token.

Using Workspace Objects

The `VariantToLocator`, `LocatorToVariant`, and `FreeLocator` functions are used to create, reference, and free APL2 workspace objects.

```
Declare Function VariantToLocator Lib "apl2vb" (ByVal Token As Long, _
   ByRef Var As Variant) As Long
Declare Function LocatorToVariant Lib "apl2vb" (ByVal Token As Long, _
   ByVal Locator As Long, ByRef Var As Variant) As Boolean
Declare Function FreeLocator Lib "apl2vb" (ByVal Token As Long, _
   ByVal Locator As Long) As Long
Dim Array As Variant
Dim Locator as Long
Dim BoolRc as Boolean
' Add code here to build Array
Locator = VariantToLocator(Token, Array)
BoolRc = LocatorToVariant(Token, Locator, Array)
FreeLocator Token, Locator
```

`VariantToLocator`'s arguments are an APL2 interpreter token and a variant. The result is a workspace object locator or zero if the variant cannot be converted.

The `LocatorToVariant` function's three arguments are an APL2 interpreter token, a workspace object locator, and a variant that is to be updated with the contents of the workspace object. The result is a Boolean that indicates whether an error occurred. False indicates success; true indicates failure.

See Data Conversion Between Visual Basic and APL2 for information on how data is converted when using this interface.

`FreeLocator`'s arguments are an APL2 interpreter token and a workspace object locator. Locators with a value of zero are ignored.

Assigning, Associating, and Expunging Names

Use the `Assign` function to assign a name to an object.

© Copyright IBM Corporation 1994, 2017

```
Declare Function Assign Lib "apl2vb" (ByVal Token As Long, _
   ByVal Name As Long, ByVal Value As Long) As Long
Dim VarName As Long
Dim VarValue As Long
VarName = VariantToLocator(Token,"VARNAME")
VarValue = VariantToLocator(Token,3.14159)
Assign(Token,VarName,VarValue)
```

`Assign`'s arguments are an APL2 interpreter token and two APL2 object locators. The first is the name to be assigned to the object. The second is the object.

`Assign`'s result is 1 if the assignment worked, 0 if it failed.

Use the `Associate` function to associate names in the APL2 workspace.

```
Declare Function Associate Lib "apl2vb" (ByVal Token As Long, _
   ByVal Class As Long, ByVal Processor As Long, ByVal Name As Long) As Long
Dim AplRc As Long
Dim Class As Long
Dim Processor As Long
Dim Name As Long
Class = VariantToLocator(Token,3)
Processor = VariantToLocator(Token,11)
Name = VariantToLocator(Token,"BEEP")
AplRc = Associate(Token,Class,Processor,Name)
If AplRc = 0 Then
   ' Association failed
End If
```

`Associate`'s arguments are an APL2 interpreter token and three APL2 object locators. The first two are used to form the left argument of `⎕NA`; the third is the right argument.

`Associate`'s result is the result from `⎕NA` - 1 for success; 0 for failure.

Use the Expunge function to expunge a name in the APL2 workspace.

```
Declare Function Expunge Lib "apl2vb" (ByVal Token As Long, _
   ByVal Class As Long)
Expunge Token, Name
```

Executing Expressions and Functions

Use the `ExecuteExpression`, `ExecuteMonadic`, and `ExecuteDyadic` functions to execute APL2 expressions and functions.

```
Declare Function ExecuteExpression Lib "apl2vb" (ByVal Token As Long, _
   ByVal Func As Long) As Long
Declare Function ExecuteMonadic Lib "apl2vb" (ByVal Token As Long, _
   ByVal Func As Long, ByVal Right As Long) As Long
Declare Function ExecuteDyadic Lib "apl2vb" (ByVal Token As Long, _
   ByVal Left As Long, ByVal Func As Long, ByVal Right As Long) As Long
Dim Exp As Long
Dim Ambi As Long
Dim Left As Long
Dim Right As Long
Exp = VariantToLocator(Token,"3+4 5 6")
```


© Copyright IBM Corporation 1994, 2017

```
Ambi = VariantToLocator(Token,"FOO")
Left = VariantToLocator(Token,45)
Right = VariantToLocator(Token,"ABC")
Result = ExecuteExpression(Token,Exp)
Result = ExecuteMonadic(Token,Ambi,Right)
Result = ExecuteDyadic(Token,Left,Ambi,Right)
```

The `ExecuteExpression` function's arguments are an APL2 interpreter token and a APL2 object locator. The locator identifies a workspace object containing an expression to be executed. The result is another locator; it is zero if no result is produced or an error occurred.

The `ExecuteMonadic` function's arguments are an APL2 interpreter token and two APL2 object locators. The first locator identifies a workspace object containing the function name; the second identifies the right argument. The result is another locator; it is zero if no result is produced or an error occurred.

The `ExecuteDyadic` function's arguments are an APL2 interpreter token and three APL2 object locators. The first locator identifies the function's left argument; the second identifies the function name; the third identifies the right argument. The result is another locator; it is zero if no result is produced or an error occurred.

Functions processed under control of these Execute functions operate in the same manner as those processed under control of `⎕EC`, and exhibit the following behavior:

- Assignment expressions return a value.
- Requests for quad input are handled the same as quad input under `⎕EC`.
- Errors and branch escape (`→`) during execution cause a zero locator to be returned.
- Stop control vectors (`S∆`) are ignored.
- Attention and interrupt signals do not occur.

Using APL Characters in Visual Basic Programs

Visual Basic does not support use of APL characters in source code. However, Visual Basic's `ChrW` function can be used to create Unicode characters including APL characters.

The following example creates an APL2 object containing the character vector `⎕TS`.

```
TS = VariantToLocator(Token, ChrW(9109) & "TS")
```

Use the `⎕UCS` function to determine the Unicode codepoint of APL characters.

Handling APL2 Errors

Use the `GetET`, `GetMsg`, and `GetStack` functions to retrieve error information from APL2.

```
Declare Function GetET Lib "apl2vb" (ByVal Token As Long) As Variant
Declare Function GetMsg Lib "apl2vb" (ByVal Token As Long) As Variant
Declare Function GetStack Lib "apl2vb" (ByVal Token As Long) As Variant
Dim ET As Variant
Dim Msg as Variant
Dim Stack As Variant
ET = GetET(Token)
Msg = GetMsg(Token)
```


© Copyright IBM Corporation 1994, 2017

```
Stack = GetStack(Token)
```

When an error occurs in APL2, the values of ⎕ET and ⎕EM and the contents of the APL2 stack are stored in the Visual Basic interface. Use `GetET` to retrieve the value of ⎕ET. Use `GetMsg` to retrieve the value of the first row of ⎕EM. Use `GetStack` to retrieve the APL2 stack as an array of character strings.

Note: All the other Visual Basic interface functions reset the stored error information. To retrieve error information, use `GetET`, `GetMsg`, or `GetStack` before calling any other Visual Basic interface functions.

Data Conversion Between Visual Basic and APL2

Visual Basic and APL2 use different internal data types. When moving data between the environments, the Visual Basic to APL2 interface performs automatic data conversion.

The following table shows the supported Visual Basic types and the APL2 data types produced by the Visual Basic to APL2 interface data conversion routines:

Data Conversion from Visual Basic to APL2

Visual Basic	**APL2**
Boolean	Integer
Character string	Vector of 4 byte characters
Currency	Floating point
Date	Floating point
Error code	Integer
Floating point (4 or 8 bytes)	Floating point (8 byte)
Integer (signed or unsigned 1, 2, or 4 byte)	Integer
Integer (signed or unsigned 8 byte values)	Floating point
Interface pointer (IDispatch or IUnknown)	Integer
Empty and Null	Empty character vector

Restrictions:

- Other Visual Basic data types are not supported and produce an error.

The following table shows the APL2 types and the Visual Basic data types produced by the Visual Basic to APL2 interface data conversion routines:

Data Conversion from APL2 to Visual Basic

APL2	**Visual Basic**
Boolean	Integer
Integer	Integer

© Copyright IBM Corporation 1994, 2017

Data Conversion from APL2 to Visual Basic	
APL2	**Visual Basic**
Floating point	Floating point
Character scalar or vector	Character string

Restrictions:

- Character arrays with rank greater than 1 are not supported
- Complex numbers are not supported
- Rank must be 60 or less

Notes on Handling Boolean Data:

When passing data from APL2 to Visual Basic programs, the Visual Basic to APL2 interface converts APL2 Boolean values of 1 to Visual Basic Integer values of 1. This provides correct behavior for Visual Basic programs with integer arguments, and normally provides correct behavior for Visual Basic programs with Boolean arguments because most Visual Basic programs treat any non-zero Boolean arguments as TRUE.

However, Visual Basic programs store Boolean TRUE values as integer -1. The Visual Basic to APL2 interface converts Visual Basic Boolean TRUE values to APL2 integer -1 values. There is no way to determine from an APL2 application whether a value of -1 returned by a Visual Basic program is a Boolean TRUE or an integer -1. Consult the documentation for the Visual Basic program to determine argument and result types.

Example Visual Basic Program

Here is a complete Visual Basic program that calls APL2 and handles errors:

```
' Visual Basic to APL2 Programming Interface Function Declarations
Private Declare Function StartApl2 Lib "apl2vb" (Optional ByRef Options As _
   Variant) As Long
Private Declare Function StopApl2 Lib "apl2vb" (ByVal Token As Long) As Long
Private Declare Function VariantToLocator Lib "apl2vb" (ByVal Token As Long, _
   ByRef Var As Variant) As Long
Private Declare Function LocatorToVariant Lib "apl2vb" (ByVal Token As Long, _
   ByVal Locator As Long, ByRef Var As Variant) As Boolean
Private Declare Function ExecuteExpression Lib "apl2vb" (ByVal Token As Long, _
   ByVal Func As Long) As Long
Private Declare Function ExecuteMonadic Lib "apl2vb" (ByVal Token As Long, _
   ByVal Func As Long, ByVal Right As Long) As Long
Private Declare Function ExecuteDyadic Lib "apl2vb" (ByVal Token As Long, _
   ByVal Left As Long, ByVal Func As Long, ByVal Right As Long) As Long
Private Declare Function Assign Lib "apl2vb" (ByVal Token As Long, _
   ByVal Name As Long, ByVal Value As Long As Long
Private Declare Function Associate Lib "apl2vb" (ByVal Token As Long, _
   ByVal Class As Long, ByVal Processor As Long, ByVal Name As Long) As Long
Private Declare Function Expunge Lib "apl2vb" (ByVal Token As Long, _
   ByVal Name As Long) As Long
Private Declare Function FreeLocator Lib "apl2vb" (ByVal Token As Long, _
   ByVal Locator As Long) As Long
Private Declare Function GetET Lib "apl2vb" (ByVal Token As Long) As Variant
Private Declare Function GetMsg Lib "apl2vb" (ByVal Token As Long) As Variant
Public Sub CallAPL2()
' This subroutine demonstrates how to call APL2 from Visual Basic under Excel.
Dim A1 As Variant
```


© Copyright IBM Corporation 1994, 2017

```
Dim B1 As Variant
Dim Token As Long
Dim Class As Long
Dim Processor As Long
Dim Func As Long
Dim Left As Long
Dim Right As Long
Dim Result As Long
Dim ET As Variant
Dim Msg As Variant
Dim BoolRc as Boolean
' Extract the values of cells A1 and B1
A1 = Worksheets("Sheet1").Range("A1:A1").Formula
B1 = Worksheets("Sheet1").Range("B1:B1").Formula
' Validate they are numeric
If Not (IsNumeric(A1) And IsNumeric(B1)) Then
    MsgBox "Cells A1 and B1 are not both numeric"
    Exit Sub
End If
' Convert them to numbers
A1 = Val(A1)
B1 = Val(B1)
' Start an APL2 interpreter
' StartApl2's optional argument is an array containing invocation options.
' The result is an APL2 interpreter token or zero if an error occurred.
Token = StartApl2(Array("-ws", "2m"))
' Create an APL2 object from an array of frequency and duration values
' VariantToLocator's arguments are an APL2 interpreter token and a variant.
' The variant can contain any of the following types of data:
'
'   Boolean values
'   Character vectors (Unicode)
'   Currency values
'   Dates
'   Error codes
'   Floating point numbers (4 or 8 bytes)
'   Integers (signed or unsigned 1, 2, 4, or 8 byte values)
'   Interface pointers (IDispatch or IUnknown)
'   Empty and Null values (converted to '')
'
' The data can be scalar or an array with any number of dimensions.
'
' The result is a APL2 object locator or zero if the variant cannot be converted.
Right = VariantToLocator(Token, Array(440, 250))
' Create some APL2 objects for use as arguments to the Associate function
Class = VariantToLocator(Token, 3)
Processor = VariantToLocator(Token, 11)
Func = VariantToLocator(Token, "BEEP")
' Associate a name
' Associate's arguments are an APL2 interpreter token and three APL2 object
' locators. The first two are the left argument of QuadNA; the third is the right
' argument.
' The result is a 0 or 1.  Note: The result is a long: 1 for success; 0 for failure.
AplRc = Associate(Token, Class, Processor, Func)
' Free the APL2 object locators for the namespace and processor.
' FreeLocator's arguments are an APL2 interpreter token and an object locator.
' Zero object locators are ignored.
FreeLocator Token, Class
FreeLocator Token, Processor
' Make sure the association worked
If AplRc = 0 Then
    MsgBox "Association with BEEP function failed!"
    FreeLocator Token, Right
    FreeLocator Token, Func
    StopApl2 Token
    Exit Sub
End If
' Call the BEEP function
' The ExecuteMonadic function's arguments are an APL2 interpreter token and two
' APL2 object locators.  The first locator is the function name; the second
```


© Copyright IBM Corporation 1994, 2017

```
' is the right argument.
' The result is a locator; it is zero if no result is produced or an error occurred.
Result = ExecuteMonadic(Token, Func, Right)
' Expunge the associated name
' The Expunge function's arguments are an APL2 interpreter token and an object
' locator for an object name.
' The result is a 0 or 1.  Note: The result is a long: 1 for success; 0 for failure.
Expunge Token, Func
' Free the function name, argument, and any result
FreeLocator Token, Func
FreeLocator Token, Right
FreeLocator Token, Result
' Create a APL2 object containing the name of an APL2 primitive
Func = VariantToLocator(Token, "+")
' Create APL2 objects from the cell values
Left = VariantToLocator(Token, A1)
Right = VariantToLocator(Token, B1)
' Execute the primitive function with two arguments
' The ExecuteDyadic function's arguments are an APL2 interpreter token and three
' APL2 object locators.  The first locator is the function's left argument;
' the second is the function name; the third is the right argument.
' The result is a locator; it is zero if no result is produced or an error occurred.
Result = ExecuteDyadic(Token, Left, Func, Right)
' If an error occurred,,,
If Result = 0 Then
    ' Get the Event Type
    ' GetET has one argument: an interpreter token.
    ' It returns a variant containing two integers corresponding to the value of
    ' QuadET after the last error.
    ET = GetET(Token)
    ' GetMsg has one argument: an interpreter token.
    ' It returns a variant containing a string corresponding to the first row of
    ' QuadEM after the last error.
    Msg = GetMsg(Token)
    MsgBox "Execution of + failed.  " _
        & vbLf & vbLf & "Event type: " & Str(ET(0)) & Str(ET(1)) _
        & vbLf & vbLf & "Error Message: " & Msg
    StopApl2 Token
    Exit Sub
End If
' Free the locators for the function name and arguments; we no longer need them.
FreeLocator Token, Left
FreeLocator Token, Right
FreeLocator Token, Func
' Extract the result
' The LocatorToVariant function's three arguments are an APL2 interpreter token,
' a locator, and a variant that is updated with the contents of the workspace object.
' The result is a Boolean that indicates whether the function failed.
' False indicates success; true indicates failure.
' Any depth 0 or 1 APL2 array can be converted except that complex numbers are not
' supported (by Visual Basic.)  Depth 2 arrays of character vectors are also supported.
BoolRc = LocatorToVariant(Token, Result, Var)
' Free the result's locator
FreeLocator Token, Result
' Put the result in cell C1
Worksheets("Sheet1").Range("C1:C1").Formula = Var
' Create a APL2 object containing the string QuadTS.
Func = VariantToLocator(Token, ChrW(9109) & "TS")
' Execute QuadTS
' The ExecuteExpression function's arguments are an APL2 interpreter token and a
' APL2 object locator.  The locator is an expression to be executed.
' The result is a locator; it is zero if no result is produced or an error occurred.
Result = ExecuteExpression(Token, Func)
' Free the locator containing the expression
FreeLocator Token, Func
' Extract the value of the result
BoolRc = LocatorToVariant(Token, Result, Var)
' Free the result's locator
FreeLocator Token, Result
' Put the value in the cells B14 to H14
```


© Copyright IBM Corporation 1994, 2017

```
Worksheets("Sheet1").Range("B14:H14").Formula = Var
' Stop the APL2 interpreter
' StopApl2's argument is an APL2 interpreter token
StopApl2 Token
End Sub
```


© Copyright IBM Corporation 1994, 2017

Writing Your Own External Routines

The following topics discuss how to write your own external routines to be called through processor 11. For more information about processor 11, see Processor 11 - Accessing External Routines.

- Using Prebuilt DLLs (Runtime Libraries)
- Creating SYSTEM Linkage Routines
- Creating FUNCTION Linkage Routines

© Copyright IBM Corporation 1994, 2017

Exception Handling in External Functions

APL2 uses the C runtime library signal routine to register handlers for the following signals:

- SIGBREAK - Ctrl+Break (Windows only)
- SIGCHLD - Child process termination (Unix systems only)
- SIGHUP - Hang up (Unix systems only)
- SIGINT - Ctrl+C
- SIGQUIT - Dump request (Unix systems only)
- SIGSEGV - Access violations
- SIGTERM - Process termination

External functions that set handlers for these signals should save APL2's handlers upon entry and restore them before returning to APL2.

APL2 does not use signal to manage floating point conditions. Instead, APL2 accesses the floating point control and status words directly. APL2 disables all floating point exceptions and uses the floating point status flags to detect error conditions.

External functions that manipulate the floating point control and status words should save APL2's settings upon entry and restore them before returning to APL2.

Note:

If APL2 itself will be called by a C or Java program using the APL2 Programming Interface (APL2PI), additional considerations apply. See Handling Exceptions under C and Handling Exceptions under Java for more information.

© Copyright IBM Corporation 1994, 2017

Loading the APL2 Fonts (Windows Only)

When the APL2 installation is performed by a user with administrator authority, the APL2 fonts supplied by IBM will be automatically installed into the Windows Fonts directory where they can be found by any program on the system. The fonts may also be installed there manually per the instructions in Installing the APL2 Fonts. However, there is no guarantee that the fonts will always be installed.

If your user-written function requires the APL2 fonts, you may not want to rely on the fonts being already present. To ensure that the fonts are loaded, you can call routine *LoadApl2Fonts* in `aplwin.dll`.

Since `aplwin.lib` is not provided for use at link time, the Windows APIs *LoadLibrary* and *GetProcAddress* must be used to load the dll and obtain the address of the routine at run time.

LoadApl2Fonts takes no arguments, and returns a Boolean result indicating success (0) or failure (1).

Example:

```
HINSTANCE dllwin;
void *ploadfonts;
BOOL boolrc;
typedef BOOL ((loadfonts)()) ;
dllwin = LoadLibrary("aplwin.dll");
if (dllwin)
{
   ploadfonts = (void *)GetProcAddress(dllwin, "LoadApl2Fonts");
   if (ploadfonts)
      boolrc =  (*(loadfonts *)ploadfonts)();
   FreeLibrary(dllwin);
}
```


© Copyright IBM Corporation 1994, 2017

Using Prebuilt DLLs (Runtime Libraries)

Note: This section applies only to Windows.

Many 32-bit DLLs can be called directly by Processor 11. These DLLs can be part of a subroutine library or program product. To use them directly, you must ensure that they can be called with 32-bit `_System` linkage. (On some non-IBM compilers, this is known as `__stdcall`.) If the DLL was meant to be called with Optlink, Pascal, or some other linkage convention, then an intervening stub program is needed. Refer to the documentation received with the library for parameter lists and linkage conventions.

For `_System` linkage DLLs, you need only create routine descriptors for the routines you wish to access. The routine descriptor can be placed in an external file (NAMES file) or passed in the left argument when associating the routine name with Processor 11.

For more information on routine descriptors see NAMES files and Routine Descriptors. For general information on using Processor 11, see Processor 11 Overview.

Example

The following routine descriptor is for GetCurrentDirectory in the Windows dll `KERNEL32.DLL`.

```
:nick.DIRECTORY :link.SYSTEM
                :lib.kernel32  :proc.GetCurrentDirectoryA
                :valence.1 1 0
                :rslt.(1 I4 0)
                :rarg.(2 G0 1 2)(1 I4 *)
                :rarg.           (G4 0)(>S1 1 *)
                :desc.DWORD GetCurrentDirectory(DWORD nBufferLength,
                :desc.                          LPTSTR lpBuffer);
```

This example uses a 2-element depth 3 array to pass the data. The return buffer has a pass by name indicator and its value is a character vector that contains the name of an APL object having the desired size and type.

```
      DD← (255⍴'+')                        ⍝ named variable
      DIRECTORY  (1+⍴DD) (⊂'DD')           ⍝ here is the call
 29
```

The result value as indicated by the `:rslt.` keyword is the length of the data stored. The value of `DD` has been updated in place. `DD` must be large enough to hold the result.

```
      DD                    ⍝ here is the updated variable
C:\Program Files\IBMAPL2W\BIN
```


© Copyright IBM Corporation 1994, 2017

Creating SYSTEM Linkage Routines

To create your own SYSTEM linkage routine to be called by Processor 11:

1. Write your subroutine as a 32-bit object using `_System` linkage. (On some non-IBM compilers, this is known as `__stdcall`.) For example:

```
/* IBM VisualAge example  */
#include <stdio.h>
#include <string.h>
int _System MyFunction(char *string)
{
  return strlen(string);
}
```

   On Unix systems, the default linkage would be used instead of `_System`:

```
/* Unix example  */
#include <stdio.h>
#include <string.h>
int MyFunction(char *string)
{
  return strlen(string);
}
```

2. Create a routine descriptor as described in NAMES Files and Routine Descriptors. The routine descriptor can be placed in a NAMES file or passed in the left argument of `⎕NA`. For this example, we will place it in a file called `P011.NAM`.

```
:nick.MY        :link.SYSTEM :lib.example :proc.MyFunction :path.C:\MYDIR
                :valence.1 1 0
                :rarg. (G4 0)(S1 1 *)
                :rslt. (I4 0)
:desc.Returns the length of a character vector to the first null.
```

   On Unix systems, the `:path.` data would follow Unix conventions:

```
:nick.MY        :link.SYSTEM :lib.example :proc.MyFunction :path./u/mydir
                :valence.1 1 0
                :rarg. (G4 0)(S1 1 *)
                :rslt. (I4 0)
:desc.Returns the length of a character vector to the first null.
```

3. Create a definition or export file as described in your C compiler documentation:

   On Windows:

© Copyright IBM Corporation 1994, 2017

example.def

```
LIBRARY
EXPORTS
    __DLL_InitTerm@8
    _MyFunction@4
```

On AIX:

example.exp

```
MyFunction
```

Note: The export file is not required on Solaris and Linux systems.

4. Compile the program.

   On Windows:

   ```
   icc /Ti /Gd- /Ge- /Gt- /O- /B"/noe /noi" example.c example.def
   ```

   On AIX Systems:

   ```
   cc -o example example.c -bE:example.exp -bM:SHR -e _nostart
   ```

   On Solaris Systems:

   ```
   cc -G -o example example.c
   ```

   On Linux Systems:

   ```
   cc -shared -o example example.c
   ```

5. Put the files `EXAMPLE.DLL` (on Unix systems, `example`) and `P011.NAM` in the directory specified by the `:path.` tag, or if `:path.` was not specified, the directory APL2 will be started from.
6. Execute the `⎕NA`:

   ```
         '<C:\MYPATH\P011.NAM>' 11 ⎕NA 'MY'
   ```

7. Test the function:

   ```
         MY ⊂ 'sssss',(⎕AF 0),'ttttt'
   5
   ```


© Copyright IBM Corporation 1994, 2017

Creating FUNCTION Linkage Routines

Like a :link.SYSTEM routine, a :link.FUNCTION routine is coded, compiled and linked as a 32-bit object with _System (or __stdcall) linkage. See Creating SYSTEM Linkage Routines for details on the compile process.

Unlike :link.SYSTEM routines, however, arguments are not passed as direct parameters to the C routine, and results are not passed back in the C routine result. All :link.FUNCTION routines have the same prototype:

```
int _System RoutineName(CALL *call)
```

Communication between the interpreter and the routine, including location of arguments and results, is handled using a control block structure.

The routine descriptors for :link.FUNCTION routines are similar to those for :link.SYSTEM routines, except that :rarg and :rslt tags are not used. Processor 11 passes the arguments directly from the workspace to the routine, and returns whatever array the routine builds for its result. The routine is responsible for validating that the arguments meet its criteria. The following is a sample nickname file routine descriptor for the routine prototyped above:

```
:nick.MYFUN     :link.FUNCTION :lib.my  :proc.RoutineName
                :valence.1 1 0
```

The CALL structure, and other constants used in this section, are defined in a macro, aplfun.h, which is shipped with APL2 in the include subdirectory of the APL2 product directory. A set of sample routines which demonstrate many of the services described in this chapter is shipped as aplfun.c, in the samples (Windows) or examples/qna (Unix) subdirectory.

- The CALL Structure
- Entry and Exit Conventions
- Handling APL2 Objects
- Callback Services
- Message Codes
- Sample Routines

The CALL Structure

The CALL structure is used for all communication between the interpreter and the :link.FUNCTION routine. The address of the structure is passed as the argument to the routine. The same structure is used as the argument to the callback services used by the routine.

```
typedef struct _call {
        short request;       /* request code on entry            */
                             /* return code on return            */
        short reason;        /* reason code if applicable        */
        PFN  service;        /* pointer to Service Routine       */
        long parm1;          /* service parameter 1              */
```


© Copyright IBM Corporation 1994, 2017

```
        long parm2;          /* service parameter 2              */
        long token;          /* routine token (user word)        */
        void *result;        /* address of result                */
        long result_token;   /* result object locator token      */
        void *left_arg;      /* address of left arg              */
        long left_arg_token; /* left arg locator token           */
        long left_fn;        /* reserved for future use          */
        long operator_token; /* reserved for future use          */
        long right_fn;       /* reserved for future use          */
        long right_arg_token;/* right arg locator token          */
        void *right_arg;     /* address of right arg             */
        long reloc_count;    /* relocation count                 */
        long parm3;          /* service parameter 3              */
    } CALL   ;
```

Entry and Exit Conventions

A `:link.FUNCTION` routine is entered with `->request` set to:

`1` (`FUNCTION_INIT`) when the name is resolved (associated).
`->service`
is the callback service routine
`->reloc_count`
is the relocation count

The routine can set `->token` to point to its own anchor block or any desired resource. The value in `->token` will be retained across calls.

`0` (`FUNCTION_CALL`) each time the name is called.
`->service`
is the callback service routine
`->token`
is the routine token from `FUNCTION_INIT`
`->left_arg_token`
is the left arg locator or 0 if there is no left argument
`->left_arg`
is the address of the left arg or NULL if there is no left argument
`->right_arg_token`
is the right arg locator or 0 if there is no right argument
`->right_arg`
is the address of the right arg or NULL if there is no right argument
`->reloc_count`
is the relocation count

The routine must process the argument(s) and set `->result_token` to the locator of an APL object to be used as the result, if one is required.

`-1` (`FUNCTION_TERM`) when the name is expunged.
`->service`
is the callback service routine
`->token`
is the routine token from `FUNCTION_INIT`

© Copyright IBM Corporation 1994, 2017

`->reloc_count`
is the relocation count

The routine should free any resources associated with `->token` before returning.

In all cases the routine must set `->request` before returning. If not set to 0, it must be set to one of the error message codes defined in Message Codes. If appropriate for the error message code chosen, `->reason` can also be set to one of the values defined for the message.

Handling APL2 Objects

Since the APL2 objects with which a `:link.FUNCTION` routine deals reside directly in the workspace, the routine must understand their structure, and be aware of some aspects of how storage is managed within the workspace.

- Locating APL2 Objects
- Processing APL2 Objects
- Building APL2 Objects

Locating APL2 Objects

Each APL object has a unique identifier, called the *locator token* or just *token*. The token is the means by which the interpreter can always locate the object, regardless of whether it has moved within the workspace.

At any given time, each APL object resides at a specific *address* in the user machine. A routine must use the address to actually access the contents of the object. The address may change when the interpreter storage management routines move objects within the workspace. The routine needs to be aware of when that can happen, and obtain new addresses for objects when necessary.

When the routine is entered, both the token and the address for the routine's argument(s) are passed in the CALL block. The routine starts with these, and may access more objects (if the arguments are nested) or build new objects (for the routine result).

As long as control remains inside the routine, objects in the workspace cannot move and the addresses remain valid. At some point, however, the routine is going to need to allocate some space in the workspace for its result. When the routine uses the ARRAYSPACE, ARRAYCONVERT or LCDR_TO_ARRAY callback services to allocate storage, or the ARRAYRESIZE callback service to lengthen an array, objects in the workspace can move during the time the interpreter has control to allocate the storage.

