IBM

International Systems Centers

AN OVERVIEW OF APL2

GG24-1627-0

AN OVERVIEW OF APL2

Document Number GG24-1627-0

September, 1985

International Systems Center
Poughkeepsie,N.Y. 12602,USA

This overview of APL2 is intended to provide a useful insight into the capabilities of APL2. The enhancements to VS APL and the links with SQL and ISPF are presented. The installation under TSO and the migration from VS APL are briefly discussed. The aim is to give to people with some modest VS APL knowledge an understanding of new APL2 features.

ESSYS LSYS VMSYS 118 pages

FIRST EDITION (SEPTEMBER 1985)

This edition applies to Release 1 of APL2, Program Product 5668-899, and to all subsequent releases until otherwise indicated in new editions or Technical Newsletters.

A form for reader's comments is provided at the back of this publication. If the form has been removed, comments may be addressed to IBM Corporation, International Systems Center, Dept. H52, Bldg. 930, P. O. Box 390, Poughkeepsie, New York U.S.A. 12602. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

PREFACE

The aim of the document is to provide an overview of APL2. It is intended to complement the existing APL manuals rather than replace them. This document is intended to provide a useful insight into the capabilities of APL2 and to assist Systems Engineers in answering queries about it.

The first section gives some background to the ideas behind APL2 and an indication of its power and usefulness.

The next section presents the APL2 language with an emphasis on the enhancements to VS APL. This section introduces the new concepts and explains their uses. It gives the reader a flavor of the power and flexibility of the language together with an understanding of the principles. A modest familiarity with the VS APL language is assumed. This section does not cover all features of the language, is not intended to replace the language reference manual, and is not a course in APL2 programming.

APL2 has opened the APL environment, providing links between APL and SQL and between APL and ISPF.

The section on SQL assumes no knowledge of SQL or relational databases. It provides an insight into the enormous potential that exists as APL functions can operate on tables from a relational database and as the SQL language is available to the APL user.

Communication between ISPF and APL2 is now possible. Each can access the same data. How to do this and what can be achieved are discussed. This section is more meaningful to the reader with some experience in ISPF.

The Shared Variable Processor has been redesigned. In the next section, the new facilities of the APL2 SVP are reviewed and main features of its design are discussed.

The next sections cover installation and migration (including a real example). They are intended as useful overviews of the process and take the form of observations, experiences, and recommendations. They are not intended as complete guides nor as replacements of the appropriate manuals.

The document concludes with a brief discussion of the performance of APL2.

Related publications are:

1. The APL2 Library
    - APL2 General information, GH20-9214
    - An Introduction to APL2, SH20-9229
    - APL2 Programming: Language Reference, SH20-9227
    - APL2 Programming: Guide, SH20-9216
    - APL2 Programming: System services Reference, SH20-9218
    - APL2 Programming: Using Structured Query Language (SQL) SH20-9217
    - APL2 Messages and Codes, SH20-9220
    - APL2 Installation and Customization under CMS, SH20-9221
    - APL2 Installation and Customization under TSO, SH20-9222
    - APL2 Migration guide, SH20-9215
    - APL2 Diagnosis guide, SY26-3931
    - APL2 Diagnosis reference, SY26-3932
    - APL2 Reference Summary SX26-3737
    - APL2 Reference Card SX26-3738
2. Structured Query Language(SQL) Publications
    - SQL/Data System General Information, GH24-5012
    - SQL/Data System Concepts and Facilities, GH24-5013
    - SQL/Data System Application Programming, SH24-5018
    - SQL/Data System Terminal User's reference, SH24-5017
    - SQL/Data System Messages and Codes, SH24-5019
3. IBM DATABASE2 (DB2) Publications
    - IBM DATABASE2 General Information, GC26-4073
    - IBM DATABASE2 Introduction to SQL, GC26-4082
    - IBM DATABASE2 Application Programming for TSO users, SC26-4081
    - IBM DATABASE2 Reference summary, SX26-3740

- IBM DATABASE2 Reference, SC26-4078
- IBM DATABASE2 Messages and Codes, SC26-4113

4. ISPF Publications
   - ISPF /PDF Reference, SC34-2139
   - ISPF /PDF Installation and customization, SC34-2143
   - ISPF Dialogue Management Services, SC34-2137

APL is an extremely powerful language that

- Handles arrays as easily as scalars
- Uses operators to create families of related functions
- Has simple rules of syntax

APL2 is an extension of APL. APL2 removes many of the restrictions of APL, generalizes many of the fundamental concepts, and extends or completes many functions and operators.

1.1 RESTRICTIONS LIFTED

VS APL has the restriction that numbers and characters cannot appear in the same array. APL2 relaxes that restriction. An array in APL2 may be a collection of numbers and characters.

VS APL is restricted as items of an array are limited to single numbers or single characters. APL2 allows any item of any array to be any other array. Such arrays are known as nested arrays.

Removing these restrictions adds the requirement for new functions. That is, we need functions that allow us to enter nested arrays at a terminal, allow us to enquire about the arrays structure, and rules of syntax to accommodate the new arrays.

1.2 DESIGN CRITERIA

APL2 was designed with the four main criteria of:

COMPATIBILITY a measure of the extent to which a proposal imposes a change - discussed further under Migration.

FORMALITY the extent to which a proposal follows rules.

SIMPLICITY a rule without any exceptions is preferable to one with exceptions.

USABILITY a measure of the ease with which the notation can be understood and applied.

1.3 DATA STRUCTURE

Programs are less reliant on data structure under APL2 than under VS APL. The structure relates to the values of, and the relationships between, the data.

Actual data is often not rectangular. Not everything has the same length name, not all products comprise the same number of subassemblies, and so on.

APL2 retains the useful properties of rectangular arrays but allows non-rectangular arrays to be represented easily in nested arrays. Nested arrays can be used just as easily as VS APL arrays. Users and programmers are not burdened with managing the data structure. More operations are controlled by the data and the need for explicit controls on these operations is removed. This makes programs less complex, easier to write, easier to use, and more flexible.

The structure of the data is handled by the language. The same primitive functions that work on simple arrays and on scalars work on nested arrays. The primitive functions also handle any necessary looping.

Nested arrays and the APL2 functions developed to handle the nested arrays allow the user to concentrate on solving the problem rather than on the array structure. It is often unnecessary to convert data from one data form to another data form to display data or to do arithmetic with the data. APL2 removes many of these concerns from the user.

The APL2 language is an enhanced version of APL, featuring:

- New data structures and types
- New functions and operators
- Enhancements to existing functions and operators

The enhancements make APL2 a very powerful and productive language. An overview of some of the more important enhancements is given in this chapter. Refer to the APL2 Language Manual (SH20-9227) for a more comprehensive discussion of the features.

2.1 APL2 ARRAYS

An array is an ordered rectangular collection of elements. In APL2 arrays, these elements may be numeric or character in the same array. Further, they may be simple scalars (that is, single characters or single numbers) or they may be other arrays.

A SIMPLE UNMIXED array is one containing either scalar character elements, or scalar numeric elements.Examples of simple unmixed arrays are:

```
      S1 ← 1 2 4 9 16

      S2 ← ⍳5

      S3 ← 'A' 'B' 'C'

      S4 ← 3 4 ⍴ 'ABCDEFGHIJKLMN'
```

Simple unmixed arrays were the only primitive arrays in VS APL.

A SIMPLE MIXED array is one containing both numeric and character scalar elements,such as:

```
      SM1 ← 'A' 'B' 'C'  1  2  3

      SM2 ← 'APL2',37 21
```

A NESTED array is an array that contains at least one non-scalar element. Examples of nested arrays are:

```
      N1 ←  'A'  (2 2 ⍴ ⍳4)  5


      TBAL ← 3 5 ⍴ 15 ?100

      'FINAL REPORT'  TBAL          ⍝   avoid  catenation
FINAL REPORT  87 25 54 46 75
              91 31 30  7 57
              44 16 85 83 41
```

Each item of a nested array is treated as one element of the array. For example:

```
      V ← 6 3 10  (2 2 ⍴ ⍳4) (3 3 ⍴ ⍳9)

      V
6 3 10  1 2   1 2 3
        3 4   4 5 6
              7 8 9

      ⍴V
5


      F ← 3 3 ⍴ 0

      (1 1⌽F) ← (1 1⍴1)(2 2 ⍴⍳4)(4 4 ⍴⍳16)

      F
1     0               0
0   1 2               0
    3 4
0     0     1  2  3  4
            5  6  7  8
            9 10 11 12
           13 14 15 16

      ⍴ F
3  3
```

Nested and mixed arrays provide an easy way to create reports as reports can simply be the display of one or more APL2 arrays. There is often no need to write functions or worry about formatting the layout. An example of how a report is produced is:

```
      A←0.60612 0.8382 65.99 0.060615 0.67222 0.89629 59.5 0.06722
      A←A,0.69225 0.923 57.78 0.06923 0.71595 0.9546 55.87 0.0716
      TAB1←'H2' 'EXPONENTIAL' 'ERLANG-2' 'CONSTANT',4 4⍴A

      TAB1

H2           0.60612  0.8382  65.99  0.060615
EXPONENTIAL  0.67222  0.89629 59.5   0.06722
ERLANG-2     0.69225  0.923   57.78  0.06923
CONSTANT     0.71595  0.9546  55.87  0.0716

      ⍴TAB1
4  5
```

The DISPLAY function, available in workspace DISPLAY in LIB 1, is useful in showing the structure of an array. Hence, the use of the DISPLAY function is a valuable aid in understanding how APL2 treats nested arrays.

Nested data is shown in boxes and the directions of the array are shown by arrows on the boxes, as can be seen in the illustration below:

```
      )COPY 1 DISPLAY DISPLAY

      DISPLAY  4.5  ¯2.4 (2 2⍴⍳4) (195  7  32)

.→-------------------------------------.
|                .→-----.  .→---------. |
| 4.5  ¯2.4      ↓ 1  2 |  | 195 7 32 | |
|                | 3  4 |  |          | |
|                '~-----'  '~---------' |
'←-------------------------------------'

      DISPLAY  ('APL' 2) ( 2 3 ⍴ ⍳ 6 )

.→-----------------------------------.
| .→----------.    .→-------.       |
| | .→----. 2 |    ↓ 1  2  3 |       |
| | | APL |   |    | 4  5  6 |       |
| | '-----'   |    '~--------'       |
| '←----------'                      |
'←-----------------------------------'

      DISPLAY  TAB1

.→-----------------------------------------------.
↓ .→-.                                           |
| |H2|          0.60615 0.8382  65.99 0.060615|
| '--'                                          |
| .→----------.                                 |
| |EXPONENTIAL| 0.67222 0.89629 59.5  0.6722  |
| '-----------'                                 |
| .→-------.                                    |
| |ERLANG-2|    0.69225 0.923   57.78 0.06923 |
| '--------'                                    |
| .→-------.                                    |
| |CONSTANT|    0.71595 0.9546  55.87 0.0716  |
| '--------'                                    |
'←-----------------------------------------------'
```

2.2 APL2 NEW FUNCTIONS

New functions were added to APL2 to deal with NESTED ARRAYS. A list of the new functions is given below. The syntax for each function will be explained in the next section, together with some brief examples.

DEPTH	≡ R
DISCLOSE	⊃ R
DISCLOSE WITH AXIS	⊃[I] R
ENCLOSE	⊂ R
ENCLOSE WITH AXIS	⊂[I] R
FIRST	↑ R
MATCH	L ≡ R
PICK	L ⊃ R
ENLIST	∈ R
FIND	L ⊆ R
CIRCLE	L ○ R

In addition, many functions have been extended or enhanced.

The functions, with examples to clarify their applications, are given in the following sections. For details or definitions, refer to the APL Programming: Language Reference (SH20-9227).

2.2.1 DEPTH

DEPTH (≡ R) analyses the degree of nesting in an array. Scalars have depth 0, simple arrays have depth 1, other arrays have depth of 1 plus the depth of the deepest nested item within the array. Hence all VS APL arrays had depth of 0 or 1.

The DEPTH symbol is formed by using the backspace character which can be displayed by the)PBS system command.

```
      ≡  2.84
0
      ≡  'A'
0
      ≡  'APL2'
1

      ≡  'ART' 'ALAN' 'MFB' 'RENE'
2

B←('ART' 'ALAN' 'MFB' 'RENE') 476 85 74 'APL2'

      ≡  B
3
```

When using the DISPLAY function, the depth of an array is the number of nested boxes containing the innermost item. For array B, above:

```
    DISPLAY B

┌→──────────────────────────────────────────────┐
│ ┌→────────────────────────┐          ┌→──┐    │
│ │ ┌→─┐ ┌→──┐ ┌→─┐ ┌→──┐ │ 476 85 74 │APL2│    │
│ │ │ART│ │ALAN│ │MFB│ │RENE│ │          └───┘    │
│ │ └──┘ └───┘ └──┘ └───┘ │                   │
│ └∈──────────────────────┘                   │
└∈──────────────────────────────────────────────┘
```

2.2.2 MATCH

MATCH (L ≡ R) compares two items or arrays. If both arguments are identical in structure and data (that is, they 'match'), a "1" is returned. If they do not match, a "0" is returned. Examples:

```
      'APL2'  ≡  'APL1'                    ⍝ different data
0

      'APL1' 'APL2'  ≡   'APL1APL2'        ⍝ different structure
0

      ' '  ≡  ⍳0                           ⍝ different structure
0                                          ⍝ and data

      'A' 'P' 'L' '2'   ≡   'APL2'
1
```

2.2.3 ENCLOSE

The enclose and disclose functions simplify the handling of nested arrays. Enclose enables us to treat an array as a single item without concern as to its structure or data. Disclose works the other way and enables us to get to the actual data. We can think of enclose as putting data into parcels so that it is easy to handle, and disclose as opening that parcel when we want to look at or to use what is inside.

ENCLOSE (⊂R) creates a scalar from its argument.

```
      V ← 'APL' (2 3 ⍴ ⍳6) (10 20 30 40)

      DISPLAY V
┌→─────────────────────────┐
│ ┌→──┐ ┌→────┐ ┌→─────────┐ │
│ │APL│ ↓1 2 3│ │10 20 30 40│ │
│ └───┘ │4 5 6│ └~─────────┘ │
│       └~────┘              │
└∊─────────────────────────┘
```

```
      ⊂ V
APL   1 2 3   10 20 30 40
      4 5 6

      DISPLAY ⊂V
.-----------------------------------.
| .→------------------------------. |
| | .→--. .→----. .→----------. | |
| | |APL| ↓1 2 3| |10 20 30 40| | |
| | '---' |4 5 6| '~----------' | |
| |       '~----'               | |
| '∈----------------------------' |
'∈-----------------------------------'
```

We see that when using ENCLOSE, the depth is increased, and the rank is lowered.

```
      ⍴ V                 ⍴⊂V                 ⍴⊂⊂V
3

      ⍴⍴V                 ⍴⍴⊂V                ⍴⍴⊂⊂V
1                   0                   0

      ≡ V                 ≡⊂V                 ≡⊂⊂V
2                   3                   4
```

ENCLOSE enables a vector of characters to be treated as a single scalar. This is very useful in application development. It is easy to find the position of a set of characters in a table, or to add a name to a table, if the items are enclosed and each name is treated as one item.

```
      TAB ← ('ART' 'ALAN' 'RENE' 'MFB')

      TAB ⍳ ⊂ 'ALAN'
2

[0]   TABB  NEWNAME
[1]  ⍝CHECK IF NEWNAME IN TAB
[2]  ⍝IF NOT, APPEND IT TO TAB
[3]   →((⊂NEWNAME) ∈ TAB)/0
[4]   TAB←TAB,⊂NEWNAME

      TAB                          ⍝   display the table
ART ALAN RENE MFB

      ⍴TAB
4

      TABB 'MICHEL'                ⍝   execution of the function

      TAB                          ⍝   display the table
ART ALAN RENE MFB MICHEL

      ⍴TAB
5
```

2.2.4 ENCLOSE WITH AXIS

ENCLOSE WITH AXIS (⊂[I]R) is used to restructure the data in a table or report. The data of R is restructured into a new array of increased depth and reduced rank. The axes "eliminated" in order to reduce the rank are specified by I. Examples:

```
      TABLE                              ⍴ TABLE
H2                                  4  11
EXPONENTIAL
ERLANG-2
CONSTANT

      ⍴⍴TABLE                            ≡ TABLE
2                                   1

      ⊂[2] TABLE
H2          EXPONENTIAL ERLANG-2    CONSTANT

      ⍴ ⊂[2] TABLE                       ⍴⍴ ⊂[2] TABLE
4                                   1

      ≡ ⊂[2] TABLE
2

      ⊂[1] TABLE
HEEC 2XRO  PLN  OAS  NNT  EGA  N-N  T2T  I    A    L

      ⍴ ⊂[1] TABLE                       ⍴⍴ ⊂[1] TABLE
11                                  1

      ≡ ⊂[1] TABLE
2

      DISPLAY ⊂[2] TABLE
.→-------------------------------------------------------.
| .→----------. .→----------. .→----------. .→----------. |
| |H2         | |EXPONENTIAL| |ERLANG-2   | |CONSTANT   | |
| '-----------' '-----------' '-----------' '-----------' |
'∈-------------------------------------------------------'

      , TABLE
H2          EXPONENTIALERLANG-2   CONSTANT

      ⍴,TABLE
44
```

```
      P ← 3 4 ⍴ ⍳12

      ⊂[2]P
1 2 3 4   5 6 7 8   9 10 11 12

      ⍴⊂[2]P
3

      ⊂[1]P
1 5 9   2 6 10   3 7 11   4 8 12

      ⍴⊂[1]P
4
```

2.2.5 DISCLOSE

DISCLOSE (⊃R) is the inverse of ENCLOSE, used to "get at" the data in nested arrays.

DISCLOSE restructures the data of R into an array of reduced depth and increased rank. The new dimensions that are added to increase the rank of R are placed last.

All items are padded on the right to match the largest dimension.

For example:

```
      D ← 'H2' 'EXPONENTIAL' 'ERLANG-2' 'CONSTANT'

      ⊃D
H2
EXPONENTIAL
ERLANG-2
CONSTANT

      ⍴D                ⍴⊃D
4                 4  11       A new dimension is placed last

      ≡D                ≡⊃D
                  1

      ⍴⍴D               ⍴⍴⊃D
1                 2
```

```
      M←(2 2⍴ 0)(2 2⍴⍳4)(2 2⍴9)

      M                              ⊃M
0 0   1 2   9 9              0 0
0 0   3 4   9 9              0 0

                             1 2
                             3 4

                             9 9
                             9 9

      ⍴M                             ⍴⊃M
3                            3 2 2

      ⍴⍴M                            ⍴⍴⊃M
1                            3

      ≡M                             ≡⊃M
2                            1
```

Disclose has no effect on simple scalars. Thus,

```
      5  <==>  ⊃5

     'K' <==>  ⊃'K'
```

2.2.6 DISCLOSE WITH AXIS

DISCLOSE WITH AXIS (⊃[I]R) is used to place new dimensions in different positions. The value of I defines the axes of the result for the restructuring of R.

The shape of the result is determined by the order in which the axes are listed in I.

All items are padded on the right to match the largest dimension.

```
      Q ← 'LOUIS' 'CROIX'

      ⍴Q                                    ⍴⍴Q
2                                       1

      ≡Q
2

      ⊃[2]Q          ⍝  equivalent to  ⊃Q
LOUIS
CROIX

      ⍴⊃[2]Q
2 5                  ⍝  new dimension placed last

      ⍴⍴⊃[2]Q                               ≡⊃[2]Q
2                                       1

      ⊃[1]Q
LC
OR
UO
II
SX

      ⍴⊃[1]Q
5 2                  ⍝  new dimension now placed first

      ⍴⍴⊃[1]Q                               ≡⊃[1]Q
2                                       1

      V2 ← (1 2 3 4) (10 12 14)

      ⊃[1]V2                                ⊃[2]V2
1  10                                    1  2  3  4
2  12                                   10 12 14  0
3  14
4   0
```

Disclose with axis can be applied to arrays of names to modify the presentation of reports. The following example illustrates how this can be done.

```
      TAB← 'MFB' 'ART' 'ALAN' 'MCF' 'XYZ' 'RST'

      ⊃TAB                                    ⍴ TAB
MFB                                       6
ART
ALAN
MCF                                           ⍴ ⊃ TAB
XYZ                                       6  4
RST

      ⊃[1]TAB
MAAMXR
FRLCYS
BTAFZT
  N


      (⊃TAB), ⍉ 5 6 ⍴⍳30
MFB  1  7 13 19 25
ART  2  8 14 20 26
ALAN 3  9 15 21 27
MCF  4 10 16 22 28
XYZ  5 11 17 23 29
RST  6 12 18 24 30


      (⊃[1]TAB),[1] ' ',[1]5 6⍴⍳30
 M  A  A  M  X  R
 F  R  L  C  Y  S
 B  T  A  F  Z  T
       N

 1  2  3  4  5  6
 7  8  9 10 11 12
13 14 15 16 17 18
19 20 21 22 23 24
25 26 27 28 29 30
```

2.2.7 PICK

PICK (L ⊃ R) selects a single item at any depth from an array R. The item is reached along the "path" given by L .