Upon return from these callback services, the routine can test to see whether it is necessary to refresh object addresses. The `->reloc_count` field in the `CALL` block can be used to do this. Before calling the callback service, save the value from `->reloc_count`. If the value has changed when the callback service returns, objects in the workspace have moved, and all addresses of objects stored locally within the routine must be refreshed.

The addresses in the CALL block (for the arguments and result, if any) are always refreshed automatically before the callback service returns to the routine. If you have made local copies of these addresses, you can simply re-copy them. For other objects, use TOKEN_TO_ADDRESS callback service to obtain new addresses.

© Copyright IBM Corporation 1994, 2017

Processing APL2 Objects

Each APL2 workspace object contains information that describes its data type, shape, size, and origin. This information is called its header, and is located at the beginning of the object. The header consists of the following fields (defined in C include file `aplobj.h`):

```
typedef struct aplobj {
  unsigned long  ptr      ;
  unsigned long  nb       ;
  unsigned long  nelm     ;
  unsigned char  type     ;
  unsigned char  rank     ;
  unsigned char  fill[2]  ;
  unsigned long  dim[1]   ;
} APLOBJ ;
```

`ptr` The locator token of the object.

`nb` The number of bytes in this APL2 object. If the datatype of this object is that of a nested array the byte count includes only this object - it does not include the subitems. (Note: this is different from the APL2 objects passed to auxiliary processors.) The length of each object must be rounded to an even multiple of 16.

`nelm` The number of elements in this APL2 object.

`type` APL2 object type:

```
0  (BOOLEAN)    Boolean             1 bit per item
1  (INTEGER)    Integer             4 bytes per item
2  (FLOAT)      Real                8 bytes per item
3  (COMPLEX)    Complex             16 bytes per item
4  (CHARACTER)  ASCII Character     1 byte per item
5  (CHARLONG)   Extended Character  4 bytes per item
6  (APV)        Progression Vector  8 bytes
7  (NESTED)     General Array       4 bytes per item
```

`rank` Rank of object (0-64).

`fill` Unused; should be set to 0.

`dim[]` Length of each dimension (number of elements in dim = rank)

Immediately following the header for each object is the data associated with the object. The length of the data for each type is shown above. If the object has more than one dimension, its elements are stored in row order (as if the APL2 primitive Ravel had been applied to the variable).

Immediately following the data are enough fill bytes to make the length of the object an even multiple of 16.

If the array is a general array, the data consists of a set of locator tokens - one for each subitem of the array. If the general array is null, the data consists of one locator token - for an array which is the prototype for the general array. To obtain the addresses for the sub-items of a nested array, use the TOKEN_TO_ADDRESS callback service.

Building APL2 Objects

There are two ways that a routine can build an APL object for its result. One method is to allocate the array (and its subitems if general) directly in the workspace with the ARRAYSPACE callback service and build the object there. The other is to build the object in local storage within the routine, and when the array is complete, use the LCDR_TO_ARRAY callback service to place the array into the workspace.

© Copyright IBM Corporation 1994, 2017

In general, building the array in the workspace is more efficient. The array contents do not have to be copied into the workspace, they will be placed directly there. However, there are several reasons that you may want to build the object in local storage first.

1. You may be in a situation (such as reading data from a pipe or port) where you do not know in advance how large the array will be. Allocating an array much larger than necessary could result in WS FULL.
2. You may be in situation (such as fetching data from a database) where you need to rearrange or compress the data after obtaining it to make the APL2 object.
3. When you have a nested array, each item and subitem must be allocated separately with a call to ARRAYSPACE. If any of the calls fails (usually because of WS FULL) the items which were allocated successfully must be freed, and depending on your routine, additional cleanup may be necessary. When you build the array locally, there is a single call to LCDR_TO_ARRAY.
4. There is an additional degree of safety in this method. When you build an array directly in the workspace, no validation of your object is done. If you make an error, you can potentially corrupt the workspace. When you build the array locally and use LCDR_TO_ARRAY, a complete validation of your object is performed before copying it to the workspace. If there are errors in the object, control will be returned to your routine, which can then perform any desired cleanup before returning an error.

When you build a result array in local storage to pass to the LCDR_TO_ARRAY service, the contents of some of the fields on the `APLOBJ` structure are different from that of workspace objects. The object format used is the same format as is used by auxiliary processors, called *Common Data Representation* (CDR). The differences between the workspace object format and CDR format are:

1. The `->ptr` field of a workspace object contains its locator token. The `->ptr` field of a CDR contains a platform identifier.
2. For general arrays, the data section of a workspace object contains the locator tokens of the subarrays, which are separate objects. The data section of a general array CDR contains the offsets to the subarrays, which must follow the general array CDR in left-list order.
3. For general arrays, the `->nb` field of a workspace object is the number of bytes in the general array only. The `->nb` field of a general array CDR is the total number of bytes in the general array and all of its subitems.

For complete information on building a CDR, see Common Data Representation.

Callback Services

A set of interpreter services to assist in handling APL arrays is available to external routines written with `:link.FUNCTION` linkage. These services are accessed by calling the service address found in the `->service` field of the `CALL` block.

Parameters and results are passed using the `CALL` block, as defined for each service.

In general, each service returns an error message code as its result, and stores that same message code in the `->request` field of the `CALL` block. When an error occurs, the routine should free any temporary resources it has allocated, and can then return with the `->request` and `->reason` fields as set by the service.

The services are:

© Copyright IBM Corporation 1994, 2017

ARRAYSPACE	Allocate an array in the workspace
ARRAYRESIZE	Change the number of elements in an allocated array
ARRAYREF	Reference an array in the workspace
ARRAYCONVERT	Convert an array to a new type
FREESPACE	Remove a reference to an array
EXECUTE_APL_ESTACK	Execute an APL expression or function, return stack on error
EXECUTE_APL	Execute an APL expression or function
TOKEN_TO_ADDRESS	Get the address of an array
QUAD_CT	Get the current comparison tolerance
QUAD_IO	Get the current index origin
LCDR_TO_ARRAY	Convert a linear CDR to an array
ARRAY_TO_LCDR	Convert an array to a linear CDR
DOAP1EVENTS	Allow AP 1 to handle pending events
GETSIGNALS	Query and/or clear the signal state

ARRAYSPACE - Allocate an array in the workspace

Parameters:

`->request` `21 (ARRAYSPACE)`

`->parm1` number of elements

`->parm2` rank

`->parm3` type, one of:

```
0  (BOOLEAN)
1  (INTEGER)
2  (FLOAT)
3  (COMPLEX)
4  (CHARACTER)
5  (CHARLONG)
6  (APV)
7  (NESTED)
```

Results:

`->request` return code - one of:

```
0 (MSG_OK) - if the request was successful.
2 (MSG_SYSTEMERROR) - if an invalid type is passed.
3 (MSG_WSFULL) - if there is no room in the workspace for this object.
4 (MSG_SYSTEMLIMIT) - if rank or number of elements is out of range.
```

`->reason` reason code, if `->request` is `MSG_SYSTEMLIMIT`:

```
8 (ET_ARRAYRANK) - if the rank requested was greater than 64.
9 (ET_ARRAYSIZE) - if the number of elements and type caused the
number of bytes required for this array to exceed the largest possible
integer value.
```

`->parm1` the locator token of the array

`->parm2` the address of the start of the array (descriptor section)

© Copyright IBM Corporation 1994, 2017

->parm3 the address of the start of the data section

Notes:

1. This service allocates an array in the workspace. If later on it is decided not to use the array as part of the routine result, or the routine returns with an error set in ->request, it should be freed with FREESPACE before returning.
2. The array descriptor fields are filled in for you, except where the array is rank 2 or higher. In that case, the dim[] fields are not filled in, and must be filled in by the function before returning the array to the interpreter.
3. ->left_arg, ->right_arg and ->result are refreshed if garbage collection occurs during this call. If local copies of these addresses exist they may need to be updated.

Example:

```
call->request = ARRAYSPACE;
call->parm1 = 2;
call->parm2 = 1;
call->parm3 = INTEGER;
if (MSG_OK == (call->service)(call)) {
   call->result_token = call->parm1;
   longdata = (LONG *)call->parm3;
   longdata[0] = 0;
   longdata[1] = 75;
}
```

ARRAYRESIZE - Change the number of elements in an allocated array

Parameters:

->request 22 (ARRAYRESIZE)
->parm1 the locator token of the array
->parm2 the new number of elements
->parm3 the new rank

Results:

->request return code - one of:

```
0 (MSG_OK) - if the request was successful.
2 (MSG_SYSTEMERROR) - if the locator token passed is invalid.
3 (MSG_WSFULL) - if there is no room in the workspace for this object.
4 (MSG_SYSTEMLIMIT) - if rank or number of elements is out of range.
```

->reason reason code, if ->request is MSG_SYSTEMLIMIT:

```
8 (ET_ARRAYRANK) - if the rank requested was greater than 64.
9 (ET_ARRAYSIZE) - if the number of elements and type caused the
number of bytes required for this array to exceed the largest possible
integer value.
```

->parm2 the address of the start of the array (descriptor section)
->parm3 the address of the start of the data section

© Copyright IBM Corporation 1994, 2017

Notes:

1. If this service is used to shorten an array, MSG_WSFULL and garbage collection will not happen. The address of the start of the array will not move. The address of the data section will move only if the rank is changed. Any data already in the array will not be moved. If rank is changed the user is responsible for moving data accordingly.
2. If this service is used to lengthen an array, WS_FULL and garbage collection are possible, and the addresses of the start of the array and data section will move. The contents of the old array are copied to the new array, but adjustments are not made for any change in rank.
3. If the rank of the array is 2 or higher, the `dim[]` fields must be filled in by the function before returning the array to the interpreter.
4. `->left_arg`, `->right_arg` and `->result` are refreshed if garbage collection occurs during this call. If local copies of these addresses exist they may need to be updated.

Example:

```
call->request = ARRAYRESIZE;
call->parm1 = call->result_token;
call->parm2 = 3;
call->parm3 = 1;
if (MSG_OK == (call->service)(call)) {
   call->result = call->parm3;
   longdata = (LONG *)call->parm3;
   longdata[2] = 99;
}
```

ARRAYREF - Reference an array in the workspace

Parameters:

`->request` `23 (ARRAYREF)`

`->parm1` the locator token of the array

Results:

`->request` return code - one of:

```
0 (MSG_OK) - if the request was successful.
2 (MSG_SYSTEMERROR) - if the locator token passed is invalid.
```

Notes:

1. This service would be used, for example, if any array in the routine's arguments will be returned as part of the result, or if an array is part of a nested array, and will be used more than once when building the nested array.
2. If later it is decided not to use the array as planned, the FREESPACE service should be used to remove the reference.

Example:

© Copyright IBM Corporation 1994, 2017

```
call->request = ARRAYREF;
call->parm1 = call->right_arg_token;
if (MSG_OK == (call->service)(call)) {
   call->result_token = call->right_arg_token;
}
```

ARRAYCONVERT - Convert an array to a new type

Parameters:

`->request` `24 (ARRAYCONVERT)`

`->parm1` the locator token of the array

`->parm2` the new type, one of:

```
0  (BOOLEAN)
1  (INTEGER)
2  (FLOAT)
3  (COMPLEX)
4  (CHARACTER)
5  (CHARLONG)
6  (APV)
7  (NESTED)
```

Results:

`->request` return code - one of:

```
0 (MSG_OK) - if the request was successful.
2 (MSG_SYSTEMERROR) - if the locator token passed is invalid
or an invalid type is passed.
3 (MSG_WSFULL) - if there is no room in the workspace for this object.
4 (MSG_SYSTEMLIMIT) - if rank or number of elements is out of range.
11 (MSG_DOMAINERROR) - if the array cannot be converted to the new type.
```

`->reason` reason code, if `->request` is `MSG_SYSTEMLIMIT`:

```
8 (ET_ARRAYRANK) - if the rank requested was greater than 64.
9 (ET_ARRAYSIZE) - if the number of elements and type caused the
number of bytes required for this array to exceed the largest possible
integer value.
```

`->parm1` the locator token of the new array

`->parm2` the address of the start of the array (descriptor section)

`->parm3` the address of the start of the data section

Notes:

1. Not all type conversions are possible. For example, numeric arrays cannot be converted to character, integer arrays can only be converted to Boolean if all the values are 0 or 1, and most nested arrays cannot be converted to a non-nested type. This service makes validity checks to ensure that the data will be representable in the new type, and returns MSG_DOMAINERROR if the conversion is not possible.
2. This service allocates a new array. WS_FULL and garbage collection are possible. The old array is not deleted. It is still present, and it, along with any other workspace objects, can move during this call.
3. If the new array allocated by this service will not be used as part of the routine result, or the routine returns with an error set in `->request`, it should be freed with FREESPACE before returning.

© Copyright IBM Corporation 1994, 2017

4. `->left_arg`, `->right_arg` and `->result` are refreshed if garbage collection occurs during this call. If local copies of these addresses exist they may need to be updated.

Example:

```
call->request = ARRAYCONVERT;
call->parm1 = call->right_arg_token;
call->parm2 = INTEGER;
if (MSG_OK == (call->service)(call)) {
   longdata = (LONG *)call->parm3;
}
```

FREESPACE - Remove a reference to an array

Parameters:

`->request` `2 (FREESPACE)`

`->parm1` the locator token of the array

Results:

`->request` return code - one of:

```
0 (MSG_OK) - if the request was successful.
2 (MSG_SYSTEMERROR) - if the locator token passed is invalid.
```

Notes:

1. This service removes a reference to an array. If the reference removed is the only remaining reference to that array, the array is then deleted, and the storage it occupied is freed for other uses.
2. This service is used to free arrays allocated with ARRAYSPACE, ARRAYCONVERT and LCDR_TO_ARRAY, and to remove references to arrays added by ARRAYREF.

Example:

```
call->request = FREESPACE;
call->parm1 = call->result_token;
if (MSG_OK == (call->service)(call)) {
   call->result_token = 0;
}
```

EXECUTE_APL_ESTACK - Execute an APL expression or function, return stack on error

`EXECUTE_APL_ESTACK` takes the same arguments and behaves in the same manner as `EXECUTE_APL`. The difference between the two is the type of object returned in `->parm1` when an error occurs.

EXECUTE_APL - Execute an APL expression or function

Parameters:

© Copyright IBM Corporation 1994, 2017

`->request`	`5 (EXECUTE_APL)` or `38 (EXECUTE_APL_ESTACK)`
`->parm1`	the locator token of the expression or function name. If a function, it can be a defined function, system function, primitive function, or the assignment arrow. If the function is the assignment arrow, the left argument is the name of the object to be assigned, and the right argument is the value to be assigned to it.
`->parm2`	the locator token of the right argument, or 0 if none
`->parm3`	the locator token of the left argument, or 0 if none

Note that the expression or function name and arguments are all passed to this service as locator tokens. This means that all required objects must exist in the workspace prior to calling the service. If executing an expression, there must be a character object containing the expression. If executing a function, there must be a character object containing the function name as well as the function itself. Objects can be created in the workspace by several methods:

1. Use of the ARRAYSPACE or LCDR_TO_ARRAY services.
2. Executing expressions which access objects in APL2 namespaces using `⎕NA`.
3. Executing expressions which establish objects using assignment, `⎕FX` or `⎕TF`.
4. Executing an assignment directly with this service.

See Handling APL2 Objects for more information.

Results:

`->request`	one of the message codes defined in Message Codes.
`->reason`	if applicable, one of the message type codes defined in Message Codes.
`->parm1`	If `->request` is `MSG_OK`: the locator token of the result, or 0 if none If `->request` is `MSG_APLERROR`: the locator token of the error message (`EXECUTE_APL`) or error stack (`EXECUTE_APL_ESTACK`)
`->parm2`	If `->request` is `MSG_OK`: the address of the start of the result (descriptor section) If `->request` is `MSG_APLERROR`: the APL event class (first element of `⎕ET`)
`->parm3`	If `->request` is `MSG_OK`: the address of the start of the data section If `->request` is `MSG_APLERROR`: the APL event type (second element of `⎕ET`)

Notes:

1. Functions processed under control of this service operate in the same manner as those processed under control of `⎕EC`, and exhibit the following behavior:
   - System commands are not allowed.
   - Assignment expressions return a value.
   - Requests for quad input are handled the same as quad input under `⎕EC`.
   - Errors generated during processing do not cause suspension of the function being processed. Error codes are returned via the `CALL` structure.
   - Branch escape (→) causes EXECUTE_APL to return.
   - Stop control vectors (`S∆`) are ignored.

© Copyright IBM Corporation 1994, 2017

- An attention signal does not cause suspension; an interrupt signal causes EXECUTE_APL to halt and return control to the caller.

2. If a result is returned in `->parm1`, a new array has been allocated. If the array will not be used as part of the routine result, or the routine returns with an error set in `->request`, it should be freed with FREESPACE before returning.
3. If an error message or error stack is returned in `->parm1`, a new array has been allocated. If the error object will not be passed on with the `MSG_APLERROR` error, or used as part of the routine result, it should be freed with FREESPACE.
4. WS_FULL and garbage collection are possible during the course of APL operations, regardless of whether a result is returned. Any previously allocated arrays may move if garbage collection occurs.
5. `->left_arg`, `->right_arg` and `->result` are refreshed if garbage collection occurs during this call. If local copies of these addresses exist they may need to be updated.

Example:

```
/* Allocate an array for the expression */
call->request = ARRAYSPACE;
call->parm1 = 3;
call->parm2 = 1;
call->parm3 = CHARACTER;
if (MSG_OK == (call->service)(call)) {
   exptoken = call->parm1;
   memcpy((char *)call->parm3,"□TS",3);
   /* Execute the expression */
   call->request = EXECUTE_APL;
   call->parm1 = exptoken;
   call->parm2 = 0;
   call->parm3 = 0;
   (call->service)(call);
   restoken = call->parm1;
   /* Free the expression array */
   call->request = FREESPACE;
   call->parm1 = exptoken;
   (call->service)(call);
}
```

TOKEN_TO_ADDRESS - Get the address of an array

Parameters:

`->request` `11 (TOKEN_TO_ADDRESS)`

`->parm1` the locator token of the array

Results:

`->request` return code - one of:

```
0 (MSG_OK) - if the request was successful.
2 (MSG_SYSTEMERROR) - if the locator token passed is invalid.
```

`->parm2` the address of the start of the array (descriptor section)

`->parm3` the address of the start of the data section

Notes:

© Copyright IBM Corporation 1994, 2017

1. This service is used to obtain addresses for sub-items of nested arrays passed as the left and right arguments to the routine, and to re-address arrays after garbage collection has taken place.
2. The addresses stored in `->left_arg`, `->right_arg` and `->result` are automatically refreshed (from the tokens in `->left_arg_token`, `->right_arg_token` and `->result_token`) by the callback service routine at the end of each callback.

Example:

```
call->request = TOKEN_TO_ADDRESS;
call->parm1 = local_token;
if (MSG_OK == (call->service)(call)) {
   local_desc = call->parm2;
   local_data = call->parm3;
}
```

QUAD_CT - Get the current comparison tolerance

Parameters:

`->request  40 (QUAD_CT)`

Results:

`->request` return code - one of:

```
0 (MSG_OK) - if the request was successful.
2 (MSG_QUADERROR) - if ⎕CT is missing or invalid.
```

`->reason` reason code, if `->request` is `MSG_QUADERROR`:

```
2 (ET_CT)
```

`->parm1` the address of a double containing the current value of `⎕CT`

Example:

```
call->request = QUAD_CT;
if (MSG_OK == (call->service)(call)) {
   local_ct = *(double *)call->parm1;
}
```

QUAD_IO - Get the current index origin

Parameters:

`->request  15 (QUAD_IO)`

Results:

`->request` return code - one of:

```
0 (MSG_OK) - if the request was successful.
2 (MSG_QUADERROR) - if ⎕IO is missing or invalid.
```


© Copyright IBM Corporation 1994, 2017

->reason reason code, if ->request is MSG_QUADERROR:

```
2 (ET_IO)
```

->parm1 the current value of □IO

Example:

```
call->request = QUAD_IO;
if (MSG_OK == (call->service)(call)) {
   local_io = call->parm1;
}
```

LCDR_TO_ARRAY - Convert a linear CDR to an array

Parameters:

->request 18 (LCDR_TO_ARRAY)

->parm2 the address of the buffer containing the CDR

Results:

->request return code - one of:

```
0 (MSG_OK) - if the request was successful.
2 (MSG_SYSTEMERROR) - if the CDR in the buffer is invalid.
3 (MSG_WSFULL) - if there is no room in the workspace for this object.
```

->parm1 the locator token of the allocated array

Notes:

1. This service allocates an array in the workspace. If later on it is decided not to use the array as part of the routine result, or the routine returns with an error set in ->request, it should be freed with FREESPACE before returning.
2. ->left_arg, ->right_arg and ->result are refreshed if garbage collection occurs during this call. If local copies of these addresses exist they may need to be updated.

Example:

```
call->request = LCDR_TO_ARRAY;
call->parm2 = buffer;
if (MSG_OK == (call->service)(call)) {
   call->result_token = call->parm1;
}
```

ARRAY_TO_LCDR - Convert an array to a linear CDR

Parameters:

->request 19 (ARRAY_TO_LCDR)

© Copyright IBM Corporation 1994, 2017

`->parm1` the locator token of the array to be converted

`->parm2` the address of a buffer to receive the CDR.
The first four bytes of the buffer must contain an integer which is the total length of the buffer.

Results:

`->request` return code:

`0 (MSG_OK) - the request was successful.`

`->parm2` the actual length of the CDR.
If the buffer was at least this length, the CDR has been built in the buffer.

Notes:

1. If you want to find out how large a buffer is needed to contain the CDR, call this service with the address of a 4-byte area containing the integer 4. On return, `->parm2` will contain the required buffer size.

Example:

```
call->request = ARRAY_TO_LCDR;
call->parm2 = buffer;
(*(long *)buffer) = buffersize;
(call->service)(call);
if (call->parm2 > buffersize) {
   buffersize = call->parm2;
   buffer = malloc(buffersize);
   call->request = ARRAY_TO_LCDR;
   call->parm2 = buffer;
   (*(long *)buffer) = buffersize;
   (call->service)(call);
}
```

DOAP1EVENTS - Allow AP 1 to handle pending events

Parameters:

`->request  35 (DOAP1EVENTS)`

Results:

`->request` return code:

`0 (MSG_OK) - the request was successful.`

Notes:

1. This service would normally be used periodically by a long-running function, to allow AP 1 to handle pending shared variable interpreter interface requests.

Example:

© Copyright IBM Corporation 1994, 2017

```
call->request = DOAP1EVENTS;
(call->service)(call);
```

GETSIGNALS - Query and/or clear the signal state

Parameters:

`->request` `32 (GETSIGNALS)`

`->parm1` indicates whether pending interrupts should be cleared

```
0 - do not clear pending interrupts
1 - clear interrupts if pending
```

Results:

`->request` return code:

```
0 (MSG_OK) - the request was successful.
```

`->parm1` the signal state:

```
0 - if there are no pending interrupts
2 - if there is a pending interrupt
```

Notes:

1. This service would normally be used periodically by a long-running function, to check whether it should terminate due to interrupts signaled by the APL2 user.
2. The DOAP1EVENTS service should be called before GETSIGNALS to allow new user interrupts to be processed.
3. The signal state returned is the state before any clear of pending interrupt is done.
4. If a function requests that pending interrupts be cleared, it must then take responsibility for that interrupt. The normal procedure is to do any internal cleanup and return with code `MSG_INTERRUPT` so the interpreter will then process the interrupt normally.

Example:

```
call->request = GETSIGNALS;
call->parm1 = 1;
(call->service)(call);
if (call->parm1 == 2) {
   fncleanup();
   call->request = MSG_INTERRUPT;
   call->reason = 0;
   return;
}
```

Message Codes

The following message codes are defined as return codes from the `apl2pi` and external routine callback services, and for use as return codes from external routines to the interpreter. The message codes correspond to

© Copyright IBM Corporation 1994, 2017

the interpreter event type codes used for ⎕ET, and will be mapped to those codes on return to the interpreter. The message codes are placed in the `->request` field of the `CALL` block.

Some messages have type, or reason, codes. When applicable, the type codes are placed in the `->reason` field of the `CALL` block.

```
0  MSG_OK
1  MSG_INTERRUPT
2  MSG_SYSTEMERROR
3  MSG_WSFULL
4  MSG_SYSTEMLIMIT

     4     ET_SYMBOLTABLE
     5     ET_INTERFACENA
     6     ET_INTERFACEQUOTA
     7     ET_INTERFACECAPACITY
     8     ET_ARRAYRANK
     9     ET_ARRAYSIZE
     10    ET_ARRAYDEPTH
     11    ET_PROMPTLENGTH
     12    ET_INTERFACEREP
     13    ET_IMPLEMENTATION
5  MSG_SYNTAXERROR

     1     ET_OMITTED
     2     ET_ILLFORMED
     3     ET_NAMECLASS
     4     ET_CONTEXT
     5     ET_COMPATIBILITY
6  MSG_VALUEERROR

     1     ET_NOVALUE
     2     ET_NORESULT
7  MSG_QUADERROR

     1     ET_PP
     2     ET_IO
     3     ET_CT
     4     ET_FC
     5     ET_RL
     7     ET_PR
8  MSG_VALENCEERROR
9  MSG_RANKERROR
10 MSG_LENGTHERROR
11 MSG_DOMAINERROR
12 MSG_INDEXERROR
13 MSG_AXISERROR
99 MSG_APLERROR
```

MSG_APLERROR

The `MSG_APLERROR` code is used to receive or return APL error information.

© Copyright IBM Corporation 1994, 2017

On return from the EXECUTE_APL and EXECUTE_APL_ESTACK services, this code indicates that an APL error occurred during processing of the APL expression or function. When MSG_APLERROR is received from the interpreter, the following additional error information is received:

->parm1	For EXECUTE_APL, the locator token of the error message (a character vector, the first row of ⎕EM). For EXECUTE_APL_ESTACK, the locator token of the error stack (a character matrix, in the same format as the third element of the result from ⎕EC).
->parm2	the APL event class (first element of ⎕ET)
->parm3	the APL event type (second element of ⎕ET)

When an external routine returns to APL2, this code may be used by the routine to signal a non-standard APL error. When sending MSG_APLERROR to the interpreter, the following fields are defined for the error information (corresponding to the arguments of ⎕ES.)

->parm1	the locator token of the error message (must be a character vector), or 0 if none
->parm2	the APL event class
->parm3	the APL event type

To signal a numeric event code:

Set ->parm1 to 0.

Set ->parm2 and ->parm3 to the error codes.

To signal a character event message:

Allocate a character vector object containing the message in the workspace.

Set ->parm1 to the locator token of the object.

Set ->parm2 to 0 and ->parm3 to 1.

To signal both a numeric event code and a character event message:

Allocate a character vector object containing the message in the workspace.

Set ->parm1 to the locator token of the object.

Set ->parm2 and ->parm3 to the error codes.

Note: An external routine may wish to return upon receiving error codes, passing those same error codes back to the interpreter. If the error code is MSG_APLERROR, care must be taken to preserve the values in the ->parm1, ->parm2 and ->parm3 fields. If the CALL block will be used to call other services before returning, these fields, along with ->request and ->reason, should be saved and restored so that all information associated with the error is returned.

Sample Routines

The following set of routines are samples of :link.FUNCTION routines written in C.

```
/*-------------------------------------------------------------------*/
/* Include files                                                     */
/*-------------------------------------------------------------------*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "aplfun.h"
#include "aplobj.h"
```


© Copyright IBM Corporation 1994, 2017

```
/*--------------------------------------------------------------------*/
/*                                                                    */
/* Function: funsample1                                               */
/*                                                                    */
/* Routine descriptor:                                                */
/*                                                                    */
/*       :nick.NEST      :link.FUNCTION                               */
/*           :lib.aplfun    :proc.funsample1                          */
/*           :valence.1 2 0                                           */
/*                                                                    */
/* Purpose:  This routine simply returns a nested array whose         */
/*           elements are copies of the left and right argument.      */
/*           Demonstrates ARRAYSPACE, ARRAYREF.                       */
/*                                                                    */
/*--------------------------------------------------------------------*/
int _System
funsample1(CALL * call)      /* EXPORT */
{
   long * dataobj;
   /*-----------------------------------------------------------------*/
   /* Handle INIT and TERM cases                                      */
   /*-----------------------------------------------------------------*/
   switch (call->request) {
      case FUNCTION_INIT:
         call->request = MSG_OK;
         call->reason = MSG_OK;
         return 0;
      case FUNCTION_CALL:
         break;
      case FUNCTION_TERM:
         call->request = MSG_OK;
         call->reason = MSG_OK;
         return 0;
      default:
         call->request = MSG_SYSTEMERROR;
         call->reason = MSG_OK;
         return 0;
   }
   /*-----------------------------------------------------------------*/
   /* FUNCTION_CALL - main logic for function                         */
   /*-----------------------------------------------------------------*/
   /*-----------------------------------------------------------------*/
   /* Make sure there are both left and right arguments               */
   /*-----------------------------------------------------------------*/
   if ((call->left_arg_token == 0) || (call->right_arg_token == 0)) {
      call->request = MSG_VALENCEERROR;
      call->reason = 0;
      return 0;
   }
   /*-----------------------------------------------------------------*/
   /* Allocate a nested array for the result                          */
   /*-----------------------------------------------------------------*/
   call->request = ARRAYSPACE;
   call->parm1 = 2;
   call->parm2 = VECTOR;
   call->parm3 = NESTED;
   if ((call->service)(call)) return 0;
   call->result_token = call->parm1;
   call->result = (void *) call->parm2;
   /*-----------------------------------------------------------------*/
   /* Set the array data to the tokens for the left and right arg     */
   /*-----------------------------------------------------------------*/
   dataobj = (long *) call->parm3;
   dataobj[0] = call->left_arg_token;
   dataobj[1] = call->right_arg_token;
   /*-----------------------------------------------------------------*/
   /* Reference the left and right arg                                */
   /*-----------------------------------------------------------------*/
   call->request = ARRAYREF;
   call->parm1 = call->left_arg_token;
```


© Copyright IBM Corporation 1994, 2017

```
   ((call->service)(call));
   call->request = ARRAYREF;
   call->parm1 = call->right_arg_token;
   ((call->service)(call));
   call->request = MSG_OK;
   call->reason = MSG_OK;
   return 0;
}
/*-------------------------------------------------------------------*/
/*                                                                   */
/* Function: funsample2                                              */
/*                                                                   */
/* Routine descriptor:                                               */
/*                                                                   */
/*       :nick.REMOVEFROM    :link.FUNCTION                          */
/*           :lib.aplfun    :proc.funsample2                        */
/*           :valence.1 2 0                                          */
/*                                                                   */
/* Purpose:  This routine removes the left argument items            */
/*           from the right.                                         */
/*           Demonstrates ARRAYSPACE, ARRAYRESIZE.                   */
/*                                                                   */
/*-------------------------------------------------------------------*/
int _System
funsample2(CALL * call)      /* EXPORT */
{
   APLOBJ *objleft, *objright, *objres;
   char *dataleft, *dataright, *datares;
   int left, right, res;
   /*----------------------------------------------------------------*/
   /* Handle INIT and TERM cases                                     */
   /*----------------------------------------------------------------*/
   switch (call->request) {
      case FUNCTION_INIT:
         call->request = MSG_OK;
         call->reason = MSG_OK;
         return 0;
      case FUNCTION_CALL:
         break;
      case FUNCTION_TERM:
         call->request = MSG_OK;
         call->reason = MSG_OK;
         return 0;
      default:
         call->request = MSG_SYSTEMERROR;
         call->reason = MSG_OK;
         return 0;
   }
   /*----------------------------------------------------------------*/
   /* FUNCTION_CALL - main logic for function                        */
   /*----------------------------------------------------------------*/
   /*----------------------------------------------------------------*/
   /* Make sure there are both left and right arguments              */
   /*----------------------------------------------------------------*/
   if ((call->left_arg_token == 0) || (call->right_arg_token == 0)) {
      call->request = MSG_VALENCEERROR;
      call->reason = 0;
      return 0;
   }
   /*----------------------------------------------------------------*/
   /* Make sure the left argument is what we need                    */
   /*----------------------------------------------------------------*/
   objleft = (APLOBJ *) call->left_arg;
   if (objleft->type != CHARACTER) {
      call->request = MSG_DOMAINERROR;
      call->reason = 0;
      return 0;
   }
   if (objleft->rank > VECTOR) {
      call->request = MSG_RANKERROR;
```


© Copyright IBM Corporation 1994, 2017

```
   call->reason = 0;
   return 0;
}
/*-------------------------------------------------------------------*/
/* Make sure the right argument is what we need                      */
/*-------------------------------------------------------------------*/
objright = (APLOBJ *) call->right_arg;
if (objright->type != CHARACTER) {
   call->request = MSG_DOMAINERROR;
   call->reason = 0;
   return 0;
}
if (objright->rank > VECTOR) {
   call->request = MSG_RANKERROR;
   call->reason = 0;
   return 0;
}
/*-------------------------------------------------------------------*/
/* Allocate an array for the result. Assume nothing will be          */
/* removed.                                                          */
/*-------------------------------------------------------------------*/
call->request = ARRAYSPACE;
call->parm1 = objright->nelm;
call->parm2 = VECTOR;
call->parm3 = CHARACTER;
if ((call->service)(call)) return 0;
call->result_token = call->parm1;
objres = (APLOBJ *) call->parm2;
datares = (char *) call->parm3;
/*-------------------------------------------------------------------*/
/* Refresh the left and right arg addresses in case they moved       */
/* during the call to ARRAYSPACE.                                    */
/*-------------------------------------------------------------------*/
objleft = call->left_arg;
objright = call->right_arg;
dataleft = (char *) &objleft->dim[objleft->rank];
dataright = (char *) &objright->dim[objright->rank];
/*-------------------------------------------------------------------*/
/* Copy the data (first pass is into the result from the right)      */
/*-------------------------------------------------------------------*/
for (right=0, res = 0; right < objright->nelm; right++) {
   if (dataright[right] != dataleft[0]) {
      datares[res] = dataright[right];
      res++;
   }
 }
 objres->nelm = res;
/*-------------------------------------------------------------------*/
/* For rest of left, do it in place  (use res as right)              */
/*-------------------------------------------------------------------*/
objright = objres;
dataright = datares;
for (left = 1; left nelm; left++) {
   for (right=0, res = 0; right < objright->nelm; right++) {
      if (dataright[right] != dataleft[left]) {
         datares[res] = dataright[right];
         res++;
      }
    }
    objres->nelm = res;
}
/*-------------------------------------------------------------------*/
/* Shorten the result to its actual size                             */
/* Return ->request and ->reason as set by ARRAYRESIZE               */
/* Chance of error is minimal since we are shortening                */
/*-------------------------------------------------------------------*/
call->request = ARRAYRESIZE;
call->parm1 = call->result_token;
call->parm2 = res;
call->parm3 = VECTOR;
```


© Copyright IBM Corporation 1994, 2017

```
   (call->service)(call);
   return 0;
}
/*-------------------------------------------------------------------*/
/*                                                                   */
/* Function: funsample3                                              */
/*                                                                   */
/* Routine descriptor:                                               */
/*                                                                   */
/*       :nick.TYPE      :link.FUNCTION                              */
/*           :lib.aplfun    :proc.funsample3                         */
/*           :valence.1 1 0                                          */
/*                                                                   */
/* Purpose:  This routine returns the type of the argument.          */
/*           (If simple, returns type, If nested, returns type       */
/*            if each subitem). Builds array in workspace.           */
/*           Demonstrates ARRAYSPACE, TOKEN_TO_ADDRESS,              */
/*           FREESPACE.                                              */
/*                                                                   */
/*-------------------------------------------------------------------*/
int _System
funsample3(CALL * call)      /* EXPORT */
{
   APLOBJ *input, *subinput, *result;
   long *restok, *inptok;
   char *typestr;
   int i;
   long reloc_count;
   /*----------------------------------------------------------------*/
   /* Handle INIT and TERM cases                                     */
   /*----------------------------------------------------------------*/
   switch (call->request) {
      case FUNCTION_INIT:
         call->request = MSG_OK;
         call->reason = MSG_OK;
         return 0;
      case FUNCTION_CALL:
         break;
      case FUNCTION_TERM:
         call->request = MSG_OK;
         call->reason = MSG_OK;
         return 0;
      default:
         call->request = MSG_SYSTEMERROR;
         call->reason = MSG_OK;
         return 0;
   }
   /*----------------------------------------------------------------*/
   /* FUNCTION_CALL - main logic for function                        */
   /*----------------------------------------------------------------*/
   /*----------------------------------------------------------------*/
   /* Make sure there is not a left argument and is a right argument */
   /*----------------------------------------------------------------*/
   if ((call->left_arg_token != 0) || (call->right_arg_token == 0)) {
      call->request = MSG_VALENCEERROR;
      call->reason = 0;
      return 0;
   }
   /*----------------------------------------------------------------*/
   /* Make sure the right argument meets criteria                    */
   /*----------------------------------------------------------------*/
   input = (APLOBJ *) call->right_arg;
   if (input->rank > VECTOR) {
      call->request = MSG_RANKERROR;
      call->reason = 0;
      return 0;
   }
   if (input->nelm == 0) {
      call->request = MSG_LENGTHERROR;
      call->reason = 0;
```


© Copyright IBM Corporation 1994, 2017

```
   return 0;
}
/*-------------------------------------------------------------------*/
/* Allocate the top-level result array                               */
/*-------------------------------------------------------------------*/
call->request = ARRAYSPACE;
if (input->type == NESTED)
   call->parm1 = input->nelm;
else
   call->parm1 = 1;
call->parm2 = VECTOR;
call->parm3 = NESTED;
if ((call->service)(call)) return 0;
call->result_token = call->parm1;
call->result = (void *) call->parm2;
result = (APLOBJ *) call->parm2;
restok = (long *) call->parm3;
/*-------------------------------------------------------------------*/
/* Recopy right arg pointer in case it moved during ARRAYSPACE       */
/*-------------------------------------------------------------------*/
input = (APLOBJ *) call->right_arg;
inptok = (long *) &input->dim[input->rank];
/*-------------------------------------------------------------------*/
/* Initialize the token pointers to 0                                */
/*-------------------------------------------------------------------*/
for (i = 0; i < result->nelm; i++)  restok[i] = 0;
/*-------------------------------------------------------------------*/
/* For each item in the input, allocate an array subitem             */
/*-------------------------------------------------------------------*/
reloc_count = call->reloc_count;
for (i = 0; i < result->nelm; i++)  {
   if (input->type == NESTED) {
      call->request = TOKEN_TO_ADDRESS;
      call->parm1 = inptok[i];
      if ((call->service)(call)) break;
      subinput = (APLOBJ *) call->parm2;
   }
   else {
      subinput = input;
   }
   switch (subinput->type) {
      case BOOLEAN:
         typestr = "BOOLEAN";
         break;
      case INTEGER:
         typestr = "INTEGER";
         break;
      case FLOAT:
         typestr = "FLOAT";
         break;
      case COMPLEX:
         typestr = "COMPLEX";
         break;
      case CHARACTER:
         typestr = "CHARACTER";
         break;
      case CHARLONG:
         typestr = "LONG CHARACTER";
         break;
      case APV:
         typestr = "PROGRESSION";
         break;
      case NESTED:
         typestr = "NESTED";
         break;
      default:
         typestr = "UNKNOWN";
         break;
   }
   call->request = ARRAYSPACE;
```


© Copyright IBM Corporation 1994, 2017

```
      call->parm1 = strlen(typestr);
      call->parm2 = VECTOR;
      call->parm3 = CHARACTER;
      if ((call->service)(call)) break;
      /*-----------------------------------------------------------------*/
      /* Recopy right arg and result addresses if needed                 */
      /*-----------------------------------------------------------------*/
      if (call->reloc_count != reloc_count) {
         input = (APLOBJ *) call->right_arg;
         inptok = (long *) &input->dim[input->rank];
         result = (APLOBJ *) call->result;
         restok = (long *) &result->dim[result->rank];
         reloc_count = call->reloc_count;
      }
      /*-----------------------------------------------------------------*/
      /* Set the data sections of the main array and subarray            */
      /*-----------------------------------------------------------------*/
      restok[i] = call->parm1;
      memcpy((char *)call->parm3, typestr, strlen(typestr));
   }
   /*--------------------------------------------------------------------*/
   /* If anything failed, back it all out                                */
   /*--------------------------------------------------------------------*/
   if (i < result->nelm) {
      int rc = call->request;       /* Save rc and reason for return */
      int reas = call->reason;
      for (i = 0; i < result->nelm; i++) {      /* Free each subitem */
         if (restok[i] != 0) {
            call->request = FREESPACE;
            call->parm1 = restok[i];
            (call->service)(call);
         }
      }
      call->request = FREESPACE;               /* Free main array    */
      call->parm1 = call->result_token;
      (call->service)(call);
      call->result_token = 0;
      call->request = rc;
      call->reason = reas;
   }
   return 0;
}
/*--------------------------------------------------------------------*/
/*                                                                    */
/* Function: funsample4                                               */
/*                                                                    */
/* Routine descriptor:                                                */
/*                                                                    */
/*       :nick.TABLE     :link.FUNCTION                               */
/*           :lib.aplfun    :proc.funsample4                          */
/*           :valence.1 1 0                                           */
/*                                                                    */
/* Purpose:  This routine returns a table of data.                    */
/*           The argument is the number of rows to return.            */
/*           It builds the array in routine storage.                  */
/*           Demonstrates ARRAYCONVERT, LCDR_TO_ARRAY, and            */
/*           the use of call->token.                                  */
/*                                                                    */
/*--------------------------------------------------------------------*/
int _System
funsample4(CALL * call)      /* EXPORT */
{
   APLOBJ *right;
   long rows;
   APLOBJ *result, *subitem;
   long *datanest;
   long *dataint;
   char *datachar;
   double *datareal;
   int i;
```


© Copyright IBM Corporation 1994, 2017

```
char *names[5] = {"Doug Aiton",
                  "Roy Bei",
                  "David Liebtag",
                  "Ed Oddo",
                  "Nancy Wheeler"};
double extension[5] = {3.3590,3.4632,3.2739,3.4167,3.4031};
/*-------------------------------------------------------------------*/
/* Handle INIT and TERM cases                                        */
/*-------------------------------------------------------------------*/
switch (call->request) {
   case FUNCTION_INIT:
      call->token = (long) malloc(4096);
      if (call->token == NULL) {
         call->request = MSG_SYSTEMLIMIT;
         call->reason = ET_INTERFACECAPACITY;
      }
      else {
         call->request = MSG_OK;
         call->reason = MSG_OK;
      }
      return 0;
   case FUNCTION_CALL:
      break;
   case FUNCTION_TERM:
      free((void *)call->token);
      call->request = MSG_OK;
      call->reason = MSG_OK;
      return 0;
   default:
      call->request = MSG_SYSTEMERROR;
      call->reason = MSG_OK;
      return 0;
}
/*-------------------------------------------------------------------*/
/* FUNCTION_CALL - main logic for function                           */
/*-------------------------------------------------------------------*/
/*-------------------------------------------------------------------*/
/* Make sure there is not a left argument and is a right argument */
/*-------------------------------------------------------------------*/
if ((call->left_arg_token != 0) || (call->right_arg_token == 0)) {
   call->request = MSG_VALENCEERROR;
   call->reason = 0;
   return 0;
}
/*-------------------------------------------------------------------*/
/* Process the right argument                                        */
/*-------------------------------------------------------------------*/
right = (APLOBJ *) call->right_arg;
if (right->rank > VECTOR) {
   call->request = MSG_RANKERROR;
   call->reason = 0;
   return 0;
}
if (right->nelm != 1) {
   call->request = MSG_LENGTHERROR;
   call->reason = 0;
   return 0;
}
if (right->type == INTEGER)  {      /* Already have what we want */
   rows = right->dim[right->rank];
}
else {                              /* Try to get it to integer   */
   call->request = ARRAYCONVERT;
   call->parm1 = call->right_arg_token;
   call->parm2 = INTEGER;
   if ((call->service)(call)) {     /* conversion impossible      */
      return 0;  /* ->request and ->reason set by ARRAYCONVERT    */
   }
   rows = *((long *)call->parm3);   /* copy the data we want      */
   call->request = FREESPACE;       /* and now we can free it     */
```


© Copyright IBM Corporation 1994, 2017

```
   (call->service)(call);           /* ->parm1 already set        */
}
if (rows < 0) rows = 0;
if (rows > 5) rows = 5;
/*-------------------------------------------------------------------*/
/* Create the APL array in the buffer                                */
/*-------------------------------------------------------------------*/
result = (APLOBJ *) call->token;
result->ptr = CDRID;
result->nb = ObjSize(VECTOR) +           /* Descriptor               */
             (4 * sizeof(long));         /* Data                     */
result->nb = ((result->nb+15)>>4)<nelm = 4;
result->type = NESTED;
result->rank = VECTOR;
result->fill[0] = 0x0;
result->fill[1] = 0x0;
result->dim[0] = 4;
datanest = (long *) &result->dim[result->rank];
/*-------------------------------------------------------------------*/
/* Create the first subitem                                          */
/*-------------------------------------------------------------------*/
datanest[0] = result->nb;                /* Offset to this item      */
subitem = (APLOBJ *) (result->nb + (char *)result); /* This item     */
subitem->ptr = CDRID;
subitem->nb = ObjSize(MATRIX) +          /* Descriptor               */
              (rows * sizeof(long));     /* Data                     */
subitem->nb = ((subitem->nb+15)>>4)<nelm = rows;
subitem->type = INTEGER;
subitem->rank = MATRIX;
subitem->fill[0] = 0x0;
subitem->fill[1] = 0x0;
subitem->dim[0] = rows;
subitem->dim[1] = 1;
dataint = (long *) &subitem->dim[subitem->rank];
for (i = 0; i < rows; i++) {
   dataint[i] = i + 1;
}
result->nb += subitem->nb;               /* Update total size        */
/*-------------------------------------------------------------------*/
/* Create the second subitem                                         */
/*-------------------------------------------------------------------*/
datanest[1] = result->nb;                /* Offset to this item      */
subitem = (APLOBJ *) (result->nb + (char *)result); /* This item     */
subitem->ptr = CDRID;
subitem->nb = ObjSize(MATRIX) +          /* Descriptor               */
              (rows * 4);                /* Data                     */
subitem->nb = ((subitem->nb+15)>>4)<nelm = rows * 4;
subitem->type = CHARACTER;
subitem->rank = MATRIX;
subitem->fill[0] = 0x0;
subitem->fill[1] = 0x0;
subitem->dim[0] = rows;
subitem->dim[1] = 4;
datachar = (char *) &subitem->dim[subitem->rank];
memcpy(datachar,"D122D132D121F326D127", rows*4);
result->nb += subitem->nb;               /* Update total size        */
/*-------------------------------------------------------------------*/
/* Create the third subitem                                          */
/*-------------------------------------------------------------------*/
datanest[2] = result->nb;                /* Offset to this item      */
subitem = (APLOBJ *) (result->nb + (char *)result); /* This item     */
subitem->ptr = CDRID;
subitem->nb = ObjSize(MATRIX) +          /* Descriptor               */
              (rows * sizeof(double));   /* Data                     */
subitem->nb = ((subitem->nb+15)>>4)<nelm = rows;
subitem->type = FLOAT;
subitem->rank = MATRIX;
subitem->fill[0] = 0x0;
subitem->fill[1] = 0x0;
subitem->dim[0] = rows;
```


© Copyright IBM Corporation 1994, 2017

```
   subitem->dim[1] = 1;
   datareal = (double *) &subitem->dim[subitem->rank];
   for (i = 0; i < rows; i++) {
      datareal[i] = extension[i];
   }
   result->nb += subitem->nb;               /* Update total size     */
   /*-------------------------------------------------------------*/
   /* Create the fourth subitem                                   */
   /*-------------------------------------------------------------*/
   datanest[3] = result->nb;                /* Offset to this item   */
   subitem = (APLOBJ *) (result->nb + (char *)result); /* This item  */
   subitem->ptr = CDRID;
   subitem->nb = ObjSize(MATRIX) +          /* Descriptor            */
                 (rows * 20);               /* Data                  */
   subitem->nb = ((subitem->nb+15)>>4)<nelm = rows * 20;
   subitem->type = CHARACTER;
   subitem->rank = MATRIX;
   subitem->fill[0] = 0x0;
   subitem->fill[1] = 0x0;
   subitem->dim[0] = rows;
   subitem->dim[1] = 20;
   datachar = (char *) &subitem->dim[subitem->rank];
   memset(datachar,' ', rows*20);
   for (i = 0; i < rows; i++) {
      memcpy(datachar,names[i],strlen(names[i]));
      datachar += 20;
   }
   result->nb += subitem->nb;               /* Update total size     */
   /*-------------------------------------------------------------*/
   /* Put the result in the workspace                             */
   /* If it fails call->parm1 will be 0, return codes already set */
   /*-------------------------------------------------------------*/
   call->request = LCDR_TO_ARRAY;
   call->parm2 = (long) result;
   (call->service)(call);
   call->result_token = call->parm1;
   return 0;
}
```


© Copyright IBM Corporation 1994, 2017

Writing Your Own Auxiliary Processors

The following topics discuss how to write your own auxiliary processors. Auxiliary processors can be written either in APL or in C.

- The basic interface for communicating between an APL auxiliary processor and the shared variable processor is the set of shared variable system functions and variables (`⎕SVC`, `⎕SVE`, `⎕SVO`, `⎕SVQ`, `⎕SVR`, and `⎕SVS`) defined in *APL2 Programming: Language Reference*.
- The basic interface for communicating between a C auxiliary processor and the shared variable processor is a set of service calls defined in SVP Programming Interface.

  The data structures used by this interface are defined in Common Data Representation

- A higher level APSERVER interface to the shared variable processor is also available for auxiliary processors written in either APL or C. This is defined in Writing Auxiliary Processors Using APSERVER.

Note: This chapter shows sample code and excerpts from C include files to be used when writing auxiliary processors. These samples and include files can be found in the subdirectories `\samples` and `\include` (Windows) or `/examples/svp` and `/include` (Unix) under the main APL2 installation directory.

© Copyright IBM Corporation 1994, 2017

Writing Auxiliary Processors Using APSERVER

The Shared Variable Processor (SVP) is a general purpose communication facility that can support various protocols, including peer-to-peer, client-server, full or half duplex, or uni-directional data flow for device drivers or instrumentation. However, the most common type of auxiliary processor (AP) is the server, which accepts requests from one or more client processors, performs some action, and returns a response.

The purpose of the APSERVER API (application program interface) is to simplify the implementation of server APs by handling all of the SVP communication and process control, leaving you to concentrate on just the AP-specific service routine. Both an APL2 function call interface and a C function call interface are provided so client-server APs can be easily written in either APL or in compiled languages such as C or Fortran.

Although the APSERVER can be used to implement dual-variable APs (typically a control and data variable), IBM recommends that a single variable interface be used for simplicity. By using APL2's general array support, it is easy to pass control information and its accompanying data through a single variable. APSERVER provides access to client processor information and shared variable name to support share restrictions when appropriate.

A server AP can run as either a dependent or independent processor, local or remote, and can be automatically started on the first incoming share offer. When using the C API, each incoming share spawns a separate execution thread to achieve the maximum benefit of multitasking or multiprocessing.

APSERVER allows common APs to be written to run on different operating systems (for example, Windows and AIX) by handling platform-dependent functions, such as process threading and event handling.

- APSERVER - APL2 Programming Interface
- APSERVER - C Programming Interface

Note: The examples shown in the following sections assume that the auxiliary processors created will be started automatically as dependent processors under the interpreter session that issues the offer to them. To start the processor independently, manually invoke the AP, passing the `-id` invocation parameter to assign the AP number. Independent processors must use an id number greater than 1000. For more information on independent vs. dependent processors, see Sending a Share Offer

APSERVER - APL2 Programming Interface

To use the APSERVER function to implement an AP written in APL2, first copy the function from the distributed library:

```
)COPY 1 UTILITY APSERVER
```

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

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

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

© Copyright IBM Corporation 1994, 2017

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

```
RC←Init_fn PID SVNAME
```

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

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

The second name in the APSERVER argument list is the name of the wait callback function. If no wait routine is supplied the default action of the APSERVER is to enter a `□SVE` wait for any shared variable event, then scan for new offers, new client requests, or shared variable retractions. The `Wait_fn` function, if provided, must be a niladic function with no explicit result. You may wish to provide your own wait function to issue `□SVE` so that you can check the state of other shared variables used for your own purposes, or so that you can provide a time-out on the `□SVE` wait (for example, to do some administrative work such as journaling). When you supply wait routine exits, the APSERVER performs the usual checking for client events.

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

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

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

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

```
Exit_fn PID SVNAME
```

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

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

An APL2 Example Using APSERVER

© Copyright IBM Corporation 1994, 2017

This example shows the creation of a .bat file for starting the AP on Windows. On Unix systems, a shell script can be used instead. A sample shell script is found in `/usr/APL2/examples/svp/ap555`.

```
      )CLEAR
CLEAR WS
      )COPY 1 UTILITY APSERVER ID SVOFFER
SAVED 1996-06-20 19.07.51 (GMT-8)
      ∇ap555
[1]   ⍝ Sample AP using 'init' and 'exit' callbacks.  This AP uses AP207
[2]   ⍝ to create a window where client requests are randomly displayed.
[3]   ⍝ Returns 'OK' if write to window was successful, or 'OOPS' if not.
[4]    APSERVER 'INIT555' '' 'PROC555' 'EXIT555'
[5]   ∇
      ∇R←INIT555 SVinfo;N;PID;SVNAME
[1]    (PID SVNAME)←SVinfo
[2]    →(~R←2=↑207 SVOFFER N←SVNAME,'207')/0
[3]    ⍎N,'←''OPEN'' (0 ''',(⍕∈ID PID),' ',SVNAME,''' ',(⍕200+?4⍴300),')'
[4]    ⍎N,'←''COLOR'' ''CYAN'''
[5]    →(R←0=↑⍎N)/0
[6]    N←⎕EX N
[7]   ∇
      ∇R←SVinfo PROC555 String;N;PID;SVNAME
[1]    (PID SVNAME)←SVinfo
[2]    N←SVNAME,'207'
[3]    ⍎N,'←''MOVE'' (?2⍴150)'
[4]    ⍎N,'←''WRITE'' ''',String,''''
[5]    R←∈↑(0=↑⍎N)↓'OOPS' 'OK'
[6]   ∇
      ∇EXIT555 SVinfo;N;PID;SVNAME
[1]    (PID SVNAME)←SVinfo
[2]    N←SVNAME,'207'
[3]    ⍎N,'←''CLOSE'' '''''
[4]    N←⎕EX N
[5]   ∇
      )FNS
APSERVER EXIT555  ID       INIT555  PROC555  SVOFFER  ap555
      )SAVE AP555
2002-02-12 13.27.30 (GMT-8) AP555
      3 10 ⎕NA 'ΔFV'
1
      (⍞ ⍞ ⍞)ΔFV 'AP555.BAT'
@ECHO OFF
REM Windows .BAT file to start AP 555
apl2win -sm off -input "')LOAD AP555' 'ap555' ')OFF'" %1 %2 %3 %4 %5 %6 %7 %8 %9
0
      )CLEAR
CLEAR WS
      555 ⎕SVO 'X'
1
      ⎕SVO 'X'
2
      ⎕SVC 'X'
0 0 0 1
      1 0 1 0 ⎕SVC 'X'
1 0 1 1
      X←'SAY HI'
      X
OK
      ⎕SVR 'X'
2
```

APSERVER - C Programming Interface

© Copyright IBM Corporation 1994, 2017

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

```
int apserver(int argc, char ** argv, int(*initfn)(void *),
                                     int(*waitfn)(void *),
                                     int(*procfn)(void *),
                                     int(*exitfn)(void *));
```

The first two arguments passed to APSERVER are the argc and argv parameters that were passed to the AP main routine. The APSERVER uses this information to determine the processor identification for signing on to the SVP. It scans for the keyword `-id` followed by one to three numeric values, with the form `id[,pid[,ppid]]`, representing the ID and, optionally, the parent and grandparent IDs for dependent processors. If no `-id` parameter is found, APSERVER uses the numeric portion of the executable module's name (of the form `ap`**`nnn`**) to determine the default processor number.

The next four arguments are the names of your supplied callback functions. These callback routines are described as follows:

`initfn` AP initialization (prior to accepting client's share)
`waitfn` Multiple event wait and non-SVP event handling
`procfn` Process a client request (the meat of the AP)
`exitfn` AP clean-up (just prior to retraction of the share)

Each of these callback routines receives a single argument that is a pointer to an SRVTOKEN structure. This token is used by subsequent APSERVER service routines that are described later. All callback functions return an integer return code, with zero indicating success.

In each case, if no callback routine is supplied, a NULL pointer must be provided instead. If a NULL value is specified for all four callback functions, the APSERVER functions as an echo AP, reflecting the client processor's request value back to the client as the server's response value.

For example, the following C function represents the simplest form of a fully functional AP (default `echo` AP), supporting multiple clients, multiple shares, and multitasking on a per-share basis:

```
/* ap555 - echo AP (source module: ap555.c, executable module: ap555) */
#include "apserver.h"
int main(int argc, char **argv) {
return(apserver(argc, argv, NULL, NULL, NULL, NULL)); }
```

If the executable ap555 module is placed in the search path for an APL2 session, the first share offer to processor 555 causes the AP to be auto-started as a dependent AP of the APL2 session. Alternatively, the AP could be started explicitly and may run as an independent processor. Optionally, by use of the TCP/IP processor profile, the server AP can also be accessed by remote client processors.

- Defining the Process Callback Function
- Defining the Init Callback Function
- Defining the Exit Callback Function

© Copyright IBM Corporation 1994, 2017

- Defining the Wait Callback Function
- Execution Environment and Exception Handling
- A C Example Using APSERVER

Defining the Process Callback Function

To be of more practical use, an AP should serve as more than just a reflector of client requests. That is where the "process" callback function comes into play. The process routine name is passed as the fifth argument to APSERVER. On entry to the process routine, the client's request value is stored in a shared variable buffer that is accessed by two APSERVER macros: SRVBUF and SRVBUFL. Note that when passing APL2 arrays, SRVBUFL is typically not required because the size of the array is defined in the self-describing array header (see Common Data Representation).