L can be a scalar or vector of depth ≤ 2. It can be an integer or empty. For example:

```
      V ← 'APL' ((3 2 ⍴ 'ABCDEF') 2 )

      2 ⊃ V
AB  2
CD
EF
      2 1 (3 1) ⊃ V
E
```

2.2.8 SELECTIVE SPECIFICATION

With nested arrays, index specification is not sufficient to allow editing of any item at depth greater than 1. SELECTIVE SPECIFICATION assigns values to items selected from an array. A selection function F applied to an array R selects items in R and assigns to those items the values contained in an array A, as shown below:

```
( F R ) ← A

( L F R ) ← A
```

Some primitive function may be used to create the selection condition.

PICK is particularly useful, as it allows the selection of an item from an array. Examples:

```
      ( 1⊃V ) ← 'APL2'

      V                                    ⍴V
APL2    AB   2                         2
        CD
        EF

      (2 2⊃V) ← 10 20 30

      V
APL2   AB   10 20 30
       CD
       EF

      M ← 3 3 ⍴⍳9

      ( 1 1 ⌽M ) ← 0

      M
0  2  3
4  0  6
7  8  0


      J ← 3 ¯4 7 ¯8 6 9 5 3 ¯10

      ((0>J)/J) ← ⊂'NEGATIVE'

      J
3  NEGATIVE  7  NEGATIVE  6 9 5 3  NEGATIVE

      ⍴J
9          ⍝  shape is unaltered as ⊂'NEGATIVE' is a scalar

      P ← 'ABCDE'

      (2↑P) ← 1 2
      P                                    ⍴P
1 2  CDE                               5

      V ← 'APL1' 1 2 3

      (1↑V) ← ⊂ 'APL2'

      V
APL2 1 2 3
```

2.2.9 FIRST

FIRST (↑R) returns the first item of its argument. Thus:

```
      ↑ 'APL2'
A
```

If an array of names is juxtaposed to a matrix of results, the first item of the resulting array is the matrix of names. This matrix of names also has a first item. For example:

```
      NM2←(4 5⍴'ROW1 ROW12ROW2 ROW3 ')
      RSLT←(4 6 ⍴ 24?1000)

      TAB2 ← NM2 RSLT

      ↑ TAB2                          ↑↑TAB2
ROW1                              R
ROW12
ROW2
ROW3

      ⍴↑TAB2                          ⍴↑↑TAB2
4  5
```

In order to determine the result of applying the function FIRST to an empty array, we must first define the type and the prototype of an array:

- The type of an array is the array in which each numeric item has been replaced by a "0" and each character item has been replaced by a "blank".

- The prototype of an array is the type of the first element of the array. The prototype is used as a fill item when padding is needed, as shown with DISCLOSE in the example below.

Hence, the result of applying the function FIRST to an empty array is the prototype of the array.

```
      M ← (2 2 ⍴⍳4) (3 3 ⍴ 10 + ⍳9)

      M
1 2  11 12 13
3 4  14 15 16
     17 18 19

      ⊃M                A PROTOTYPE  is  0
 1  2  0
 3  4  0
 0  0  0

11 14 17
12 15 18
13 16 19

      ,⊃M
1 2 0 3 4 0 0 0 0 11 12 13 14 15 16 17 18 19
```

2.2.10 FIND

FIND (L ⍷ R) returns a boolean array of the same shape as R.

An item of the result is 1 if the pattern given by L begins in the corresponding position of R, 0 otherwise. Example:

```
      U ← 3 3 ⍴ 1 0 0 1 1 1 0 0 1

      U
1  0  0
1  1  1
0  0  1

      ( 2 2 ⍴ 1 1 0 1 )  ⍷  U
0  0  0
0  1  0
0  0  0

      E ← 'INDIVISIBLE'

      'IS'  ⍷    E
0  0  0  0  0  1  0  0  0  0  0
```

2.2.11 ENLIST

ENLIST (∈R) returns a simple vector, comprising each simple scalar in the argument.

The function ENLIST is needed as the function RAVEL, applied to a nested array, returns a nested vector and not a simple vector. The example below illustrates that point.

```
      S ← 'H2' 'EXPONENTIAL' 'ERLANG-2' 'CONSTANT'

      ⍴S
4

      DISPLAY S
┌→──────────────────────────────────────┐
│ ┌→┐ ┌→──────────┐ ┌→───────┐ ┌→───────┐ │
│ │H2│ │EXPONENTIAL│ │ERLANG-2│ │CONSTANT│ │
│ └──┘ └───────────┘ └────────┘ └────────┘ │
└∈──────────────────────────────────────┘

      ,S                                    ⍴ ,S
H2 EXPONENTIAL ERLANG-2 CONSTANT          4

      DISPLAY ,S
┌→──────────────────────────────────────┐
│ ┌→┐ ┌→──────────┐ ┌→───────┐ ┌→───────┐ │
│ │H2│ │EXPONENTIAL│ │ERLANG-2│ │CONSTANT│ │
│ └──┘ └───────────┘ └────────┘ └────────┘ │
└∈──────────────────────────────────────┘

      ∈S                                    ⍴ ∈S
H2EXPONENTIALERLANG-2CONSTANT             29

      DISPLAY ∈S
┌→──────────────────────────┐
│H2EXPONENTIALERLANG-2CONSTANT│
└───────────────────────────┘
```

The ENLIST function allows you to know if a particular character or sequence of characters is contained in a nested array.

```
      'X'  ∈ S
0

      'X'  ∈ ∈ S
1
```

```
      V ← 'APL2' (2 3 ⍴ ⍳6) (10 20 30 40)

      ⍴V
3

      'L' ∈ V
0

      ∈V
 APL2  1 2 3 4 5 6 10 20 30 40

      ⍴∈V
14

      'L' ∈∈ V
1
```

2.3 APL2 NEW OPERATORS

New functions called Derived Functions can be created by applying operators to existing functions.

APL2 allows the user to define operators in addition to the operators the language provides. Thus, APL2 has an unlimited set of operators, whereas VS APL has a limited set of operators.

All operators, whether provided by APL or user defined, can be applied to any function to produce newly defined functions.

Operators in VS APL:

REDUCE	F/ R
	F⌿ R
	F/[I] R
SCAN	F\ R
	F⍀ R
	F\[I] R
EXPAND	LO⍀ R
	LO\[I] R
OUTER PRODUCT	L ∘.F R
INNER PRODUCT	L F.G R

APL2 provides one completely new primitive operator and enhanced VS APL operators.

The new operator is:

EACH	F " R L F" R

The enhanced VS APL operators are:

N-WISE REDUCE	N F/ R
REPLICATE	LO / R

All existing VS APL operators are extended in that they can be applied to user defined functions as well as to primitive functions.

All of the operators, new and enhanced, are discussed in the following sections.

2.3.1 EACH

EACH (⍤) can be monadic or dyadic.

The monadic form of EACH (F⍤ R), applies the function F to each item of the array R. The result has the same shape as the argument. That is, R being a vector of 3 arrays A, B, and C, if

```
Z  <==>  F⍤ R
```

then

```
F⍤ A B C   <==>  (F  A)(F  B)(F  C)
```

Examples:

```
      ⍳⍤ ⍳4
1   1 2   1 2 3   1 2 3 4

      ⍴ ⍳⍤ ⍳4
4

      T ← (2 12)(3 14)(1 20)(1 5)

      ↑⍤ T                                    +/⍤ T
2  3  1  1                                14 17 21 6

      (1 = ↑⍤T)/T
1  20    1  5

      ⎕←V←'APL' (2 3⍴⍳6) (2 4⍴'ABCDEFGH')
APL   1 2 3   ABCD
      4 5 6   EFGH

      ⍴⍤ V                                    ⍴⍤ ⍴⍤ V
3    2 3    2 4                             1  2  2

      ,⍤V                                     ⍴,⍤V
APL   1 2 3 4 5 6  ABCDEFGH                  3

      ∈V                                      ⍴ ∈V
APL 1 2 3 4 5 6  ABCDEFGH                    17
```

EACH can be applied to any function, primitive or user written.

The expression below shows EACH applied to the primitive ⎕CR:

```
      2  1 ⍴ ⎕CR⍤  'FN1'  'FN2'
```

where FN1, FN2 are functions. The canonical representations of each function are displayed one after the other.

The example below illustrates the application of EACH to a user written function.

```
    [0]  Z←RANK R
    [1]  Z←↑ρρR

          V ← 'APL2' (2 2 ρ ⍳4) (10 20 30 )

          RANK ⍤ V                          RANK  V
    1  2  1                            1
```

The dyadic form of EACH (L F⍤ R) applies the function F to corresponding pairs of arrays from L and R.Examples of the use of the dyadic form of EACH are:

```
          T1 ← 1 2 3 4
          T2 ← 10 12 14 16

          T1 ,⍤ T2                          ρ T1 ,⍤ T2
    1 10   2 12   3 14   4 16          4

          T1 , T2                           ρ T1 , T2
    1 2 3 4 10 12 14 16                8

          1 ¯1 ¯1 1 ↓⍤ T1 ,⍤ T2
    10  2  3  16

          3  2  ρ⍤ 4  5
    4 4 4   5 5

          Z ← 5↑⍤ 0ρ0 0 0
          ρZ                                ρ↑Z
    0                                  5

          3 ρ⍤ 'ABCD'  'XY'                 3 ρ 'ABCD' 'XY'
    ABC  XYX                           ABCD  XY  ABCD
```

The last example illustrates the mechanism of scalar extension. When one argument of a dyadic function is a scalar and the other argument is an array, the scalar argument is extended to an array with the shape of the non-scalar argument. The function is applied to its arguments after this extension has occurred.

When the dyadic form of the EACH operator is used with the CATENATE function as shown below:

```
      Q ← 'LOUIS' 'CROIX'


      Z ← 'ST. ' ,"Q
LENGTH ERROR
      Z ← ST.  ,"Q
        ^      ^
```

a length error occurs. Scalar extension is a way to obtain a correct result. As the right argument is a vector, the left argument has to behave like a scalar. This is achieved by enclosing the left argument, as shown below:

```
      Q ← 'LOUIS' 'CROIX'


      Z ← (⊂'ST. '),"Q
      Z


 ST. LOUIS ST. CROIX
```

Other examples of the use of dyadic EACH are:

```
      (⊂2 3) ⍴" 'ABC'                    2 3 ⍴ 'ABC'
AAA   BBB   CCC                      ABC
AAA   BBB   CCC                      ABC


      M1 ← 3 3 ⍴ 10?100
      M2 ← 3 3 ⍴ 10?100

      M1                                   M2
 87  24  54                           49  17  95
 46  75  91                           72  46  32
 31  30   7                           84  58  86

      ⌹" M1 M2
 0.01206  ¯0.00793  0.01021    0.02878  0.05547  ¯0.05243
¯0.01366   0.00582  0.02971   ¯0.04802 ¯0.05161   0.07225
 0.00517   0.01020 ¯0.02964    0.00427 ¯0.01937   0.01411

      ⍴ ⌹" M1 M2
2

      B ← 3 1 ⍴ 8 7 10

      (⊂B) ⌹" M1  M2
0.14291    0.094228     ⍝ these are the solutions to 2 sets
0.22852   ¯0.02294      ⍝ of 3 simultaneous equations with
0.18365    0.039714     ⍝ 3 unknowns.
```

EACH is equivalent to the DO loop in other programming languages. In most cases, loops in APL2 can be eliminated by using EACH.

```
      .
      .
L1: Z←Z,⊂F  ↑V
V ← 1↓V
→(0≠⍴V)/L1
      .
      .
```

This loop can be replaced by:

```
Z ← F¨ V
```

2.3.2 DEFINED OPERATORS

The user can define operators in the same way as he defines functions. In the following example, the derived function (F SEE) displays the arguments of F before applying F to its arguments.

```
    [0]   Z ← L (F  SEE) R
    [1]  ⍝SEE THE ARGS&RESULTS AS  F  EXECUTES
    [2]   →(0=⎕NC 'L')/MON
    [3]   Z←L  F  R
    [4]   ⎕←Z '←'  L  'f'  R
    [5]   →0
    [6]  MON:  Z←F R
    [7]   ⎕←Z '←f'  R

          1 + 2  3 + 4
  7  8


          1 + SEE  2 3 + SEE 4

     6  7  ←  2  3  f  4
     7  8  ← 1  f  6 7
  7  8

          -/ 12  2  5
   15

          -/ SEE  12  2  5

     15  ←f  12  2  5
   15

          +/ SEE  2  3  4   × SEE  4  5  6

     8 15 24  ←  2 3 4   f   4 5 6
     47  ←f   8 15 24
   47
```

2.3.3 N-WISE REDUCE

N-WISE REDUCE (N F⌿R), is an extension to the operator REDUCE. F is applied to the right argument, taking N items at a time. If the left argument, N, is negative, the items of R are reversed before the reduction. Examples:

```
      R ← 1 2 3 4 5 6
      4 +/ R
10  14  18

      S ← (1 5) (3 7) (8 9)

      2 +/S
4 12    11 16

      ¯2 -/1 3 5 7 9 11
2  2  2  2  2
```

```
      5 +/ R
15  20

      ⍴ 2 +/S
2

      ¯2 -/2*⍳10
2 4 8 16 32 64 128 256 512
```

It is possible to indicate the axis, or axes, along which the reduction will take place. (N F/[I] R)

```
      M ← 4 4 ⍴ ⍳16

      2 +/M
 3   5   7
11  13  15
19  21  23
27  29  31

      2 +/[1] M
 6   8  10  12
14  16  18  20
22  24  26  28
```

```
      3 +/M
 6   9
18  21
30  33
42  45
```

Any APL2 function can be used as the left argument to the slash operator. For example:

```
      2 ,/R
1 2   2 3   3 4   4 5   5 6

      ¯2 *⌿1 2 3 4 5
2  9  64  625

      (1,2≠⌿B)⌿B←2 3 3 3 5 8 9 9 12   ⍝ unique elements in B
2  3  5  8  9  12                    ⍝ when B sorted and 2≤⍴B
```

```
      ⍴ 2 ,/R
5
```

2.3.4 REPLICATE

SLASH (LO/ R) operator now accepts any numeric vector as its left operand. If an item of LO has the value n (positive), the corresponding item of R is repeated n times in the result. If an item of L is negative, n zeroes or blanks are inserted, according to the type of R. The number of non-negative items in L must be equal to the number of items in R.

REPLICATE also accepts an axes specification.

Examples:

```
      1 2 3 4/'ABCD'
ABBCCCDDDD

      1 2 ¯1 3 ¯2/ 8 9 10
8 9 9 0 10 10 10 0 0

      (⍳5)/⍳5
1 2 2 3 3 3 4 4 4 4 5 5 5 5 5

      M ← 2 2 4 ⍴ ⍳16
      M
 1   2   3   4
 5   6   7   8

 9  10  11  12
13  14  15  16

      1 ¯1 1 ¯1 /[1]M
 1   2   3   4
 5   6   7   8

 0   0   0   0
 0   0   0   0

 9  10  11  12
13  14  15  16

 0   0   0   0
 0   0   0   0

      2 ¯1 1 2 / (2 2 ⍴⍳4)(3 3⍴'ABC')(1 2 3)

1 2  1 2  0 0  ABC  1 2 3  1 2 3      ⍝ Prototype of
3 4  3 4  0 0  ABC                   ⍝ first matrix is
               ABC                   ⍝ used as fill element
```

```
      2  ¯1  1/[2] M
 1   2   3   4
 1   2   3   4
 0   0   0   0
 5   6   7   8

 9  10  11  12
 9  10  11  12
 0   0   0   0
13  14  15  16
```

2.3.5 SCAN

SCAN (LO \ R) has been extended similarly to SLASH. The SCAN operator in APL2 accepts any function as its left operand.

Examples:

```
      ×\ 1 2 3 4 5
1  2  6  24  120

      +\ (1 2) (3 4)(5 6)
1  2    4  6   9  12

      ,\1 2 3 4
1   1 2   1 2 3   1 2 3 4

      ⊃ ,\1 2 3 4
1  0  0  0
1  2  0  0
1  2  3  0
1  2  3  4

      ×\3 4 ⍴⍳12
1  2   6    24
5 30 210  1680
9 90 990 11880

      ,\3 4 ⍴⍳12
1  1 2   1 2 3    1 2 3 4
5  5 6   5 6 7    5 6 7 8
9  9 10  9 10 11  9 10 11 12

      DISPLAY  ,\[1] 3 4 ⍴ ⍳12
┌→──────────────────────────────────────────┐
↓ 1          2          3          4       │
│ ┌→──┐     ┌→──┐      ┌→──┐      ┌→──┐    │
│ │1  5│     │2  6│      │3  7│      │4  8│    │
│ └~──┘     └~──┘      └~──┘      └~──┘    │
│ ┌→────┐   ┌→────┐    ┌→────┐    ┌→────┐  │
│ │1  5  9│ │2  6  10│ │3  7  11│ │4  8  12│ │
│ └~────┘   └~────┘    └~────┘    └~────┘  │
└←──────────────────────────────────────────┘

      ×[1] 3 4 ⍴ ⍳12
2

      ≡  ,\[1] 3 4 ⍴ ⍳12
2
```

2.3.6 OUTER PRODUCT

OUTER PRODUCT (Z ← L ∘.RO R) is used to construct tables. It applies the function RO between pairs of items, one from L and one from R, in all combinations.

RO may be any function, either primitive or user defined.

Examples:

```
      (⍳4) ∘.× ⍳4
 1  2   3   4
 2  4   6   8
 3  6   9  12
 4  8  12  16

      76  85 ∘., 7 26 4
76 7   76 26   76 4
85 7   85 26   85 4

      '⊥' '⍎' ∘., 10 20 30
⊥ 10   ⊥ 20   ⊥ 30
⍎ 10   ⍎ 20   ⍎ 30

      ∇ Z←L POWER R
[1]   Z← L * R  ∇

      (⍳5) ∘.POWER  ⍳5
 1  1   1   1    1
 2  4   8  16   32
 3  9  27  81  243
 4 16  64 256 1024
 5 25 125 625 3125

      1 3 2 4 ∘.⊃ V←'APL2' (5⍴⍳6)(10 20 30 40) 'ABCDEF'
A 1 10 A
L 3 30 C
P 2 20 B
2 4 40 D
```

```
      (⍳4) ∘.≤ ⍳4
1  1  1  1
0  1  1  1
0  0  1  1
0  0  0  1

      ⍴76 85 ∘., 7 26 4
2  3

      '⊥' '⍎'  ,⍥ 10 20 30
LENGTH ERROR
      '⊥' '⍎'  ,⍥ 10 20 30
       ^        ^
```

2.3.7 INNER PRODUCT

INNER PRODUCT (Z ← L LO.RO R) combines subarrays along the last axis of L with subarrays along the first axis of R by applying an RO outer product. Then an LO reduction is applied to each item of that result.

The +.× inner product is the same function as the matrix product used in matrix algebra.

For simple vectors: +/V × C <==> V +.× C

INNER PRODUCT is extended to allow LO and RO to be any functions, either primitive or user defined.

Examples:

```
      V ← (1  2)(2  3)(3  4)(4  5)
      C ← 4  5  6  2

      +/ V × C                          V +.× C
40  57                                40  57

      T                                 U
1  2                                 0.1   0.2
3  4                                 0.3   0.4
5  6

      T ,.+ U
1.1  2.3    1.2  2.4
3.1  4.3    3.2  4.4
5.1  6.3    5.2  6.4

      M1
1  2  0  0
3  4  0  0
0  0  0  0
0  0  0  0

      M1 ∧.= M1
0  0  0  0
0  0  0  0
0  0  1  1
0  0  1  1
```

```
      M ← 3 3 ⍴ 9?100

      M
87  25  54
46  75  91
31  30   7

      M +.× M
10393   5670   7351
10273   9505   9946
 4294   3235   4453
```

(L +.∈ R) is useful for counting the number or elements of an array that are also elements of another array, or to determine whether a name is in a table.For example,

```
      K              A   ⍴ K  is  3  8
SATURDAY
8/15/85
AUG. 15

      K +.∈ '0123456789'
0  5  2

      R              A   ⍴ R   is   3  4
GOOD
WELL
BAD

      R ∧.= 'WELL'
0  1  0
```

```
      ∇ Z← L PLUS R                ∇ Z←L TIMES R
[1]   Z← L + R   ∇           [1]   Z← L × R   ∇

      (2 2⍴⍳4) PLUS.TIMES 2 3 ⍴15
 45  45  45
105 105 105

      3 4 PLUS . TIMES    3 4
25
```

2.4 COMPLEX NUMBERS

Complex numbers can be used in APL2 just like real numbers: assigned to variables, used as arguments to functions, and so on.