All APSERVER macros take the SRVTOKEN as an argument. A brief summary of the service macros follows:

```
       (void *)SRVBUF(srvtoken)  - pointer to shared variable buffer
(unsigned long)SRVBUFL(srvtoken) - length of shared variable buffer
 (struct xid *)SRVPXID(srvtoken) - pointer to client's SVP ID
       (char *)SRVNAME(srvtoken) - pointer to shared variable name
(unsigned long)SRVNTOK(srvtoken) - SVP event notification token
       (void *)SRVUTOK(srvtoken) - user token (for your use)
 (struct srb *)SRVSRBP(srvtoken) - pointer to SVP share request block
```

On return from the process routine, the AP stores the result of the client's request in the shared variable buffer. If the size of the buffer on entry is not large enough for the result, the AP can request reallocation of storage via the `srv_alloc` service routine. The syntax of the `srv_alloc` call is:

```
void * srv_alloc(void * srvtoken, unsigned long size);
```

The first argument is the SRVTOKEN pointer and the second is the number bytes of storage required. The SRVTOKEN pointer is returned. Note that `srv_alloc` does not preserve the previous contents of the buffer, so if the client request data is to be used later, a copy must be made. The SRVBUF macro is used to get addressability to the newly-allocated buffer. A NULL pointer is returned by SRVBUF if the allocation failed.

To calculate the storage requirement for an APL2 array, the service routine `aplobjsize` can be called. The definition of `aplobjsize` is:

```
unsigned long aplobjsize(unsigned char type, unsigned char rank,
                         unsigned long nelm);
```

See Common Data Representation for a description of the valid argument values. The function returns the number of bytes of storage required.

Since `srv_alloc` only reallocates shared memory when necessary, you do not need to keep track of the buffer size to achieve optimum performance, but can simply call `srv_alloc` to request space for the result at any time.

© Copyright IBM Corporation 1994, 2017

If the AP process routine enters a potentially long wait (for example, due to a blocking read), it should release the lock that is implicitly held on the shared memory buffer. This is done by calling the `srv_free` service routine, which has the following definition:

```
srv_free(void * srvtoken);
```

After the `srv_free` service call, the buffer is no longer addressable and SRVBUF returns a NULL pointer.

For best performance in non-blocking AP process routines that return a response to the client, `srv_free` should not be called. The AP holds the lock on the shared variable between the reference of the request and the setting of the result value.

You can also write an AP server that accepts client requests, performs some action, but does not send a response back to the client. In this case, `srv_free` is called prior to return from the process callback routine, with no intervening `srv_alloc` calls.

Defining the Init Callback Function

While practical APs can be written using only the process callback routine, three other callbacks are provided for additional flexibility: init, wait, and exit. The `init` routine is called by APSERVER prior to accepting a new share from a client processor.

Using the SRVNAME and SRVPXID macros, the AP can determine the name of the shared variable and the identity of the client. By setting the return code SRV_REJECT_SHARE, the AP `init` routine can instruct the APSERVER to ignore the incoming share. A return code of 0 results in the accepting of the incoming share to complete the coupling with the client processor.

Typical things done in an `init` routine are file opens, environment initialization, and allocation of global data structures. The APSERVER maintains a user token for you to use as an anchor to a malloc'd data area. The SRVUTOK macro can be used to store and retrieve the user token.

Defining the Exit Callback Function

The `exit` callback routine usually provides the inverse function of the `init` routine. It is called by APSERVER just prior to retraction of the shared variable and termination of the execution thread. Using SRVUTOK to get the user token, which is typically the pointer to malloc'd storage, the `exit` routine can be used to close open files and windows, and to free any dynamic storage that was allocated in the `init` routine.

Defining the Wait Callback Function

The `wait` callback function is used only for special APs that need to wait on multiple events. If no wait routine is provided, the default action of the APSERVER is to wait for a client specification or retraction of the shared variable.

APSERVER obtains a separate event notification token for each shared variable as it spawns a new execution thread. The SRVNTOK macro can be used to retrieve the SVP event notification token. For APs that need to monitor additional file or window handles, the SVP notification token can be included in a multi-wait call (for

© Copyright IBM Corporation 1994, 2017

example, select, poll, or muxwait). After handling non-SVP events, the `wait` routine should return to allow processing of the next client request.

Execution Environment and Exception Handling

One of the goals of the APSERVER is to assume responsibility for most of the exception handling required for a robust auxiliary processor, including interrupt or termination signal trapping, re-dispatching of SVP requests in the case of temporary shared variable interlocks, and the orderly signoff and shutdown of the processor, including cases of severe error conditions.

In the case of an unrecoverable error on a shared variable SVP call, the APSERVER issues an error message with return code information, then calls the `exit` registered callback function prior to retracting the shared variable and terminating the execution thread. The execution states of threads associated with other shared variables are generally unaffected by the one share failure.

When the APSERVER is started, it immediately defines a signal handler to capture SIGTERM (software termination request) and SIGINT (interrupt signal). The signal handler is the same function that handles normal SVP-broadcast SHUTDOWN events. It issues a standard SVP processor SIGNOFF request before termination of the auxiliary processor.

When a child process is spawned for each shared variable, the SIGTERM and SIGINT signals are reset to the default handlers. The auxiliary processor's `init` registered callback function can be used to establish the desired exception handling environment for the main "process" routine.

In the unlikely event of an unrecoverable SVP error while monitoring the processor event queue, an error message is issued and an attempt is made to complete a normal processor SIGNOFF. During a normal processor SIGNOFF, the parent (dispatcher) process waits up to a maximum of 15 seconds for the child threads to complete a normal retraction of the associated shared variables and terminate.

A C Example Using APSERVER

```
/*-P-ap999-------------------------------------------------------------------
 *
 * Module Name: ap999.c
 *
 * Descriptive Name: Auxiliary Processor 999
 *
 * Function: Sample Auxiliary Processor implementing client-server
 *           protocol.  For hard working programmers with deadlines,
 *           this sample server will "catch some ZZZZZZ's" for a
 *           client who has no time to sleep for himself.
 *
 *-Z-------------------------------------------------------------------------*/
#include
#include
#include "aplobj.h"
#include "apserver.h"
#define INVALID_REQ  -1                /* catch-all rc for unfriendly AP   */
#define SYSTEM_LIMIT -2                /* requested space is unavailable   */
int _System sleep_init(void *);        /* AP Server "init" callback fn     */
int _System sleep_server(void *);      /* AP Server "process" callback fn  */
int _System sleep_exit(void *);        /* AP Server "exit" callback fn     */
static int error_code(void *, int);    /* error handler                    */

/*-F-main--------------------------------------------------------------------
```


© Copyright IBM Corporation 1994, 2017

```
 *
 * Purpose:    Auxiliary Processor 999 - main routine runs the general
 *             AP Server with standard callback to one or more of the 4
 *             AP Server callback routines identified in the parameter list.
 *             The 4 callback routines are named in parms 3-6 of apserver, and
 *             provide the following function:
 *                1. AP initialization, prior to accepting the client share
 *                2. multiple event wait and non-SVP event handling
 *                3. process a client request (passed in shared variable)
 *                4. AP exit cleanup on retraction of the share
 *             (Note: if a callback function is not provided, set parm to NULL)
 *
 * Arguments:  int      argc        Number of elements in argv
 *             char **  argv        Argument pointer array
 *             char **  envp        Environment pointer array
 *
 * Results:    int                  Exit Return Code
 *
 *---------------------------------------------------------------------------*/
int main(int argc, char **argv, char **envp) {
   return(apserver(argc, argv, sleep_init, NULL, sleep_server, sleep_exit));
/*                            init        wait  process       exit        */
}

/*-F-sleep_init------------------------------------------------------------
 *
 * Purpose:    Server initialization callback - called by the AP
 *             Server prior to accepting the share to fully couple
 *             with the client.  The typical things done by an AP
 *             here are to open files, allocate global storage,
 *             initialize a global data structure and store its
 *             address in the user token field of the srvtoken, etc...
 *             Via SRVPXID and SRVNAME, the AP can examine the client's
 *             fully-qualified SVP identification and the name of
 *             the share.  If the AP decides not to accept the share,
 *             it returns from this init routine with return code
 *             SRV_REJECT_SHARE.
 *
 * Arguments:  void * srvtoken       pointer to AP Server token
 *
 *             SRVUTOK(srvtoken)   macro returns user token (AP defined)
 *             SRVNAME(srvtoken)   macro returns shared variable name
 *             SRVPXID(srvtoken)   macro returns partner's extended ID
 *             SRVSRBP(srvtoken)   macro returns SVP share request block ptr
 *
 * Results:    int    Return Code
 *                    0 = success
 *                    SRV_REJECT_SHARE = AP Server just quietly ends thread
 *                    other = error condition (Server message & end thread)
 *
 *---------------------------------------------------------------------------*/
int _System sleep_init(void * srvtoken) {
   SRVUTOK(srvtoken) = malloc(8);       /* fake control block as an example */
   if (SRVUTOK(srvtoken) == NULL)
      return(SRV_REJECT_SHARE);         /* reject share if malloc failed    */
   memset(SRVUTOK(srvtoken),'Z',8);     /* initialize fake control block    */
   return(0);
}

/*-F-sleep_server----------------------------------------------------------
 *
 * Purpose:    Auxiliary processor 999 is a sample server AP which
 *             uses the general AP Server interface (apserver.c) to
 *             implement a client-server type protocol through the
 *             Shared Variable Processor.  It uses a single shared
 *             variable interface, with no name restrictions, and
 *             runs each share request stream asynchronously under
 *             a separate process (Unix) or thread (OS/2, Windows).
 *
 *             AP999 accepts 2 element integer vectors, where the first
```


© Copyright IBM Corporation 1994, 2017

```
 *             value is the number of milliseconds to sleep, and the second
 *             element is the size of the resulting character vector to
 *             return.  To demonstrate the handling of short versus long
 *             requests, AP999 uses a fast path through the SVP for a
 *             sleep of 1 second or less (holding the shared variable
 *             lock from pre-ref until post-spec), while releasing the
 *             lock (via srv_free call) for longer requests.
 *
 * Arguments:  void * srvtoken      pointer to AP Server token
 *
 *             SRVBUF(srvtoken)     macro returns ptr to shared var buffer
 *             SRVBUFL(srvtoken)    macro returns shared var buffer length
 *             SRVUTOK(srvtoken)    macro returns user token (AP defined)
 *             SRVNAME(srvtoken)    macro returns shared variable name
 *             SRVPXID(srvtoken)    macro returns partner's extended ID
 *             SRVSRBP(srvtoken)    macro returns SVP share request block ptr
 *
 * Results:    int            Return Code
 *                            0 = success
 *                       other = error condition (Server retracts & exits)
 *
 *             On return, the AP stores the result of the client's request
 *             in the shared variable buffer, enlarging it if necessary (or
 *             reallocating it if freed with srv_free) using the srv_alloc
 *             callback to the AP Server.  Note that srv_alloc will NOT
 *             preserve the previous contents of the buffer, and SRVBUF
 *             must be used to get addressability to the buffer after the
 *             call to srv_alloc.
 *
 *-------------------------------------------------------------------------*/

int _System sleep_server(void * srvtoken) {
   int rc = 0;
   APLOBJ *obj = (APLOBJ *) SRVBUF(srvtoken);  /* request is an APL2 object */
   int objsize;
   int msecs = obj->dim[1];             /* 1st item of request is sleep time*/
   int shape = obj->dim[2];             /* 2nd is shape of char array result*/
   if ((obj->type != INTEGER && obj->type != BOOLEAN) ||
        obj->rank != VECTOR || obj->dim[0] != 2) {
      rc = error_code(srvtoken, INVALID_REQ);  /* generate error return code*/
      return rc;
      }
   if (obj->type == BOOLEAN) {          /* if Boolean, convert to integer   */
      msecs = ((unsigned long)obj->dim[1]) >> ((8*sizeof(long))-1);
      shape = (((unsigned long)obj->dim[1]) << 1) >> ((8*sizeof(long))-1);
      }
   if (msecs <= 1000) {                 /* short wait - OK to hold the lock */
      if (msecs > 0) {
         svsleep(msecs);
         }
      }
   else {                               /* long wait                        */
      srv_free(srvtoken);               /* AP Server releases space, unlocks*/
      obj = NULL;                       /* no longer have addressability    */
      if (msecs <= 60000) {             /* bypass sleep if more than 60     */
         svsleep(msecs);                /*    seconds in this sample AP     */
         }
      }
   objsize = aplobjsize(CHARACTER, VECTOR, shape);          /* result size? */
   obj = (APLOBJ *) SRVBUF(srv_alloc(srvtoken, objsize));   /* ask for space*/
   if (obj == NULL) {                                /* if allocation failed, */
      rc = error_code(srvtoken, SYSTEM_LIMIT);     /* set error return code */
      return rc;
      }
   obj->ptr = CDRID;                    /* build the CDR header for result  */
   obj->nb = objsize;
   obj->nelm = shape;
   obj->type = CHARACTER;
   obj->rank = VECTOR;
   obj->dim[0] = shape;
```


© Copyright IBM Corporation 1994, 2017

```
   memset(&obj->dim[1], 'Z', shape);    /* return some Z's for sleepy client*/
   return rc;
}

/*-F-error_code------------------------------------------------------------
 *
 * Purpose:    Return an error return code to client for this request
 *
 * Arguments:  void * srvtoken       pointer to AP Server token
 *             int        rc         AP return code
 *
 * Results:    int            Return Code (from this function)
 *                            0 = success
 *                           -1 = error condition (srv_alloc failed)
 *
 *-------------------------------------------------------------------------*/
int error_code(void * srvtoken, int rc) {
   APLOBJ *obj;                         /* ptr to APL2 object in CDR format */
   int objsize;
   objsize = aplobjsize(INTEGER, SCALAR, 1);               /* result size? */
   obj = (APLOBJ *) SRVBUF(srv_alloc(srvtoken, objsize));   /* ask for space*/
   if (obj == NULL) {                            /* we're in trouble if we   */
      return (-1);                               /* can't get 32 bytes - die!*/
      }
   obj->ptr = CDRID;                             /* construct result header  */
   obj->nb = objsize;
   obj->nelm = 1;
   obj->type = INTEGER;
   obj->rank = SCALAR;
   obj->dim[0] = rc;
   return (0);
}

/*-F-sleep_exit------------------------------------------------------------
 *
 * Purpose:    Server exit callback - called by the AP Server prior to
 *             shared variable retraction and termination of this process
 *             thread.  Typical use by the AP is for freeing any dynamic
 *             storage it allocated, closing files and pipes, etc.
 *
 * Arguments:  void * srvtoken       pointer to AP Server token
 *
 *             SRVUTOK(srvtoken)     macro returns user token (AP defined)
 *
 * Results:    int             Return Code
 *                             0 = success
 *                         other = error condition (Server prints error msg)
 *
 *-------------------------------------------------------------------------*/
int _System sleep_exit(void * srvtoken) {
   free(SRVUTOK(srvtoken));             /* free the fake control block      */
   return(0);
}

/*=========================================================================*/
/*   The following wait callback routine is not used in this sample AP, but */
/*   is provided for documentation purposes.                               */
/*=========================================================================*/
/*-F-multi_wait------------------------------------------------------------
 *
 * Purpose:    Server callback to allow wait on multiple file descriptors,
 *             semaphores, or message queues, including the shared variable
 *             event queue.
 *
 *             Return from this function allows the AP Server to handle
 *             shared variable (SVP) events.  Typically an AP will wait on the
 *             notification token along with window, file, or pipe descriptors.
 *             The AP handles non-SVP events, and simply returns from this
 *             function when it wants the AP Server to process SVP events
 *             (which normally result in a call-back to the AP "process" exit
```


© Copyright IBM Corporation 1994, 2017

```
 *             function to handle the client request).
 *
 * Arguments:  void * srvtoken       pointer to AP Server token
 *
 *             SRVNTOK(srvtoken)     macro returns shared variable event
 *                                   notification token (system dependent)
 *             SRVUTOK(srvtoken)     macro returns user token (AP defined)
 *
 * Results:    int             Return Code
 *                             0 = success
 *                         other = error condition (Server retracts & ends)
 *
 *-------------------------------------------------------------------------*/
int _System multi_wait(void * srvtoken) {
/* Get message queue id (Unix) or semaphore handle (OS/2, Windows)               */
   unsigned long wait_token = SRVNTOK(srvtoken);
/* AP typically does poll/select (Unix) or wait (OS/2, Windows)                  */
/* and continues processing non-SVP events in a while loop until the only   */
/* event outstanding is an SVP event, at which time it simply returns       */
/* to the AP Server to read the share event queue and handle the event.     */
   return(0);
}
```


© Copyright IBM Corporation 1994, 2017

SVP Programming Interface

Interface to the SVP is by C function call. There are three entry points based on the service desired. The following sections describe the structures used on these calls:

`int _System svpp(struct prb *);` for SIGNON and SIGNOFF
`int _System svpe(struct wrb *);` for SVEVENT (for example, wait)
`int _System svps(struct srb *);` for all other SVP requests

Notes:

1. The APL2 interpreter passes all of its command line arguments to any auxiliary processor that is started automatically. An auxiliary processor should be prepared to receive arguments that might not be applicable. The auxiliary processor should, however, always look for the `-id` parameter and use the information from the values given when issuing an SVSIGNON. If no `-id` parameter is found, the auxiliary processor should sign on with its own default processor number, and no parent or grandparent.
2. The `ntoken` returned on an SVSIGNON and SVSHARE is the handle of an event semaphore. Normally this token is only used on an SVEVENT call, but it can be waited on directly or added to a multiple wait. It should never be posted or reset by the auxiliary processor.
3. A SHUTDOWN event is posted to a dependent processor by the SVP when the parent issues an SVSIGNOFF, or when the **Shutdown** option is selected from the **Info->Processors** menu item of the *SVP Monitor* window. It is the responsibility of the auxiliary processor to retract all variables and issue an SVSIGNOFF when this signal is received.

- PRB Requests
- WRB Requests
- SRB Requests
- SVP Control Blocks

PRB Requests

- SVSIGNON - Sign on to the SVP
- SVSIGNOFF - Sign off from the SVP

SVSIGNON - Sign on to the SVP

```
prb.req = SVSIGNON
```

Sign on to the Shared Variable Processor. The following additional `prb` fields must be provided:

`prb.xid.pparent` Grandparent processor number.
`prb.xid.parent` Parent processor number.
`prb.xid.procid` Processor ID of caller.

Note: If the auxiliary processor will be run independently, `pparent` and `parent` must both be specified as `0`.

© Copyright IBM Corporation 1994, 2017

The following fields are optional:

`prb.user` An arbitrary token returned on `SVEVENT` for incoming offers or shutdown.

`prb.emask` Event conditions to be masked (not signaled). See WRB Requests for names of the events.

`prb.PRBFNPOST` If true, disable posting to event queue.

`prb.PRBFNOTRC` If true, disable tracing of SVP events for this processor.

The following `prb` fields are returned:

`prb.rc` One of the following return codes:

```
SVP_OK              Success
SVP_UNAVAILABLE     SVP not available
SVP_ERROR_INVARG    Argument error
SVP_ERROR_ASO       Already signed on as xid
SVP_ERROR_PUSED     Another process signed on as xid
```

`prb.PRBFOFFS` `TRUE` if one or more offers exist.

`prb.ntoken` Token used for waiting on SVP events.

`prb.pcbx` Processor token that must be used on all subsequent SVP calls.

Note: If the SVP shared memory control area has not been initialized, this call automatically starts the process that establishes and owns that area.

SVSIGNOFF - Sign off from the SVP

```
prb.req = SVSIGNOFF
```

Sign off from the Shared Variable Processor. The following additional `prb` field must be provided:

`prb.pcbx` The processor token that was returned by `SVSIGNON`.

`prb.rc` One of the following return codes:

```
SVP_OK              Success
SVP_UNAVAILABLE     SVP not available
SVP_ERROR_INVARG    Argument error
SVP_ERROR_NSO       Processor not signed on
```

Notes:

1. If the SVP was automatically started by a signon, a usage count of processors is maintained and the SVP process is shut down when that count reaches 0.
2. The SVP retracts all variables shared with or offered from the processor during signoff.

WRB Requests

```
wrb.req = SVEVENT
```

Wait for an SVP event. This can be either a processor event (such as an incoming offer) or a shared variable event (such as variable set by partner). The following additional `wrb` fields must be provided:

© Copyright IBM Corporation 1994, 2017

`wrb.pcbx` The processor token that was returned by `SVSIGNON`.

`wrb.ntoken` A notification token returned by either `SVSIGNON` or `SVSHARE`.

`wrb.timeout` Maximum time to wait in milliseconds if nonnegative. Use `-1` to wait indefinitely for an event.

The following flag is optional:

`wrb.WRBFFLUSH` If true, causes the event queue to be flushed on return.

The following `wrb` fields are filled in on return:

`wrb.rc` One of the following return codes:

```
SVP_OK                Success
SVP_UNAVAILABLE       SVP not available
SVP_ERROR_NSO         Processor not signed on
SVP_ERROR_INVARG      Invalid argument
SVP_ERROR_NOEVENT     No event found before time expired
SVP_ERROR_TIMEOUT     Timeout occurred with no event
```

`wrb.user` For `SVP_EVENT_SHUTDOWN` or `SVP_EVENT_OFFEREX` this is the value provided in `prb.user` at `SVSIGNON`. For all other events it is the value provided in `srb.user` at `SVSHARE`.

`wrb.event` One of the following event codes:

```
SVP_EVENT_SHUTDOWN    SVP shutdown
SVP_EVENT_OFFEREX     Offer extended
SVP_EVENT_OFFERAC     Offer accepted
SVP_EVENT_SMAVAIL     Shared memory available
SVP_EVENT_PRETRACT    Partner retracted
SVP_EVENT_PSETACV     Partner set ACV
SVP_EVENT_PSPEC       Partner specified
SVP_EVENT_PREF        Partner referenced
SVP_EVENT_FSPEC       Partner failed specified
SVP_EVENT_FREF        Partner failed reference
SVP_EVENT_PRELEASE    Partner released variable
```

`wrb.scbx` (except for `SVP_EVENT_SHUTDOWN`) For `SVP_EVENT_OFFEREX`, a variable token that can be used for a matching `SVSHARE`. For other events, the variable token that was used when sharing the variable.

`wrb.osn` (`SVP_EVENT_OFFEREX` only) A sequence number that can be used for a matching `SVSHARE` to verify that the offer being reported is still outstanding.

Note: Only one event is returned. The auxiliary processor should continue to make calls (to retrieve all events) until `SVP_ERROR_NOEVENT` is received.

SRB Requests

The following sections discuss the SRB requests.

- Offering and Retracting Variables
- Setting and Using Data in Variables
- Obtaining or Setting Information about Variables

© Copyright IBM Corporation 1994, 2017

Offering and Retracting Variables

- SVSHARE - Offer a variable
- SVSHARE - Match an offer
- SVRETRACT - Retract a share

SVSHARE - Offer a variable

`srb.req = SVSHARE srb.scbx = 0`

Use this service to offer a new variable to some processor. The following additional `srb` fields must be provided:

`srb.pcbx`	The processor token that was returned by `SVSIGNON`.
`srb.pxid.procid`	The identification number of the partner to whom the offer is made.
`srb.scbx`	Must be zero to indicate a new offer.

The following fields are optional:

`srb.osn`	Offer sequence number. Only offers with osn greater than this can be used to match the offer.
`srb.user`	User field, returned on `SVEVENT`.
`srb.acv`	Access control (see SVP Control Blocks).
`srb.name`	Variable name (up to 255 chars).
`srb.SRBFNEWQ`	True to request a separate event queue to be used for this variable. If false, the processor event queue is used for all events.
`srb.SRBFNPOST`	True to request that no events be signaled for this variable.
`srb.SRBFNOFFR`	True to disable automatic reoffer on partner's retract. It is recommended that all auxiliary processors turn this flag on.

The following `srb` fields are filled in on return:

`srb.rc` One of the following return codes:

```
SVP_OK              Success
SVP_UNAVAILABLE     SVP not available
SVP_ERROR_NSO       Processor not signed on
SVP_ERROR_SHRSLF    Offer to self
SVP_ERROR_INVARG    Invalid argument
```

`srb.pxid`	Extended ID of corresponding offer, if found.
`srb.osn`	Offer sequence number.
`srb.scbx`	A token associated with this share, that must be used on all subsequent requests.
`srb.ntoken`	A signaling token. This is unique to the variable if `srb.SRBFNEWQ` was set, otherwise the processor `ntoken` is returned.
`srb.acv`	Access control (see SVP Control Blocks).
`srb.state`	Shared variable state.

© Copyright IBM Corporation 1994, 2017

`srb.coupling` Degree of coupling.

Note: If a share request is made and no corresponding offer is found, a search is made for a file named `ap`*`nnn`*, where *`nnn`* is the `srb.procid`. If this file is found, it is executed.

SVSHARE - Match an offer

`srb.req = SVSHARE srb.scbx =` *`variable_token`*

Match an offer that has already been made to this processor. This service requires information (including `srb.scbx`) that has been provided by `SVSCAN` or `SVEVENT`. The following additional `srb` fields must be provided:

`srb.pcbx`	The processor token that was returned by `SVSIGNON`.
`srb.scbx`	As returned by `SVSCAN` or `SVEVENT`.
`srb.osn`	As returned by `SVSCAN` or `SVEVENT`.
`srb.acv`	Access control (see SVP Control Blocks).
`srb.SRBFNEWQ`	True to request a separate event queue to be used for this variable. If false, the processor event queue is used for all events.
`srb.SRBFNPOST`	True to request no event queue to be used for this variable.
`srb.user`	(Optional) User field, returned on `SVEVENT`.

The following `srb` fields are filled in on return:

`srb.rc` One of the following return codes:

```
SVP_OK               Success
SVP_UNAVAILABLE      SVP not available
SVP_ERROR_NSO        Processor not signed on
SVP_ERROR_SHRSLF     Offer to self
SVP_ERROR_INVARG     Invalid argument (including OSN mismatch)
SVP_ERROR_NOOFFER    No offer found
```

`srb.pxid`	Extended ID of corresponding offer, if found.
`srb.ntoken`	A signaling token. This is unique to the variable if `srb.SRBFNEWQ` was set, otherwise the processor `ntoken` is returned.
`srb.acv`	Access control (see SVP Control Blocks).
`srb.state`	Shared variable state.
`srb.coupling`	Degree of coupling.

SVRETRACT - Retract a share

`srb.req = SVRETRACT`

Retract a shared variable or share offer. The following additional `srb` fields must be provided:

`srb.pcbx` The processor token that was returned by `SVSIGNON`.

© Copyright IBM Corporation 1994, 2017

`srb.scbx` The variable token that was returned or used by `SVSHARE`.

The following `srb` fields are filled in on return:

`srb.rc` One of the following return codes:

```
SVP_OK              Success
SVP_UNAVAILABLE     SVP not available
SVP_ERROR_NSO       Processor not signed on
SVP_ERROR_INVARG    Invalid argument
```

`srb.coupling` Degree of coupling.

Setting and Using Data in Variables

In order to gain access to shared variable data, a processor must obtain a temporary lock, so that the partner does not attempt to update the data or change the variable's state during the accessing process. The processor commonly gets a lock implicitly through an `SVPREREF` or `SVPRESPEC` request. It is also possible to issue any of the `SVPREREF` or `SVPRESPEC` requests when the variable's lock is already held. Eventually, the processor must issue `SVRELEASE`, or more typically `SVREF` or `SVSPEC`, to release the lock. Applications should minimize the time that they hold the lock. No count is maintained, so a single unlocking request unlocks the variable no matter how many lock requests were issued.

The second function provided by the locking requests is to give the calling program access to a data buffer. When `SVPREREF` is used, this buffer contains the current value of the variable. For `SVPRESPEC`, a buffer is provided of a caller-specified size. In either case, the buffer is available only until the next locking or unlocking request is issued.

The three unlocking requests differ only in the effects they have on the access state of the variable. The system does not check for any correspondence between the locking request and the subsequent unlocking request. The normal sequence is, of course, to use `SVPREREF` followed by `SVREF` to reference a variable, and `SVPRESPEC` followed by `SVSPEC` to specify a new value for a variable. However, other sequences that can be useful on occasion include:

- `SVPREREF` or `SVRELEASE` to peek at a value set by your partner without actually referencing it.
- `SVPREREF`, `SVPRESPEC`, or `SVSPEC` to look at a value set by your partner and then replace it without losing the lock between reference and specification.
- `SVPREREF` or `SVSPEC` to update part of the value in place, and inform your partner that the change has occurred.
- `SVRELEASE` to look at the last value assigned to a variable even though you have already referenced it.

- SVPREREF - Start variable reference
- SVREF - Finish variable reference
- SVPRESPEC - Start variable assignment
- SVSPEC - Finish variable assignment
- SVRELEASE - Release a preRef or preSpec

SVPREREF - Start variable reference

© Copyright IBM Corporation 1994, 2017

```
srb.req = SVPREREF
```

First stage of referencing (obtaining the current value of) a variable. This call provides a pointer to the variable's value, and locks the variable until SVREF is issued. The partner cannot issue further requests against the variable until it is unlocked by an SVSPEC, SVREF, or SVRELEASE. The following additional srb fields must be provided:

srb.pcbx The processor token that was returned by SVSIGNON.

srb.scbx The variable token that was returned or used by SVSHARE.

The following srb fields are filled in on return:

srb.rc One of the following return codes:

```
SVP_OK              Success
SVP_INTERLOCK       Variable interlocked
SVP_UNAVAILABLE     SVP not available
SVP_ERROR_NSO       Processor not signed on
SVP_ERROR_NOVALUE   No new value
SVP_ERROR_INVARG    Invalid argument
```

srb.vlen Size of object.

srb.value Pointer to object.

srb.acv Access control (see SVP Control Blocks).

srb.state Shared variable state.

srb.coupling Degree of coupling.

SVREF - Finish variable reference

```
srb.req = SVREF
```

Second stage of referencing (obtaining the current value of) a variable. This call notifies the SVP that the value has been extracted, and unlocks the variable. The value pointed to by srb.value on return from SVPREREF can no longer be accessed after issuing this call. The following additional srb fields must be provided:

srb.pcbx The processor token that was returned by SVSIGNON.

srb.scbx The variable token that was returned or used by SVSHARE.

The following srb fields are filled in on return:

srb.rc One of the following return codes:

```
SVP_OK              Success
SVP_UNAVAILABLE     SVP not available
SVP_ERROR_NSO       Processor not signed on
SVP_ERROR_INVARG    Invalid argument
```

srb.acv Access control (see SVP Control Blocks).

srb.state Shared variable state.

srb.coupling Degree of coupling.

© Copyright IBM Corporation 1994, 2017

SVPRESPEC - Start variable assignment

```
srb.req = SVPRESPEC
```

First stage of specifying a value for a variable. This call provides a pointer to a location where the new value for the variable is to be placed, and locks the variable until `SVSPEC` is issued. The partner cannot issue further requests against the variable until it is unlocked. The following additional `srb` fields must be provided:

`srb.pcbx`	The processor token that was returned by `SVSIGNON`.
`srb.scbx`	The variable token that was returned or used by `SVSHARE`.
`srb.vlen`	Size of object.
`srb.SRBFIGNV`	True to ignore any unreferenced value.

The following `srb` fields are filled in on return:

`srb.rc`	One of the following return codes:

```
SVP_OK              Success
SVP_UNAVAILABLE     SVP not available
SVP_ERROR_NSO       Processor not signed on
SVP_ERROR_INVARG    Invalid argument
SVP_SMNOSPACE       Space not available
SVP_INTERLOCK       Variable interlocked
SVP_ERROR_VOS       Partner's value not read
```

`srb.acv`	Access control (see SVP Control Blocks).
`srb.state`	Shared variable state.
`srb.value`	Pointer to area to write object.
`srb.coupling`	Degree of coupling.

SVSPEC - Finish variable assignment

```
srb.req = SVSPEC
```

Second stage of specifying a value for a variable. This call notifies the SVP that the new value is in place, and unlocks the variable. The area pointed to by `srb.value` on return from `SVPRESPEC` can no longer be accessed after issuing this call. The following additional `srb` fields must be provided:

`srb.pcbx`	The processor token that was returned by `SVSIGNON`.
`srb.scbx`	The variable token that was returned or used by `SVSHARE`.

The following `srb` fields are filled in on return:

`srb.rc`	One of the following return codes:

```
SVP_OK              Success
SVP_UNAVAILABLE     SVP not available
SVP_ERROR_NSO       Processor not signed on
SVP_ERROR_INVARG    Invalid argument
```

`srb.acv`	Access control (see SVP Control Blocks).

© Copyright IBM Corporation 1994, 2017

`srb.state` Shared variable state.
`srb.coupling` Degree of coupling.

SVRELEASE - Release a preRef or preSpec

```
srb.req = SVRELEASE
```

Release a variable previously locked by `SVPREREF` or `SVPRESPEC` without changing its state. The area pointed to by `srb.value` on return from the locking call can no longer be accessed after issuing this call. The following additional `srb` fields must be provided:

`srb.pcbx` The processor token that was returned by `SVSIGNON`.
`srb.scbx` The variable token that was returned or used by `SVSHARE`.

The following `srb` fields are filled in on return:

`srb.rc` One of the following return codes:

```
SVP_OK              Success
SVP_UNAVAILABLE     SVP not available
SVP_ERROR_NSO       Processor not signed on
SVP_ERROR_INVARG    Invalid argument
```

`srb.acv` Access control (see SVP Control Blocks).
`srb.state` Shared variable state.
`srb.coupling` Degree of coupling.

Obtaining or Setting Information about Variables

- SVSHARE - Inquire about a share offer
- SVSCAN - Scan for a share offer
- SVSEEACV - Query access control
- SVSETACV - Set access control
- SVSTATE - Query variable state

SVSHARE - Inquire about a share offer

`srb.req = SVSHARE srb.scbx =` *`variable_token`*

Obtain information about a previous share offer. The following additional `srb` fields must be provided:

`srb.pcbx` The processor token that was returned by `SVSIGNON`.
`srb.pxid` Partner extended ID, or zero. If zero, this value is returned.
`srb.scbx` Variable token from original offer.

The following `srb` fields are filled in on return:

© Copyright IBM Corporation 1994, 2017

`srb.rc` One of the following return codes:

```
SVP_OK              Success
SVP_UNAVAILABLE     SVP not available
SVP_ERROR_NSO       Processor not signed on
SVP_ERROR_INVARG    Invalid argument
SVP_ERROR_NOOFFER   No offer found
```

`srb.pxid` Extended ID of partner.

`srb.acv` Access control (see SVP Control Blocks).

`srb.state` Shared variable state.

`srb.coupling` Degree of coupling.

SVSCAN - Scan for a share offer

```
srb.req = SVSCAN
```

Scan for an incoming shared variable offer. The following additional `srb` fields must be provided:

`srb.pcbx` The processor token that was returned by `SVSIGNON`.

`srb.pxid` If pxid.procid is zero, scan for offers from any processor. Otherwise, only offers from the specified processor are checked.

`srb.osn` Offer sequence number. If non-zero, and `srb.scbx=0`, only offers with a sequence number greater than that specified are checked. This allows the scan to be repeated to search for additional offers, and also assists in rotating limited resources among requestors.

If both `srb.osn` and `srb.scbx` are non-zero, the `osn` value is used as a verification against the offer identified by the `scbx`.

`srb.name` A null-delimited character string. If a non-empty name is specified, only offers for this variable name are checked.

`srb.scbx` Normally must be zero, but can be set to identify a specific partner.

The following `srb` fields are filled in on return:

`srb.rc` One of the following return codes:

```
SVP_OK              Success
SVP_UNAVAILABLE     SVP not available
SVP_ERROR_NSO       Processor not signed on
SVP_ERROR_INVARG    Invalid argument
SVP_ERROR_NOOFFER   No offer found
```

`srb.pxid` Extended ID of corresponding offer, if found.

`srb.osn` Offer sequence number.

`srb.scbx` SCB index of offer.

`srb.name` Variable name.

SVSEEACV - Query access control

```
srb.req = SVSEEACV
```


© Copyright IBM Corporation 1994, 2017

Query access control. The following additional `srb` fields must be provided:

`srb.pcbx` The processor token that was returned by `SVSIGNON`.
`srb.scbx` The variable token that was returned or used by `SVSHARE`.

The following `srb` fields are filled in on return:

`srb.rc` One of the following return codes:

```
SVP_OK              Success
SVP_UNAVAILABLE     SVP not available
SVP_ERROR_NSO       Processor not signed on
SVP_ERROR_INVARG    Invalid argument
```

`srb.acv` Access control (see SVP Control Blocks).
`srb.state` Shared variable state.
`srb.coupling` Degree of coupling.

SVSETACV - Set access control

```
srb.req = SVSETACV
```

Set access control. The following additional `srb` fields must be provided:

`srb.pcbx` The processor token that was returned by `SVSIGNON`.
`srb.scbx` The variable token that was returned or used by `SVSHARE`.
`srb.acv` Access control desired (see SVP Control Blocks).

The following `srb` fields are filled in on return:

`srb.rc` One of the following return codes:

```
SVP_OK              Success
SVP_UNAVAILABLE     SVP not available
SVP_ERROR_NSO       Processor not signed on
SVP_ERROR_INVARG    Invalid argument
```

`srb.acv` Access control (see SVP Control Blocks).
`srb.state` Shared variable state.
`srb.coupling` Degree of coupling.

SVSTATE - Query variable state

```
srb.req = SVSTATE
```

Query shared variable state. The following additional `srb` fields must be provided:

`srb.pcbx` The processor token that was returned by `SVSIGNON`.
`srb.scbx` The variable token that was returned or used by `SVSHARE`.

© Copyright IBM Corporation 1994, 2017

The following `srb` fields are filled in on return:

`srb.rc` One of the following return codes:

```
SVP_OK              Success
SVP_UNAVAILABLE     SVP not available
SVP_ERROR_NSO       Processor not signed on
SVP_ERROR_INVARG    Invalid argument
```

`srb.acv` Access control (see SVP Control Blocks).

`srb.state` Shared variable state.

`srb.coupling` Degree of coupling.

Notes:

1. In addition to the return codes listed, SVP_ERROR_SYSTEM can be returned if the SVP receives an error from an operating system call. If this return code is received, prb.reason (or srb.reason or wrb.reason) provides an additional reason code. (See SVP Reason Codes for a listing of these reason codes beginning with SVP_ERRSYS_xxx.)
2. Similarly, if the SVP_ERROR_INVARG return code is received, the reason code is set to one of the SVP_ARGERR_xxx values.

SVP Control Blocks

The control blocks shown here are defined in C include file `aplap.h`. See also Common Data Representation.

- Extended ID Structure
- Processor Request Block
- Share Request Block
- Wait Request Block
- SVP Requests
- SVP Return Codes
- SVP Reason Codes
- SVP Event Codes
- Function Prototypes
- Access Control Constants

Extended ID Structure

```
struct xid {                    /* Extended ID structure              */
    unsigned long pparent;      /*  Parent's Parent ID                */
    unsigned long parent;       /*  Parent ID                         */
    unsigned long procid;       /*  Processor ID                      */
    unsigned long hostid;       /*  Host ID (TCP/IP address)          */
    unsigned long res1;         /*  Reserved                          */
    unsigned long res2;         /*  Reserved                          */
    char userid[8];             /*  User ID                           */
    char surrid[8];             /*  Surrogate ID                      */
    };
struct flags {
    unsigned int b1:1;
    unsigned int b2:1;
    unsigned int b3:1;
    unsigned int b4:1;
```


© Copyright IBM Corporation 1994, 2017

```
    unsigned int b5:1;
    unsigned int b6:1;
    unsigned int b7:1;
    unsigned int b8:1;
    unsigned int b9:1;
    unsigned int b10:1;
    unsigned int b11:1;
    unsigned int b12:1;
    unsigned int b13:1;
    unsigned int b14:1;
    unsigned int b15:1;
    unsigned int b16:1;
    unsigned int b17:1;
    unsigned int b18:1;
    unsigned int b19:1;
    unsigned int b20:1;
    unsigned int b21:1;
    unsigned int b22:1;
    unsigned int b23:1;
    unsigned int b24:1;
    unsigned int b25:1;
    unsigned int b26:1;
    unsigned int b27:1;
    unsigned int b28:1;
    unsigned int b29:1;
    unsigned int b30:1;
    unsigned int b31:1;
    unsigned int b32:1;
    };
```

Processor Request Block

```
struct prb {
    short req;                  /*  Request code              */
    short rc;                   /*  Return code               */
    short reason;               /*  Reason code               */
    short res1;                 /*  Reserved                  */
    struct xid xid;             /*  Extended ID               */
    void *user;                 /*  User field                */
    unsigned long emask;        /*  Event mask                */
    unsigned long ntoken;       /*  Notification token        */
    unsigned long pcbx;         /*  PCB index                 */
    unsigned long socket;       /*  Socket number             */
    unsigned long socksem;      /*  Socket semaphore          */
    char ** argv;               /*  Command line args ptr     */
    int argc;                   /*  Count of args             */
    void *res2;                 /*  Reserved                  */
    struct flags prbflags;      /*  Processor flags           */
    char socksemname[20];       /*  Socket semaphore name     */
    char reserved[12];          /*  Reserved                  */
    };
#define PRBFOFFS  prbflags.b1  /*  Offers outstanding        */
#define PRBFNPOST prbflags.b3  /*  Don't post                */
#define PRBFNOTRC prbflags.b4  /*  Don't trace               */
```

Share Request Block

```
struct srb {
    short req;                  /*  Request code              */
    short rc;                   /*  Return code               */
    short reason;               /*  Reason code               */
    short res1;                 /*  Reserved                  */
    void *user;                 /*  User field                */
    unsigned long ntoken;       /*  Notification token        */
    unsigned long pcbx;         /*  PCB index                 */
```


© Copyright IBM Corporation 1994, 2017

```
    unsigned long scbx;         /*  SCB index                 */
    struct xid pxid;            /*  Extended Partner ID       */
    long osn;                   /*  Offer sequence number     */
    long vlen;                  /*  Length of value           */
    void *value;                /*  Pointer to value          */
    unsigned long pvrba;        /*  Reserved                  */
    unsigned short acv;         /*  Access control            */
    unsigned short state;       /*  State                     */
    unsigned short coupling;    /*  Coupling                  */
    unsigned short res2;        /*  Reserved                  */
    struct flags srbflags;      /*  Flags                     */
    unsigned long socket;       /*  socket number             */
    unsigned long socksem;      /*  socket send sem-id (UNIX) */
    char socksemname[20];       /*  Socket semaphore name     */
    char reserved[12];          /*  Reserved                  */
    char name[256];             /*  Variable name             */
    };
#define SRBFIGNV  srbflags.b1  /*  Ignore value waiting      */
#define SRBFNEWQ  srbflags.b2  /*  Request new event queue   */
#define SRBFNPOST srbflags.b4  /*  Don't post                */
#define SRBFNOFFR srbflags.b5  /*  No offer back on retract  */
```

Wait Request Block

```
struct wrb {
   short req;                  /* Request code               */
   short rc;                   /* Return code                */
   short reason;               /* Reason code                */
   short res1;                 /* Reserved                   */
   void *user;                 /* User field                 */
   unsigned long ntoken;       /* Notification token         */
   unsigned long pcbx;         /* PCB index                  */
   unsigned long scbx;         /* SCB index                  */
   unsigned long osn;          /* offer sequence number      */
   long timeout;               /* Timeout value              */
   unsigned long event;        /* Event code                 */
   struct flags wrbflags;      /* Flags                      */
   char reserved[16];          /* Reserved                   */
   };
#define WRBFFLUSH wrbflags.b1  /*  Flush event queue         */
```

SVP Requests

```
#define SVINIT         0
#define SVSIGNON       1
#define SVSIGNOFF      2
#define SVSCAN         3
#define SVSHARE        4
#define SVSEEACV       5
#define SVSETACV       6
#define SVREF          7
#define SVSPEC         8
#define SVRELEASE      11
#define SVRETRACT      12
#define SVSTATE        13
#define SVQUERY        14
#define SVPREREF       15
#define SVPRESPEC      16
#define SVEVENT        17
#define SVPOST         18
#define SVGETTOKEN     19
```

SVP Return Codes

© Copyright IBM Corporation 1994, 2017

```
#define SVP_ERROR_SYSTEM  - 3     /* System Error                */
#define SVP_SMNOSPACE     - 2     /* No space for object         */
#define SVP_INTERLOCK     - 1     /* variable interlocked        */
#define SVP_OK              0     /* A-OK                        */
#define SVP_UNAVAILABLE     1     /* SVP is unavailable          */
#define SVP_ERROR_PROTEXCP  2     /* Protection exception        */
#define SVP_ERROR_NSO       3     /* Not signed on               */
#define SVP_ERROR_ASO       4     /* Already signed on           */
#define SVP_ERROR_PUSED     5     /* Processor in use            */
#define SVP_ERROR_SHRSELF   8     /* Share with self             */
#define SVP_ERROR_VTL       10    /* Value too large             */
#define SVP_ERROR_NOVALUE   11    /* No value                    */
#define SVP_ERROR_NOOFFER   12    /* No offer found              */
#define SVP_ERROR_INVREQ    13    /* Invalid request             */
#define SVP_ERROR_VOS       14    /* Unread value exists         */
#define SVP_ERROR_INVARG    15    /* Invalid argument            */
#define SVP_ERROR_NOEVENT   16    /* No event found              */
#define SVP_ERROR_TIMEOUT   17    /* Timeout on SVEVENT          */
#define SVP_ERROR_INTERRUPT 18    /* Interrupt on wait           */
```

SVP Reason Codes

```
#define SVP_ERRSYS_GETNMEM   100   /* Error getting named sh mem    */
#define SVP_ERRSYS_FREEMEM   101   /* Error freeing sh mem          */
#define SVP_ERRSYS_SUBFREE   102   /* Error freeing pcb/scb space   */
#define SVP_ERRSYS_NOPCBS    103   /* Limit on processors exceeded  */
#define SVP_ERRSYS_NOSCBS    104   /* Limit on variables exceeded   */
#define SVP_ERRSYS_GETESEM   105   /* Error creating event sem      */
#define SVP_ERRSYS_POSTQUEUE 106   /* Error on post of queue        */
#define SVP_ERRSYS_WAITESEM  107   /* Error on wait for semaphore   */
#define SVP_ERRSYS_RESETESEM 108   /* Reset semaphore error         */
#define SVP_ERRSYS_OPENSEM   109   /* Error opening a semaphore     */
#define SVP_ERRSYS_REQSEM    110   /* Error requesting sm semaphore */
#define SVP_ERRSYS_RELSEM    111   /* Error releasing sm semaphore  */
#define SVP_ERRSYS_CREATESEM 112   /* Error creating sm semaphore   */
#define SVP_ERRSYS_GETXID    113   /* Error in side file lookup     */
#define SVP_ERRSYS_GETQUEUE  114   /* Error creating msg queue      */
#define SVP_ERRSYS_FREESEM   115   /* Error freeing semaphore       */
#define SVP_ERRSYS_FREEQUEUE 116   /* Error freeing queue           */
#define SVP_ERRSYS_WAITQUEUE 117   /* Error waiting on queue        */
#define SVP_ERRSYS_PEEKQUEUE 118   /* Error peeking queue           */
#define SVP_ERRSYS_OPENFAIL  119   /* Error opening file            */
#define SVP_ERRSYS_QEMPTY    120   /* Error queue is empty          */
#define SVP_ERRSYS_PURGEQUE  121   /* Error purging queue           */
#define SVP_ERRSYS_VARLOCK   122   /* Error acquiring variable lock */
#define SVP_ERRPRF_FNF       201   /* Profile not found             */
#define SVP_ERRPRF_SNF       202   /* Svopid not found              */
#define SVP_ERRPRF_INVF      203   /* Invalid file                  */
#define SVP_ERRPRF_IOERR     204   /* I/O Error                     */
#define SVP_ERRPRF_INVWC     205   /* Invalid wild card             */
#define SVP_ERRPRF_NOAUTH    206   /* Authorization failed          */
#define SVP_ERRPRF_CRYPT     207   /* Could not load crypt routine  */
#define SVP_ARGERR_INVPCBX   301   /* Invalid PCB Index             */
#define SVP_ARGERR_INVSCBX   302   /* Invalid SCB Index             */
#define SVP_ARGERR_INVNAME   303   /* Invalid Variable name         */
#define SVP_ARGERR_NOLOCK    304   /* Lock not held on post-op      */
#define SVP_ARGERR_INVTOKEN  305   /* Invalid token on SVEVENT      */
#define SVP_ARGERR_INVPARENT 306   /* Parent not signed on          */
#define SVP_ARGERR_INVOSN    307   /* Invalid Offer Sequence Number */
#define SVP_ARGERR_INVSPI    308   /* Invalid Process Number        */
```

SVP Event Codes

```
#define SVP_EVENT_OFFEREX     0x01     /* Offer extended          */
```


© Copyright IBM Corporation 1994, 2017

```
#define SVP_EVENT_OFFERAC     0x02     /* Offer accepted           */
#define SVP_EVENT_SMAVAIL     0x04     /* Shared Memory Available  */
#define SVP_EVENT_PRETRACT    0x08     /* Partner retracted        */
#define SVP_EVENT_PSETACV     0x10     /* Partner set ACV          */
#define SVP_EVENT_PSPEC       0x20     /* Partner specified        */
#define SVP_EVENT_PREF        0x40     /* Partner referenced       */
#define SVP_EVENT_FSPEC       0x80     /* Partner failed spec      */
#define SVP_EVENT_FREF        0x100    /* Partner failed ref       */
#define SVP_EVENT_SHUTDOWN    0x200    /* SVP shutdown             */
#define SVP_EVENT_PRELEASE    0x400    /* Partner released var     */
```

Function Prototypes

```
int _System svpp(struct prb *);
int _System svps(struct srb *);
int _System svpe(struct wrb *);
void _System svsleep(unsigned long);
unsigned long _System aplobjsize(char, char, unsigned long);
void * _System APV2Integer(void *);
void * _System Boolean2Integer(void *);
void * _System Matrix2VectorOfVectors(void *, int);
int _System Nested2Simple(void *);
int _System CompareCdrs(void *, void *);
char * _System GetEnvOpt(char *, int, char **);
char * _System GetEnvLib(int);
char * _System GetSvpOpt(char *);
char * _System GetLibraryStrings(void);
int _System msgs_to_qets(long, long, long *, long *);
```

Access Control Constants

```
/* sv state    */
#define INITSTATE      0x03
#define PARTSPEC       0x05
#define USERSPEC       0x0A
/* sv coupling */
#define ISSHARED       0x02
#define ISOFFER        0x01
#define ISNTSHAR       0x00
/* sv acv      */
#define ACVMYSET       0x08
#define ACVPARTSET     0x04
#define ACVMYUSE       0x02
#define ACVPARTUSE     0x01
```


© Copyright IBM Corporation 1994, 2017

Common Data Representation

Data passed from APL2 to an auxiliary processor, and data passed back to APL2, must be in a special format. If you pass invalid data to APL2, unpredictable errors may occur.

Each APL2 object contains information that describes its data type, shape, size, and origin. This information is called its header, and is located at the beginning of the object. The header consists of the following fields (defined in C include file `aplobj.h`):

```
typedef struct aplobj {
  unsigned long  ptr      ;
  unsigned long  nb       ;
  unsigned long  nelm     ;
  unsigned char  type     ;
  unsigned char  rank     ;
  unsigned char  fill[2]  ;
  unsigned long  dim[1]   ;
} APLOBJ ;
```

`ptr` An identifier that indicates the system on which the object was built. The valid values are indicated by a set of C #define's:

`#define CDRid6000 0x40400000`
indicates a 32-bit system without numeric byte reversal (AIX, Solaris).

`#define CDRidOS2 0x00002020`
indicates a 32-bit system with numeric byte reversal (OS/2, Linux, Windows).

Auxiliary processors should set the value of this field for objects they create. The constant CDRID provides the correct value for the running system.

```
     #if defined(__OS2__)  || defined(__WINDOWS__) || defined(WIN32) ||
defined(LINUX)
        #define CDRID 0x00002020
     #else
        #define CDRID 0x40400000
     #endif
```

`nb` The number of bytes in this APL2 object. If the datatype of this object is that of a nested array the byte count must include this object and all of its subitems. (Note: this is different from the APL2 objects passed to `:link.FUNCTION` routines.) The length of each object must be rounded to an even multiple of 16.

`nelm` The number of elements in this APL2 object.

`type` APL2 object type:

```
0  (BOOLEAN)    Boolean             1 bit per item
1  (INTEGER)    Integer             4 bytes per item
2  (FLOAT)      Real                8 bytes per item
3  (COMPLEX)    Complex             16 bytes per item
4  (CHARACTER)  ASCII Character     1 byte per item
5  (CHARLONG)   Extended Character  4 bytes per item
6  (APV)        Progression Vector  8 bytes
7  (NESTED)     General Array       4 bytes per item
```

`rank` Rank of object (0-63).

`fill` Unused; should be set to 0.

© Copyright IBM Corporation 1994, 2017

`dim[]` Length of each dimension (number of elements in dim = rank)

Immediately following the header for each object is the data associated with the object. The length of the data for each type is shown above. If the object has more than one dimension, its elements are stored in row order (as if the APL2 primitive Ravel had been applied to the variable).

Immediately following the data are enough fill bytes to make the length of the object an even multiple of 16.

Special Notes for General Arrays

1. General arrays (type 7) have a recursive structure. Their data section consists of one 4-byte word per item, each containing the offset from the beginning of the general object to the corresponding subitem. The subitems (each of which can also be a general array) follow in left-list order.
2. General arrays that are empty (nelm=0) must have a prototype definition, so their data section consists of a single 4-byte offset field, which is the offset from the beginning of the general array to the beginning of the object that contains the prototype.

Byte Reversal

On systems where numbers are stored in byte-reversed order, all numeric 4-byte header fields (`nb`, `nelm`, and `dim`) and all numeric data are expected to be stored in the object in byte-reversed order. However, note that Boolean data is stored as 1 bit per element, 8 bits per byte. It is not stored in byte-reversed order.

© Copyright IBM Corporation 1994, 2017

The SVP Monitor Facility

Note: This section does not apply to Unix systems. On Unix systems, the SVP Monitor is a simple output-only window for display of trace messages.

The APL2 SVP monitor facility provides a variety of services to aid in the understanding of shared variable processor events.

Error messages and traces of those events are displayed in its window. It includes dialogs which provide statistics of SVP usage, information about processors that are signed on to the SVP, and variables that are shared by those processors. The dialogs can also be used to control tracing selectively and to send shutdown or kill requests to processors. Menu items are also provided to start, stop, and issue commands to the APL2 port server (used for cooperative processing).

There is never more than one monitor window active at a time, and it shows events for all processors on the system that are using the SVP.

Note: You might want to start the SVP monitor manually before starting APL2 to avoid the overhead of SVP initialization when invoking APL2. If you have TCP/IP installed, the SVP monitor must be started after TCP/IP is fully initialized.

- Starting the SVP Monitor Facility
- Event Traces and Messages
- Menu Options

© Copyright IBM Corporation 1994, 2017

Starting the SVP Monitor Facility

The SVP Monitor is started automatically only if the invocation option `-svptrace` is specified when starting a processor.

It can also be started explicitly by clicking on the *SVP Monitor* icon or by invoking `apl2svpt.exe` from an operating system command window.

The syntax is:

```
apl2svpt [optional parameters]
```

The following invocation options may be used:

`-trace [on|`**`off`**`|log|both]`
`-svptrace [on|`**`off`**`|log|both]`

Enables display and/or logging of shared variable events.

`on` sends trace messages to the *SVP Monitor* window.
`log` sends trace messages to a file.
`both` sends trace messages to the window and a file.

The default file name used is `apl2svp.trc` in the current directory. This can be changed by setting environment variable APL2SVPLOG, or the SVPLOG keyword in the [Shared Variable Processor] section of the `apl2.ini` file, to any valid operating system file name.

`-trace` and `-svptrace` are equivalent, except that `-svptrace` takes effect immediately and `-trace` takes effect after the SVP Monitor is initialized.

`-listen [on|`**`off`**`]`
`-svplisten [on|`**`off`**`]`

Specifying `on` starts the APL2 port server and TCP/IP server for cooperative processing.

`-listen` and `-svplisten` are equivalent, except that `-svplisten` is processed immediately and `-listen` is processed after the SVP Monitor is initialized.

Both of the options can also be specified in the `apl2.ini` configuration file, or by using environment variables. In `apl2.ini`, add keyword definitions for SVPLISTEN and/or SVPTRACE in the [Invocation Options] section. The environment variables correspond to the invocation options with the character "APLSVP" prefix. For example:

```
SET APLSVPTRACE=ON
```

is equivalent to starting `apl2svpt` with the `-svptrace on` option.

© Copyright IBM Corporation 1994, 2017

Event Traces and Messages

In most cases, SVP requests generate two trace messages; one on entry to the service and one on exit. If the exit return code is not 0, an additional message can be generated. During initialization trace messages are also displayed which give information about the system environment, including the time and date that the SVP was started and the amount of shared memory allocated for processor and variable control blocks.

When SVP tracing is in effect, trace messages are output to the *SVP Monitor* window.

When SVP logging is in effect, trace messages are written to a file. The log file is always cleared when it is opened, so it shows trace information from only one instance of the trace window. The default file name used is `apl2svp.trc` in the current directory. This can be changed by setting environment variable APL2SVPLOG, or the SVPLOG keyword in the [Shared Variable Processor] section of the `apl2.ini` file, to any valid operating system file name.

SVP trace messages are color coded as follows:

Blue	SVP initialization
Black	SVP trace
Red	SVP error
Green	APL2 port server
Light Cyan	APL2 TCP/IP server
Dark Cyan	Cross-system SVP trace messages

© Copyright IBM Corporation 1994, 2017

Menu Options

This section describes each of the menu options available on the *SVP Monitor* window.

- File
- Options
- Actions
- Info
- Help

File

The following choices appear in the **File** menu:

Printer Setup
: Presents a dialog to select and set up the printer.

Print
: Prints the SVP trace queue.

Exit
: Closes the window and shuts down the SVP monitor facility.

Options

The **Options** menu allows you to control the display of SVP trace messages. The following choices appear in the Options menu:

Trace on
: Toggle for displaying SVP trace entries in the trace window.

Logging on
: Toggle for writing SVP trace entries to a file. The default file name is `apl2svp.trc` in the current directory. This can be changed by setting environment variable APL2SVPLOG, or the SVPLOG keyword in the [Shared Variable Processor] section of the `apl2.ini` file, to any valid operating system file name.

Timestamp on
: Toggle for including timestamps with SVP trace entries.

Actions

The **Actions** menu allows you to affect the behavior of the Shared Variable Processor. The following choices appear in the Actions menu:

Cross System
: Starts and stops the APL2 port server and APL2 TCP/IP server. These programs must be started to use cooperative processing with a remote APL2 system.