The availability of complex numbers greatly increases and facilitates the use of APL2 in the engineering and scientific area. In many fields, such as electrical engineering, electromagnetism, nuclear physics, hydraulics, and acoustics, behavior is described with complex numbers.

COMPLEX numbers can be entered in rectangular or polar form. For the CIRCLE function (L ○ R), the value of L has been extended to range from ¯12 to 12. The extended values provide the real part, imaginary part, magnitude, and phase of a complex number.

The conjugate of a complex number is obtained by using (+C), the conjugate primitive function.

```
      C ← 3J2

      C
3J2

      9 ○ C                A  real part of C
3

      10 ○ C               A  magnitude of  C
3.605551274

      11 ○ C               A  imaginary part of C
2

      12 ○ C               A  phase  of  C
0.5880026035

      ¯10 ○ C              A  complex conjugate of C
3J¯2

      + C                  A  complex conjugate of C
3J¯2
```

The solution of equations sometimes results in complex numbers.

Polynomial equations can be solved by using the POLYZ function from workspace MATHFNS in public library no. 1

```
      POLYZ 1 1 1                    A  sol. of (X*2) + X + 1 = 0
 0.5J0.86602540   ¯0.5J¯0.86602540

      11 ○ POLYZ 1 1 1               A imaginary part of the
 0.86602540   ¯0.86602540            A sol. of  (X*2) + X + 1 = 0
```

2.5 ERROR HANDLING

APL2 provides several new facilities to assist with problem determination and correction. Errors are now indicated by 2 carets, one to show how far the execution of the statement had progressed, and one to show the likely point of the error.

The state indicator now includes errors made in immediate execution as well as those made in functions. The)SIS command displays the statement in error as well as the function and line.

The)RESET command clears the status indicator. It clears all executions suspended due to error. In VS APL, a right arrow had to be entered for each error entry, clearing them one by one.

```
      )SIS                      ⍝ status indicator with statements

*  1 3 2 4+2 3 4                ⍝ most recent  interruption
   ^      ^                    ⍝ two carets to indicate the error
*  5+0
   ^^
TIMES[1] Z←L×R                  ⍝ error indication with the statement
           ^ ^
*  1 2 TIMES 1 2 3
   ^   ^

      )RESET                    ⍝ reset the status indicator
      )SIS
```

After an error has occurred, the ⎕L and ⎕R system variables contain the left and right arguments of the function. They are especially useful for the programmer during debugging functions with shared variables. Left and right arguments of suspended functions can be modified and execution can then be resumed from the point of interruption in the statement by entering →⍳0 . For example, suppose the user defined function has been incorrectly called,as follows:

```
     3 4 PLUS 3 4 5             ⍝ using user defined function

LENGTH ERROR
PLUS[1]  Z←L+R
           ^ ^

      ⎕L
3 4

      ⎕R
3 4 5

      ⎕L ← 1 2 3                ⍝ correct error

      →⍳0                       ⍝ resume function execution
4 6 8
```

EXECUTE ALTERNATE (L ⎕EA R) attempts to execute the expression in the right argument. If there is an error, ⎕EA executes its left argument.

Thus ⎕EA provides a way to retain program control after an error. If an application is likely to produce a specific error that can be anticipated, together with a correction, ⎕EA is ideal.

Programmers are often advised that their programs should do extensive error checking. End users must not be thrown into APL with confusing error messages. However, it is not feasible to search for all possible errors. Some errors are unpredictable.

Often a function will make extensive checks to ensure that the argument being supplied to a function will be acceptable to that function. For example, the function checks that data being fed to a multiplication is numeric or that a matrix being fed to an inverse is not singular. The function spends much of its time just checking to ensure that it will work. It would be nice to just try it and back off if it fails. Using ⎕EA allows the programmer to have APL do the testing.

The function to be executed is entered as the right argument to ⎕EA. If the data is not acceptable and the function fails, ⎕EA passes control to the left argument. This could then perform appropriate remedial action, such as asking the user to reenter the data. Program control has thereby been maintained and the complicated testing has been done by APL.

However, ⎕EA will pass control to the left argument if the right argument fails for any reason. This could be a WS FULL condition. The left argument has to determine what error caused the right argument to fail.

The error determination can be done using the two system variables ⎕ET and ⎕EM which contain the error type and error message, respectively, after each error.

The ⎕EA system function should be considered as a mean of retaining program control.

The system function ⎕ES, event simulation, allows the user to act just as the system does when an error occurs.

An example of the use of these error handling variables and functions is given below. A function PLUS is defined. The first statement uses ⎕EA to keep program control when the evaluation of L+R fails. The third statement checks the error type with ⎕ET.

- If there is a length error, statements 5 to 13 allow the user to correct it.

- In all other cases, the fourth statement simulates the occurrence of the event specified by ⎕ET. The function now behaves as if there was no ⎕EA.

```
      ∇PLUS [□] ∇
    ∇
[0]   Z←L PLUS R
[1]  L0:'→L1' □EA 'Z←L+R'
[2]    →0
[3]  L1:→(□ET ≡ 5 3)/ERR1
[4]   □ES □ET
[5] L3:⍞←W←'WHICH ARGUMENT DO YOU WANT TO REDEFINE, RIGHT OR LEFT?(R L)'
[6]   →('RL'∈1↑(⍴W)↓⍞)/L4,L5
[7]   →L3
[8]  L4:'REDEFINE RIGHT ARGUMENT'
[9]   R←□
[10]  →L0
[11] L5:'REDEFINE LEFT ARGUMENT'
[12]  L←□
[13]  →L0
[14] ERR1:→L3,0⍴□←' * *  ARGUMENTS OF DIFFERENT LENGTH  * * '
    ∇ 1984-11-06  9.00.02 (GMT-5)


      1 2 3  PLUS  3 4 5
4 6 8

      2 3  PLUS  2 3 4

 * *  ARGUMENTS OF DIFFERENT LENGTH  * *
WHICH ARGUMENT DO YOU WANT TO REDEFINE, RIGHT OR LEFT ? (R L) R
REDEFINE RIGHT ARGUMENT
□:
      2 3
4 6

      2 3  PLUS  2 3 4

 * *  ARGUMENTS OF DIFFERENT LENGTH  * *
WHICH ARGUMENT DO YOU WANT TO REDEFINE, RIGHT OR LEFT ? (R L) L
REDEFINE LEFT ARGUMENT
□:
      1 2 3
3 5 7
```

□ET and □EM together with □ES allow the user to receive more meaningful error messages. The programmer now has great control over APL's handling of errors and can tailor error handling to each individual application. For example,

```
      3 4 TIMES 2 3 4    ⍝  execution of user function

LENGTH ERROR
TIMES[1]  Z←L×R
           ^ ^

      ⎕EM
LENGTH ERROR
TIMES[1]  Z←L×R
           ^ ^

      ⎕ET
5 3
```

Using the NATIONAL LANGUAGE facility of APL2, all system commands and system messages are displayed in the language assigned to the ⎕NLT ← 'xxxxxxx' system variable. APL2 has already defined 9 different national languages.

2.6 APL2 EDITORS

APL2 provides two editors for entering and editing defined functions and operators. The)EDITOR command is used to query and select the editor. The two editors are:

1. EDITOR 1: This is a line editor similar to the VS/APL line editor. One significant improvement is the ability to delete multiple lines with a single command.

2. EDITOR 2: This is a fullscreen editor requiring GDDM release 3. Most of the commands are the same as those used for the line editor. Editor 2 has additional commands that allow you to locate occurrences of character strings, make global changes, and copy lines within a function or from another function. The fullscreen editor can also be used to edit simple character vectors or matrices.

The fullscreen editor is different from the CMS editors, the ISPF/PDF editor, or the fullscreen editor previously available with the VS/APL Fullscreen Support field developed program. The differences are:

- There is no specific command line on the screen. Commands are entered between brackets at the start of any line on the screen.

- There is no 'insert' or 'add' command. If new lines are entered starting in column 1, overtyping existing lines, they are inserted before the first line overtyped. The line editor method of inserting lines can also be used.

- APL symbols are used for commands.

To users of other fullscreen editors, the APL2 fullscreen editor might appear strange at first. However, it does have the advantage of being similar to the line editor, it is a powerful editor, and it is easy to learn to use its full capabilities.

If the systems editors are preferred over the new APL2 editor, functions can easily be developed which allow use of other editors for APL2 functions. The systems editors however do not offer an APL execution command [±]. "Using the PDF Editor for APL2 Functions" on page 76 and "Appendix C. Sample APL2 Function to Use CMS Editor" on page 109 show how the ISPF/PDF editor or the CMS editor can be used from APL2. These methods do require that an 'edit' function be in the workspace and that the object to be edited be entered in quotes.

3.0 APL2 AND RELATIONAL DATA BASES.

This chapter deals with the connection between APL2 and the relational data bases SQL/DS and DB2. APL is no longer confined to its own environment. APL2 has a new auxiliary processor, AP127, which allows communication with a relational data base either in VM/SP or in MVS. AP127 is shipped with the distribution tape. SQL commands can be processed directly from APL2 and the result returned as an APL2 array which can then be operated on by APL2 functions. APL2 has become a very convenient language to deal with relational data bases.

The concept of nested arrays and the concept a relational data bases evolved independently. Even so, tables from relational data base map nicely into the new concept of nested arrays which was introduced in APL2. The connection between APL2 and relational data bases allows APL2 to exploit the facilities of the data base products in dealing with collections of data.

This chapter provides an introduction to APL2 with SQL/DS or DB2 for the system engineer. The subjects covered are:

- An overview of relational data bases
- Organization of APL2 and SQL/DS in VM/SP
- Organization of APL2 and DB2 in MVS
- Working with a relational data base in APL2
- Examples of APL2 using DB2 or SQL/DS
- Conclusion

3.1 WHAT IS A RELATIONAL DATA BASE ?

Two relational data base management systems are available :

- DB/2, which runs under MVS
- SQL/DS which runs in VM/SP, VSE, or SSX.

The two data base management systems have a number of common characteristics.They are handled by the Structured Query Language (SQL). Both systems allow a user to access shared data for on-line, interactive, and batch systems.

The two data base management systems simplify the task of handling data. The language SQL provides facilities for querying data and manipulating

data. Data operations which can be complex in APL2 can frequently be done more easily with the SQL language.

SQL/DS and DB2 both provide a catalog which manages all the information that they can handle. The catalog contains information about data, storage, programs, and authorizations. It is often managed by a Data Base Administrator who grants the authorizations.

SQL/DS and DB2 have comprehensive and integrated recovery schemes with disk logging, automatic recovery on restart, and utilities. Data is protected from three types of failure: system, media, and application program. With APL2, there is protection against failure of APL2 or of the auxiliary processor.

Utilities are shipped with SQL/DS and DB2 to help process large amounts of data with batch jobs. The utilities are the DBS utility for SQL/DS, invoked by the procedure SQLDBSU, and the DBS utilities option of DB2I (DB2 interactive) for DB2.

DB2 and SQL/DS both use the relational model of data. A RELATION in the relational data model can be thought of as a simple two-dimensional table having a specific number of columns and some unordered rows. We will consider, in terms of the tables:

- The structure of the tables
- The operations we may perform on them
- The commands which permit us to execute these operations
- The data types to specify
- Different views of the data

3.1.1 STRUCTURE OF TABLES.

A table is identified by a table name. Each column relates to a given characteristic, contains data of the same kind, and has a name. Each row relates to a specific object,contains data of different kinds, and has no name. A row is similar to a record in a conventional data set. An example of a table is shown in Figure 1 on page 45

BIN	YEAR	TYPE	STORLOC	COST	COLOR
C11	1971	RIOJA	IMPORT DEPARTMENT	4.94	R
C12	1971	RIOJA	WINE CLUB SHOW CASE	3.94	R
F16	1983	ROSE	WINDOW	-	-
G12	1974	MERLOT	BASEMENT	8.94	R
I10	1979	BARDOLINO	ANNEX	2.25	R
K10	1981	CHABLIS	ON ORDER	3.94	W
B10	1973	CHAMBERTIN	COUNTER	10.94	R
B11	1966	BORDEAUX	SPECIALITY CORNER	8.94	R
B12	1974	BORDEAUX	SPECIALITY CORNER	10.94	R
K11	1981	RIESLING	SHELF	6.25	W
I11	1979	VAPOLICELLA	ANNEX	2.25	R

Figure 1. Example of Table Named WINE.

This array has six columns and some rows and represents a relation in the relational data model. The intersection of one row and one column is the smallest unit which can be handled. It is called a value in the terminology of DB/2 and a field in the terminology of SQL/DS. When a value (field) is missing, it has a NULL value

Rows have no inherent order. If the data is to be retrieved in a specific order, the user must specify that order.

There are two things to notice about the array in Figure 1. The array is an ordinary matrix of eleven rows and six columns. In addition, the array looks suspiciously like an APL2 array.

3.1.2 OPERATIONS ON TABLES.

A data base system allows a variety of operations to be performed. Basic operations on tables are:

- Creating or dropping (deleting) tables
- Retrieving data, whole tables, rows or parts of rows
- Updating, inserting, or deleting data
- Copying data from one table into another
- Performing table utility operations, such as bulk data loading, data reorganization, and printing

An operation unique to the relational data model is called JOINING. This operation causes the data base system to merge data from different tables.

Figure 2 presents a table named ORDERS. This table contains the quantities of each wine ordered by customers.

CUST	BI	QUANTITY
ALAN	C11	3
ALAN	C12	4
MANUEL	G12	5
MICHEL	B10	2
MICHEL	B11	1

Figure 2. Example of Table Named ORDERS.

The syntax of the SQL command that joins the table ORDERS in Figure 2 with the table WINE in figure 1 is:

```
SELECT CUST,BIN,TYPE,COST,QUANTITY,QUANTITY*COST
FROM ORDERS,WINE
WHERE BI = BIN
```

The result of the joining of the tables is:

```
ALAN   C11 RIOJA       4.94 3 14.82
ALAN   C12 RIOJA       3.94 4 15.76
MANUEL G12 MERLOT      8.94 5 44.7
MICHEL B10 CHAMBERTIN 10.94 2 21.88
MICHEL B11 BORDEAUX    8.94 1  8.94
```

For each order, the type, the cost, the quantity, and the price are listed.

Joining table as above shows some of the power that the relational data base systems offer. A single statement can merge two tables and can perform an operation between two columns. The query specifies what the user wishes to see, which tables contain the desired data, a search condition and the required operation between two columns.

3.1.3 STRUCTURED QUERY LANGUAGE (SQL)

SQL is a high-level language for handling data. With SQL you specify what you want, not how to get it. You do not have to know how or where data is stored. Most programming and data languages process data one record at a time. To use them you code a sequence of instructions explaining

how to get the data, what to look for, and what to do with it. With SQL you do not have to specify all this information; you select all the data you want with a single statement.

SQL commands consist of command verbs, one or more optional clauses, language keywords, and parameter operands. SQL commands can be entered at the terminal, contained in programs, and now used in APL2 with the auxiliary processor. In APL2, the data is received directly into an array. APL2 functions operate on data all at once without the need of loops. The APL2 language and the SQL language fit together nicely.

The most commonly used SQL commands are shown in Figure 3 on page 48 and are grouped into five types.

Query Command :	
SELECT	Retrieves data from one or more tables
Data Manipulation Commands :	
INSERT	Places a new row in a table
UPDATE	Changes data fields in one or more rows
DELETE	Removes one or more rows from a table
Data Definition Commands :	
CREATE	Defines new tables, views, indexes, synonyms, dbspaces for SQL/DS, tablespaces for DB2
ALTER	Changes the description of tables, tablespaces in DB2, dbspaces in SQL/DS
ACQUIRE	Acquires a dbspace in which tables and indexes can be created (SQL/DS only)
DROP	Erases a tablespace, a view, an index, or a table
Authorization Commands :	
GRANT REVOKE	Control access to data and privileges on the data base system
Control commands :	
COMMIT ROLLBACK	Permit explicit control of the disposition of a unit of work
LOCK TABLE	Lock a table or a tablespace in DB2

Figure 3. Summary of SQL Commands.

3.1.4 DATA TYPES.

When working with a relational data base system, the user has to be aware of the data type of the columns. Figure 4 is a summary of the different data types. When the data is fetched, basic operations such as addition, multiplication, or averaging can be applied on columns.

DATA TYPE	Descriptions
DECIMAL (m,n)	Decimal data, where m is the total number of digits and n the number of decimal digits
INTEGER	Large positive or negative number (4 bytes)
SMALLINT	The same as INTEGER but less than 32,767 (2 bytes)
FLOAT	Floating point number : from 5.4E-79 to 7.2E+75
CHAR(n)	Fixed length character string up to 255 bytes
VARCHAR(n)	Varying character string up to 255 bytes
LONG VARCHAR	Varying character string up to 32,767 bytes

Figure 4. Types of Data in Relational Tables.

When data of DECIMAL, INTEGER, SMALLINT or FLOAT is fetched, the resulting APL2 variable is numeric.

3.1.5 VIEWS AND INDEXES.

A VIEW is a logical table that is derived from one or more tables. VIEW DEFINITIONS can be stored in a relational data base system.

Views look like stored tables and can be used as if they were tables. However, some operations are restricted.

Views are used to simplify data retrieval commands or to limit user access. An example of how to create a view WINE2 which is a subset of the table WINE is:

```
CREATE VIEW   WINE2     AS
SELECT YEAR,TYPE,COST,COLOR FROM WINE WHERE COLOR ='R'
```

SQL/DS and DB2 are able to handle large amounts of data (up to 64 billion bytes for DB2). To improve the performance of retrieving data from the data base, INDEXES can be created on columns. They greatly improve the performance of a retrieval. An example of how to build an index on the table WINE for a specific column YEAR is:

```
CREATE UNIQUE INDEX  IWINE
ON WINE  (YEAR ASC)
```

3.2 OPERATING SYSTEM IN SQL/DS

To use SQL/DS data bases,you should know about the environment and ensure that the access to SQL/DS has been prepared.

3.2.1 ENVIRONMENT IN SQL/DS

A SQL/DS data machine is a VM/SP virtual machine that owns the minidisks where the data bases are stored and handles the data base. A data base machine is active for only one data base at a time. SQL/DS is initiated by an SQL procedure SQLSTART EXEC and terminated by an operator command SQLEND. The different modes of operation that are available are:

1. Single User Mode.

   In single user mode, SQL/DS, its preprocessors, and application programs run in a single VM/SP virtual machine.

2. Multiple User Mode.

   In multiple user mode, one or more users or applications concurrently access the same database. For this mode of operation, SQL/DS runs in a VM/SP virtual machine while one or more APL2 users, or batch users, or interactive users operate in other virtual machines.

3. Multiple Data Base Operation.

   In multiple data base operation, several SQL/DS data base machines run in multiple user mode under the same VM/SP.

These modes of operation are illustrated in Figure 5 on page 51

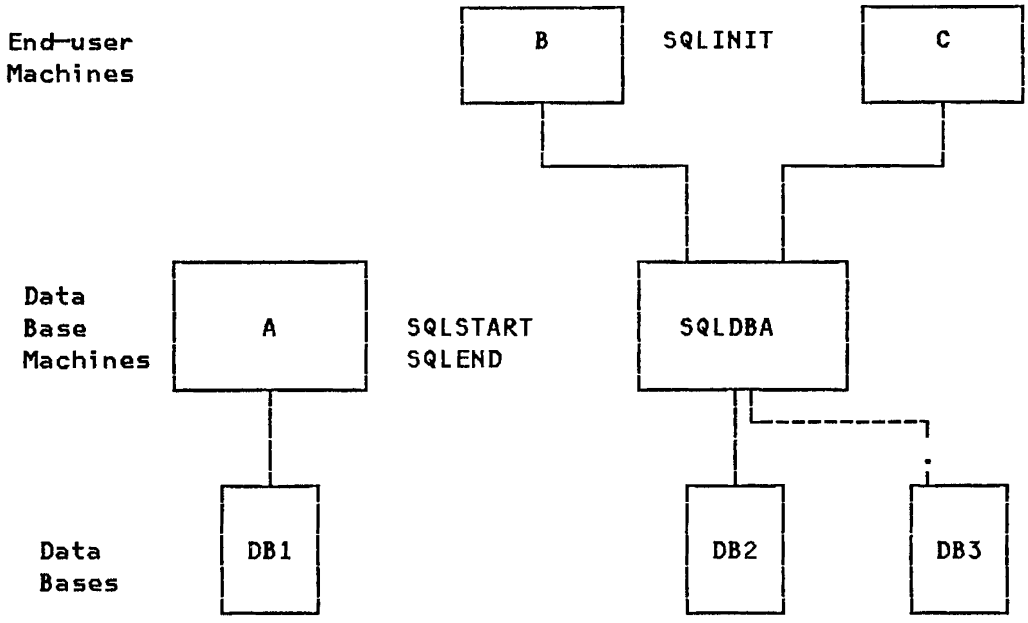


Figure 5. Single or Multiple Access to Data Bases.