Port Server
: Issues commands to the APL2 port server.

© Copyright IBM Corporation 1994, 2017

Purge Queue
Empties the SVP trace queue. This does not affect the log file, if any.

A dialog appears while starting the cross system programs. The Port Server is used to identify processors by their TCP/IP port number. You can select the TCP/IP port number that it uses for itself. The default is 31415.

Info

The **Info** menu contains choices that display information about processors using the SVP.

Statistics

The **Statistics** dialog displays the following information:

Maximum Processors
The maximum number of processors allowed to sign on to the SVP.
Processors signed on
The total number of processors currently signed on to the SVP.
Maximum Variables
The maximum number of variables that can be shared at one time.
Variables in use
The total number of variables currently shared or offered.
Maximum Shared Memory
The maximum size (in bytes) of APL2 shared memory.
Shared Memory in use
The total amount of memory (in bytes) currently in use by shared variables for data.
Maximum per Variable
The maximum size (in bytes) of one shared variable.

The **Maximum** values are controlled by the SVP parameter file. The APL2SVPPARMS environment variable, or the SVPPARMS keyword value in the [Shared Variable Processor] section of the `apl2.ini` configuration file, can be used to specify the name of this file. If the file name is not established using either of those methods, the default is `apl2svp.prm`. For more information, see The SVP Parameter File.

Processors

The **Processors** dialog displays a list of processors that are signed on to the SVP. Dependent processors have their parent processor and grandparent processor (if any) displayed in parentheses after the processor number. For example:

```
120 (1001)
```

This refers to processor 120, which is a child of processor 1001.

If a processor is selected in the list, the following buttons are activated:

Info
Displays information about the selected processor.

© Copyright IBM Corporation 1994, 2017

Variables

Displays a dialog listing the variables currently shared with, offered to, or offered from the selected processor.

Shutdown

Sends an SVP Shutdown signal to the selected processor. See SVP Programming Interface for more information.

Kill

Removes all information about the processor from the SVP and sends a kill signal to the process associated with the selected processor. This should only be attempted as a last resort. The SVP is not always able to clean up properly after an auxiliary processor has been killed.

Help

Use the choices on the **Help** menu to display:

- An index of help topics
- General help about the SVP Monitor Facility
- Information about using Help
- A list and description of the keys you can use
- Product level information

© Copyright IBM Corporation 1994, 2017

The APL2 Runtime Library

The APL2 Runtime Library is a subset of the full APL2 system. It provides the environment necessary to run applications written using the full APL2 system, but within the restrictions documented for the Runtime Library. The APL2 Runtime Library is available for all platforms supported by APL2 Version 2.

The APL2 Runtime Library enables developers to freely distribute applications which do not require the development facilities of the full APL2 system. Using the APL2 Runtime Library end-users can run APL2 applications without purchasing and installing the full APL2 product.

The APL2 Runtime Library can be used to run:

- Namespaces created using the external function `CNS`, using the `-rns` invocation option.
- APL2 statements and scripts, using the `-input` and `-sm piped` invocation options.
- Packages created by the APL2 Runtime Environment for Windows workspace packager, using the `-run` invocation option.

The APL2 Runtime Library cannot be used to run applications which require interactive input through the session manager or use library system commands (such as `)LOAD`) to manage the application code.

- Restrictions of the Runtime Library
- Testing your Application with the Runtime Interpreter
- Distributing the Runtime Library
- Installing the Runtime Library

© Copyright IBM Corporation 1994, 2017

Restrictions of the Runtime Library

The APL2 Runtime Library interpreter supports all the features of the full APL2 interpreter except for the following restrictions:

- The `-sm` invocation option does not support the value `on`. The default value is `off`.
- Even if a CONTINUE workspace exists, it is not loaded.
- The `)CONTINUE` system command does not save a CONTINUE workspace.
- The following system commands are not supported:

```
)COPY       )LIB        )PIN
)DROP       )LOAD       )SAVE
)EDITOR     )OUT        )WSID
)IN         )PCOPY
```

- When the `-sm off` invocation option is used, the interpreter terminates when the statements supplied in the `-input` invocation option are exhausted and the AP 101 stack is empty.

© Copyright IBM Corporation 1994, 2017

Testing your Application with the Runtime Interpreter

The APL2 Runtime Library need not be installed on a system where the full APL2 product is already installed.

You can test your application to make sure it will run properly with the APL2 Runtime Library, using the full APL2 product. The runtime version of the interpreter is shipped with the full APL2 product, and sample invocation command files are provided.

On Windows, sample file `apl2runt.bat` is provided in the `\IBMAPL2W\samples` directory. On Unix systems, shell script `apl2runt` is provided in the `/APL2/bin` directory.

You will need to either modify the sample script to add the invocation parameters needed to start your application, or pass the invocation parameters to the script on the command line. Without additional input from invocation parameters, the runtime interpreter will terminate immediately.

© Copyright IBM Corporation 1994, 2017

Distributing the Runtime Library

Once your application has been tested with the runtime interpreter, you will want distribute it to your users together with the APL2 Runtime Library.

The APL2 Runtime Library may be redistributed without charge. The APL2 Runtime Library includes all the components of the full APL2 product except the following:

- Full APL2 interpreter
- Session manager
- Object editor
- Dialog editor
- File editor
- Library Manager
- Public workspaces (Objects from public workspaces may be copied into your applications and saved with them before distribution).
- Certain supplied external routines:

  ```
  APL2CFG     COMBROWSE   EDITOR_2    IDIOMSG     PCOPY       WSCOMP
  APL2LM      COPY        IDIOMS      LIB         PRINTWSG
  ```

- Online documentation

For each operating system platform supported, the `/runtime` subdirectory of the main APL2 installation directory contains installation files for the APL2 Runtime Library. You must distribute the complete installation file as provided. Individual components of APL2 may not be redistributed.

Your application containing the APL2 Runtime Library must be labeled as follows:

> CONTAINS Runtime Modules of IBM Workstation APL2 for Multiplatforms Version 2.0
> (c) Copyright IBM Corporation 1991-2008 All Rights Reserved

See the **LICENSE INFORMATION** section of the `README` file in the main APL2 directory for complete terms and conditions of redistribution.

You may incorporate the installation of the APL2 Runtime Library into your application's installation process, or you may require the user to install the APL2 Runtime Library as a pre-requisite to installing and running your application. The next section describes the installation process for each platform.

© Copyright IBM Corporation 1994, 2017

Installing the Runtime Library

AIX

The `/runtime` subdirectory of the main APL2 directory contains an installp image named `apl2ar20.installp`

To install the APL2 Runtime Library on AIX:

1. Copy the `.installp` file to the end-user's machine in binary mode.
2. Use the `installp` command to install the product:

   ```
   installp -ac -X -d APL2run.obj
   ```

3. Exit from root authority.

Use `apl2run` to invoke the APL2 Runtime Library interpreter. You will probably want to provide a cover shell script that invokes `apl2run` with the appropriate parameters to start your application.

Linux

The `/runtime` subdirectory of the main APL2 directory contains an install script named `apl2lr20` and a tarred, gzipped install file named `apl2lr20.tgz`.

To install the APL2 Runtime Library on Linux:

1. Copy the install script and `.tgz` file to the end-user's machine in binary mode.
2. Change to the directory where the files have been copied.
3. Make the install script executable:

   ```
   chmod 755 apl2lr20
   ```

4. Switch to root authority:

   ```
   su root
   ```

5. Run the install:

   ```
   ./apl2lr20
   ```

6. Exit from root authority.

Use `apl2run` to invoke the APL2 Runtime Library interpreter. You will probably want to provide a cover shell script that invokes `apl2run` with the appropriate parameters to start your application.

Sun Solaris

© Copyright IBM Corporation 1994, 2017

The `/runtime` subdirectory of the main APL2 directory contains an install script named `apl2sr20` and a tarred, compressed install file named `apl2sr20.tarz`.

To install the APL2 Runtime Library on Solaris:

1. Copy the install script and `.tarz` file to the end-user's machine in binary mode.
2. Change to the directory where the files have been copied.
3. Make the install script executable:

   ```
   chmod 755 apl2sr20
   ```

4. Switch to root authority:

   ```
   su root
   ```

5. Run the install:

   ```
   ./apl2sr20
   ```

6. Exit from root authority.

Use `apl2run` to invoke the APL2 Runtime Library interpreter. You will probably want to provide a cover shell script that invokes `apl2run` with the appropriate parameters to start your application.

Windows

The `\runtime` subdirectory of the main APL2 directory contains a self-extracting executable named `apl2wr20.exe`.

To install the APL2 Runtime Library on Windows:

1. Copy the self-extracting executable to the end-user's machine in binary.
2. Run `apl2wr20.exe`.

Use `apl2run.exe` to invoke the APL2 Runtime Library interpreter. You will probably want to provide a custom icon or command file that invokes `apl2run.exe` with the appropriate parameters to start your application.

Note: On some versions of Windows, if the installation process is performed from a userid that is a member of the administrator group, you may be prompted to choose whether the installation will be for the current user or all users.

© Copyright IBM Corporation 1994, 2017

Using The X Window System Interface

Note: This section applies only to Unix systems.

AP144 is an interface between APL2 and the X Window System. It enables the full set of X Window System Xlib calls and data structures to be used from the APL2 environment, and in so doing, enables APL2 to use a true windowing environment. Several APL2 sample programs using the interface are provided in the DEMO144 workspace. One of these sample programs, the `HelloWorld` function, is explained in detail in The HelloWorld Function

To use this tutorial, you should be able to read and write programs in APL2 and C, have a good general knowledge of X Window System and be able to use the underlying operating system's facilities.

- Invoking X Window System Calls
- Deviations From Standard X Window System Call Syntax
- AP144 System Commands
- How To Use X Data Structures And Constants
- AP144 Structure Commands
- System Structures
- The HelloWorld Function

© Copyright IBM Corporation 1994, 2017

Invoking X Window System Calls

AP144 enables X Window System calls to be issued from within APL2 by associating the Xlib calls with AP144 commands. These commands are defined in several command tables, along with other information describing the command parameters and the actual C function to be executed.

The AP144 command interface:

1. Verifies the X Window System input parameters
2. Passes control to a designated X Window System call
3. Returns any output generated by this call
4. Returns extended return codes identifying any errors, if any occurred

The command interface includes some built-in *system commands*. These commands deal with the operation of the interface itself, not with the X Window System environment that can be reached through the interface. The names of these commands all start with a `)`, in keeping with APL2.

Calling X Window System from APL2

X Window System is called from APL2 using the `XWIN` function supplied in the AP144 Workspace.

The following example of the *XOpenDisplay* call illustrates the close correspondence between an X Window System call issued from C, and the same call issued from APL2. In C, the call might look like:

```
dp = XOpenDisplay("");
```

In APL2, the same call is issued through AP144 as:

```
dp ← XWIN 'XOpenDisplay' ''
```

The X Window System call name becomes the first parameter and everything else remains the same. The name is case sensitive and must be given exactly as required.

If additional parameters are needed to complete the X Window System call (and most do), they are specified in a vector, with the command name forming the first element in the vector. All parameters are mapped to either numeric or character items. This mapping is done with the help of type codes provided for each call. Each call has an associated set of input and output type codes that control the syntactic mapping of data between APL2 and X Window System. The type codes that are used by AP144 are listed in AP 144 Commands and Structures.

The `)Syntax` command lists the type codes for a specific command. `)Cmds` lists all commands as well as their associated type codes:

```
      3 XWIN ')Syntax' 'XParseGeometry'
 0    Xlib XParseGeometry S IIIII
```

See AP144 System Commands for more on these two system commands.

© Copyright IBM Corporation 1994, 2017

Results

A large proportion of the available X Window System calls generate results when they are executed. When available, these results are passed back as an explicit result of calling the `XWIN` APL2 function. The X Window System calls can produce results in two forms, explicit and implicit. Explicit X Window System results are always passed back as the first parameter of the APL2 result. Any additional output parameters (parameters specified in the X Window System documentation with a `_return` suffix) follow.

The parameter type codes are also used for the mapping of the result and output parameters. Note that one extra level of indirection for the output types is required by the C language function syntax, but the interface automatically handles this requirement.

If no result is returned by a given X Window System call, then `XWIN` does not return anything. If you try to assign the (nonexistent) result of such a call to a variable, APL2 stops with an error:

```
      dp←XWIN 'XCloseDisplay' dp
VALUE ERROR
      dp←XWIN 'XCloseDisplay' dp
         ^
```

Each call to the command interface results in a return code being generated and passed back to the caller. A non-zero return code is an indication that the command failed to execute for some reason.

If a non-zero return code is returned, and the error is found in the content of the parameter list, additional return codes can be returned in the second return parameter. These parameter return codes relate one-to-one to the parameters passed. The correctness of the first parameter (the command name) is indicated by the first parameter code. These return codes help you to locate the source of the error.

The parameter return codes also extend into nested arrays to match the structure of the parameter list.

The list of all possible return codes can be found in AP 144 Return Codes.

© Copyright IBM Corporation 1994, 2017

Deviations From Standard X Window System Call Syntax

One of the objectives of AP144 has been to provide an implementation of the X Window System Xlib in APL2 that remains as close to the C version as possible. However, some differences are inevitable due to the differences between an interactive, interpreted environment such as APL2, and the compiled environment of C. The parts of this section detail these differences.

Naming Conventions

The names of the implemented X Window System functions and structures are the same as found in the X Window System Xlib. X Window System macros are implemented in their function form, that is, with a leading `X` prefix. This is also true for the `Is...Key` group of macros. These macros do not have a function form equivalent in the native X Window System, but are implemented in AP144 as though they did:

`IsCursorKey`	is renamed `XIsCursorKey`
`IsFunctionKey`	is renamed `XIsFunctionKey`
`IsKeypadKey`	is renamed `XIsKeypadKey`
`IsMiscFunctionKey`	is renamed `XIsMiscFunctionKey`
`IsModifierKey`	is renamed `XIsModifierKey`
`IsPFKey`	is renamed `XIsPFKey`

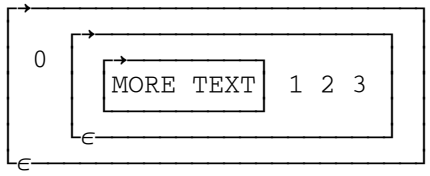
Character Strings

When a C function requires a character string as input or returns it on output, the content of the string is not passed to the function, but rather a pointer to the string. The function can then access the content of the string via the passed pointer. Another characteristic of C character strings is that they are null-terminated. That is, an extra, null character is appended at the end of the string to indicate the end of, and thus the extent of, the string.

APL2, on the other hand, passes all data by value. Not only is the data content always present, the type, rank, and dimensions are also always available to the program.

The AP144 interface performs the appropriate transformations between these two schemes. When a character string is called for by an X Window System call, it should be given as a regular APL2 character vector. For example:

```
      XWIN 'Put' 'XTextItem' h ('More text' 1 2 3)
      DISPLAY 3 XWIN 'Get' 'XTextItem' h
```



Numeric or Character String (argv) Arrays

© Copyright IBM Corporation 1994, 2017

Arrays must be passed by value from APL2 on any X Window System call requiring such input, and values are returned on output in the same way. This is analogous to the treatment of character strings described above. AP144 generates the appropriate C array and substitute the pointer to this array on the call to X Window System.

Character string array (argv) parameters must be given as an APL2 (nested) vector of strings. It is not accepted in the form of an APL2 character matrix. The generated C construct is similar in structure to the familiar C `argv` structure, that is, an array of string pointers, including an additional, final null pointer.

```
xy← 4 2⍴0 0 10 0 10 10 0 0
c ← CoordModeOrigin
XWIN 'XDrawLines' dp w gc (,xy) (↑⍴xy) c
XWIN 'XSetFontPath' dp (⊂'/usr/lpp/fonts/') 1
```

Specifying X Window System Input Parameters

AP144 maintains as close a fidelity to the X Window System call syntax as possible. One area where this may seem out of place in the APL2 environment is on calls where strings or arrays are passed as input parameters. These calls require the specification of the string or array length in addition to the data itself. Although it is possible to determine the length of any APL2 array, the AP144 interface still requires that this information be given explicitly on the call. This may be seen as a bad choice, but it does maintain consistency with the base X Window System documentation. For example:

```
t ← 'Some text to be shown'
XWIN 'XDrawString' dp w gc x y t (⍴t)
xy← 4 2⍴0 0 10 0 10 10 0 0
c ← CoordModeOrigin
XWIN 'XDrawLines' dp w gc (,xy) (↑⍴xy) c
XWIN 'XSetFontPath' dp (⊂'/usr/lpp/fonts/') 1
```

Result and Output Parameters

The X Window System calls can produce two types of output to the calling program (in addition to their effect on the windows displayed on the workstation). Some calls have explicit function results, others pass back results in or via storage addresses provided as parameters to the call, and yet others combine the two approaches. In the X Window System documentation, parameters that are used to return information usually have a name that ends with a `_return` suffix.

APL2 functions do not have the capability of passing back results in predetermined storage locations (ignoring for the moment using global variables to accomplish this). They do, however, have the possibility of returning multiple values as part of their explicit return. We use this capability for those X Window System calls that return multiple results.

If an X Window System call produces an explicit result, this result is the first element of the data returned to APL2. Any results returned via storage addresses (if any) are appended to the explicit result. These result parameters must not be specified as input to the X Window System call. They are generated automatically by the AP144 interface.

© Copyright IBM Corporation 1994, 2017

Note: The type codes of the parameters returned for a given X Window System call can be listed by using the `)Syntax` call. The return type codes are the second element returned, following the input type codes. For example:

```
      XWIN ')Syntax' 'XParseGeometry'
Xlib XParseGeometry S IIIII
      XWIN 'XParseGeometry' '10x20-30+40'
31 ┗30 40 10 20
```

Calls Using Event Structures

When structures are used in X Window System calls, the general rule is that they are passed or returned by reference. It is up to the program to set up the structure instance before the call, or to import it after the call has returned, using the AP144 structure support.

The calls that use any of the event-type structures are an exception to this general rule.

Events are always returned to APL2 by value, and as a result the value is immediately available for use. The reason for this change is performance. In the X Window System itself, the *XNextEvent* call also specifies a second parameter, a pointer to an *XAnyEvent* structure. *XNextEvent* fills in the structure fields, thereby making the event values available to the calling program.

To follow the same strategy in AP144 would require three calls through the interface:

1. Call *XNextEvent* with a pointer to an *XAnyEvent* structure.
2. Retrieve the event type by using *Get XAnyEvent*. All structures have the first four fields in common, one of which is the event type.
3. Find out which type of event it is. Use this information to retrieve the event fields by using the appropriate event structure.

The calls covered by this are:

```
XCheckMaskEvent          XNextEvent
XCheckTypedEvent         XPeekEvent
XCheckTypedWindowEvent   XPutBackEvent
XCheckWindowEvent        XRefreshKeyboardMapping
XLookupKeysym            XSendEvent
XMaskEvent               XWindowEvent
```

For example:

```
      XWIN 'XNextEvent' dp
2 11 0 6949392 8388609 524395 0 1193985641 183 135
      554 756 1 17 1
```

Events are also used as input on a few X Window System calls. They can either be specified by value, for example, in the same form as they are returned from X Window System, or they can be specified by a pointer to an event structure that has been previously allocated.

© Copyright IBM Corporation 1994, 2017

XGetEventBuffer

This call is not part of the standard X Window System Xlib set of calls, but is specific to AP144. It returns a pointer to the event buffer that is used to hold all events being returned to APL2, before the event is being translated into an APL2 array of values. It can be used on those calls that require an event pointer to be passed as a parameter, as long as the buffer has not been modified. The buffer content is updated whenever a call is made to a command that returns an event structure.

Structures Within Structures

Some structures contain sub-structures within their definition. *XSizeHints* is an example of such a structure:

```
typedef struct {
        ....
        struct {
                int x;  /* numerator */
                int y;  /* denominator */
        } min_aspect, max_aspect;
        ....
} XSizeHints;
```

In AP144, such substructures are fully expanded. Thus, in the case of *XSizeHints*, the structure fields are listed as:

```
      3 XWIN 'GetFields' 'XSizeHints'
 0    long flags        X
      ....
      int  max_aspect.x I
      int  max_aspect.y I
      int  base_width   I
      int  base_height  I
      int  win_gravity  I
```

Note: The APL2 function `axGetFF` uses the field names to generate APL2 variables by the same name. Since periods (.) cannot be a part of a valid APL2 name, they are replaced by an overbar (¯). See AP144 Workspace for more information on `axGetFF`.

Error Handling

The X Window System error handling calls *XSetErrorHandler* and *XSetIOErrorHandler* use a pointer to a C function as their one and only parameter. When an error occurs, this C function gets control, rather than the standard X Window System error handler.

It is not possible to mimic this behavior from APL2. It is not possible to set up an APL2 function that assumes control when the error occurs. Instead, AP144 has defined an error handler of its own. This error handler is a bit more lenient than the one provided by X Window System, and it enables the program or programmer to continue beyond the point at which an error occurred.

The input parameter to each of the two X Window System calls has been replaced by a `name` parameter. The valid names are:

© Copyright IBM Corporation 1994, 2017

Default The standard X Window System error handler

XErrorHandler The AP144 error handler

None The standard X Window System error handler (same as **Default**)

For example:

```
p ← XWIN 'XSetErrorHandler' 'XErrorHandler'
p ← XWIN 'XSetIOErrorHandler' 'Default'
```

The AP144 error handler is the initial one activated, when the interface is first activated.

Warning: AP144 is not notified when a user closes an AP144 window by some means other than issuing an Xlib call through AP144. For instance, some window managers allow you to close a window by clicking a button, or entering a special key sequence. Since AP144 commands typically include a pointer to the window, a fatal error occurs when the Xlib subroutine tries to access a window that no longer exists.

X Window System Calls Requiring Special Changes In APL2

XFetchBuffer and XFetchBytes
: Both of these functions return only the buffer content. The length of the buffer is not returned. The content is returned by value, hence there is no need to explicitly free the space.

XGetClassHints
: The status is returned in all cases. The class hints are returned by value, if present.

XGetKeyboardControl
: The keyboard control parameters are returned by value.

XGetNormalHints
: The status is returned in all cases. The hints are returned by value, if present.

XGetSizeHints
: The status is returned in all cases. The hints are returned by value, if present.

XGetWindowAttributes
: The status is returned in all cases. The attributes are returned by value, if present.

XGetWindowProperty
: If the specific property does not exist, this function returns an empty string, else it returns the values.

XGetWMHints
: The status is returned in all cases. The hints are returned by value, if present.

XGetZoomHints
: The status is returned in all cases. The hints are returned by value, if present.

XLookupString
: The event structure is the only input parameter from APL2. The call returns the count, the string buffer, the key symbol, and a pointer to an *XComposeStatus* structure.

XMatchVisualInfo
: The status is returned in all cases. The hints are returned by value, if present.

XQueryTree
: Status, Root, Parent, Children, and NChildren are returned as explicit parameters if Status ≠ 0; else only status is returned. The function frees up the Children array malloc space before returning.

© Copyright IBM Corporation 1994, 2017

AP144 System Commands

AP144 contains some system commands and structures that help you to control and interrogate the interface itself. The following describes these commands that can be used in the same way as the X Window System calls.

)Cmds - List the available commands

Returns an array of command names available for use by AP144 with associated parameter requirements and expected results. Separate three-column tables of structure names are returned for the specified environment, or for all active environments if no name is given on the command line.

```
(rc cmds) ← 3 XWIN ')Cmds' [env]
```

`[env] ...`
: An optional environment name. If this parameter is specified, only a single table is returned, listing the commands defined within the given environment. If the parameter is not specified, then there is a table of commands returned for each active environment.

`rc`
: The operation return code.

`cmds`
: Lists all commands available in the user command table. The list of commands is made up a three-column matrix containing a row for each command available, with the rows made up of the command name, parameter types, and expected return:

  Command: The name of the call or structure

  Input parms: The type codes of the required inputs

  Output parms: The type codes of the output parameters

Example:

```
3 XWIN ')Cmds' 'Structs'
```

Note: The absence of commands in either user or system command tables may indicate that no commands are available, or that no list command is available for the table.

)Env Get - Get the current or default list of environments

`)Env Get` gets the current environment list, or gets a list of all available environments.

All commands and structures available through AP144 reside in tables called environment tables, or just environments for short. An environment is typically a group of related commands and structures. As an example, all the X Window System calls are grouped into a single environment called Xlib. When searching for a command, the environment tables are searched in the order given by the environment list. This environment list can be manipulated by the `)Env Get` and `)Env Set` commands.

```
(rc env_list) ← 3 XWIN ')Env' 'Get' ['Default']
```


© Copyright IBM Corporation 1994, 2017

`rc`
: The operation return code.

`env_list`
: If `'Default'` is specified, then the default list of environments is returned. If it is not specified, the currently active environment list (as set by `)Env Set`) is returned.

Examples:

```
3 XWIN ')Env' 'Get'
3 XWIN ')Env' 'Get' 'Default'
```

)Env Set - Change the list of environments

This command allows you to change the list of environments. You can rearrange the order in which the environments appear in the list, leave out some environments entirely, or add ones previously not active. Using the command completely replaces the current list; thus if you want to retain any or all of the current environments, you need to specify them in the command.

```
rc ← 3 XWIN ')Env' 'Set' env_list
```

`env ...`
: The new list of environments to be used, in the order given. Only valid environment names will be accepted; invalid names are just ignored. The present list is left intact if no valid names are specified.

`rc`
: The operation return code.

Example:

```
3 XWIN ')Env' 'Set' 'System'
```

Note: Commands can be accessed even though the environment they are included in are not part of the current environment list. To do so you must specify the environment name as part of the command itself, separated from the command name by a period (.). As an example, `'System.)Cmds'` can be used to list all available commands, even though `System` is not currently part of the environment list.

)RC - List a return code message

Return the message associated with a single return code.

```
(rc erc) ← 3 XWIN ')RC' rc_no
(rc_no name msg) ← erc
```

`rc_no`
: The return code number whose message you want retrieved.

`rc`
: The operation return code.

`name`
: An internal name used to identify the message within AP144.

`msg`
: The message relating to that return code. The message returned is the C character string used during parameter verification tracing, so it may contain C *printf()* substitution variables.

Example:

```
3 XWIN ')RC' 24
```


© Copyright IBM Corporation 1994, 2017

)Structs - List the available structures

Return an array of structures available for use by AP144. Separate one-column tables of structure names are returned for the specified environment, or for all active environments if no name is given.

```
(rc structs) ← 3 XWIN ')Structs' [env]
```

`[env]`
: An optional environment name. If this parameter is specified, only a single table is returned, listing the structures defined within the given environment. If the parameter is not specified, a table of structures is returned for each active environment.

`rc`
: The operation return code.

`structs`
: If `[env]` is set, a single, one-column table listing all the structures defined within this environment. This may be an empty table.
If `[env]` is not set, then `structs` contains an n-by-1 matrix of tables, with one table for each active environment. Each of these tables lists the structures defined within the matching environment.

Example:

```
3 XWIN ')Structs' 'Structs'
```

)Syntax - List the syntax of a specific command

Return the input and output type codes for a named command. .

```
(rc syntax) ← 3 XWIN ')Syntax' cmd
(env cmd intypes outtypes) ← syntax
```

`cmd`
: Name of the command.

`rc`
: The operation return code.

`env`
: Environment in which the command is defined.

`intypes`
: A vector of type codes used to validate input parameters to the specific command.

`outtypes`
: A vector of type codes specifying the function return types.
If the first element is specified, it always refers to the explicit return from the C function. If no explicit return is available, the type code is given as a _, and no return from the (missing) explicit return is passed back to APL2. Subsequent type codes refer to the output parameters.

Example:

```
3 XWIN ')Syntax' 'XParseGeometry'
```

)Version - List the AP144 version identifier

Return a character string identifying the currently implemented version of AP144.

© Copyright IBM Corporation 1994, 2017

```
(rc version) ← 3 XWIN ')Version'
```

`version`
: A character string identifying the installed version of AP144.

`rc`
: The operation return code.

Example:

```
3 XWIN ')Version'
```


© Copyright IBM Corporation 1994, 2017

How To Use X Data Structures And Constants

X defines a large number of C data structures and constants for its own use. These structures and constants are also available for use by APL2 via the AP144 interface.

Each defined C data structure is mapped to an APL2 vector. The vector may be simple or nested, depending on the underlying C definition. When in APL2, the array can be processed in normal APL2 fashion. For each structure defined to the interface there is a corresponding command that lets you create, change, and delete structure instances of the given type. Each of these commands use a common set of subcommands that is described later in this chapter. The instances are stored in space controlled by C and thus directly available to X. To be processed by APL2 the instances must be imported or exported to and from APL2. The subcommands perform this task, as well as the basic chores of allocating space for a new structure instance, getting data into and out of it, and freeing up the space when it is no longer needed.

The structure field names and types are also available to APL2 via the interface. This is useful when using the data structures from within APL2, in that it associates each element in the vector with its related field name in C.

Constants defined in the C header files are also available in the same way. An APL2 utility function is provided to load these constants into the active workspace. The constants are grouped with the structures they relate to, so only relevant constants are loaded.