Once a data base machine has been activated in multiple user mode, many users can access the SQL/DS data base simultaneously in batch mode, or dynamically, or with APL2. For this type of operation, users normally must have:

- a proper SQL/DS authorization
- a VM/SP IUCV path to the data base machine
- read access to the SQL/DS production disk
- executed the SQLINIT exec to establish the current data base association.

In example 1 of Figure 5, an SQLSTART exec has activated SQL/DS data base A in single user mode (A is the data base machine for the data base DB1) APL2 must be started in this machine if access to the data base DB1 is required.

In example 2 of Figure 5, SQLSTART EXEC has activated the SQL/DS data base machine SQLDBA in multiple user mode (SQLDBA is the data base machine for data base DB2). User virtual machines B and C used the SQLINIT EXEC to select DB2 as their SQL/DS data base. They can then start APL2 in their machine and pass the appropriate functions.

Starting the data base machine in a multiple users mode is usually done by the data base administrator or by a procedure.

The auxiliary processor AP127 is like any application program and can operate in different modes in VM/SP. In all modes, access to the SQL/DS production minidisk is required.

3.2.2 PREPARING ACCESS TO SQL/DS

To ensure that APL2 and SQL/DS can communicate with each other, with SQL/DS, you have to preprocess the source AP2V127I ASMSQL must be pre-processed. An entry in the table SYSACCESS of SQL/DS will be created to control the access of APL2. The distribution tape contains the source and the text of the auxiliary processor.

3.2.2.1 WHAT THE SQL/DS DATA BASE ADMINISTRATOR MUST DO

- Preprocess the source AP2V127I ASMSQL. For example:

```
SQLPREP ASM PP(NOPU,NOPR,PREP=AP2V127I,
USER=SQLDBA/SQLDBAPW) IN(AP2V127I)
```

  SQLDBA is the name of the machine which handles the data base and SQLDBAPW is the password.

- Authorize users to run the program AP2V127I. In order to give this authorization, the command

```
GRANT RUN  ON AP2V127I TO PUBLIC;
```

  must be passed to ISQL, the interactive way to use SQL/DS. For this case, any APL2 end user may use SQL/DS.

3.2.2.2 WHAT THE INDIVIDUAL USER NEEDS

- Authority to use the program AP2V127I and CONNECT authority to the data base.
- Authority to use a DBSPACE in which to create tables.
- Link and access the SQL/DS production disk.
- Run SQLINIT exec to create the access module ARISRMBT on his disk. An example of SQLINIT which makes the link with SQLDBA is:

```
SQLINIT DBNAME(SQLDBA)
```

- Start APL2 and try some SQL commands with the functions in the workspace SQL, shipped with the distribution tape.

3.3 OPERATING SYSTEM IN DB2

In an MVS environment, DB2 operates as a formal subsystem of MVS. Applications that access DB2 resources can run in batch, TSO, IMS, or CICS environments. Let us consider how APL2 can access DB2 and what has to be done to prepare this access.

3.3.1 ENVIRONMENT IN MVS.

APL2 can access DB2 in the TSO environment as any program runs in TSO with DB2. Figure 6 gives an overview of APL2 in an MVS environment. Notice that APL2 with AP127 can reach all the data handled by DB2.

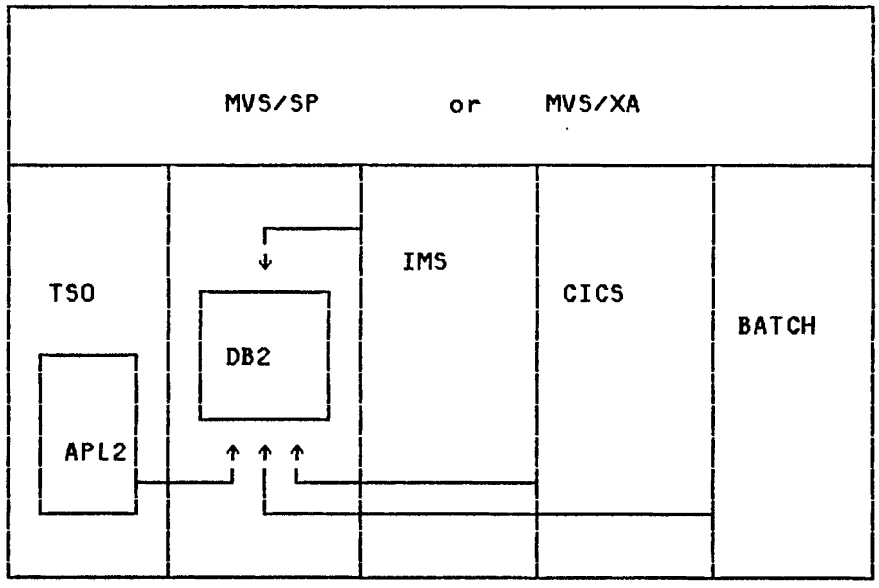


Figure 6. APL2 in an MVS Environment

The auxiliary processor AP127 is an assembler program containing embedded SQL statements as does any other DB2 program. Four steps must be performed before it can be run:

- Precompilation: check SQL syntax, produce a modified source program, and produce a database request module (DBRM) which is an intermediate form of an SQL statement.

- Compilation: translate the modified source program.

- Bind: process the DBRM to produce an application plan, the executable code representing one or more SQL statements.

- Link-edit: produce the final object module.

Binding is the activity that converts the DBRM, a set of syntactically correct SQL statements, into a set of executable instructions to DB2. If all the SQL statements are correct and if the binder is authorized to access the data, DB2 builds an application plan that contains information about the program and the data the program uses. Figure 7 illustrates this process.

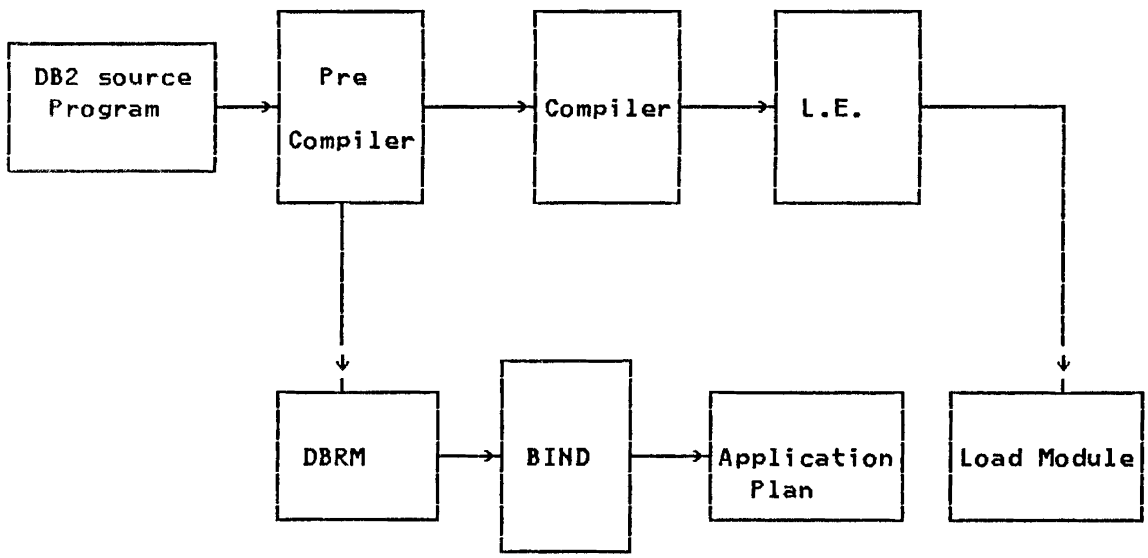


Figure 7. Process and bind of a DB2 program

3.3.2 PREPARING ACCESS TO DB2

The load module AP2T127 of AP127 is a member of the linklib of APL2. The DBRM of this program is stored in APL2.SYMBLIB with the name AP2TDBRM. The level of the DBRM module must be checked with the level of the AP2T127 load module.

3.3.2.1 WHAT THE DB2 DATA BASE ADMINISTRATOR MUST DO

- Bind the APL2 application (AP127) to DB2

- Grant any AP127 user RUN authority to DB2 through the APL2 application plan

The sample job stream AP2JBIND can be processed after customization according to the installation. The default APL2 application plan is APL2PLAN. This name can be changed. If the name is changed, it is necessary to specify the name of the plan during the invocation of AP127 when APL2 is started.

The two previous operations can be done with DB2I, which is an interactive way to work with DB2.

3.3.2.2 WHAT THE INDIVIDUAL USER NEEDS

- Authority to use the auxiliary processor AP127

- Authority to use a TABLESPACE in which a table can be created

- Start APL2 and try some queries with the functions in the workspace SQL, shipped with the distribution tape.

3.4 HOW TO USE THE SQL WORKSPACE.

An APL2 user can work with a relational data base by personally managing the shared variables. However, it is easier to use the workspace SQL which is shipped with the program product. The use of this workspace greatly facilitates access to the data bases. The rules to access DB2 or SQL/DS are very similar. Any differences will be noted.

Check with the Data Base Administrator about the rules of the data base and to ensure that you have the appropriate authorizations to do what you want to do. This checking is to be done prior to working with DB2 or SQL/DS. For our work, we asked for a dbspace TEST because we want a space to create tables in an SQL/DS data base. If we were working with DB2 we would have asked for a tablespace in a data base.

In this chapter we will demonstrate:

- How to communicate with DB2 or SQL/DS

- The functions in the workspace SQL
- The structure of the result data

3.4.1 AN EASY WAY TO COMMUNICATE WITH A RELATIONAL DATA BASE.

This section contains examples of some basic operations using the workspace SQL.

The APL2 function SQL is used to create, insert, and query a sample table. The SQL command is passed as the right argument to the SQL function and the auxiliary processor returns a vector of three items:

- The first item is a five-element return code vector
- The second item is the result data array
- The third item is a request stack vector

For example, to create a table WINE in the dbspace TEST, an SQL command CREATE must be executed. CREATE specifies the characteristics of the desired table. To execute CREATE, a variable CWINE in an APL2 character matrix or vector, containing the CREATE command of the SQL language, is created. (In DB2 there is no DBSPACE. You must specify the name of the database and the tablespace. For instance: IN DBTEST.TSTEST)

Each column type must be defined. If a value is required, NOT NULL must be added.

The SQL command CREATE is processed from APL2 by passing the variable CWINE as the right argument of the function SQL:

```
     SQL CWINE
0 0 0 0 0
```

AP127 returns a five-element code vector, the first result item returned by SQL. All zeros shows a successful completion.

Similarly, if the appropriate SQL command is put into a character matrix INSERTTAB, a row can be inserted in the table, as shown below.

```
          CWINE

 CREATE    TABLE       WINE
 (BIN      CHAR(3)     NOT NULL,
  YEAR     SMALLINT            ,
  TYPE     VARCHAR(12) NOT NULL,
  STORLOC  VARCHAR(20)         ,
  COST     DECIMAL(6,2)        ,
  COLOR    CHAR(1)             )
 IN TEST
```

This operation inserts just one row in the table WINE. With APL2, rows can also be inserted in bulk. We will be discuss in the next section.

The function COMMIT makes all changes to the data base since the last successful shared offer or since the last COMMIT operation permanent. It is necessary to commit modifications if they are to be available to other end-users.

Suppose that many rows have been inserted in the data base. The data base can be queried by:

```
R←SQL 'SELECT * FROM WINE'
```

(WINE is a table given in Figure 1 on page 45)

R is a three-element result. The second element is called a relation in the language of SQL. For the example, the second item of R is:

```
C11 1971 RIOJA        IMPORT DEPARTMENT     4.94  R
C12 1971 RIOJA        WINE CLUB SHOW CASE   3.94  R
F16 1983 ROSE         WINDOW
G12 1974 MERLOT       BASEMENT              8.94  R
I10 1979 BARDOLINO    ANNEX                 2.25  R
K10 1981 CHABLIS      ON ORDER              3.94  W
B10 1973 CHAMBERTIN   COUNTER              10.94  R
B11 1966 BORDEAUX     SPECIALITY CORNER     8.94  R
B12 1974 BORDEAUX     SPECIALITY CORNER    10.94  R
K11 1981 RIESLING     SHELF                 6.25  W
I11 1979 VAPOLICELLA  ANNEX                 2.25  R
```

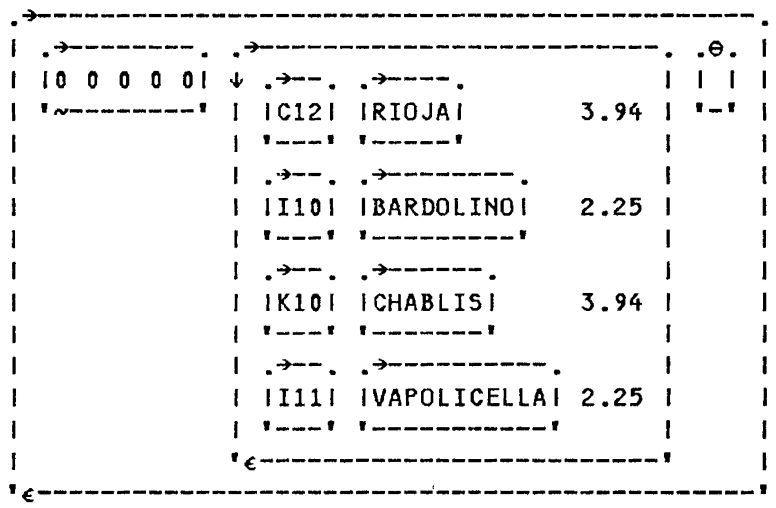
The result returned by the function SQL is always a three-item vector. The second or third item can be empty. We will now consider two examples, looking at them in greater detail.

```
      INSERTTAB

INSERT INTO  WINE
       ( BIN   , YEAR , TYPE   , STORLOC  )
VALUES ( 'F16' , 1983 , 'ROSE' , 'WINDOW' )

      SQL INSERTTAB
0 0 0 0 0
      COMMIT
 0 0 0 0 0

     R← SQL 'SELECT BIN , TYPE , COST FROM WINE WHERE COST <= 4'
     DISPLAY R
```

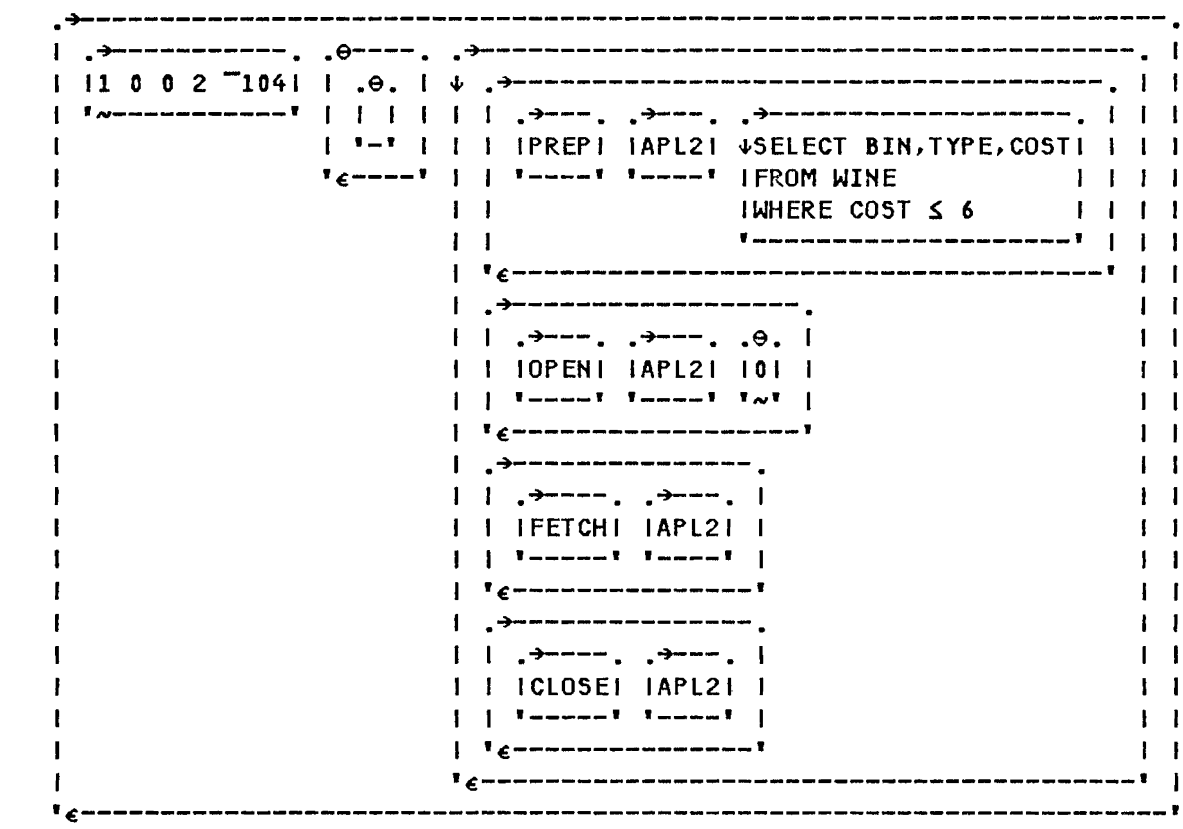


Where:

- The first item is a five-element return code vector
- The second item is the result data array. If no result data is required or if none is obtained, this item is empty. There will be no result data array if an error occurs that prevents the process from completing.
- The third item is a request stack vector, containing incomplete requests. This item is an empty vector if all requested operations have completed.

The next example shows a query with an error, so the second item is empty and the stack is filled by the requested operations.

```
R← SQL 'SELECT BIN , TYPE , COST FROM WINE WHERE COST ≤ 4'
DISPLAY R
```



The return code vector is not equal to zero so there is an error. The 1 indicates that an error exists. The 2 indicates that the error was detected in either DB2 or in SQL/DS. The ¯104 is the return code of the data base system and is, in fact, a syntax error. The error is ≤ . This function does not conform to the syntax of an SQL command. <= is used to define less than.

In this case the second item is an empty vector and the third item is the stack vector which contains the sequence of functions which should be processed. The error can be corrected directly in APL2 by editing the stack. Processing can then be continued by the function RESUME which is provided in the workspace SQL.

Authorization is required to access a table belonging to another user. The name of the creator must be specified before the name of the table. For example, to list the main catalog in the SQL/DS data base system:

```
SQL 'SELECT TNAME,CREATOR,DBSPACENAME FROM SYSTEM.SYSCATALOG'
```

SYSTEM is the name of the creator of the table SYSCATALOG.

```
SQL 'SELECT * FROM SYSTEM.SYSCATALOG WHERE CREATOR=''MICHEL''  '
```

This query will give all the information for all tables created by MICHEL.

3.4.2 FUNCTIONS IN THE WORKSPACE SQL

APL2 can easily communicate with a data base system such as DB2 or SQL/DS as the SQL function does the necessary work. The SQL function handles the shared variables and the communication with AP127. However, it is necessary to know the functions in the SQL workspace in order to develop applications or to manage AP127 in an efficient manner.

An overview of the SQL workspace is presented in Figure 8 on page 61. In addition, Figure 8 contains the functions in the SQL workspace, organized into five types.

There are two different ways to communicate with the auxiliary processor:

- Immediately, using a single request (EXEC,PREP,FETCH...)
- With the SQL function which builds the statements

For example, consider the following query:

```
SELECT TYPE,COST
FROM WINE
WHERE TYPE LIKE 'B%'
```

The function SQL builds a stack of four functions which will be processed. Each function shares data with the auxiliary processor.

Statement Type	Statement in SQL language	Basic functions in W.S. SQL	Auxiliary functions in workspace SQL.
Query	SELECT	SQL	PREP 'S1' SEL1 OPEN 'S1' VALUES FETCH 'S1' CLOSE 'S1'
Data Manipulation	DELETE INSERT UPDATE		- EXEC INSERTAB - PREP 'S2' IWINE CALL 'S2' VALUES
Data Definition	CREATE ALTER DROP ACQUIRE		EXEC
Authorization	LOCK TABLE GRANT REVOKE		
Control	COMMIT ROLLBACK	COMMIT ROLLBACK	

Figure 8. Statements in SQL and the Access Operations

```
      DISPLAY ⊃STACK
   .→-------------------------------------.
   ↓ .→---.  .→---. .→-------------------. |
   | |PREP|  |APL2| ↓SELECT TYPE,COST    | |
   | '----'  '----' |FROM WINE           | |
   |                |WHERE TYPE LIKE 'B%'| |
   |                '--------------------' |
   | .→---.  .→---. .⊖.                    |
   | |OPEN|  |APL2| |0|                    |
   | '----'  '----' '~'                    |
   | .→----. .→---.                        |
   | |FETCH| |APL2|                        |
   | '-----' '----'                        |
   | .→----. .→---.                        |
   | |CLOSE| |APL2|                        |
   | '-----' '----'                        |
   '←-------------------------------------'
```

When the stack is processed, the following operations are done:

- PREP gives the name APL2 to a prepared statement, passes the query and prepares the dynamic SQL statement for the data base system.

- OPEN opens the prepared statement and passes a vector of values(value-list). In the example, the value-list is empty,as the value has been passed with the PREP statement

- FETCH returns the shared data which is the result table. The number of rows returned can be controlled by the function SETOPT or by a right parameter in the FETCH function.

- CLOSE closes the OPEN statement.

The SQL function builds a stack of auxiliary functions like PREP, OPEN, FETCH, CLOSE, and is convenient to use. However, in an application it might be better to use the auxiliary functions directly rather than using the SQL function as each step can then be processed and checked.

The SQL language used by AP127 is slightly extended so that SQL statements can contain references to APL2 arrays. It is a powerful implementation as the APL2 user no longer has to work on a row by row basis.

An SQL statement can refer directly to an APL2 array. When referring to an APL2 array, the SQL statement can specify a column in the array by specifying the column number prefixed by a colon.

It is easy to pass values in an array. For example, consider an array called CHAMPAGNE and a variable INSERT containing an SQL statement. A display of INSERT and CHAMPAGNE is:

```
      DISPLAY   INSERT     CHAMPAGNE
.→-------------------------------------------------------------------.
| .→-------------------------. .→-----------------------------------. |
| ↓INSERT INTO WINE          | ↓ .→--.      .→----------.          | |
| |(BIN , YEAR , TYPE , COST)| | |J12| 1977 |DON PERIGNON|    13.95 | |
| |VALUES(:1,:2,:3,:4)       | | '---'      '------------'          | |
| '--------------------------' | .→--.      .→--------------.       | |
|                              | |J13| 1978 |MOET ET CHANDON| 11.2  | |
|                              | '---'      '---------------'       | |
|                              | .→--.      .→------.               | |
|                              | |J14| 1981 |POMMERY|         10.5  | |
|                              | '---'      '-------'               | |
|                              '€-----------------------------------' |
'€-------------------------------------------------------------------'
```

Issuing the SQL function

```
      SQL INSERT CHAMPAGNE
```

results in

```
      DISPLAY ⊃STACK
.→-------------------------------------------------.
↓ .→---. .→---. .→-------------------------.       |
| |PREP| |APL2| ↓INSERT INTO WINE          |       |
| '----' '----' |(BIN , YEAR , TYPE , COST)|       |
|               |VALUES(:1,:2,:3,:4)       |       |
|               '--------------------------'       |
| .→---. .→---. .→-------------------------------.  |
| |CALL| |APL2| | .→--.      .→----------.      |  |
| '----' '----' | |J12| 1977 |DON PERIGNON| 13.95 |  |
|               | '---'      '------------'      |  |
|               '€--------------------------------'  |
| .→---. .→---. .→-------------------------------. |
| |CALL| |APL2| | .→--.      .→--------------.    | |
| '----' '----' | |J13| 1978 |MOET ET CHANDON| 11.2 | |
|               | '---'      '---------------'    | |
|               '€--------------------------------' |
| .→---. .→---. .→-------------------------.       |
| |CALL| |APL2| | .→--.      .→------.     |       |
| '----' '----' | |J14| 1981 |POMMERY| 10.5 |       |
|               | '---'      '-------'     |       |
|               '€-------------------------'       |
'€-------------------------------------------------'
```

In the processing of the SQL function,the two variables are passed to SQL as a right argument and the SQL function generates as many CALL functions as there are rows of the variable CHAMPAGNE in the stack. The indexes (:1,:2,:3,:4) point towards the columns of the matrix CHAMPAGNE which contain the list of values.

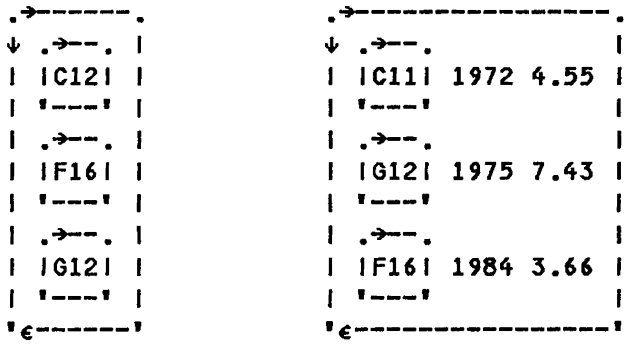
This example points out a very important characteristic: The APL2 user inserted three rows at one time, rather than doing it row by row. Thus,

in APL2 several rows can be inserted in a table with a bulk type of insertion.

Here are some more examples.

```
SQL 'DELETE FROM WINE BIN=:1'  LIST1

SQL 'UPDATE WINE SET YEAR=:2 COST=:3 WHERE BIN=:1'  LIST2
```



3.4.2.1 STRUCTURE OF RESULT DATA.

When a FETCH is processed, AP127 returns the result table in one of the two data structures:

- A matrix of variable length items
- A vector of simple matrices

The control of the data structure can be specified by two parameters, MATRIX or VECTOR, either in the SETOPT parameter list or in the FETCH request. If SELECT is the variable containing the query

- and MATRIX was specified
  - The result is in an array in which each NULL element of the table gives the empty vector
  - Each item has its exact length (even for data type VARCHAR)
- and VECTOR was specified
  - There is no difference between 0 and the NULL value for numeric data or blank and the NULL value for character data
  - AP127 pads character items to the length of the longest item

If LENGTH is also specified, there is one more item which provides the length of each data.

Example:

```
      SELYEAR
SELECT YEAR,TYPE,COST,COLOR
FROM WINE
WHERE YEAR BETWEEN 1980 AND 1983
```

```
      SETOPT 'MATRIX'
0 0 0 0 0
      DISPLAY 2⊃R←SQL SELYEAR
.→-------------------------.
↓      .→---.     .⊖.   .⊖. |
| 1983 |ROSE|     |0|   | | |
|      '----'     '~'   '-' |
|      .→------.         .→. |
| 1981 |CHABLIS|   3.94 |W| |
|      '-------'        '-' |
|      .→-------.       .→. |
| 1981 |RIESLING|  6.25 |W| |
|      '--------'       '-' |
'←-------------------------'
```

```
      SETOPT 'VECTOR' 'LENGTH'
0 0 0 0 0
      DISPLAY 2⊃R←SQL SELYEAR
.→-----------------------------------------.
| .→---. .→-------. .→---. .→. .→------. |
| ↓1983| ↓ROSE    | ↓0   | ↓ | ↓1 4 0 0| |
| |1981| |CHABLIS | |3.94| |W| |1 7 1 1| |
| |1981| |RIESLING| |6.25| |W| |1 8 1 1| |
| '~---' '--------' '~---' '-' '~------' |
'←-----------------------------------------'
```

3.4.2.2 OTHER SQL FUNCTIONS.

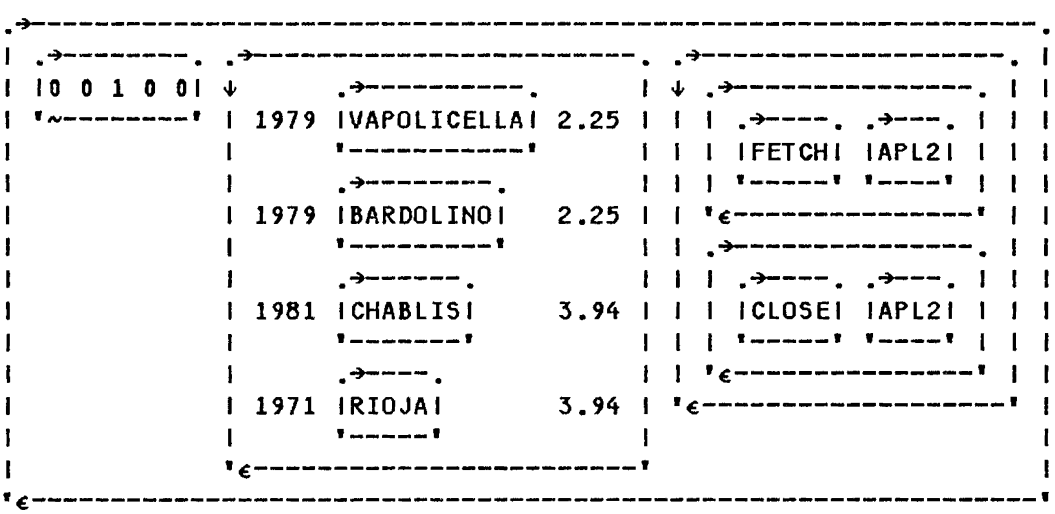
The workspace SQL provides other functions. A list of the functions contains:

- MESSAGE - explain the return code
- SHOW - display the error message, the result, and the stack
- GETOPT - query the current values of the options
- COMMIT - record any modifications in the data base
- ROLLBACK - back out all changes made in the data base since the last successful COMMIT
- RESUME - resume the stack, for instance, when the result table returned by AP127 is larger than the limit given by SETOPT
- SETOPT - control the number of rows that can be accessed at each FETCH

Examples of the use of SETOPT and RESUME:

```
      SETOPT 4
0 0 0 0 0

      R ← SQL 'SELECT YEAR,TYPE,COST FROM WINE ORDER BY COST'
      DISPLAY R
```



The function SETOPT controls the maximum number of rows returned by each FETCH function. If SELECT returns more than 4 rows, it is necessary to process the stack using the function RESUME.

```
      MESSAGE 1⊃R
FETCH INCOMPLETE: RESULT TABLE MAY CONTAIN MORE ROWS

      RESUME 3⊃R
0 0 0 0 0    1971 RIOJA    4.94
             1981 RIESLING 6.25
             1974 MERLOT   8.94
             1966 BORDEAUX 8.94
```

It is necessary to RESUME until the third element of the return code vector is equal to 0.

3.5 MISCELLANEOUS

The workspace SQL is shipped with APL2 and contains useful functions.

To these functions can be added user written functions. Some examples are given in the appendix. They should be useful, and also provide ideas for more functions. In this section we consider the functions in SQL/DS. The functions work similarly in DB2.

3.5.1 FUNCTIONS WHICH DEAL WITH THE TABLES OF THE CATALOG.

It is useful for the user to know the system tables, his own tables, the names of the columns of a specific table, or a quick view of a table. The results of some functions that are helpful are given in this section.

SQLSYSTEM - a function which gives the names of all tables in the catalog. Useful information can be retrieved for each table. For instance:

- SYSACCESS gives the name of the programs, such as AP127, which can work with SQL/DS
- SYSUSERAUTH gives information about the users who are authorized to work with SQL/DS

```
      SQLSYSTEM                    (for SQL/DS)

 TNAME        CREATOR  TABLETYPE NCOLS REMARKS DBSPACENO DBSPACENAME

 SYSACCESS    SYSTEM   R             9 COMMENT         1 SYS0001
 SYSCATALOG   SYSTEM   R            18 COMMENT         1 SYS0001
 SYSCOLAUTH   SYSTEM   R             6 COMMENT         1 SYS0001
 SYSCOLUMNS   SYSTEM   R            14 COMMENT         1 SYS0001
 SYSDBSPACES  SYSTEM   R            12 COMMENT         1 SYS0001
 SYSDROP      SYSTEM   R             3 COMMENT         1 SYS0001
 SYSINDEXES   SYSTEM   R            16 COMMENT         1 SYS0001
 SYSPROGAUTH  SYSTEM   R             6 COMMENT         1 SYS0001
 SYSSYNONYMS  SYSTEM   R             4 COMMENT         1 SYS0001
 SYSTABAUTH   SYSTEM   R            15 COMMENT         1 SYS0001
 SYSUSAGE     SYSTEM   R             7 COMMENT         1 SYS0001
 SYSUSERAUTH  SYSTEM   R             6 COMMENT         1 SYS0001
 SYSVIEWS     SYSTEM   R             4 COMMENT         1 SYS0001
```

SQLTAB - gives details of a user's own tables.

```
      SQLTAB
 TNAME      CREATOR  TABLETYPE NCOLS REMARKS DBSPACENO DBSPACENAME

 CUSTOMERS  PR465ERG R             4                10 TEST
 ORDERS2    PR465ERG R             4                10 TEST
 ORDERS3    PR465ERG R             3                10 TEST
 STOCKS     PR465ERG R             5                10 TEST
 WINE       PR465ERG R             6                10 TEST
```

SQLCOLNAME - gives the specifications of the columns of a table.

```
      SQLCOLNAME 'WINE'

 COLNO CNAME    COLTYPE   LENGTH

     1 BIN      CHAR           3
     2 YEAR     SMALLINT
     3 TYPE     VARCHAR       12
```

```
     4 STORLOC VARCHAR       20
     5 COST    DECIMAL   ( 7, 2)
     6 COLOR   CHAR            1
```

SQLDISP - gives a view of a table

```
      SQLDISP 'ORDERS'

 CUST   BIN QUANTITY

 ALAN   C11        3
 ALAN   C12        4
 MANUEL G12        5
 MICHEL B10        2
 MICHEL B11        1
```

3.5.2 FUNCTIONS WHICH DEAL WITH A TABLE

A great deal of work with tables consists of producing reports with break points or with cross results. Two functions, REPORT and ACROSS, are examples of how APL2 can facilitate this.

In the following example, we ask for an average on column 5 for each type of wine before 1980.

```
      '3 AVG 5' REPORT SQL 'SELECT * FROM WINE WHERE YEAR < 1980'

 BIN  YEAR  TYPE        STORLOC                COST  COLOR

 C11  1971  RIOJA       IMPORT DEPARTEMENT     4.94  R
 C12  1971  RIOJA       WINE CLUB SHOW CASE    3.94  R
            RESULT                            4.44
 G12  1974  MERLOT      BASEMENT               8.94  R
            RESULT                            8.94
 B10  1973  CHAMBERTIN  COUNTER                10.94  R
            RESULT                            10.94
 B11  1966  BORDEAUX    SPECIALITY CORNER      8.94  R
 B12  1974  BORDEAUX    SPECIALITY CORNER     10.94  R
            RESULT                            9.94
```

3.5.3 FUNCTIONS INVOKING GDDM.

When working with a relational data base, it can be useful to produce some graphics from the results. An example of the use of SQLICU, which calls the ICU utility of GDDM/PGF, is presented. SQLICU takes the result of an SQL query as its right argument and the column numbers as the left argument.

```
      QICU
SELECT TYPE,AVG(COST)
FROM WINE
WHERE COST IS NOT NULL
GROUP BY TYPE
      '1 2' SQLICU SQLX QICU
```

Many different graphics can be obtained from ICU, such as linear diagram, bar chart, pie chart and even tower chart.

4.0 USING ISPF WITH APL2 UNDER TSO

ISPF 2.1.2 permits the use of APL2 for the development of dialog functions. Appendix G of ISPF Dialog Management Services (SC34-2137) describes this capability.

This chapter introduces the interaction between APL2 and ISPF. Notes on the installation requirements, and lists of ways in which APL2 and ISPF interaction improve application development are also presented.

4.1 ESTABLISHING AN ISPF-APL2 ENVIRONMENT

The basis for communication between APL2 and ISPF lies in the ability of both products to access the same variables. ISPF can use the APL2 workspace variables as its dialog function pool. This means that APL2 variables can be referenced in dialog panels, and values altered by the panels can subsequently be accessed by APL2 functions. There are, however, two restrictions on APL2 variables which are used by ISPF:

1. The names of the variables must be acceptable to ISPF. Valid names must not be more than eight characters long and must not contain special APL2 characters.

2. The values of the variables must be simple character strings. APL2 general data types are not allowed.

An ISPF-APL2 environment must be created to allow this communication to take place. The environment is created by invoking APL2 as an ISPF command with the special new parameter LANG(APL). The APL2 command can be used with ISPSTART, directly from an ISPF menu, or in a CLIST which is part of an ISPF dialog. An advantage of using a CLIST is that datasets required by APL2 need only be allocated immediately before they are used.

See "Appendix B. Sample Panel and Clist for Initiating ISPF-APL2" on page 107 for an example of how to initiate an ISPF-APL2 environment using a CLIST from the PDF primary menu.

4.2 INSTALLATION AND INITIATION

The auxiliary processor (AP) used by APL2 to access ISPF is called ISPAPAUX. This AP is distributed as part of the ISPF product in the ISPPLIB dataset. Depending on the options chosen at installation time, ISPAPAUX may also be in LINKLIB or LPALIB. The AP should be explicitly requested in the APL2 invocation command by including ISPAPAUX in the APNAMES parameter. There are two points to note:

1. Specifying APNAMES(ISPAPAUX) overrides the defaults. To add the ISPF AP to the defaults, code the following parameter:

```
APNAMES(AP2X104,AP2T127(PLAN(APL2PLAN) SSID(DSN )),ISPAPAUX)
```

2. APL2 will look for the ISPAPAUX in the following datasets:

   - The APL2 LOADLIB specified in the invocation command or allocated prior to the command being issued
   - The ISPF task library allocated to ISPLLIB
   - The normal MVS search sequence: STEPLIB;LPALIB;LINKLIB

The ISPAPAUX AP is known to APL2 as AP317. Although it is possible to communicate directly with AP317 by sharing a variable, an ISPEXEC function is distributed with ISPF to simplify use of ISPF.

The workspace containing the ISPEXEC function is distributed in transfer form in member ISPFWS of the ISPALIB dataset. During ISPF installation, this workspace should be saved in an APL2 public library to simplify access by all users.

The steps taken to save the APL2 workspace are:

1. Invoke APL2.

2. Enter)IN 'ISP.V2R1M2.ISPALIB(ISPFWS)' to convert to an active APL2 workspace.

3. Enter)SAVE 1 ISPFWS to save the workspace in public library 1 with the name ISPFWS. See "Loading APL2 Public Workspaces" on page 89 for further information about saving workspaces in public libraries.

4.3 USING ISPF SERVICES FROM APL2

When APL2 has been invoked in an ISPF-APL2 environment, you request ISPF dialog services using the ISPEXEC function. The right argument is a vector of characters representing the parameters to be passed to the dialog service, as in the example shown below. The format of the vector is the same as that for dialog service statements for command languages (CLIST and EXEC). The ISPF return code for the selected service is returned by ISPEXEC and should be stored for analysis.

```
RC←ISPEXEC 'SELECT PANEL(MENU1)'
```

The default action taken by ISPF for any error with a return code of 12 or higher is to terminate the function. In the ISPF-APL2 environment, this means that APL2 is terminated and control is returned to the PDF primary menu. You can use the ISPF CONTROL command, as shown below, to ensure that all errors are returned to the APL2 function.

```
RC←ISPEXEC 'CONTROL ERRORS RETURN'
```

When the variable names used in the APL2 function are valid ISPF names, the values of the variables can be set or referenced in either APL2 or a dialog service. Figure 9 shows how APL2 can use an ISPF panel to request input from a terminal user and display results.

ISPF Panel : DATA

```
)ATTR DEFAULT(%+⍎)
)BODY
% Using an ISPF Panel for APL2 data entry
+
+ Enter five numbers: ⍎A  ⍎B  ⍎C  ⍎D  ⍎E  +
+
+ The Sum is .......:%&SUM                +
+
```

APL2 Function

```
[0]  ISPFVAR;RC SUM A B C D E
[1]  SUM←A←B←C←D←E←'0'
[2]  ⍝ DISPLAY ISPF PANEL
[3] L1:RC←ISPEXEC 'DISPLAY PANEL(DATA)'
[4]  ⍝ SUM THE NUMBERS LETTING APL2 DETECT INVALID INPUT
[5]  SUM←'''INVALID INPUT''' ⎕EA '⍎+/⍎¨A B C D E'
[6]  →(RC=0)/L1   ⍝ LOOP IF NOT 'END' OR ANY ERROR
```

Figure 9. Using Panel Display from an APL2 Function.

4.4 USING ISPF AND APL2 TO CREATE DIALOGS

In addition to being able to use ISPF services from APL2, two additional features are provided to simplify the development of dialogs. The features are:

1. An APL2 function can be used as a dialog function.

2. An ISPF-APL2 environment can be created and used to execute APL2 functions without the terminal user being aware that APL2 has been invoked.