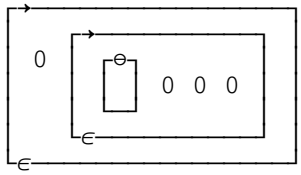
Introductory Examples

The structure commands are illustrated by way of examples. The examples use the *XTextItem* structure found in the X Window System Xlib.h file. This structure is defined as follows in the X Window System:

```
typedef struct {
        char *chars;    /* Pointer to string                  */
        int   nchars;   /* Number of characters               */
        int   delta;    /* Delta between strings              */
        Font  font;     /* Font to be used, None don't change */
} XTextItem;
```

The first set of commands performs some very basic manipulation on a simple structure:

```
⍝ Create a new XTextItem instance
      s ← XWIN 'New' 'XTextItem'
      s
5897984
⍝ Initially it is empty
      XWIN 'Get' 'XTextItem' s
  0 0 0
⍝ The first item is hard to spot, so ...
      DISPLAY 3 XWIN 'Get' 'XTextItem' s
```



© Copyright IBM Corporation 1994, 2017

```
⍝ Now fill it with data
      XWIN 'Put' 'XTextItem' s ('Simple' 1 2 3)
⍝ Verify that the data made it in
      XWIN 'Get' 'XTextItem' s
 Simple 1 2 3
```

At this point, the data structure is ready and available for use from any X call requiring a pointer to a structure as an input parameter:

```
⍝ Data can also be reset
      XWIN 'Clear' 'XTextItem' s
⍝ ... as can be seen here
      XWIN 'Get' 'XTextItem' s
 0 0 0
⍝ Remember to free when all done
      XWIN 'SFree' 'XTextItem' s
```

XDrawText is one of the X Window System calls that use the *XTextItem* structure. Ignoring for a second the preceding calls needed to set up the proper display, window and graphics context variables, a call to *XDrawText* using the above structure (before it was freed) looks like:

```
      XWIN 'XDrawText' dp w gc s
```

Continuing .....

```
⍝ Create and Put can be combined
      s ← XWIN 'NewPut' 'XTextItem' ('New & Put' 4 5 6)
⍝ ... as can be seen here
      XWIN 'Get' 'XTextItem' s
 New & Put 4 5 6
⍝ Again, remember to free when done
      XWIN 'SFree' 'XTextItem' s
```

X often refers to multiple instances of a given structure via a single pointer. The structures are stored adjacent to one another. AP144 uses the *MNew*, *MGet* and *MFree* commands to support these constructs; essentially they are multistructure versions of the *New*, *Get*, *Put*, and *Free* calls.

```
⍝ Multiple instances are also possible
      s ← XWIN 'MNew' 'XTextItem' 3
⍝ Fill the first instance like this
      XWIN 'Put' 'XTextItem' s ('First' 1 11 111)
⍝ ... or like this
      XWIN 'MPut' 'XTextItem' s 0 ('First' 1 11 111)
⍝ Store the second like this
      XWIN 'MPut' 'XTextItem' s 1 ('Second' 1 11 111)
⍝ Or both at once
      aaa ← ('aaa' 1 1 1)
      bbb ← ('bbb' 2 2 2)
      XWIN 'MPut' 'XTextItem' s 0 aaa bbb
⍝ Get them all at once
      XWIN 'MGet' 'XTextItem' s 0 3
  aaa 1 1 1    bbb 2 2 2     0 0 0
⍝ Clear them
      XWIN 'MClear' 'XTextItem' s 0 3
⍝ MFree frees all in one call
      XWIN 'MFree' 'XTextItem' s 3
```


© Copyright IBM Corporation 1994, 2017

The mapping of data from APL2 to the X Window System or back is controlled by a definition stored internally in AP144. This definition ensures that the needed validations and proper conversions take place before the data is actually copied from one environment to the other. The information in this definition is also directly available to the user of the AP144 interface:

```
⍝ Get all XTextItem fields
      XWIN 'GetFields' 'XTextItem'
 char *chars S
 int nchars  I
 int delta   I
 Font font   I
⍝ ... and constants
⍝     (but XTextItem doesn't have any defined)
      XWIN   'GetConst' 'XTextItem'
⍝ ... so get the constants in XSizeHints
⍝     (another X structure) instead
      XWIN 'GetConst' 'XSizeHints'
 USPosition  X    1
 USSize      X    2
 PPosition   X    4
 PSize       X    8
 PMinSize    X   16
 PMaxSize    X   32
 PResizeInc  X   64
 PAspect     X  128
 PBaseSize   X  256
 PWinGravity X  512
 PAllHints   X 1020
⍝ An XTextItem instance requires this many bytes:
      XWIN 'GetSize' 'XTextItem'
16
```


© Copyright IBM Corporation 1994, 2017

AP144 Structure Commands

All the defined structure commands use the same format:

```
(rc [result]) ← 3 XWIN command struct [parms]

or,

[result] ← XWIN command struct [parms]
```

`struct`
: The name of the structure to be used.

`command`
: The name of the action to be performed. These are all defined below.

Some of the commands require that one or more additional parameters be specified, as indicated by the `[parms]` items. Two common parameters are:

`handle`
: The handle (reference) to the structure instance

`values`
: The structure instance values as a nested array

If the `C` interface function is used , then a return code is returned in all cases:

`rc`
: The operation return code

Clear - Clear a structure instance

Clear is used to reset a structure instance to the state of a new instance. Numeric fields are set to `0`, and character strings and other pointer fields to NULL. Any space pointed to by the character string fields are assumed to have been malloc'd and are freed by the call.

```
rc ← 3 XWIN 'Clear' struct handle
```

`struct`
: The name of a command referring to an implemented structure.

`handle`
: Handle to the structure instance to be used.

`rc`
: The operation return code.

Example:

```
3 XWIN 'Clear' 'XSizeHints' 3524424
```

Get - Build an APL2 vector from a C structure instance

© Copyright IBM Corporation 1994, 2017

Get extracts the data previously stored in a structure instance and returns it to APL2 in the form of a vector of elements. The structure instance is identified by its handle.

```
(rc array) ← 3 XWIN 'Get' struct handle
```

`struct`
: The name of a command referring to an implemented structure.

`handle`
: Handle to the structure instance to be used.

`rc`
: The operation return code.

`array`
: A vector of elements representing each field in the referenced structure. The type of each element is determined by the field type as defined in the structure definition.

Example:

```
3 XWIN 'Get' 'XSizeHints' 3524424
```

GetConst - Get all constants belonging to a particular structure

Retrieve a list of all the constants associated with a given structure. These constants are defined at the same time as the actual structure, and tend to contain C's #define constants used by the structure, although there is no requirement that this be the case.

```
(rc const) ← 3 XWIN 'GetConst' struct
```

`struct`
: The name of a command referring to an implemented structure.

`rc`
: The operation return code.

`const`
: An n-by-3 matrix, with each row listing a defined constant in the structure. The first column in the matrix contains the name of the constant, almost invariably the same name as used in a C #define statement. The second column returns the constant type, again using the same type codes as used for parameter verification. The third column contains the value of the constant.

Example:

```
3 XWIN 'GetConst' 'XSizeHints'
```

GetFields - Get all fields of a particular structure

Get the list of a structure's field definitions and types.

```
(rc fields) ← 3 XWIN 'GetFields' struct
```

`struct`
: The name of a command referring to an implemented structure.

`rc`
: The operation return code.

`fields`

© Copyright IBM Corporation 1994, 2017

An n-by-2 matrix, with each row listing a defined field in the structure. The first column contains the field name, and the second column the field type. The field type uses the same parameter type codes that were used to describe the command parameters.

Example:

```
3 XWIN 'GetFields' 'XSizeHints'
```

GetSize - Get the size of a structure instance when allocated

Get the space needed (in bytes) to store an instance of a structure. The space needed comes out of the pool of space managed by the C environment.

```
(rc size) ← 3 XWIN 'GetSize' struct
```

`struct`
: The name of a command referring to an implemented structure.

`rc`
: The operation return code.

`size`
: The number of bytes needed to store an instance.

Example:

```
3 XWIN 'GetSize' 'XSizeHints'
```

MClear - Clear a number of consecutive structure instances

MClear resets a number of structure instances that are stored in consecutive storage locations. The structure instances are identified by the overall handle, the starting offset and the number of structures to be cleared.

```
rc ← 3 XWIN 'MClear' struct handle start count
```

`struct`
: The name of a command referring to an implemented structure.

`handle`
: Handle to the structure instance to be used.

`start`
: The index (in origin 0) of the first structure to be cleared.

`count`
: The number of structure instances to be cleared.

`rc`
: The operation return code.

Example:

```
3 XWIN 'MClear' 'XTextItem' 5897984 0 2
```

MFree - Free multiple structure instances at once

MFree is used to free the storage space occupied by multiple consecutive structure instances, when none of them are no longer needed. It is the responsibility of the user to determine when an instance is no longer needed, and to call the MFree subcommand to make the space available for other use.

© Copyright IBM Corporation 1994, 2017

```
rc ← 3 XWIN 'MFree' struct handle count
```

`struct`
: The name of a command referring to an implemented structure.

`handle`
: Handle to the structure instance to be used.

`count`
: The number of structure instances stored in consecutive storage areas.

`rc`
: The operation return code.

Example:

```
3 XWIN 'MFree' 'XTextItem' 5897984 2
```

MGet - Build an APL2 nested array from multiple C structure instances

MGet extracts the data previously stored in one or more structure instances stored consecutively and returns it to APL2 in the form of a vector of structures, where each structure is a vector of elements. The structure instances are identified by the overall handle, the starting offset and the number of structures to be returned.

```
(rc vals) ← 3 XWIN 'MGet' struct handle start count
```

`struct`
: The name of a command referring to an implemented structure.

`handle`
: Handle to the structure instance to be used.

`start`
: The index (in origin 0) of the first structure to be returned.

`count`
: The number of structure instances to be returned.

`rc`
: The operation return code.

`vals`
: A vector of structure instances. Each structure is a vector itself, with its elements representing the fields in the referenced structure. The type of each element is determined by the field type as defined in the structure definition.

Example:

```
3 XWIN 'MGet' 'XTextItem' 5897984 0 2
```

MNew - Create multiple new abutting structure instances

Create new instances of a particular structure, allocating its space from C's free-space pool. The call returns a handle to the new instances of the structure. This handle must be used on all subsequent calls to the structure, as it uniquely identifies the particular instance.

This mimics declaring a structure in C. In both cases some amount of storage is set aside to be used by an instance of the structure.

© Copyright IBM Corporation 1994, 2017

```
(rc handle) ← 3 XWIN 'MNew' struct count
```

`struct`
: The name of a command referring to an implemented structure.

`count`
: The number of structure instances to be allocated. The structures are stored in consecutive storage areas.

`rc`
: The operation return code.

`handle`
: The structure instance handle that must be used in all future reference to the structure instance. It uniquely identifies the new instances. Only the handle to the first structure is returned. The subsequent instances can be accessed by using the MClear, MFree, MGet and MPut commands.

Example:

```
3 XWIN 'MNew' 'XTextItem' 2
```

Note: The instances created with MNew can only be freed as a block. Use MFree to free all at once.

MPut - Fill multiple structure instances with data

Fill multiple structure instances with data passed from APL2. The data is stored in structure instances assumed to be stored in consecutive storage locations. The initial storage location is identified by a handle and an offset count.

```
rc ← 3 XWIN 'MPut' struct handle start array ...
```

`struct`
: The name of a command referring to an implemented structure.

`handle`
: Handle to the structure instance to be used.

`start`
: The offset count of the first structure instance to be used to store data in.

`array`
: The data to be stored in the structure. The data must be specified in form of a vector of elements. Each data element is verified to be of a correct type for that field before being stored in the structure.

  Multiple value arrays can be specified. They are stored in consecutive elements, starting at the `start` element. It is the caller's responsibility to ensure that the indices remain within the allocated range.

`rc`
: The operation return code.

Example:

```
3 XWIN 'MPut' 'XTextItem' 5897984 0
```

New - Create a new structure instance

Create a new instance of a particular structure, allocating its space from C's free-space pool. The call returns a handle to this new instance of the structure. This handle must be used on all subsequent calls to the structure, as it uniquely identifies the particular instance.

© Copyright IBM Corporation 1994, 2017

This mimics declaring a structure in C. In both cases some amount of storage is set aside to be used by an instance of the structure.

```
(rc handle) ← 3 XWIN 'New' struct
```

`struct`
: The name of a command referring to an implemented structure.

`rc`
: The operation return code.

`handle`
: The structure instance handle that must be used in all future reference to the structure instance. It uniquely identifies the new instance of the structure.

Example:

```
3 XWIN 'New' 'XSizeHints'
```

NewPut - Create a new structure instance and fill it with data

Create a new instance of a particular structure, allocating its space from C's free-space pool. Then initialize the structure with data passed on the call. The call returns a handle to the new instance. This handle must be used on all subsequent calls to the structure, as it uniquely identifies the particular instance.

```
(rc handle) ← 3 XWIN 'NewPut' struct values
```

`struct`
: The name of a command referring to an implemented structure.

`values`
: The data to be stored in the structure. The data must be specified in form of a vector of elements. Each data element is verified to be of a correct type for that field before being stored in the structure.

`rc`
: The operation return code.

`handle`
: The structure instance handle that must be used in all future reference to the structure instance. It uniquely identifies the new instance of the structure.

Example:

```
3 XWIN 'NewPut' 'XSizeHints' (⍳15)
```

Put - Copy an APL2 nested array to a C structure instance

Replace all data in structure with the new set of data. The data must be specified in the form of a vector. Each element in the vector is verified to have a type appropriate to that structure field. The data is saved in the structure if they all verify correctly, and the normal command return code indicates that this has happened:

```
rc ← 3 XWIN 'Put' struct handle array
```

`struct`
: The name of a command referring to an implemented structure.

`handle`

© Copyright IBM Corporation 1994, 2017

Handle to the structure instance to be used.

`array`
The data to be stored in the structure. The data must be specified in form of a vector of elements. Each data element is verified to be of a correct type for that field before being stored in the structure.

`rc`
The operation return code.

Example:

```
3 XWIN 'Put' 'XSizeHints' 3524424 (⍳15)
```

SFree - Free a structure instance

SFree is used to free the storage space occupied by the structure instance, when it is no longer needed. It is the responsibility of the user to determine when an instance is no longer needed, and to call the SFree command to make the space available for other use. (The name of the command is *SFree* rather than the expected *Free*. This is due to *free* being used as a counterpart to *malloc* in keeping with normal C conventions.

```
rc ← 3 XWIN 'SFree' struct handle
```

`struct`
The name of a command referring to an implemented structure.

`handle`
Handle to the structure instance to be used.

`rc`
The operation return code.

Example:

```
3 XWIN 'SFree' 'XSizeHints' 3524424
```


© Copyright IBM Corporation 1994, 2017

System Structures

A small number of system structures have been defined. Each of these is available independently of the X Window System, although they can also be usefully used in conjunction with the X Window System. They all define single-element data structures. As an example, `I` defines the following C data structure:

```
struct {
  long I;
} I;
```

These structures can be used to process arrays returned as a pointer using the *Get* or *MGet* structure commands:

C1 One-byte character
B8 One-byte integer (uchar)
I,I4 Four-byte integer (long or ulong)
I2 Two-byte integer (short or ushort)
P Four-byte pointer (void *)
S Four-byte character string pointer (char *)
X,X4 Four-byte integer (long or ulong)
X2 Two-byte integer (short or ushort)
E8 Eight-byte floating point

© Copyright IBM Corporation 1994, 2017

The HelloWorld Function

The `HelloWorld` function included in the DEMO144 workspace demonstrates two fundamental concepts of windows-based systems: manipulation of the window and responding to user-generated events. As such, it serves as an excellent vehicle to introduce these concepts. The following text describes the `HelloWorld` function section-by-section and explains the X Window System calls used:

Initialization

```
[0]    HelloWorld;⎕IO;dp;w;gc;s;e;k;rw;bp;wp;m;hello;hi
       ;done;None;hp;hints;rc;nl;x
[1]   ⍝ Sample X program, based on helloworld.c from
[2]   ⍝ Oliver Jones:
[3]   ⍝ Introduction to the X Window system
[4]   ⍝ Prentice-Hall, 1989;  ISBN 0-13-499997-4
[5]    ⎕IO←0
[6]
[7]   ⍝ Define some constant text-strings
[8]    hello←'Hello, World.'
[9]   ⍝ The exclamation point makes hi ugly:
[10]   hi←'Hi',('A'=⎕AF 65)⊃⎕AF 90 33
[11]
```

All that happens in this section is the setting up of the text strings that are shown in the window later on, and definition of some local identifiers.

Note that the exclamation point (after `'Hi'`) is defined so that a suitable character is given in both mainframe and workstation environments.

Connect to X Window System

```
[12]  ⍝ Initialization
[13]   →(0=dp←XWIN 'XOpenDisplay' '')↓lopen
[14]   ⎕←'XOpenDisplay failed ... HelloWorld aborted'
[15]   →lexit
[16]  lopen:
[17]
```

The first call to the X Window System is to open a connection to the display. The call results in a *handle* or magic number that are used to identify the connection in all subsequent X Window System call requests. The handle is saved in the `dp` variable.

- All X Window System calls pass through the `XWIN` function. The first parameter to this function is always the name of the X Window System call.
- If the X Window System call requires parameters to be specified, these are given following the name of the call. The *XOpenDisplay* call has one parameter: the name of the display to be used. This is given as an APL2 character string. (The default display is used, because we have specified an empty string.)
- Character strings are passed (and returned) by value. AP144 converts these parameters to the C *char ** pointers that X Window System ultimately requires. AP144 also handles null-termination of the character strings.
- All output from invoking an X Window System call is returned as an explicit result of `XWIN`.

© Copyright IBM Corporation 1994, 2017

- The `HelloWorld` function performs minimal error checking. This is obviously undesirable for programs designed for a production environment.

Get More Information

```
[18] ⍝ Default pixel values
[19]  s←XWIN 'XDefaultScreen' dp
[20]  bp←XWIN 'XBlackPixel' dp s
[21]  wp←XWIN 'XWhitePixel' dp s
[22]
```

Once the connection to the display is established, X Window System can provide the calling program with various items of information. Here we retrieve the number designating the default screen and the pixel values for black and white.

- All input and output parameters on these three calls are integers. They are passed to (and returned from) the X Window System by value.
- AP144 handles any required conversion between Boolean and integer data types.

Using X Constants

```
[23] ⍝ Define an X constant
[24]  None←0
[25]
[26] ⍝ Prepare to set window position and size
[27]  (rc nl)←'H¯' axGetFF 'XSizeHints'
[28]  m←+/PPosition PSize
[29]
```

X Window System defines a large number of named constants in its C header files. These constants should also be available to APL2. However, there are too many constants defined to keep them all in the workspace. (Although technically feasible, it would define so many symbols in the workspace that it would obscure the *real* application variables.

The simplest solution to this problem is to specify these constants by value rather than by name. However, by so doing, we lose a lot of the information associated with the constant, and we increase any future maintenance effort.

Another simple solution is to define the required constants as APL2 variables, and then to use these variables whenever the constant is needed. The variables can either be defined as global variables, or as local variables within the applicable functions, as is done with `None`. This is *not* a recommended solution, though, as it relies on the constants remaining unchanged. (`None` is used further down in the function, in the call to *XSetStandardProperties*.)

A better way is to use the support for constants built into AP144. AP144 groups the constants by categories. Each group can be brought into the workspace independently of other groups, ensuring that the latest values are used. The `axGetFF` function loads a named group of constants and establishes them as APL2 variables in the workspace. `PPosition` and `PSize` are examples of two such constants.

© Copyright IBM Corporation 1994, 2017

- `axGetFF` returns a list of names of constants established in the workspace. Use the list later on to clear away the variables when they are no longer needed.
- The name of the group of constants is specified as the right argument to `axGetFF`. This name is the name of an X Window System data structure. This topic is discussed in additional detail below.
- `PPosition` and `PSize` each contain 32 individual bits, although passed to APL2 as four-byte integers. The simple addition works here because their bits are orthogonal. In general this cannot be assumed to be the case, and the following expression would be better:
-
-
```
m←2⊥∨/(32⍴2)⊤PPosition PSize
```

Creating a New Structure Instance

```
[30] ⍝ Build an XSizeHints structure instance
[31]  hp←XWIN 'New' 'XSizeHints'
```

AP144 provides extensive support for X Window System data structures. A number of these commands exist to help you manipulate the X Window System data structures from APL2. The command is specified as the first parameter, and the structure name as the second. Additional parameters are often required, and follow the structure name on the call.

In this case, *New* allocates space for a new instance of the *XSizeHints* structure and returns the address of this instance to APL2. The structure instances are stored in C free storage space (using calls to the C malloc() routine).

Importing a Structure Into APL2

```
[32]  hints←XWIN 'Get' 'XSizeHints' hp
```

To be used from within the APL2 workspace, the structure must be imported into the workspace using the *Get* command. This command copies the values of the structure into an APL2 vector. Each field in the structure is mapped to an element in the vector in the same order as in the X Window System structure definition. AP144 handles the appropriate data conversions when copying the data.

Changing The Values in a Structure (Within APL)

```
[33]  hints[H¯flags H¯x H¯y]←m 200 300
[34]  hints[H¯width H¯height]←350 250
```

Once copied into the workspace, the structure instances can be used like any regular APL2 array. Here five of the fields are updated.

The indexes of the elements to be changed are given by constant variables prefixed by `H¯`. These constants were generated by the `axGetFF` function when the *XSizeHints* structure definition was loaded in line 25. To recall:

```
[27]  (rc nl)←'H¯' axGetFF 'XSizeHints'
```


© Copyright IBM Corporation 1994, 2017

The axGetFF function establishes a number of global variables in the workspace. These variables can be divided into two separate groups:

Field indexes:

These variables are used to indicate the location of a given field in a structure. They provide a rough equivalence between the C structure member operation *hints.mask* and the APL2 version hints[H¯mask].

Since structure field names all tend to be very common identifiers, for example, *x*, *name*, or *size*, it is possible (and advisable) to specify a prefix that is applied to each field name before it is established in APL2. This avoids name clashes between the various structure's field-name index variables. H¯ is the prefix used in the example given here.

Constants:

We met these earlier on. They are combined with the structure definitions because they are often used in conjunction with them. The valid values for a given field in a structure are commonly given by constants, but they can be used in other ways. Names of constants are *never* changed; the prefix does not apply to constants.

Updating a Structure Instance (Within C)

```
[35]  XWIN 'Put' 'XSizeHints' hp hints
[36]
```

Changing the values of a structure variable only has effect within APL2. The *Put* command is necessary to make the updated structure values available to the X Window System calls.

The *Put* command expects exactly three parameters: the name of the structure ('XSizeHints'), a pointer to the storage location containing an instance of the structure (hp), and the data values (hints) to be inserted into the structure instance. The structure data values must be specified as a single nested array in the fourth element of the command vector (the vector passed to XWIN).

Creating a Simple X Window System Window

```
[37] ⍝ Window creation
[38]  rw←XWIN 'XDefaultRootWindow' dp
[39]  x←hints[⎕IO+1 2 3 4],5 bp wp
[40]  w←XWIN 'XCreateSimpleWindow' dp rw,x
```

We are now in a position to create our application window that is accomplished with these two calls.

The hints values can be used like any other APL2 variable. Here four of them are used to specify the window start and extent.

Using The Structure Pointer

```
[41]  x←hello hello None ('A' 'test') 2 hp
[42]  XWIN 'XSetStandardProperties' dp w,x
```


© Copyright IBM Corporation 1994, 2017

The *XSizeHints* structure created is now put to use. As the name of the call implies, it defines defaults for a number of the window attributes.

- *XSetStandardProperties* uses an *argv* parameter. This type of parameter is specified in APL2 as a vector of character vectors. (*argv* and *argc* are well-known C variable types. X Window System uses them in several of its calls.
- The *argv* parameter is followed by an *argc* parameter. The information content of this parameter is, in a sense, redundant in APL2, since the information can be determined by examining the *argv* variable. One of the general design rules in building the AP144 interface has been to maintain a fidelity to the standard X Window System call syntax, even though at times, it means specifying more information in the call than is strictly necessary.

Freeing The Structure After Use

```
[43]  XWIN 'SFree' 'XSizeHints' hp
[44]
```

It is important that the structure is freed after it is no longer needed. If this is not done, it continues to occupy space in storage, and can lead to storage fragmentation.

The name *SFree* is used in place of *Free*, so as not to conflict with the *free* associated with the C *malloc* function.

Creating a Graphics Context

```
[45] ⍝ Graphics Context creation and initialization
[46]  gc←XWIN 'XCreateGC' dp w 0 0
[47]  XWIN 'XSetBackground' dp gc bp
[48]  XWIN 'XSetForeground' dp gc wp
[49]
```

A graphics context is the X Window System version of a magic paintbrush. It is used on any drawing calls to define colors, patterns, and other attributes. We define one here, so we can use it later to write some text to the window.

Show The Window Outline

```
[50] ⍝ Window mapping
[51]  XWIN 'XMapRaised' dp w
[52]
```

We are now ready to show the window to the user. This call does just that, but note that we have yet to write any text into it, nor are we quite ready to accept input from the user.

What Events Are We Interested In?

```
[53] ⍝ Input event selection
[54]  m←'ButtonPressMask' 'KeyPressMask'
[55]  m←m,⊂'ExposureMask'
```


© Copyright IBM Corporation 1994, 2017

```
[56]  (rc m)←m axGetFF1 'XEvent'
[57]  XWIN 'XSelectInput' dp w(+/m)
[58]  ep←XWIN 'XGetEventBuffer'
[59]
```

XSelectInput tells the X Window System what events we are interested in being notified about. The last parameter of this call is a bit-mask. The various constants defining the individual bits are defined in the *XEvent* structure. We could now use the `axGetFF` function to load in all fields and constants, and then use the variables thus established. However, we choose to take a slightly different route:

The `axGetFF1` function is used in place of `axGetFF`. Instead of establishing variables based on the names of the fields and constants, it returns the values of a set of named items. Because we are only interested in three specific constants, we use `axGetFF1` to retrieve their values.

- The values retrieved represent individual bits. It is therefore possible to simply add them instead of performing a bit-wise *or*.
- Bit-flags are returned in *packed* form, as integers.

The *XGetEventBuffer* call establishes a pointer to the buffer used to hold events. The same buffer is used for all events, hence we only need to make this call once per session.

Get The Event Codes

```
[60] ⍝ Get some more constants
[61]  m←'KeyPress' 'ButtonPress'
[62]  m←m,'Expose' 'MappingNotify'
[63]  (rc m)←m axGetFF1 'XEvent'
[64] ⍝ ... and some event structure layouts
[65]  nl←nl,1⊃'K_' axGetFF 'XKeyEvent'
[66]  nl←nl,1⊃'B_' axGetFF 'XButtonEvent'
[67]  nl←nl,1⊃'E_' axGetFF 'XExposeEvent'
[68]
```

X Window System notifies us of the user's actions by returning events to us when requested. The type of events is limited to ones enabled by the *XSelectInput* call just executed. One of the fields in the returned event specifies the type of event that is being returned. These event types are also defined in the *XEvent* structure. To be able to distinguish between the returned events by type, a vector has been set up with the expected types by using the `axGetFF1` function.

The field definition variables describing the event structures that might be received are also loaded. These help access the correct fields in the structures.

The Event Loop

```
[69] ⍝ Main event-reading loop
[70]  done←0
[71] levent:→(done=0)↓lend
[72]
```

We now sit in an event loop, responding to the incoming events as they come in from the display, until done.

XNextEvent

© Copyright IBM Corporation 1994, 2017

```
[73] ⍝ Read and process the next event
[74]  x←lKeyPress lButtonPress lExpose lMappingNotify
[75]  →(m=K_type⊃e←XWIN 'XNextEvent' dp)/x
[76]
```

XNextEvent returns the next event in the queue to us. Once the event has been returned, the event type is compared with the mask defined earlier to set up the proper branch. Effectively this serves as a *case-on-event* statement.

The event is returned as a vector of structure fields, in much the same way as the structure *Get* call would return a structure instance.

This differs from the way that the *XNextEvent* call is handled in X Window System. In the native X Window System, the *XNextEvent* call also specifies a second parameter, a pointer to an *XAnyEvent* structure. *XNextEvent* fills in the structure fields, thereby making the event values available to the calling program. This is an example of how AP144 differs from native X Window System by automatically handling the call's output parameters.

Handling The Expose Events

```
[77] lExpose: ⍝ Repaint window on expose events
[78]  →e[E_count]↑levent   ⍝ Count > 0 ?
[79]  x←e[E_display E_window],gc,50 50
[80]  XWIN(⊂'XDrawImageString'),x,hello(⍴hello)
[81]  →levent
[82]
```

Expose events are generated whenever a window is made visible on the screen. Multiple expose events can be generated, so it makes sense only to respond to the last one. X Window System caters to this approach by providing a count of future, already pending, expose events as one of the fields in the expose event itself. We use this field to ignore all but the last expose event.

When we receive the final expose event we want to write the content of the `hello` variable to the screen. We do this using the *XDrawImageString* call.

- All events provide the value of the originating display connection in the `e[E_display]` item in the event structure. This identifies the origins of an event, in case multiple concurrent display connections have been established. Here its value equals that of `dp`.
- Most (but not all) events also specify the window that generated the event (some events do not come from windows). This equals the value of `w` here.
- The last parameter (`(⍴hello)`) may seem superfluous. This is certainly the case in the APL2 environment, where the information could be derived from the preceding parameters. However, as stated earlier, AP144 follows, as closely as possible, the syntax of the X Window System calls.