4.4.1 USING APL2 DEFINED FUNCTIONS AS DIALOG FUNCTIONS

In an ISPF-APL2 environment, an APL2 function in the active workspace can be used as a dialog function from a selection panel, other dialog functions, or application command tables. You request execution of an APL2 function by using the same instruction as for other dialog functions. However, you must include the LANG(APL) parameter. For example:

```
SELECT CMD(APLFUNC parameters) LANG(APL)
```

For dialog functions written as CLISTs or in other high level languages, ISPF creates a separate function pool for each new function. ISPF does not create a separate function pool with APL2 functions. An APL2 function selected from ISPF is executed (⍎ in APL2 terms) in the APL2 workspace. This applies to dialogs running under APL2 and to use of the split screen ISPF capability. There is, therefore, only one active workspace, used as the function pool for all APL2 functions in the ISPF-APL2 environment. Thus, any variables created or altered by an APL2 function will not be reset or deleted prior to subsequent functions.

4.4.2 INCLUDING APL2 DIALOGS IN NORMAL ISPF DIALOGS

Users of dialogs do not usually wish to see unnecessary APL2 screens or messages. The error handling features of APL2 make it possible to protect users from meaningless error messages. The usual APL2 initiation messages can be hidden from the user with the following features:

1. The INPUT invocation parameter on the APL2 command provides a simple method of specifying APL2 commands or functions to be executed after APL2 has started. Multiple commands may be entered. Each command is enclosed in quotation marks and separated from the next command by blanks or commas.

2. The QUIET invocation parameter prevents APL2 from displaying any output until APL2 prompts for input. If you are using the session manager, the QUIET parameter does not suppress the display of the initial session manager screen.

3. The PROFILE invocation parameter without a profile name - PROFILE() -, stops a profile from being loaded and suppresses the display of the session manager screen. The display of the session manager could also be suppressed by specifying the PROFILE parameter with the name of a profile coded with DISPLAY OFF.

Figures 10 and 11 on page 73 illustrate the above techniques. The APL2 invocation would probably be from a CLIST to allow any special APL2 allocations to be done. If the TSO procedure or ISPF CLIST does all the allocations, the APL2 invocation could be directly from a menu panel.

APL2 Invocation Command (in CLIST or ISPF menu)

```
ISPEXEC SELECT CMD(APL2 -
                   AP(ISPAPAUX) INPUT( -
                    ')LOAD AVRWSPC' -
                    'ISPEXEC ''SELECT PANEL(MENU1)''' -
                    ')OFF' ) -
                   QUIET PROFILE() -
                 ) LANG(APL)
```

Figure 10. Invoke APL2 with Automatic Function Execution

Selection Panel (MENU1)

```
)ATTR DEFAULT(%+⍟)
)BODY
% Select APL2 Function as a Dialog Function
%
%SELECTION ===>⍟ZCMD          +
+    %1+ Select APL2 Average Function
+
+        Enter five numbers: ⍟A ⍟B ⍟C ⍟D ⍟E +
+
+        Result is.........:%&RESULT                    +
+
)PROC
   &ZSEL=TRANS(TRUNC(&ZCMD,'.')
               1,'CMD(AVERAGE) LANG(APL)'
               *,'?')
)END
```

APL2 Function <AVERAGE>

```
[0]  AVERAGE;RESULT A B C D E
[1]  ⍝ SINCE SELECTION PANEL USES SHARED POOL,
[2]  ⍝ WE MUST USE VARIABLE SERVICES TO GET VALUES OF VARIABLES
[3]  RC←ISPEXEC 'VGET (A B C D E) SHARED'
[4]  RESULT←'''INVALID INPUT''' ⎕EA '⍕+/⍷"A B C D E'
[5]  RC←ISPEXEC 'VPUT RESULT SHARED'
```

Figure 11. Using an APL2 Function as a Dialog Function

4.5 POSSIBLE APPLICATION AREAS

APL can no longer be considered as existing in a world of its own!

Using ISPF and APL2, it is now possible to develop applications using the tools most appropriate to any particular task. APL2 can use the function provided by ISPF, or any tool supported by ISPF (including PDF). Similarly APL2 functions can be incorporated into dialogs developed using CLISTs or other programming languages. This flexibility opens the doors to a host of different possibilities. A few of these possibilities are mentioned in this section.

4.5.1 USING THE PDF EDITOR FOR APL2 FUNCTIONS

Figure 12 shows how the functions available in workspace TSO in public library 2 can be used with the ISPEXEC function to pass a function to the PDF editor. The PUTFILE and GETFILE functions produce several messages while doing the allocations, so this example is unlikely to be used in a production environment. The method is, nevertheless, sound and can be enhanced to meet user requirements.

```
[0]   EDIT FUNC;X
[1]   X←⎕CR FUNC
[2]   ⍝ RESHAPE <X> BECAUSE <PUTFILE> NEEDS FIXED 80 CHAR RECS
[3]   (((1↑⍴X),80)⍴X) PUTFILE 'APL2.APLTEMP(OLD KEEP'
[4]   ISPEXEC 'EDIT DATASET(APL2.APLTEMP)'
[5]   ⎕FX GETFILE 'APL2.APLTEMP'
```

Figure 12. Executing the ISPF/PDF Editor from APL2

4.5.2 USING SUBROUTINES WRITTEN IN OTHER PROGRAMMING LANGUAGES

It is now possible to execute program subroutines from APL2 using the ISPF interface. Figure 13 on page 77 shows how a very simple Fortran program can be executed from APL2. Some special code is required to access the ISPF shared variable pool. The code can be put into the subroutine itself, or a special front-end could be written. Because ISPF variables have to be in character format, the maintenance of precision between APL2 and a scientific subroutine could limit the effectiveness of this technique.

Sample Fortran Program CHANGE

```
C     CHANGE THE ORDER OF A CHARACTER STRING PASSED
C     IN THE ISPF SHARED VARIABLE POOL
      INTEGER*4 ISPLNK,LASTRC
      CHARACTER*8 A,B
C     DEFINE VARIABLE AS BEING IN THE FUNCTION POOL
      LASTRC = ISPLNK ('VDEFINE','A ',A,'CHAR',8)
C     GET THE SHARED POOL VALUE
      LASTRC = ISPLNK ('VGET','A ')
      B(5:8) = A(1:4)
      B(1:4) = A(5:8)
      A = B
C     RETURN NEW VALUE TO SHARED POOL
      LASTRC = ISPLNK ('VPUT','A ')
      STOP
      END
```

Using the Fortran Program CHANGE from APL2

```
      A←'1111XXXX'
      ISPEXEC 'VPUT (A )'           ⍝ PUT VALUE IN SHARED POOL
0
      ISPEXEC 'SELECT PGM(CHANGE)' ⍝ CALL SUBROUTINE
0
      ISPEXEC 'VGET (A )'           ⍝ GET NEW VALUE FORM SHARED POOL
0
      A                             ⍝ SHOW CHARACTERS CHANGED
XXXX1111
```

Figure 13. Executing a Fortran Program from APL2

4.5.3 APL2 AS A PROTOTYPING TOOL

You can develop applications extremely quickly with APL2. As a prototyping tool, this is likely to far outweigh other possible criticisms of APL2. Using the ISPF-APL2 interface, you can develop dialogs in APL2 and, subsequently, convert them to other programming languages without the end user being aware of the conversion. The use of ISPF facilities for screen interactions, error messages, and help screens further enhances APL2 productivity and simplifies later conversion.

If you are developing ISPF dialogs, you should also consider the use of APL2 to produce working applications in the minimum time possible.

4.6 CONCLUSION

ISPF was developed as a powerful product for the development of interactive dialog applications. The support of APL2 by ISPF means that APL2 can be used in the areas where it is most beneficial.

APL2 is no longer in a world of its own. It is now an important component of an integrated set of productivity tools.

The Shared Variable Processor has been redesigned and rewritten in APL2 to provide significant new function, performance, serviceability, and extensibility. In both the CMS and TSO environments it provides for sharing of variables between APL users and between single user and multi-user auxiliary processors. In addition to allowing for a new APL2 auxiliary processor interface and data format, it also provides compatible interfaces for VSAPL and APLSV auxiliary processors.

The new facilities of the APL2 Shared Variable Processor are reviewed and a number of the salient features of its design are discussed.

5.1 INTRODUCTION

Since the Shared Variable Processor (SVP) was first defined by Lathwell in 1973 and implemented in APLSV, it has been widely misunderstood and has been implemented with substantially different definitions in various APL systems. This has lead to difficulties in migrating APL applications or auxiliary processors (APs) from one APL implementation to another. Further, because of the various misconceptions and incompatible implementations, it has been difficult to extend the underlying formal model and the resulting facilities in the APL language.

With APL2, an attempt has been made to formalize the SVP model and to rationalize its implementation as a separate system component. As a result, it has been possible to extend the SVP facilities and, at the same time, to provide compatible interfaces for APs written for APLSV and VSAPL.

5.2 APL2 SVP CHARACTERISTICS

In APL2, the Shared Variable Processor is implemented as a separate component of the system, formally interfacing with the rest of the system through the Executor. In certain environments, it is initiated and controlled independently of APL, and can be used as a general communication mechanism between independent, asynchronous processes.

Earlier SVP implementations have been entirely passive with regard to the format of data passing through shared memory. Typically, the data format has been defined by the APL interpreter to be identical to that stored internally in an APL workspace. While this approach was simple and efficient, it created problems in migrating from one APL implementation to another, and restricted modifications or extensions to the data formats used within the workspace. Further, this approach required any AP com-

municating with APL to be cognizant of the internal APL data formats and to map to and from them.

To overcome these problems and to accommodate the new data structures introduced with APL2, a new data format, the Common Data Representation (CDR), was introduced. This format is independent of the APL2 internal data format used in the workspace and is, therefore, less sensitive to change in that area. It also accommodates a superset of those data types used by APL2. In particular, most 370 data types, including packed and zoned decimal and various floating point formats, are acceptable. This means that APs written for the CDR format may be freed of many of the data conversion requirements which were previously necessary. The APL2 interpreter, in its interface to the SVP, accepts and produces data in CDR format and handles the necessary mapping and conversion to and from the internal format used within the workspace.

As the SVP provides compatible interfaces for APLSV and VSAPL APs, it must accept their data formats in addition to the new CDR format. It does this, and maps between the three formats to allow processors using dissimilar protocols to communicate with each other. For auxiliary processors which operate with the APLSV or VSAPL protocols, a new SVP call has been added to allow them to receive data in the new CDR format. Through the use of this call and with appropriate modifications, APLSV and VSAPL APs can be upgraded to accept the new APL2 data types such as complex numbers, and data structures such as nested arrays. A new return code has also been added for those APs for situations where data cannot be represented in the APLSV or VSAPL format.

In addition to the new CDR data format, the APL2 SVP also provides improved interface protocol and signalling rules for the APL2 interpreter and new auxiliary processors. These facilities provide a more consistent interface with extended function and better performance. As with data formats, the APL2 SVP also accepts the APLSV and VSAPL interface protocols and signalling rules.

The APL2 SVP that operates in the VM/CMS, MVS, and MVS/XA environments, allows for communication between processors in different address spaces or virtual machines as well as between processors in the same address space or virtual machine. As a result, the sharing of variables between APL2 users is supported as are multi-user auxiliary processors. Two new facilities, □SVS and □SVE have been added to the APL language to allow implementation of multi-user auxiliary processors written in APL. An example of a simple multi-user server has been included in Appendix D.

The ability to communicate between asynchronous processes in different address spaces or virtual machines represents an important advance. APL2 is the first general purpose programming language to offer these facilities in the VM/CMS or MVS/TSO environments. As auxiliary processors can be written in languages other than APL and as these auxiliary processors can communicate with each other, APL2 SVP also extends its facilities to processors written in other languages.

Finally, in implementing APL2 SVP, an attempt has been made to improve performance and to provide significant diagnostic facilities to debug

problems in the SVP or in processors interfacing with it. More detailed information on this subject is presented later in this section.

5.3 APL2 SVP IMPLEMENTATIONS IN VM/CMS, MVS AND MVS/XA

In each of its working environments, APL2 SVP consists of executable code and a work area called shared memory. Shared memory is used to contain tables and work areas for the SVP and to temporarily hold the values of shared variables as they pass between processors. At any given time, SVP, which operates as a subroutine of its caller, works with a single shared memory. One shared memory is allocated for each active APL user. This area is referred to as the user's "local" shared memory. To support sharing of variables between APL users (and multi-user APs), an additional area called "global" shared memory can be allocated on a system wide basis. Theoretically, many global shared memories can be allocated and used for communication between groups of APL users. No one APL user, however, can use more than a single global shared memory.

The local shared memory is located in the APL user's address space or virtual machine .It is used for communicating between the APL user and AP's located in the same address space or virtual machine. When dealing with this shared memory, the SVP holds no system locks. The APL executor provides synchronization to resolve contention between tasks attempting to access local shared memory. In the MVS/XA environment, local shared memory may be located above the line.

Global shared memory is an optional facility. To include it, it must be specified in the APL2 installation procedure and then initialized. It is initialized by starting a system task in MVS or a separate virtual machine in VM. Global shared memory is allocated in CSA in the MVS and MVS/XA environments and in a writeable, discontiguous, shared segment in VM. Therefore, it is available to all APL2 users in the system. Users contend for use of global shared memory, which is synchronized by the CMS lock in MVS and by a lock implemented in the SVP in VM.

The multi-level shared memory structure was chosen over the single global shared memory approach used by APLSV and VSPC to provide compatible support for VSAPL local APs, to simplify the design of local APs, and to provide optimal performance.

The APL user need not be aware of the multi-level shared memory structure. The SVP switches back and forth between global and local shared memory as necessary and in a fashion transparent to the APL user. Offers to specific processors are first attempted in local shared memory. If the specified processor is not signed on in the local shared memory, the offer is extended to global shared memory. General offers are always extended to global shared memory. Queries are performed against both shared memories, and all other operations are directed to the appropriate shared memory.

The APL user is identified to the global SVP by an account number which may be specified at APL2 initialization with the ID parameter. This account number may be assigned, verified, accepted, or changed by installation exits. The account number becomes the first element of ⎕AI and is used in SVP communication as the user's private library number. If the APL user does not wish to use the global SVP ,he does not specify an account number. The account number then defaults to 1001, as in VSAPL.

Processors written to the APLSV or VSAPL SVP protocols may choose to sign on to either local or global shared memory, but not both. Interface routines which facilitate this process are provided for local VSAPL APs and for global VSAPL APs in MVS only. APs that sign on globally in the MVS environment must be MVS authorized programs. Processors written to the APL2 SVP protocol may sign on to local or global shared memory or, as the interpreter does, to both simultaneously.

5.4 APL2 SVP DIAGNOSTIC FACILITIES.

APL2 SVP provides certain tracing facilities and a consistency checking option which can be helpful in diagnosing problems in SVP or in auxiliary processors.

Two types of traces are available. The first, which is available in VM/CMS as well as MVS/TSO, provides a log on the user's terminal of SVP calls, signals, and errors. This terminal tracing is requested by specifying TRACE(1) at APL2 invocation or on the)CHECK SYSTEM command; it can be terminated by issuing a)CHECK SYSTEM TRACE(-1) command.

The second type of SVP trace, which is provided for the global SVP in MVS only, uses an in memory wrap around trace table to trace SVP calls, returns, signals, and errors. It also provides the ability to patch trace calls into virtually any location in the SVP. The trace table is allocated by the system task which is run to initialize the global SVP. Tracing can be activated by modifying that task with the parameters TRACE,ON and terminated with TRACE,OFF. The trace table can be dumped to a SNAP data set by modifying the task with a SNAP parameter.

The SVP provides a consistency checking option which checks the format and consistency of shared memory at the completion of every call. This facility is enabled for local shared memory by specifying the SYSDEBUG(2) option at APL2 invocation, or on the)CHECK SYSTEM command. When consistency checking is enabled and the SVP encounters a problem in shared memory, it causes an 0C1 program check in the CSECT AP2XCSS with a code in register one that indicates the type of error found.

Detailed information on the tracing facilities, trace formats, and consistency checking error codes can be found in the APL2 Diagnosis Reference manual.

5.5 CONCLUSIONS

The APL2 Shared Variable Processor represents a significant improvement over SVPs implemented as part of previous IBM APL systems. It offers a new function, compatibility for APLSV and VSAPL APs, better performance, and improved diagnostic facilities.

The installation manual for APL2 stresses the need for thorough preparation prior to installing the product. The actual installation follows the normal SMP route: load installation jobs; allocate datasets; and RECEIVE/APPLY/ACCEPT the product. Preparation is required because of the need to decide on certain systems options, and options related to end user requirements.

The installation manual and program directory are the primary installation references. This section serves only to stress the minimum preparation requirements, and to expand on the installation steps where appropriate.

6.1 PREPARATION FOR THE INSTALLATION OF APL2

The need for preparation stems from the following considerations:

1. You must decide whether APL2 libraries and files will be supported by SAM or VSAM.

2. If SAM is to be used for these files, you must decide on a naming convention acceptable to APL2.

3. APL2 interfaces with many other IBM program products. You must know which of these are installed and how they will be used by the APL2 system.

4. APL2 itself has certain optional capabilities which are not essential to basic operation, but might well be required in your installation.

5. You must decide on default invocation options and on appropriate authority levels for access to APL2 libraries and files.

6. Some of the above decisions affect the APL2 System Options. You will probably have to change these options. Changing options is best done by receiving an SMP usermod prior to the SMP APPLY step of the APL2 installation.

6.1.1 SELECTION OF ACCESS METHODS TO BE USED BY APL2

APL2 public, project, and private libraries can be sequential or VSAM datasets. You must decide on which method will be used in your installation before attempting to install APL2 as this decision affects several of the installation jobs.

The APL2 Installation and Customization manual (SH20-9222) describes the relative merits of the two access methods. However, if the APL2 naming conventions are acceptable in your installation, the use of the sequential access method for workspaces appears to offer more advantages. This is particularly true if HSM or RACF are being used in your installation.

File libraries must be VSAM clusters. A private file library is used to store APL2 data files and to save the session manager log. The auxiliary processor, AP121, is used to read and write APL2 variables stored in an APL2 data file. Although simple APL2 applications are unlikely to use AP121, the ability to save the session manager log is of value to all APL2 users. If a private file library will be needed, a separate VSAM cluster must be defined for each user and allocated to the user before APL2 initiation.

6.1.2 NAMING CONVENTION FOR SAM LIBRARIES

The following are examples of the default file names for the various APL2 libraries:

```
PRIVATE = 'USER1.V.PRIVATE'
PROJECT = 'USER1.V0002000.PROJECT'
PUBLIC  = 'APL2.V0000001.PUBLIC'
```

Within the APL2 naming structure some flexibility is available. The default character strings 'APL2' and 'V' can be changed to suit your installation. Although the character 'V' is not very descriptive, it does allow for maximum freedom for project library numbers and is unlikely to be used as the middle qualifier for other dataset names. If 'APL2' is used instead of 'V', any dataset created by a user with APL2 as the middle qualifier appears to be an APL2 workspace.

If the basic naming structure cannot be accommodated in your installation, the User Exit must be modified to implement a standard which is acceptable.

6.1.3 OTHER IBM PRODUCTS WHICH INFLUENCE APL2 INSTALLATION

The use of any of the following products in your installation will have some bearing on the APL2 installation process:

1. GDDM: This product is required in order to use the APL2 Session Manager. GDDM provides APL2 users with powerful graphics capability using either the GPAPHPAK or the GDDM Auxiliary Processors. The use of the Session Manager influences the choice of access method (see "Selection of Access Methods to be Used by APL2" on page 85) and in-

vocation options. The use of GRAPHPAK requires special character sets (see Step 16 of the installation procedure).

2. DB2: If APL2 users will access DB2, Step 11 of the installation procedure must be run to define the APL2 Auxiliary Processor AP127 to DB2. AP127 must be included in the APL2 invocation options.

3. 3800 and DCF: Special APL2 character sets for the 3800 are distributed with APL2.

6.1.4 APL2 INSTALLATION OPTIONS FOR TSO

The default installation options have been chosen to be as general as possible. However, you will probably have to make minor changes for your installation. In particular, the following options are likely to need tailoring:

- DASD volumes which will be used to allocate datasets created by APL2 users

- Users with the authority to save workspaces in public libraries

- Default invocation options. In particular, options related to other installed products and to the amount of storage to be allocated to users.

- Descriptive data about the system which can be retrieved by any APL2 user.

The Installation Options are specified in member AP2TIOPT. This member is in one of the following datasets during APL2 installation:

- JLG1110.F3 After Receive
- SYS1.SMPSTS After Apply
- SYS1.AP2SOURC After Accept

AP2TIOPT has all the Installation Options, and contains the following logical parts:

1. Systems Options - relate mainly to dataset naming rules and are specified using the AP2TIOPT macro.

2. Default and Override Invocation Parameters - specified as constants using the DC operation code of the Assembler Language. Since default parameters can be set in the CLIST used to initiate APL2, changes to this part of the Installation Options are only essential if you wish to alter the override values.

3. Authorized Library Ranges - specified using the USERL macro. The default table does not grant authority for any user to write to the public libraries. However, the supplied User Exit Routine grants

write access to users having TSO OPERATOR status, irrespective of the values in this table. If you are installing APL2 and do not have OPERATOR status you must change this table to allow you to save the APL2 public libraries.