Handling The ButtonPress Events

```
[83] lButtonPress: ⍝ Process mouse-button presses
[84]  x←e[B_display B_window],gc,e[B_x B_y]
[85]  XWIN(⊂'XDrawImageString'),x,hi(⍴hi)
[86]  →levent
[87]
```


© Copyright IBM Corporation 1994, 2017

The processing here is much the same as for Expose events. The only noteworthy change is the use of `events[B_x B_y]` to indicate the position where the writing is to start. This coordinate pair is returned as part of the ButtonPress event structure, and indicates the position of the tip of the mouse pointer when its button was pressed.

The effect of this code fragment is to write the text "Hi!" at the position of the mouse pointer when a mouse button is pressed.

Handling The KeyPress Events

```
[88] lKeyPress: ⍝ Process keyboard input
[89]  →(done←(↑k←Δ1⊃XWIN 'XLookupString' ep)Δ'qQ')↑levent
[90]  x←e[K_display K_window],gc,e[K_x K_y]
[91]  XWIN(⊂'XDrawImageString'),x,k(⍴k)
[92]  →levent
[93]
```

KeyPress events are generated when the user presses any of the keys on the keyboard. However, information about the key pressed is not part of the returned event structure, so we need to use the *XLookupString* call to acquire this information. This call has one input parameter, the event whose key we want to ascertain. This parameter can be specified either as the set of values returned on the *XNextEvent* call, or using the pointer to the event buffer that we set up earlier on. The latter method is faster, as it does not involve passing a lot of information through the interface, so we use it here.

The key pressed is returned as the second element of the **XLookupString** return. We test this field to see if it matches a *q*. If so, we terminate the program. If not, we print the key character at the current mouse pointer location.

All event structures are exactly as defined by X Window System, except for the *XKeyEvent* structure returned by the KeyPress (and KeyRelease) events. The *XKeyEvent* structure contains one additional field not found in the X Window System definition: the character representation of the key. This last field is named *key*, and is appended onto the end of the structure.

Mapping Notify

```
[94] lMappingNotify: ⍝ Reset keyboard
[95]  XWIN 'XRefreshKeyboardMapping' e
[96]  →levent
[97]
```

The *MappingNotify* event is used to ensure that your keyboard mapping is restored to the state it was in when the window was last used. Each window in the X Window System can define its own keyboard mapping, so this call is needed to ensure that you get the proper definition restored whenever you return to this window.

Termination

```
[98] lend: ⍝ Termination
[98]  XWIN 'XFreeGC' dp gc
[100] XWIN 'XDestroyWindow' dp w
[101] XWIN 'XCloseDisplay' dp
[102] nl←⎕EX⍠nl
```


© Copyright IBM Corporation 1994, 2017

```
[103]lexit:
```

Before terminating the `HelloWorld` sample program we free up the resources we have used. Some of these resources are controlled by the X Window System, so the X Window System must free them.

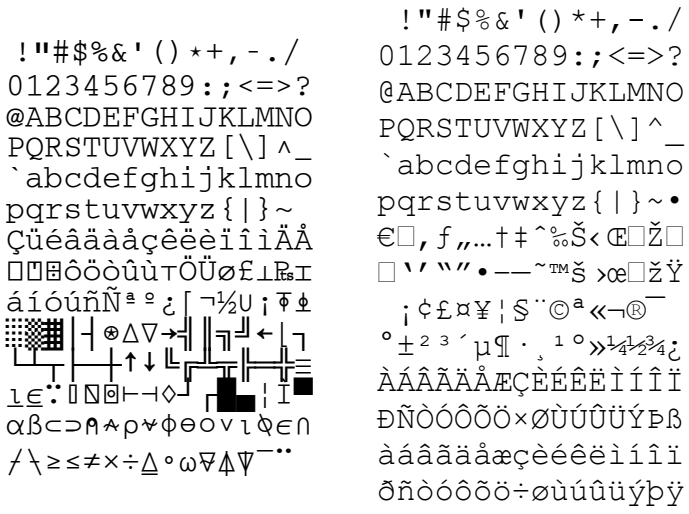
We also clear the workspace of all the AP144-defined variables. These could conceivably be saved with the workspace, obviating the necessity of loading them every time the application is run. A production application probably should take this latter approach, unless the underlying structures change very frequently.

© Copyright IBM Corporation 1994, 2017

Windows Character Set Support

APL2 and Windows use different character sets. The character sets contain different characters and some of the characters are in different positions. Examine the following figure which shows the two character sets with the characters in their native arrangements:

APL2 **Windows**



The Windows character set does not contain APL characters. In addition, notice that the accented characters are in different locations. This can cause confusion if you use both APL2 and other Windows applications to view character data containing these characters.

The APL2 display and printing facilities (the Session Manager, Editors, AP 124, AP 145, and AP 207) automatically use the APL2 character set when you use the APL2 Image, APL2 Italic, Courier APL2, Courier APL2 Bold or any font that contains glyphs for the Unicode codepoints for APL2 characters. When you use any other font, APL2 uses the Windows character set.

APL2's other facilities for accessing data outside the workspace (AP 100, AP 119, AP 127, AP 227, AP 210, and `FILE`) do *not* translate between the character sets. If you use these facilities to share data with non-APL2 applications, you need to translate character data between the APL2 and Windows character sets to ensure accented characters are displayed properly.

The WINDOWS workspace includes Character Set Translation tools for translating such character data.

Note: These tools do *not* provide a reversible one-to-one translation. Translation in either direction causes some information loss.

© Copyright IBM Corporation 1994, 2017

Double-Byte Character Set Support

The next sections discuss various aspects of support for double-byte character sets.

- Overview of Double-Byte Characters
- APL2 Support of Double-Byte Characters
- DBCS in the APL2 Programming Environment
- Migration of DBCS Data from APL2/370

© Copyright IBM Corporation 1994, 2017

Overview of Double-Byte Characters

Workstation operating systems support two primary character-encoding schemes:

Multibyte

> Each character is represented by 1 or more bytes. The system algorithmically determines how many bytes are used by each character. The algorithm used depends on the current code page. Some code pages support only single-byte characters; some code pages support Multibyte characters. The code page can be specified explicitly during conversion or implied by the current input locale.
>
> Single-byte APL characters can be used with single-byte character code pages. Multibyte character sets, such as Shift-JIS, do not support APL characters.

Unicode

> Each character is represented by the operating system using 2 bytes.
>
> Unicode supports APL characters.

Windows Support of Double-Byte Characters

Some parts of Windows store data in Multibyte; some parts of Windows store data in Unicode. When Multibyte data is passed to a Unicode part or when Unicode data is passed to a Multibyte part Windows automatically converts the data as necessary.

Windows uses several techniques to determine which codepage to use to convert data between Multibyte and Unicode:

- When data is transferred between applications and the operating system, Windows uses the current input locale to determine which code page to use.
- When printing Multibyte data, Windows uses the character set of the currently selected font. If the character set includes only single-byte characters, then one character is drawn for each byte. If the character set includes Multibyte characters, the code page implied by the character set is used for decoding the data.
- When printing Unicode data, no conversion is performed.
- When displaying data on the screen, some classes of windows use the current input locale and some classes use the character set of the currently selected font.

Note:

The input locale is set either during installation or with the **Language for non-Unicode Programs** setting in the **Regional and Language Options** dialog.

© Copyright IBM Corporation 1994, 2017

APL2 Support of Double-Byte Characters

Internally, APL2 stores characters using either 1 or 4 bytes per character. If all the characters in an array are elements of AV, the array may be stored using 1 byte per character. If any of the characters are not elements of □AV, the array will be stored using 4 bytes per character.

Conceptually, APL2 treats all characters as Unicode characters. The use of 1 byte per character is merely a technique to conserve storage. The APL2 interpreter automatically converts data between single-byte and Unicode as necessary. These automatic conversions always use the Unicode codepoints for characters that are elements of □AV.

The following character conversion tools are provided with APL2:

- Use the □UCS system function to convert between APL2 characters and numeric Unicode code points.
- Use the CTUTF and UTFTC external functions to convert between APL2 character arrays and the UTF-7 and UTF-8 transformation formats used to send Unicode characters across networks.
- **Windows Only:** Use the CTK and KTC external functions to convert between APL2 character arrays and Multibyte format.

The following sections provide more detail about APL2 support for double-byte characters:

- Accessing Files
- Printing
- Text Windows
- Graphic Windows
- GUI Windows

Accessing Files

When using APL file formats (AP 211, AP 210 type 'A', and Processor 12 type 'A'), any APL2 array may be stored in the file.

When using text file formats (FILE, AP 210 type 'D' or 'C', Processor 10 functions, and Processor 12 type 'F'), only single-byte character data is supported.

All APL2 file access functions and processors support Multibyte filenames. The operating systems use the current input locale to interpret Multibyte file names.

On Windows, the FILE external function and the UNIFILE function from the GUITOOLS workspace also support Unicode filenames.

Printing

On Windows, AP 145's printing facilities support both Multibyte and Unicode data.

The CREATE_PRINTER function is used to create Multibyte printer objects. APL2 single-byte data sent to a Multibyte printer object is passed directly to Windows. APL2 4-byte data sent to a Multibyte printer object is

© Copyright IBM Corporation 1994, 2017

converted to single-byte characters and then passed to Windows. Characters that are not in ⎕AV are converted to the ⍵ character.

When using a Multibyte printer object, the *APL2 Italic* and *Courier APL2* fonts can be used to print APL characters. The *APL2 Image* font cannot be used because it is a screen-only bitmap font.

The CREATE_UNICODE_PRINTER function is used to create Unicode printer objects. APL2 4-byte data sent to a Unicode printer object is passed directly to Windows. APL2 single-byte data sent to a Unicode printer object is converted to Unicode by AP 145 using the Unicode codepoints for the APL characters and then passed to Windows.

To print Multibyte data with a Unicode printer and bypass AP 145's conversion of data using Unicode codepoints for APL characters, use the external function KTC to convert the data before sending it to the printer.

When using a Unicode printer object, the *APL2 Unicode Italic* and *Courier APL2 Unicode* fonts can be used to print APL characters.

Text Windows

AP 124 does not support display or entry of double-byte characters.

Graphic Windows

AP 207 supports display of Multibyte characters when a Multibyte system font is selected. Entry of Multibyte characters is not supported.

On Windows, AP 207 supports display and entry of Unicode characters.

See Support for Double-byte Characters in the AP 207 documentation for more information.

GUI Windows

On Windows, AP 145 and the GUITOOLS workspace are used to create Multibyte and Unicode GUI windows.

The following GUITOOLS functions can be used to create Multibyte dialogs, menus, and controls:

- CREATEDLG
- CREATEMENU
- EDIT
- FILE
- FILEDLG
- FOLDERDLG
- MSGBOX
- POPUPMENU
- ∆FV

The following functions can be used to create Unicode dialogs, menus, and controls:

- UNICREATEDLG

© Copyright IBM Corporation 1994, 2017

- UNICREATEMENU
- UNIEDIT
- UNIFILE
- UNIFILEDLG
- UNIFOLDERDLG
- UNIMSGBOX
- UNIPOPUPMENU
- UNI∆FV

Multibyte and Unicode properties are used for passing data to and from windows.

The following properties use the Multibyte encoding scheme:

- CELL DATA (Range property)
- CONTEXT HELP
- DATA
- DATA LIST
- DROPPED FILES
- FONT
- HEADINGS
- PATH
- PATH SELECTION
- PICTURE
- PICTURE LIST
- STATUS DATA
- TOOL TIP

The following properties use the Unicode encoding scheme:

- UNICODE CELL DATA (Range property)
- UNICODE CONTEXT HELP
- UNICODE DATA
- UNICODE DATA LIST
- UNICODE FONT
- UNICODE HEADINGS
- UNICODE STATUS DATA
- UNICODE TOOL TIP

How Multibyte properties work

When the Multibyte properties are used to send data to a window, AP 145 converts each APL2 4-byte character to a single-byte character. 4-byte characters that are not elements of □AV are converted to ⍵. The single-byte characters are then passed to Windows. If the recipient window uses the Multibyte scheme to encode data, Windows then simply delivers the data. If the recipient window encodes data in Unicode, Windows first converts the data from Multibyte to Unicode and then delivers it.

When the Multibyte properties are used to retrieve data from a window that uses the Multibyte encoding scheme, Windows delivers the data directly to AP 145. The Multibyte data is then stored in the workspace as a vector of APL2 single-byte characters.

© Copyright IBM Corporation 1994, 2017

When the Multibyte properties are used to retrieve data from a window that uses the Unicode encoding scheme, Windows converts the data from Unicode to Multibyte using the current codepage. Characters which cannot be represented by the current code page are replaced with a substitution character. The Multibyte data is then stored in the workspace as a vector of APL2 single-byte characters.

How Unicode properties work

When the Unicode properties are used to send data to a window, AP 145 converts each APL2 character to a Unicode character. Single-byte characters are converted to their equivalent Unicode characters. The data is then passed to Windows. If the recipient window uses the Unicode scheme to encode data, Windows then simply delivers the data. If the recipient window uses the Multibyte scheme to encode data, Windows first converts the data from Unicode to Multibyte and then delivers it.

To send Multibyte data with a Unicode property and prevent AP 145 converting the data as single byte APL characters, use the external function `KTC` to convert the Multibyte data to Unicode.

When the Unicode properties are used to retrieve data from a window that uses the Multibyte encoding scheme, Windows converts the data from Multibyte to Unicode using the current code page. The Unicode data is then stored in the workspace as a vector of APL2 4-byte characters.

When the Unicode properties are used to retrieve data from a window that uses the Unicode encoding scheme, Windows delivers the data directly AP 145. The Unicode data is then stored in the workspace as a vector of APL2 4-byte characters.

How to use APL and National Language Characters

Workstation operating systems use codepages that reserve codepoints greater than 127 for single-byte national language characters and first bytes of double-byte characters. APL2 reuses many of those codepoints for APL characters. This can cause errors if data is incorrectly converted between Multibyte and Unicode.

Depending on whether your application uses characters from the upper half of `⎕AV`, you may be able to use Multibyte dialogs, controls, and properties. You can use Multibyte components if:

- Your application only uses characters in the lower half of `⎕AV` *or*,
- Your application will only run on machines that have the same input locale as your development machines

Otherwise, to guarantee correct display and processing of APL and national language characters worldwide, use Unicode dialogs, controls, and properties.

© Copyright IBM Corporation 1994, 2017

DBCS in the APL2 Programming Environment

The following sections describe the DBCS support provided by the APL2 programming environment:

Interpreter

- 4-byte characters may be used in character constants and comments
- 4-byte characters are converted to ⍵ characters in default display and error messages.
- 4-byte characters are displayed as ⍵ characters in)EDITOR 1. Changing a line containing 4-byte characters permanently changes them to ⍵ characters.

Session Manager

- Double-byte characters are not supported in the session log.
- 4-byte characters not in □AV are displayed as ⍵ characters.
- Multibyte characters are displayed as single-byte characters.
- Multibyte characters are supported in file names.
- Double-byte characters may be entered in the Find and Function Key dialogs but they will be converted to omegas.

Open Object Dialog

- Double-byte characters may be entered in the Open Object dialog but they will produce unpredictable results.

Object Editor

- Double-byte characters are supported in the editing area.
- Double-byte characters are supported in the Find dialog.

Dialog Editor

- Multibyte characters are supported in the **Text** entry field.
- APL characters are only supported in the event handlers dialogs.
- On systems configured to use a single-byte character codepage, the default font used by the dialog editor is MS Sans Serif. On systems configured to use the Japanese ANSI codepage 932, the default font used by the dialog editor is MS Mincho. On systems configured to use other double-byte character codepages, the dialog editor uses the operating system's default font. Use the Presentation Properties dialog to select other fonts.

File Editor

- Double-byte characters are supported in the editing area.
- Double-byte characters are supported in the Find dialog.
- Multibyte characters are supported in file names.

Other Tools

© Copyright IBM Corporation 1994, 2017

- The `UNIEDIT` function in the GUITOOLS workspace may also be used to edit arrays and program definitions that contain 4-byte characters. By selecting an appropriate Unicode font and input locale, characters that are not elements of `⎕AV` can be displayed and entered.

© Copyright IBM Corporation 1994, 2017

Migration of DBCS Data from APL2/370

APL2/370 uses a 4-byte format for storing double-byte characters. The first two bytes contain the code page identifier (CPGID), the value specified in the APL2/370 DBCS invocation option, or zero. If the first two bytes are not zero, the second two bytes contain the code point in the code page. If the first two bytes are zero, the third byte is also zero and the fourth byte is an EBCDIC character.

The workstation APL2 systems use a different format for 4-byte characters. The values are Unicode code points.

Data containing 4-byte characters can be migrated from APL2/370 to workstation APL2 systems. The `)IN` command, cross-system shared variables and AP 211 provide translation for mainframe 4-byte characters. Characters will be translated when they are in the EBCDIC format or when the CPGID is one of the following:

Code Page	Description
037	CECP: USA, Canada (ESA*), Netherlands, Portugal, Brazil, Australia ...
290	Japanese Katakana Host Extended SBCS
300	Japanese Latin Host Double-Byte including 4370 UDC
833	Korean Host Extended SBCS
834	Korean Host Double-Byte including 1880 UDC
835	Traditional Chinese Host Double-Byte including 6204 UDC
836	Simplified Chinese Host Extended SBCS
837	Simplified Chinese Host Double-Byte including 1880 UDC
930	Japanese Katakana-Kanji Host Mixed including 4370 UDC, Extended SBCS
933	Korean Host Mixed including 1880 UDC, Extended SBCS
935	Simplified Chinese Host Mixed including 1880 UDC, Extended SBCS
937	Traditional Chinese Host Mixed including 6204 UDC, Extended SBCS
939	Japanese Latin-Kanji Host Mixed including 4370 UDC, Extended SBCS
1027	Japanese Latin Host Extended SBCS
1200	Unicode

For code pages not in this list, the data will be left unchanged.

The following procedure can be used to extract the code page identifier and code point from a 4-byte character which has not been translated:

1. Use the `⎕AF` system function to convert the 4-byte character to an integer.
2. Use the `⊤` primitive to convert the integer into two base 65536 integers. The first integer is the code page; the second integer is the code point.

Note: APL2/370 uses EBCDIC, which defines a different set of code pages from the code pages defined for workstation systems. Thus the code points extracted in this way do not match their workstation counterparts.

© Copyright IBM Corporation 1994, 2017

Implementation Limits

In this section the term *workstation* refers to all workstation implementations except APL2/PC.

The APL2 interpreter has the following implementation limits:

Largest and smallest representable numbers in an array
- Workstations `1.7976931348E308` and `¯1.7976931348E308`
- Mainframes `7.2370055773E75` and `¯7.2370055773E75`

Most infinitesimal (near `0`) representable numbers in an array
- Workstations `2.2250738585E¯308` and `¯2.2250738585E¯308`
- Mainframes `5.397605346934E¯79` and `¯5.397605346934E¯79`

Maximum rank of an array
- Workstations `64`
- Mainframes `64`

Maximum length of any axis in an array
- `¯1+2*31 (2147483647)`

Maximum product of all dimensions in an array
- `¯1+2*31 (2147483647)`

Maximum depth of an array applied with the primitive functions depth (`≡R`) and match (`L≡R`)
- `181`

Maximum depth of a shared variable
- `181`

Maximum depth of a copied variable
- `181`

Maximum number of characters in the name of a shared variable
- `255`

Maximum number of characters in a comment (minus leading blanks)
- Workstations `4090`
- Mainframes `32764`

Maximum length of line
- Workstations `32767`
- Mainframes N/A

Maximum number of lines in a defined function or operator
- Workstations `¯1+2*15 (32767)`
- Mainframes `¯1+2*31 (2147483647)`

Maximum number of labels in a defined function or operator
- Workstations Limited by number of lines
- Mainframes `32767`

Maximum number of local names (excluding labels) in a defined function or operator
- Workstations Limited by lengths of lines and names
- Mainframes `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.

© Copyright IBM Corporation 1994, 2017

Workstations N/A

Mainframes 32767

Maximum value of ⎕PW

Workstations 254

Mainframes 390

Maximum value of ⎕PP

Workstations 16

Mainframes 18

Maximum number of users with whom a user can share cross-system variables

Workstations N/A

Mainframes 59

© Copyright IBM Corporation 1994, 2017

Deviations from *APL2 Programming: Language Reference*

This appendix describes the areas in which this implementation of APL2 differs from the APL2 language as defined in *APL2 Programming: Language Reference*.

The differences are classified as follows:

Deviation
: The feature is implemented, but the implementation is different from that defined in *APL2 Programming: Language Reference*.

Restriction
: The feature is not fully implemented

System dependency
: The feature is not required or is implemented differently due to system requirements.

- General Language Differences
- APL2 Statement Entry
- APL2 Error Reporting
- Display of Output
- System Functions and Variables
- Defined Functions and Operators
- System Commands

© Copyright IBM Corporation 1994, 2017

General Language Differences

Restriction

- Selective specification is not implemented for all cases available in APL2/370. For example:

```
(L0¨V)←A
```

  gives DOMAIN ERROR.

  Any usage of bracket indexing in a selective assignment expression, such as

```
(2↑V[⍳4])←3
```

  gives SYNTAX ERROR. The index primitive (⌷) may be used instead of bracket indexing to accomplish the same result.

  Selective specification of a shared variable also gives SYNTAX ERROR.

© Copyright IBM Corporation 1994, 2017

APL2 Statement Entry

Deviation

- Some rules are relaxed for syntax validation. For example, combinations such as `A□`, `□□` and `⍞⍞` are accepted as `A □`, `□ □` and `⍞ ⍞` (respectively), instead of giving `SYNTAX ERROR`.

System Dependency

- Exact fidelity of statements is not maintained. Function and operator statement lines are tokenized, and when redisplayed, these may not appear exactly as entered. This can cause redundant spaces to be removed and the display format of numbers to be changed.

© Copyright IBM Corporation 1994, 2017

APL2 Error Reporting

Deviation

- Where multiple errors occur in an operation, a different error may be reported. For example:

```
      'AB'⌽3 3⍴⍳9      ⍝ Workstation implementations
DOMAIN ERROR
      'AB'⌽3 3⍴⍳9
      ^   ^


      'AB'⌽3 3⍴⍳9      ⍝ APL2/370
LENGTH ERROR
      'AB'⌽3 3⍴⍳9
      ^   ^
```


© Copyright IBM Corporation 1994, 2017

Display of Output

Deviations

- An array that exceeds the printing width (⎕PW) is wrapped line by line, rather than being folded plane by plane.
- Default display of columns of data of an array of mixed type right-justifies all numeric items. APL2/370 aligns numeric items in the same column on the decimal point. For example:

```
      3 2⍴'APL2' 2 (○1)
```

gives:

```
       APL2           2
3.141592654        APL2
          2 3.141592654
```

instead of:

```
       APL2 2
3.141592654        APL2
2           3.141592654
```

- Dyadic format with negative left argument leaves additional spaces to allow for an exponent containing up to three digits, and a negative sign. For example:

```
      ⍴⎕←¯1⍕1 1
```

gives:

```
 1E0    1E0
14
```

instead of:

```
 1E0 1E0
8
```


© Copyright IBM Corporation 1994, 2017

Also:

```
      5 ¯1⍕1
```

gives:

```
DOMAIN ERROR
      5 ¯1⍕1
      ^   ^
```

instead of:

```
      5 ¯1⍕1
1E0
```

Restriction

- Dyadic format by specification gives `DOMAIN ERROR` if formatting an element of an array creates output longer than 32767 characters. For example:

```
      0⍕⊂33000⍴'ABC'
DOMAIN ERROR
      0⍕⊂33000⍴'ABC'
      ^^
```


© Copyright IBM Corporation 1994, 2017

System Functions and Variables

Restrictions

- ⎕L and ⎕R are not recognized; they return `VALUE ERROR`.
- ⎕NLT can be referenced, but not set. National language is set with the `-nlt` invocation option.
- The contents of ⎕AI are incorrect after they reach `2*31`. This can occur for sessions active longer than 24 days.
- General shared variable offers are not supported.

© Copyright IBM Corporation 1994, 2017

Defined Functions and Operators

Deviation

- The header line (line 0) of a function does not accept a comment. Any comment given is ignored, and no error is generated. For example:

```
      ∇TEST ⍝ COMMENT
[1]    LINE 1
[2]    [□]
[0]   TEST
[1]    LINE 1
[2]
```

Restrictions

- Recursive editing and entry of system commands while editing are not allowed.
- Editing of a line with the line editor removes stop (S∆) and trace (T∆) controls for that line.

© Copyright IBM Corporation 1994, 2017

System Commands

Deviations

- The maximum length of the workspace name parameter to)SAVE,)COPY,)PCOPY,)LOAD,)IN,)PIN, and)OUT is 64 rather than 8.
-)DROP accepts a suffix in a quoted filename format for deleting transfer form files.
-)ERASE erases and)COPY replaces the most local, rather than the global, referent of an object.
-)FNS,)VARS,)OPS,)NMS, and)LIB allow specification of multiple ranges of names on one command, where the language defines only one range per command. The syntax for specification of multiple ranges is:

  Workstations: `[[first | first-last][...]`
  APL2/370: `[first][-][last]`

- The)SYMBOLS command gives a different report: size of table (bytes), number of bytes in use, number of symbols.
-)SYMBOLS n contracts or expands symbol table to n bytes.

Restrictions

-)EDITOR 2 is not implemented. See the EDITOR_2 function, described in the EDIT Workspace, for details of a defined function that is equivalent to the APL2/370 Editor 2.
-)EDITOR name is implemented with some restrictions. It enables a system editor to be invoked, and allows ∇name to be used to edit a function, operator or a simple character matrix. The restrictions are the same as those imposed by "⎕FX ⎕CR name"; for instance, all stop (S∆) and trace (T∆) controls on the function are removed.
-)MORE always displays NO MORE INFORMATION.
-)OUT of the entire workspace with a nonempty stack is not allowed. The error reports SI WARNING and NOT SAVED are given.
-)SAVE does not save shared variables.

System Dependencies

-)EDITOR name accepts an option, -u, which causes Unicode data representation to be used in the temporary editor file.

```
)EDITOR editorname [-u]
```

-)IN accepts a library number.

```
)IN [libno] filename [obj1 [obj2]...]
```


© Copyright IBM Corporation 1994, 2017

-)PIN accepts a library number.

```
)PIN [libno] filename [obj1 {obj2}...]
```

-)LIB accepts a different syntax and lists operating system files. The syntax is:

```
)LIB [libno | 'path'] [initials] [{.apl | .atf}]
```

  This displays files with names that start with the characters initials and that have an extension of .apl or .atf in the library indicated by libno or 'path'.

```
)LIB [libno | 'path'] [first]-[last] [{.apl | .atf}]
```

  This displays files with names that fall alphabetically between first and last, and have an extension of .apl or .atf in the indicated library. If either first or last is omitted, that end of the range is unbounded.

  If neither initials nor first-last are given, all files are listed that meet the other qualifications. For both syntax variations, if no library number is given, the files for library ↑⎕AI are listed. This is normally defined as being the directory from which APL2 is invoked. If neither .apl nor .atf are specified, both are assumed.

  Examples:

)LIB 1
  Lists all .apl and .atf files in library 1
)LIB 2 AP
  Lists all .apl and .atf files with names beginning with AP in library 2
)LIB AP .apl
  Lists all .apl files with names beginning with AP
)LIB -C
  Lists all .apl and .atf files beginning with letters A through C
)LIB C-
  Lists all .apl and .atf files beginning with letters C through z
)LIB A-C
  Lists all .apl and .atf files beginning with letters A through C
)LIB A C
  Lists all .apl and .atf files beginning with letters A and C

-)OUT accepts a library number.

```
)OUT [libno] filename [obj1 {obj2}...]
```


© Copyright IBM Corporation 1994, 2017

Bibliography

- APL2 Publications
- Other Books You Might Need

© Copyright IBM Corporation 1994, 2017

APL2 Publications

The following table shows all the books in the APL2 library, organized by the tasks for which they are used. The APL2 library can be found on the web at *http://www.ibm.com/software/awdtools/apl/library.html*.

Information	**Book**	**Publication Number**
Introductory language material	An Introduction to APL2 APL2 Language Summary	SH21-1073 SX26-3851
Common reference material	APL2 Programming: Language Reference APL2 Reference Summary APL2 GRAPHPAK: User's Guide and Reference APL2 Programming: Using SQL APL2 Migration Guide	SH21-1061 SX26-3999 SH21-1074 SH21-1057 SH21-1069
Mainframe programming	APL2 Programming: System Services Reference APL2 Programming: Using the Supplied Routines APL2 Programming: Processor Interface Reference APL2 Installation and Customization under CMS APL2 Installation and Customization under TSO APL2 Messages and Codes APL2 Diagnosis	SH21-1054 SH21-1056 SH21-1058 SH21-1062 SH21-1055 SH21-1059 LY27-9601
Workstation programming	APL2 User's Guide APL2 Programming: Developing GUI Applications APL2 Programming: Using APL2 with WebSphere	SC18-7021 SC18-7383 SC18-9442

© Copyright IBM Corporation 1994, 2017

Other Books You Might Need

The following book is recommended:

APL2 at a Glance, by James Brown, Sandra Pakin, and Raymond Polivka, published by Prentice-Hall, ISBN 0-13-038670-7 (1988). (Copies can be ordered from IBM as SC26-4676.)

A set of *APL2 Keyboard Decals*, SC33-0604, are included with this product. Additional sets of these decal sheets can be ordered from IBM.

© Copyright IBM Corporation 1994, 2017