4. System Information - declared as Assembler constants. An APL2 user can access this information with a system command. Although this information will only be critical if APL2 functions have dependencies on other installed products, you should keep the information as accurate as possible.

The only one of the above sections which requires further explanation is the Systems Options.

The AP2TIOPT macro definition has default values specified for all possible parameters. Most, but not all, of the parameters are also specified when the macro is used in the sample AP2TIOPT. For example, the sample AP2TIOPT does not show which auxiliary processors are loaded with APL2 (parameter RESAPS). To clarify the actual defaults in effect, Figure 14 shows all parameters defined by the macro and their default values.

```
B     APLID=V,
M     ATASKS=,
B     BLKSIZE=4240,
B     CSVPID=CSVP,
I     DEFAULT=(DEFAULTA,DEFAULTZ),
B     LIBKEEP=YES,
B     LIBQLFR=APL2,
B     LIBSER=,
B     LIBUNIT=,
M     OPTUSER=AP2TIUSR,
I     OVERRIDE=(OVERRIDA,OVERRIDZ),
M     PREFIX=TOP,
B     PUBQLFR=APL2,
B     QNLT=ENGLISH,
M     QTZDEC=0,
M     QTZINT=-7,
M     RESAPS=(AP2T100,AP2T101,AP2T102,AP2T111,AP2T123,AP2T210,
              AP2X120,AP2X121,AP2X126),
M     DSECT=NO
```

The first column indicates the source of the default value -
M = Value is default provided by the macro.
I = Value in AP2TIOPT overrides macro default.
B = Override value is the same as the macro default.

Figure 14. Default AP2TIOPT Values

6.2 NOTES ON APL2 INSTALLATION STEPS

The following notes refer to, and are intended to supplement, the installation steps described in Chapter 3 of APL2 INSTALLATION AND CUSTOMIZATION UNDER TSO (SH20-9222)

6.2.1 SMP CONSIDERATIONS

STEPS 3 and 4: The sample jobs assume that a new SMP environment will be used for APL2. This is satisfactory if the installation is purely for testing purposes. If you will be applying PUT maintenance to the APL2 system in the future, you should consider updating your current SMP environment and PROCLIB members rather than using the sample jobs as distributed.

In this case, step 3 should not be run and only the DEFINE USERCAT part of step 4 should be run. The DD statements for the APL2 target and distribution should be added to the appropriate SMP PROC or defined to SMP/E if the dynamic allocation feature is being used.

6.2.2 LOADING APL2 PUBLIC WORKSPACES

STEP 15: Since this step loads the APL2 public libraries, APL2 must know whether the public libraries are to be accessed using VSAM or SAM. The CLIST tailored in Step 14 should, therefore, be used to invoke APL2. VSAM clusters are used for the public libraries if files W1 and W2 have been allocated before the APL2 command is issued.

You must also be authorized to write to these libraries. Refer to the heading Authorized Library Ranges in "APL2 Installation Options for TSO" on page 87

7.0 MIGRATION

The term 'migration' is used to describe the process by which applications running under VS APL are 'converted' to run under APL2. Migration includes the transfer of files from one environment to the other, making changes to code, testing, debugging, and bringing the application into production.

The overall message is that migration is easy, the literature is good, and aids are available. However, there are many potential 'problems' which should be checked.

In this chapter, we will:

1. Explain why migration is necessary
2. Present an overview of the process
3. Explain its different steps
4. Illustrate it with an example
5. List available publications
6. Formulate concluding remarks

7.1 WHY MIGRATION IS NECESSARY

APL2 is a new language. One criterion used in its development was compatibility with VS APL. However some of the features of APL2 do not provide complete compatibility with VS APL. That is, an application written to run under VS APL may need some modifications before it can run under APL2.

There are four possible types of incompatibility:

1. An error in VS APL produces an answer in APL2
2. An answer in VS APL is a different answer in APL2
3. An answer becomes an error
4. An error is a different error

The consequences of these incompatibilities are respectively:

1. This is not serious. It is the natural way by which languages develop. It would be a problem only if an application's logic was dependent

on an error. This is unlikely as VS APL has no facility for retaining program control after an error.

2. This is very serious. It was allowed in a small number of cases, and only when there was a very strong reason for doing so. An example is the indexing of a constant numeric vector which, in APL2, requires parentheses as bracket binding has to be stronger than vector binding. The)MCOPY command will, in many cases, insert the required parentheses.

3. This is not as serious as it seems as APL2 makes it easy to locate the exact point of an error and its cause.

4. This is not a problem for reasons similar to 1. Note, however, that it will be an important issue when considering further development of APL2 as the logic may depend on a particular error, especially as program control can be maintained after an error.

Migration consists of searching for incompatibilities and 'fixing' them in order to preserve the logic of the application.

Offending pieces of code are identified by:

- Searching for known problems
- Using the programmer's knowledge of the application
- Testing of the program producing incorrect results

7.2 OVERVIEW OF PROCESS

1. Planning - decide what will be migrated and when
2. Make some changes to applications in VS APL
3. Transfer applications from VS APL to APL2
4. Identify possible problem areas
5. Fix them
6. Test the application thoroughly
7. Repeat stages 4,5,6 until the application operates properly

Migration is then complete and the application can be put into production.

7.3 DETAIL OF PROCESS

1. In order to decide whether APL2 should completely replace VS APL or APL2 and VS APL should run concurrently, CPU availability, expected life of applications, and interdependence of applications and their data should be considered. In other words, it is not necessary to migrate all existing applications.

   Plan for migration by reading the literature and attempt to anticipate areas likely to need attention.

   APL2 training is necessary.

2. APL2 uses EBCDIC rather than z-codes (the interchange code unique to APL).

   This means that ⎕AV is different. The)MCOPY command (see 3) makes allowance for this difference. That is,)MCOPY ensures that the character vector 'JULIE' remains 'JULIE' in APL2 despite the different ⎕AV. There are two possible situations where the "MCOPY command is not sufficient to overcome the differences in ⎕AV. The two situations are:

   a. The entry in ⎕AV in VS APL has no corresponding entry in ⎕AV in APL2.)MCOPY would fail as an attempt is being made to use an illegal character. Illegal characters have to be identified within VS APL.

      The TRANSFER workspace (supplied with APL2 in public library 2) contains a function BADCHARS. This must be put into the VS APL workspace to be migrated. This can be done using the Session Manager. The instruction BADCHARS ⎕NL 3 will examine all functions in the workspace for illegal characters. If any illegal characters are found, they must be altered before migration can proceed.

   b. The logic of the application depends on a character's position in ⎕AV In this case, the functions CHARIND and INDCHAR in the TRANSFER workspace must be used (instead of)MCOPY) to transfer the concerned character data to APL2. The use of these functions is explained on pages 7 and 8 of the Migration Guide.

3. The transfer from VS APL to APL2 can now take place.

   In APL2 enter:

```
)CLEAR

)MCOPY [libno] wsname

)SAVE [libno] wsname
```

The contents of the specified VS APL workspace will be copied into a saved workspace under APL2. As the VS APL workspace remains intact, the same name may be used, if desired.

The VS APL workspaces to be transferred must reside on the same CMS or MVS system as APL2. If not, the VS APL workspaces must first be brought into the same system as APL2. The appropriate installation reference manual gives details on how to do this.

)MCOPY ensures that the correct ⎕AV is used and makes a number of other adjustments to functions and variables. These will often be sufficient to allow use of the workspace in APL2. If the workspace in APL2 still does not operate properly, additional fixes are required.

4. Once the application has been transferred, an iterative process of testing and debugging begins. This often will be relatively simple as APL2 has been designed with compatibility in mind.

   APL2 comes supplied with a workspace TRANSFER, (in public library 2), which has helpful functions and advice. Two useful functions are FLAG_ and FIX_. FLAG_ searches functions in the workspace (including those from TRANSFER if you are not careful) for given character strings. FIX_ replaces the given character strings by others, supplied by the user.

   The variable FLAGMVSAPL_ in the workspace TRANSFER provides a list of strings that are known to be likely areas of concern (for example ⎕NC is included because under certain circumstances it can return different answers under VS APL and APL2).

   After possible problem areas have been identified, each possible problem should be examined closely to determine if it is indeed a problem.

   Pages 26-32 of the Migration Guide are helpful here as they contain explanations of the circumstances under which differing results will occur.

   FLAG_ identifies pieces of code that may produce a different result. It does not necessarily follow that a different result will be produced. The logic of each particular application must be examined in conjunction with the literature to determine if there is any circumstance under which an incorrect result would arise in that application.

   Having determined that code needs to be amended, the function FIX_ may be used. The right argument of FIX_ specifies the functions to be edited. The left argument of FIX_ is a vector. Each item of the vector specifies two character strings. For each item, FIX_ replaces all occurrences of the first string, by the second string, in all the functions specified. Thus, it should be used with caution. The use of FIX_ should be reserved for 'fixing' widespread problems occurring several times in a workspace.

Remember that FIX_ and FLAG_ are tools and not panaceas.

Even though an application appears to work, it may not be fully migrated. All paths through the code and all possible arguments to the functions should be checked. It is the extreme cases (for example, when an array is empty) that most likely cause the problems.

Also, flagging an item does not mean it must be changed. Flagging merely states that you are using a piece of code that, under some circumstances, could produce a different result. Your use of the piece of code may be unaltered.

The process of identification, testing and fixing continues until the application behaves exactly as it should under all possible circumstances.

At this point migration is complete and the application may be put into production under APL2 and discontinued under VS APL.

7.4 EXAMPLE

This example illustrates the mechanics of migrating a workspace from VS APL to APL2.

There is no such thing as a typical migration as, by nature, it is the unusual and unlikely events that cause the problems.

This example was chosen because it is likely to be reasonably familiar to most people and does actually require some function editing (many applications do not).

Ignore for a moment the fact that it is unlikely that a lesson in using VS APL would be migrated to APL2 (or is it?).

1. In VS APL, load LESSON3

   Spend a few moments familiarizing yourself with the application if necessary.

2. A copy of the function BADCHARS from workspace 2 TRANSFER in APL2 is needed in our workspace. Invoke APL2 with the Session Manager on.

3. Display BADCHARS.

4. Return to VS APL, load LESSON3.

   Define the function BADCHARS by

   - Entering function definition mode
   - Scrolling back through the Session Manager log

- Entering each line of BADCHARS by altering one character (for example, blank to blank)

There is a bug in BADCHARS which prevents it from examining the first function in the list given by ⎕NL 3

The required fix is:

insert line [1.5] I←¯1

edit existing line [2] ⎕IO←0

Save the workspace with a new name.

5. Type BADCHARS ⎕NL 3 .This will examine all functions in the workspace for illegal characters.

   There should not be any.

   Check that all functions have been inspected by using)FNS

6. Return to APL2.

7. Enter the following commands:

)CLEAR

)MCOPY wsid

)SAVE wsid

   If any error messages result, use pages 12-13 of the Migration Guide to investigate.

   Bring in a copy of the TRANSFER workspace by entering)PCOPY 2 TRANSFER

8. Display and read DESCRIBE.

9. Enter FLAGMVSAPL_ FLAG_ ALL_.

   Use pages 253-254 of APL Programming: An Introduction to APL2 (SH20-9229) to decide on your own right arguments to FLAG_, and try them.

10. Use Chapter 3 of the Migration Guide together with the Language Manual to establish the significance, if any, of all items flagged.

11. Make any necessary code changes.

12. Test the migrated workspace by typing:

)SAVE wsid

```
)LOAD wsid
```

You should get a DOMAIN ERROR

13. Use pages 27,38 and 39 of the Migration Guide to identify the cause of the problem.

14. Edit the offending piece of code.

15. Save and load the workspace again.

16. Test the workspace.

17. Easy, is it not?

7.5 PUBLICATIONS

APL2 Migration Guide, SH20-9215, is the primary source for all information and guidance on migration. It is advisable to study the TRANSFER workspace itself before using the workspace.

APL2 Programming: Language Reference, SH20-9227

APL2 Installation and Customization under CMS, SH20-9221

APL2 Installation and Customization under TSO, SH20-9222

7.6 CONCLUDING REMARKS

The experiences of customers involved in the Early Support Program were similar to what we expected. That is, that)MCOPY is easy to use and successful in most cases. Further, that the literature is good, but that "badly written" APL can cause problems.

Migration of the vast majority of applications will be relatively easy.

Features of VS APL that are incompatible with APL2 are either taken care of by)MCOPY or are rather 'perverse', that is, unlikely to arise in an actual application. Pieces of code that will fail are the sort of thing that relied on the 'internals' of APL rather than the language itself. For example, some ADI functions relied on the fact that in VS APL a pendent function cannot be expunged. This is not the case in APL2.

Beware of the 'clever' programmer who does things in a non-standard manner.

Remember that compatibility with VS APL was one of the criteria by which all possible features of APL2 were evaluated by the designers.

Remember, too, that the options to not migrate, or to rewrite applications to take advantage of the enhanced features of APL2, are always present.

The objective of the designers of APL2 was to produce a powerful, productive language which would provide a rich set of tools for problem solving.

The criteria against which APL2 is measured are compatibility, formality, simplicity, and usability. (For a more complete discussion of these criteria see "Design Criteria" on page 1)

APL2 was not explicitly designed with CPU performance in mind. However, once APL2 itself had been designed, each primitive function was written in Assembler code by experienced systems programmers. Most of the primitives have been optimized for many special cases and some APL 'idioms' have been coded directly. That is, APL2 will 'recognize' an idiom and process it with code dedicated to that idiom. For example ⍴⍴ is recognized as rank and does not use the code for ⍴ twice.

APL2 derives its powerful performance from its ability to handle arrays as easily as scalars.

8.1 QUESTIONS A USER MIGHT ASK.

There are, perhaps, two valid questions on performance that a user might ask:

1. If I simply migrate an existing VS APL application to APL2 (that is, I make no use of the APL2 language enhancements), how will the performance compare?,

2. if I develop a new application in APL2, will it perform better than if I had developed it in VS APL?

The answer to the first question is "probably similar".

The answer to the second question is "probably better with APL2".

It is difficult to commit to anything much more precise than this.

However a number of simple tests have been carried out.

The results are summarized below.

8.2 TIMING

Functions were timed in both VS APL and APL2 by using ⎕AI[2] - which keeps a cumulative record of CPU time used in an APL session. The method used, the results obtained, and the conclusions to keep in mind are presented below.

8.2.1 METHOD

Care must be taken to make valid use of ⎕AI[2], to ensure that what is being timed is what was intended to be timed.

Accessing and displaying ⎕AI[2] will use CPU time and, hence, alter its own value.

Putting a trace or a stop on a function in order to time individual statements will greatly increase the CPU time since the trace or stop uses CPU time.

Functions that display values run much slower than functions that only assign values to variables. That is, time to display can be a very significant proportion of total time and can lead to misleading comparisons.

Timing of very fast functions or primitives can lead to unreliable results as ⎕AI[2] is not sensitive enough to record low values accurately.

Care must also be taken when ⎕AI[2] "goes round the clock" and is reset to zero while the function being timed is running.

A function was written which, basically, accesses ⎕AI[2] immediately prior to, and immediately subsequent to, executing the character string supplied by the right argument. This is repeated a number of times and average, minimum, and maximum data are calculated.

Each function was timed frequently so that the data displayed stability.

Any outlying values were investigated.

8.2.2 RESULTS

As primitive functions run very fast, meaningful results can only be obtained when using reasonably large arrays.

Two random arrays of size 50×50 and two of size 200×200 were generated in VS APL. Fourteen simple operations such as addition, subtraction, rotation, comparison, selection, and catenation were timed on both sizes. The arrays were then transferred to an APL2 workspace by the)MCOPY com-

mand and the same operations were timed again. APL2 was faster (typically by 25%-30% for the 50×50s and by 35%-40% for the 200×200s) in eleven of the operations. APL2 immeasurably outperformed VS APL in performing ⍎ and VS APL outperformed APL2 with ↑ and catenate.

To use smaller arrays, more complex functions were tested. Some performed better in APL2 and others were faster in VS APL. In particular, a function which generated arrays and then did various sorts and conversions performed better in VS APL (up to 40% improvement) on arrays of all sizes up to 50×50. With larger arrays APL2 again outperformed VS APL with the margin increasing as the amount of data increased.

In attempting to analyse which types of function were better in each, the most significant factor seemed to be that functions with many loops and "GO TOs" (that is, written Fortran style) were better in VS APL.

The function SORTALF (available in workspace UTILITY in public library 1) was timed. It performed better in VS APL on all arrays of size less than 100×100. SORTALF does an alphabetic sort on character arrays. It is a function which is present in one form or another in numerous real world applications. APL2 has overcome the necessity of using functions like SORTALF. The GRADE functions (⍋,⍒) have been greatly enhanced and alphabetic sorting is now a primitive function. This means that performance is improved enormously. GRADE was used to do the same sorts as SORTALF. The improvements in performance ranged from about 25% with small arrays to over 3000% with large arrays.

8.2.3 CONCLUSIONS

The facts that emerge are:

- No major difference between VS APL and APL2 in a majority of situations. Most real world applications will comprise a variety of functions some of which will run faster in APL2 and some slower in APL2.
- APL2 performs significantly better than VS APL in handling large arrays.
- The extensions of APL2 permit applications to be written in less lines of code.
- The extensions to the primitive functions reduce the need for user-written functions. This has very significant implications for performance.

8.3 SUMMARY

The major conclusion is that users generally need have no worries about performance when migrating to APL2.

The message to the user is that APL is not so much about CPU productivity as it is about end-user productivity and in reducing application development time. APL2 improves upon VS APL in these areas.

The design of APL2 has made it even more suited to rapid prototyping and development of applications. There is now an even wider range of powerful primitive functions that perform well. These primitives can be combined in very few lines of code to produce applications that are shorter than their equivalents in other languages and, more importantly, take less time to code. As APL is interactive, each line of code can be tested as it is written. This, combined with the extensive new features for handling errors, makes debugging, maintenance, and development easier, more accurate, and quicker. In addition, the lifting of restrictions on data types makes representation of data structures more logical and more natural, which can reduce time spent on analysis and program design.

The poor performance of Fortran style functions should not be a major problem. Hopefully, there are few such functions. Such functions either should have been written correctly originally or are trying to do something for which APL is not appropriate. APL has always suffered from programmers who have learned a "traditional" language first and then not changed their ways of thinking. It is almost as if APL2 is exacting vengeance on such people!

APL is not generally used for huge applications running numerous times per day. It is used most frequently in a personal computing environment where the time to develop an application is of prime importance. Many such applications may only be run once or twice.

APL's strength lies in the speed at which applications can be developed. This should not be construed as implying that APL's performance is poor. Far from it, its ability to handle arrays and its variety of primitives see to that.

What we are saying is that performance is not, or should not be, a major consideration because the APL environment is one in which quick, effective applications can be prototyped and developed swiftly.

Those are precisely APL's strengths.

A.1 SQLSYSTEM

```
     ∇SQLSYSTEM[□]∇
[0]   SQLSYSTEM;S;MAT;D
[1]   S←'SELECT TNAME,CREATOR,TABLETYPE,NCOLS,REMARKS,DBSPACENO,DBSPACENAME'
[2]   MAT←2⊃SQL S,' FROM SYSTEM.SYSCATALOG WHERE CREATOR=''SYSTEM'''
[3]   □PW←256
[4]   □ES(~2=127 □SVO 'DAT')/'DATA NOT OFFERED'
[5]   D←DAT_ 'DESCRIBE' 'APL2'
[6]   →(0≠⍴Z←MESSAGE↑D)/0
[7]   D←¯1 0↓2⊃D
[8]   (D,[1]' '),[1]MAT
```

A.2 SQLTAB

```
     ∇SQLTAB[□]∇
[0]   SQLTAB;S
[1]   S←'SELECT TNAME,CREATOR,TABLETYPE,NCOLS,REMARKS,DBSPACENO,DBSPACENAME'
[2]   MAT←2⊃SQL S,' FROM SYSTEM.SYSCATALOG WHERE CREATOR=USER'
[3]   □PW←256
[4]   □ES(~2=127 □SVO 'DAT')/'DATA NOT OFFERED'
[5]   D←DAT_ 'DESCRIBE' 'APL2'
[6]   →(0≠⍴Z←MESSAGE↑D)/0
[7]   D←¯1 0↓2⊃D
[8]   (D,[1]' '),[1]MAT
```

A.3 SQLCOLNAME

```
     ∇SQLCOLNAME[□]∇
[0]   SQLCOLNAME A;S1;S2
[1]   S1←'SELECT COLNO,CNAME,COLTYPE,LENGTH '
[2]   S2←'FROM SYSTEM.SYSCOLUMNS WHERE TNAME=''',A,''' ORDER BY COLNO'
[3]   TITLE SQL S1,S2

     ∇TITLE[□]∇
[0]   Z←TITLE MAT;D;□PW
[1]  ⍝ADD COLUMN NAMES AND DESCRIPTOR TO R
[2]  ⍝ MAT IS A MATRIX RESULT FOR CURSOR 'APL2
[3]   □PW←256
[4]   □ES(~2=127 □SVO 'DAT')/'DAT NOT OFFERED'
[5]   D←DAT_ 'DESCRIBE' 'APL2'
[6]   →(0≠⍴Z←MESSAGE↑D)/0
```

```
[7]   D←¯1 0↓D←2⊃D
[8]   Z←(D,[1]' '),[1]2⊃MAT
```

A.4 SQLDISP

```
     ∇SQLDISP[□]∇
[0]   SQLDISP S1
[1]   TITLE SQL 'SELECT * FROM ',S1
```

A.5 REPORT

```
      ∇REPORT[□]∇
[ 0]   Z←A REPORT R;V;C;MAT;B1;OP;BB;ZZ;CO;XX;RES;I;RE;N
[ 1]   R←2⊃R
[ 2]   R←R[SORTSEQ⍋(⍒,[⍳0]R[;±1↑A]);]
[ 3]   C←V∘.≡V←⊂[2],[⍳0]R[;±1↑A]
[ 4]   BB←1+⌽2≤/1,⌽B1←+/[1]C×MAT←∨\((⍴V),(⍴V))⍴1,(⍴V)⍴0
[ 5]   OP←((⊂A[3 4 5])≡"('MAX' 'MIN' 'SUM' 'AVG'))/'⌈L++'
[ 6]   I←∈(1(1 ¯1))[BB]
[ 7]   ZZ←I/[1]R
[ 8]   CO←R[;±¯1↑A]
[ 9]   XX←⊂[1]C×(CO∘.×(⍴CO)⍴1)
[10]   XX←FGX"XX
[11]   RES←(B1=1)/XX
[12]   RE←±OP,'/" RES'
[13]   ±(A[3 4 5]≡'AVG')/' RE←RE+⍴"RES '
[14]   N←(,⊃0=1↑"0⍴"R[1;])/(⍳1↓⍴R)
[15]   ((I=¯1)/[1]ZZ[;N])←⊂⍳0
[16]   ((I=¯1)/[1]ZZ[;±¯1↑A])←RE
[17]   ((I=¯1)/[1]ZZ[;±1↑A])←⊂'RESULT'
[18]   □ES(~2=127 □SVO 'DAT')/'DAT NOT OFFERED'
[19]   Z←((¯1 0↓(2⊃DAT_ 'DESCRIBE' 'APL2')),[1]' '),[1]ZZ

      ∇FGX[□]∇
[0]   Z←FGX V
[1]   Z←(V≠0)/V
```

A.6 SQLICU

```
      ∇SQLICU[□]∇
[ 0]   A SQLICU R;CHRCTL;CTL;DATCTL;HEADING;KEYS;LABELS;X;Y;□IOCHART;□IO
[ 1]  ⍝R IS THE RESULT TABLE OF A SQL QUERY.A SPECIFIES WHICH
[ 2]  ⍝COLUMNS OF THE RESULT ARRAY ARE PLOTTED BY SQLICU.
[ 3]   □IO←1
[ 4]   KEYS←,¯1 0↓2⊃DAT_ 'DESCRIBE' 'APL2'
```

```
[ 5]   KEYS←KEYS[A←⍋A]
[ 6]   KEYS←((⌈/⍴⍤KEYS)↓4)↑⍤KEYS
[ 7]   ⎕SVR 'DAT'
[ 8]   ⎕IO←1
[ 9]   R←R[;A]
[10]   Y←0 1↓R
[11]   X←,⍳1↑⍴Y
[12]   LABELS←R[;1]
[13]   LABELS←(,(⌈/⍴⍤LABELS))↑⍤LABELS
[14]   HEADING←'AVERAGE PRICE FOR THE WINE BY TYPE'
[15]   →(2≠1↑R←126 ⎕SVO 2 3⍴'CTLDAT')/0
[16]   CHRTCTL←76⍴' '          ⍝   INIT CHART CTL TO BLANKS
[17]   CHRTCTL[⍳4]←4 ⎕IO 0     ⍝   LEVEL 0
[18]   CHRTCTL[4+⍳4]←4 ⎕IO 2   ⍝   DISPLAY 1
[19]   CHRTCTL[8+⍳4]←4 ⎕IO 0   ⍝   HELP 0
[20]   CHRTCTL[12+⍳4]←4 ⎕IO 0 ⍝   ISOLATE 0
[21]   CHRTCTL[17]←'*'         ⍝   FORMNAME '*'
[22]   CHRTCTL[25]←'*'         ⍝   DATANAME '*'
[23]   CHRTCTL[32+⍳4]←4 ⎕IO 0  ⍝   PAIRING 0
[24]   CHRTCTL[36+⍳4]←4 ⎕IO 1↓⍴Y ⍝   NUMBER OF DATA GROUP
[25]   CHRTCTL[40+⍳4]←4 ⎕IO 1↑⍴Y ⍝   NUMBER OF ELEMENT BY DATA GROUP
[26]   CHRTCTL[44+⍳4]←4 ⎕IO 4  ⍝   LENGTH OF EACH STRING  IN KEYS
[27]   CHRTCTL[48+⍳4]←4 ⎕IO⊃(1↑⍴⍤LABELS) ⍝  LENGTH OF EACH STRING IN LABELS
[28]   CHRTCTL[52+⍳4]←4 ⎕IO⍴HEADING  ⍝   HEADINGL 7
[29]   CHRTCTL[57]←'*'         ⍝   PRINTNAME
[30]   LABELS←∈LABELS
[31]  ⍝ PRTDEP 0 PRTWID 80 PRTCOPY 2
[32]   CHRTCTL[64+⍳12]←,4 ⎕IO 0 80 2
[33]   DATACTL←0⍴0 ⍝ NO DATACTL DATA
[34]   KEYS←∈1↓KEYS
[35]   DAT←CHRTCTL,KEYS,LABELS,HEADING
[36]   Y←,Y
[37]   CTL←¯10,(⍴CHRTCTL),(⍴DATACTL),DATACTL,(⍴X),X,(⍴Y),Y,(⍴KEYS),(⍴LABELS),(⍴
       HEADING)
[38]   DAT←CHRTCTL,KEYS,LABELS,HEADING
[39]   R←CTL
[40]   ⎕SVR 'DAT'

       ∇SQLX[⎕]∇
[0]    Z←SQLX SQL_STMT;ZZ;⎕IO
[1]    ⎕IO←1
[2]    ZZ←SQL SQL_STMT
[3]    E←1⊃ZZ
[4]    ⍋(0∨.≠↑ZZ)/'MESSAGE E'
[5]    Z←2⊃ZZ
[6]    ⎕ES(2≠⍴⍴Z)/'RESULT TABLE EMPTY'
```

```
%----------------------  ISPF/PDF PRIMARY OPTION MENU  -------------------------
%OPTION  ===>_ZCMD                                                            +
%                                                          +USERID    - &ZUSER
%   0 +ISPF PARMS  - Specify terminal and user parameters   +TIME      - &ZTIME
%   1 +BROWSE      - Display source data or output listings +TERMINAL - &ZTERM
%   2 +EDIT        - Create or change source data           +PF KEYS   - &ZKEYS
%   3 +UTILITIES   - Perform utility functions
%   4 +FOREGROUND  - Invoke language processors in foreground
%   5 +BATCH       - Submit job for language processing
%   6 +COMMAND     - Enter TSO command or CLIST
%   7 +DIALOG TEST - Perform dialog testing
%   8 +LM UTILITIES- Perform library management utility functions
%   C +CHANGES     - Display summary of changes for this release
%   T +TUTORIAL    - Display information about ISPF/PDF
%   A +APL2        - Execute APL2 with default options
%   X +EXIT        - Terminate ISPF using log and list defaults
%
+Enter%END+command to terminate ISPF.
%
)INIT
  .HELP = ISR00003
  &ZPRIM = YES         /* ALWAYS A PRIMARY OPTION MENU      */
  &ZHTOP = ISR00003    /* TUTORIAL TABLE OF CONTENTS        */
  &ZHINDEX = ISR91000 /* TUTORIAL INDEX - 1ST PAGE          */
  VPUT (ZHTOP,ZHINDEX) PROFILE
)PROC
  &ZSEL = TRANS( TRUNC (&ZCMD,'.')
                0,'PANEL(ISPOPTA)'
                1,'PGM(ISRBRO) PARM(ISRBRO01)'
                2,'PGM(ISREDIT) PARM(P,ISREDM01)'
                3,'PANEL(ISRUTIL)'
                4,'PANEL(ISRFPA)'
                5,'PGM(ISRJB1) PARM(ISRJPA) NOCHECK'
                6,'PGM(ISRPTC)'
                7,'PGM(ISRYXDR) NOCHECK'
                8,'PANEL(ISRLPRIM)'
                C,'PGM(ISPTUTOR) PARM(ISR00005)'
                T,'PGM(ISPTUTOR) PARM(ISR00000)'
                A,'CMD(%APL2)'
              ' ',' '
                X,'EXIT'
                *,'?' )
  &ZTRAIL = .TRAIL
)END
```

Figure 15. PDF Menu Altered to Include APL2

```
PROC 0
CONTROL MAIN NOFLUSH NOPROMPT NOMSG NOLIST NOCONLIST
ALLOC FILE(F0) SHR DA('&SYSUID.TSOUSER.FILES')
/*  ALLOCATE THE SYMBOL SET LIBRARIES.                                  */
ALLOC FI(ADMSYMBL) DA('APL2.SYMBLIB') SHR
/*  ALLOCATE THE FILE(LOADLIB), IF NEEDED AND IF NOT SPECIFIED IN       */
/*  THE LOADLIB INVOCATION PARAMETER.                                   */
ALLOC FILE(LOADLIB) SHR DA( +
             'ISP.V2R1M2.ISPLOAD')
/*  ALLOCATE THE FILE(APLDUMP).                                         */
ALLOC FILE(APLDUMP) SYSOUT(T)
/*  SPECIFY INVOCATION PARAMETER VALUES IN THE APL2 COMMAND             */
/*  DEPENDING UPON THE DEFAULT INVOCATION PARAMETERS IN THE SYSTEM      */
/*  OPTIONS MODULE, AP2TIOPT.                                           */
ISPEXEC SELECT CMD( -
APL2 AP(ISPAPAUX) FREE(100K) CODE(32791) SM(OFF) -
     INPUT(')LOAD 1 ISPFWS') -
      ) LANG(APL)
/*  FREE COPY WORKFILES.                                                */
FREE FI(F0)
/*  FREE SYMBOL SET LIBRARY(S).                                         */
FREE FI(ADMSYMBL)
/*  FREE SPECIAL LOAD LIBRARY(S).                                       */
FREE FI(LOADLIB)
/*  FREE APL2 DUMP DATA SET.                                            */
FREE FI(APLDUMP)
```

Figure 16. APL2 CLIST Executed from PDF Menu

```
      ∇EDITER[□]∇
[ 0]   EDITER FONC;TEMP;RC;CMS;□IO;CTLD;DATD;I;NC;REC;NCC
[ 1]  ⍝
[ 2]  ⍝  EDIT AN APL FUNCTION  WITH  XEDIT
[ 3]  ⍝
[ 4]  ⍝ DO NOT FORGET THE FOLLOWING LINES IN AN EXEC PROCEDURE
[ 5]  ⍝  APLEDIT EXEC
[ 6]  ⍝ &CONTROL OFF
[ 7]  ⍝ &STACK LRECL a
[ 8]  ⍝ &STACK VER 1 a
[ 9]  ⍝ &STACK TRUNC a
[10]  ⍝ &STACK ZONE 1 a
[11]  ⍝ X script APLEDIT A (WIDTH a
[12]  ⍝
[13]   □IO←1
[14]   CMS←'CMS'
[15]   RC←100 □SVO 'CMS'
[16]   CMS←'ERASE ',FONC,' APLEDIT'
[17]   DATD←FONC,' APLEDIT A ( FIX U 192'
[18]   CTLD←FONC,' APLEDIT A ( CTL'
[19]   RC←110 □SVO 2 4⍴'CTLDDATD'
[20]   RC←1↑⍴TEMP←□CR FONC
[21]   I←0
[22]  L3:→(RC<I←I+1)/L4
[23]   DATD←TEMP[I;]
[24]   →L3
[25]  L4:RC←□EX 2 4⍴'CTLDDATD'
[26]   CMS←'EXEC APLEDIT ',FONC,' 255'
[27]   DATD←FONC,' APLEDIT A ( 192 FIX'
[28]   CTLD←FONC,' APLEDIT A ( CTL'
[29]   RC←110 □SVO 2 4⍴'DATDCTLD'
[30]   RC←¯1+1↑2↓DATD
[31]   TEMP←(RC,NCC←⍴TEMP)⍴TEMP←DATD
[32]   I←1
[33]  L8:→(RC<I←I+1)↑L9
[34]   →(NCC≥NC←⍴REC←DATD)↑L10
[35]   TEMP←(RC,NCC←NC)↑TEMP
[36]  L10:TEMP[I;]←NCC↑REC
[37]   →L8
[38]  L9:□FX TEMP
[39]  ⍝  OPTION LIGNE SUIVANTE ERASE DU FICHIER EDITE
[40]  ⍝ CMS←'ERASE ',FONC,' APLEDIT A'
[41]   CMS←'ERASE ',FONC,' APLEDIT A'
```

Figure 17. APL2 Function to Call Xedit

The following APL functions implement a simple server which will accept offers from one or more APL users via the global SVP. The various functions can be expanded as noted to include the processing logic for a specific application.

It should be noted that the server does not set access control when a variable is shared. This must be done by the user offering the variable or the server will not be signalled on references or specifications of the shared variable.

D.1 CLEANUP

```
      ∇CLEANUP[□]∇

[0]   CLEANUP
[1]   ⍝ FUNCTION TO HANDLE HOUSEKEEPING AND/OR STATISTICS.
[2]   ⍝
[3]   ⍝ This function is called at the completion of each wait interval.
[4]   ⍝
[5]    (6↑□TS)'- ACTIVE SHARES:'(↑⍴'⍙' □NL 2)'OFFER SEQUENCE NO:' OFFNO
```

D.2 PROCESS

```
      ∇PROCESS[□]∇

[0]   PROCESS VAR;VALUE
[1]   ⍝ PROCESS A NAMED SHARED VARIABLE
[2]   ⍝
[3]   ⍝ This function can be extended to handle the unique processing
[4]   ⍝ requirements of a specific application.  Minimally, it must
[5]   ⍝ specify the variable named by the argument, causing its state
[6]   ⍝ to change.
[7]   ⍝
[8]    VALUE←⍎VAR                ⍝ get the shared variable value
[9]    ⍎VAR,'←□TS'               ⍝ specify the shared variable
[10]  VAR VALUE
```

D.3 RETRACT

```
      ∇RETRACT[□]∇

[0]   RETRACT VAR
[1]   ⍝ RETRACT AND EXPUNGE A NAMED SHARED VARIABLE
```

```
[2]  ⍝
[3]  ⍝ This function can be extended to handle the uniques requirements
[4]  ⍝ of a specific application.  Minimally, it must expunge the named
[5]  ⍝ variable.
[6]  ⍝
[7]   →0⍴⎕EX VAR
```

D.4 SERVER

```
      ∇SERVER[⎕]∇

[0]   SERVER INTERVAL;OFFNO;PROCS;RETRACTS;SETS;VARS
[1]  ⍝ GENERAL PURPOSE SERVER FOR APL AUXILIARY PROCESSORS
[2]  ⍝
[3]  ⍝ This function acts as a dispatcher for an auxiliary processor
[4]  ⍝ written in APL.  It waits for shared variable events, reciprocates
[5]  ⍝ offers made to it, calls a function to process work when the partner
[6]  ⍝ specifies a shared variable and retracts variables when the degree
[7]  ⍝ of coupling drops below 2.
[8]  ⍝
[9]  ⍝ The functions SHARE, PROCESS and RETRACT are called as required to
[10] ⍝ handle shared variable events.  These functions may be extended
[11] ⍝ to handle the unique requirements of a specific application.
[12] ⍝
[13] ⍝ This function takes an argument INTERVAL which specified a maximum
[14] ⍝ wait time.  Each time this interval expires, the function CLEANUP
[15] ⍝ is called.  This function may also be extended to handle the unique
[16] ⍝ requirements of a specific application.
[17] ⍝
[18] ⍝ The semi-global variable OFFNO is an offer number used to attempt
[19] ⍝ to ensure a unique shared variable name.
[20] ⍝
[21]  →SETSVE,⍴⎕SVE,⎕SVE←OFFNO←0                ⍝ clear prior events
[22] RESET: CLEANUP                              ⍝ called at end of interval
[23] SETSVE:⎕SVE←INTERVAL                        ⍝ prepare to wait
[24] RUN:→(0≠⍴PROCS←⎕SVQ⍳0)/OFFER                ⍝ check for offers
[25]  →(∨/SETS←(⎕SVS VARS←'⍙' ⎕NL 2)∧.=0 1 0 1)/SET   ⍝ check state
[26]  →(∨/RETRACTS←2≠⎕SVO VARS)/EXPUNGE          ⍝ check for retraction
[27]  →(0=⎕SVE)↓RUN RESET                        ⍝ wait for an event
[28] ⍝
[29] OFFER:SHARE⍤⍤PROCS                          ⍝ couple any offers
[30]  →RUN
[31] ⍝
[32] SET:PROCESS⍤⍤⊂[⎕IO+1]SETS⌿VARS              ⍝ process specified variables
[33]  →RUN
[34] ⍝
[35] EXPUNGE:RETRACT⍤⍤⊂[⎕IO+1]RETRACTS⌿VARS      ⍝ retract as necessary
[36]  →RUN
```

D.5 SHARE

```
      ∇SHARE[□]∇

[0]   SHARE PROC;MY_NAMES;HIS_NAMES
[1]   ⍝  SHARES VARAIBLES WITH A SPECIFIED PROCESSOR
[2]   ⍝
[3]   ⍝  Variable names offered by the specified processor
[4]   ⍝  are prefixed with ⍙, the processor id and _
[5]   ⍝  and then suffixed with _ and OFFNO to create
[6]   ⍝  unique names which are then used to share.
[7]   ⍝
[8]   HIS_NAMES←□SVQ PROC
[9]   MY_NAMES←(⊃[2](↑⍴HIS_NAMES)⍴⊂'⍙',(⍕PROC),'_'),HIS_NAMES
[10]  MY_NAMES←MY_NAMES,'_','000000000'⍕,[□IO+.5]OFFNO+⍳↑⍴MY_NAMES
[11]  →0⍴PROC □SVO MY_NAMES,' ',HIS_NAMES
[12]  OFFNO←OFFNO+↑⍴MY_NAMES          ⍝ update offer sequence number
```

□

A

B

C

D

U

V

W

READER'S
COMMENT
FORM

AN OVERVIEW OF APL2

GG24-1627

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or the IBM branch office serving your locality.

Possible topics for comments are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name, company, mailing address and date:

What is your occupation ? ----------

Reader's Comment Form

Cut or Fold Along Line

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y.

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM International Systems Center
Department H52, Building 930
P.O. Box 390
Poughkeepsie, New York 12602
U.S.A.

Fold and tape | Please Do Not Staple | Fold and tape

IBM®

READER'S
COMMENT
FORM

AN OVERVIEW OF APL2

GG24-1627

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or the IBM branch office serving your locality.

Possible topics for comments are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name, company, mailing address and date:

What is your occupation ? ----------

Cut or Fold Along Line

Reader's Comment Form

Fold and tape | **Please Do Not Staple** | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM International Systems Center
Department H52, Building 930
P.O. Box 390
Poughkeepsie, New York 12602
U.S.A.

Fold and tape | **Please Do Not Staple** | Fold and tape

IBM®

READER'S
COMMENT
FORM

AN OVERVIEW OF APL2

GG24-1627

You may use this form to communicate your comments about this publication, its organization, or subject matter, with the understanding that IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Your comments will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

Note: Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or the IBM branch office serving your locality.

Possible topics for comments are:

Clarity Accuracy Completeness Organization Coding Retrieval Legibility

If you wish a reply, give your name, company, mailing address and date:

What is your occupation ? ----------

Cut or Fold Along Line

Reader's Comment Form

Fold and tape | **Please Do Not Staple** | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, N.Y

POSTAGE WILL BE PAID BY ADDRESSEE:

IBM International Systems Center
Department H52, Building 930
P.O. Box 390
Poughkeepsie, New York 12602
U.S.A.

Fold and tape | **Please Do Not Staple** | Fold and tape

IBM®

GG24-1627-0

An Overview of APL2

GG24-1627-0

PRINTED IN THE U.S.A.