Alice in Packageland

Brent Hawks
IBM APL2 Development
555 Bailey Avenue
San Jose, CA 95141
408/463-3588

Installation Code: IBM
Project: APL
Session: D601

Abstract

APL2 can take you to a marvelous new place called "packageland." Alice fell into a package (rabbit?) hole and is confused about what packages are, what to package, how to do it, and how to use it. We'll go in after Alice, and find a fascinating world of new ways to make APL2 more productive.

Once Upon a Time...

APL was originally confined to working only with objects within the workspace. When APL2 V1 R2 came along you could then access objects in Assembler, FORTRAN, and REXX. Now, with "the then current release" of APL2 you can add APL2 to that list.

I must be dreaming...

During the FIP (field introduction program) we had one customer give us a fantastic report after using packages. Several APL2 Release 2 workspaces were converted into APL2 Release 3 packages. The customer reported exciting improvements in elapsed time, CPU time, and programmer productivity. These gains were attributed to sharing code among multiple users, the ability to eliminate function files (resulting in simpler code), and the need to only maintain a single copy of a packaged workspace.

What is a package?

A package is simply another form of a *saved* workspace. Using `)SAVE` you store the active workspace out onto disk. The PACKAGE function takes the saved workspace and makes an object deck out of it. You can then use the linkage editor to make an object module out of it. However, it is still just a saved workspace, we've just put it into a wrapper that the operating system can understand.

You can get at objects in the packaged workspace with `⎕NA`. For example, `3 11 ⎕NA 'TIME'` will get you the TIME function from the TIME packaged workspace. If you do `)FNS` you'll see *TIME*; however, all you really have in the active workspace is a pointer to the TIME function in the package (if you try `⎕CR 'TIME'`, the result will be an empty matrix). The function itself is still in the package. The package, if not already in memory, will be loaded when you do the first `⎕NA` to the package.

A simple packaging example.

It's quite simple to package a workspace. For example, we can quickly package the DISPLAY workspace:

```
      )CLEAR
CLEAR WS

                                  ⍝ PACKAGE A SAVED WORKSPACE
      3 11 ⎕NA 'PACKAGE'
1
      PACKAGE 'DISPLAY V0000001'          ⍝ PACKAGE IT
DISPLAY TEXT A
      )HOST LKED DISPLAY (LIBE PKGLIB     ⍝ LINK EDIT IT
. CMS(0)

                                ⍝ NOW ⎕NA TO THE PACKAGE
      )CLEAR
CLEAR WS
      'PKGLIB.DISPLAY' 11 ⎕NA 'DISPLAY'
1
      'PKGLIB.DISPLAY' 11 ⎕NA 'ΔLX ⎕LX'
1
      DISPLAY ΔLX                   ⍝ USE FN AND VAR FROM PACKAGE
.→-----.
|COIBM|
'------'
      A←'ABCDE'                     ⍝ VAR FROM THE ACTIVE WORKSPACE
      DISPLAY A                     ⍝ USE FN FROM PACKAGE
.→-----.
|ABCDE|
'------'
```

You would normally "clean" the workspace before packaging it. You clean it by)*COPY*ing it into a clear workspace, resetting the system variables, and then resaving it.

The example is from CMS (you can tell because the result of the PACKAGE function is a TEXT deck). It turns out that in CMS, you can use a TEXT deck without doing the linkedit, but it is *NOT* recommended.

In the example we find out what ⎕*LX* in the DISPLAY workspace is. Doing)*NMS* in the workspace would give you Δ*LX*.2 *A*.2 *DISPLAY*.3. However, only *A* actually exists in the workspace, the others are pointers into the packaged DISPLAY workspace.

You'll notice that we could not do ⎕*NA* to ⎕*LX* directly. A quad name may only be accessed via ⎕*NA* as a surrogate name. Also notice, that *DISPLAY* and Δ*LX* *act* as if they were in the active workspace.

Why Use Packages?

There are several advantages to using packages:

- Name Isolation
- Run Only Applications
- Shared Code

Name Isolation. Many applications that expect to share a workspace must go to extraordinary lengths to keep from having name conflicts. For example, look at the names of things in the PRINTWS workspace; I tried to print out a workspace that contained a function called HOST: it turns out that PRINTWS also has a function called HOST, so mine got obliterated. If PRINTWS were in a package, then it could use its own HOST function *and* print mine at the same time.

Run Only Applications. It's easier to hide API from a user, when using packages. It is also possible to make decommenting and unmeaningful names an automatic part of the packaging process.

Shared Code. There are a number of advantages to sharing code among multiple users and/or workspaces.

Maintenance
Performance
    Eliminate Function Files
    Reduce storage requirements

The customer noted above attributed the gains to the use of shared code. The performance was due to the elimination of function files, and the productivity came from reduced maintenance costs and ease of implementation (versus function files). Please understand, the package is still APL and runs at the same speed as APL in your active workspace. The performance improvements are due to things such as reduced paging (multiple users share *one* copy of the package), less logic required than for function files, less I/O, etc.

What should I package?

Obviously, anything that you want to share (amongst users or workspaces) is a candidate for packaging. Things that you wish to hide from the active workspace (maybe to avoid clutter or name conflicts) or utility functions that require frequent maintenance; "end user" applications are also good candidates for packaging.

Inside of Packages

Now we'll talk a little about name isolation versus localization, changes in a package name scope, and moving around between name scopes.

Name Isolation

Name isolation is NOT the same as localization! Within the active workspace is a name table that points to the objects in the workspace. The packaged workspace also has a name table that points objects within the packaged workspace. When you switch execution from the active workspace to the packaged workspace, you work from the name table in the packaged workspace.

This means that if you have a function FN that you have accessed via `⎕NA` then the name table in the active WS (workspace) will have an entry for FN that points to the name table in the package which actually points to FN in the packaged WS.

Now suppose that we have variables A and B in the active WS and variable B in the packaged WS. Now when we execute FN, we switch name tables and FN will only see the B that exists in the package. An attempt to reference something called A will result in a VALUE ERROR because A exists only in the active WS. If FN changes B, it will change only within the context of the package, the B that's in the active WS will remain untouched.

Changes in a packaged WS

As you may have noticed above, FN can CHANGE things in the package. However, the package may be in non-writable memory, and anyway, I may not want other users of the package to see the change. So, what do I do?

It so happens that when a package is first accessed, its name table is copied into the active workspace. It still points to objects in the package, but now I can make changes to the name table and they affect only my active WS, not someone else's. Now if anything in the package changes (or is created), then the new/changed version is put into the active workspace and pointed to by the package's name table (not the active WS's name table), so it acts as if the package had changed.

If I `)SAVE` the active WS after having made changes in the package, the changes will still be there when I re-`)LOAD` the workspace.

One point of interest. In the CURRENT implementation of packages, Any variable that gets referenced (not just new/changed) will get copied into the active WS. Functions are copied only if new or changed.

Packages can call other packages

We already know about *⎕NA* as a means for accessing a package from the active WS. It can also be used to access a package from a package, or to access the active WS from a package. There is an external function *EXP* that can be used to access the *previous* name scope. Consult the System Services Reference and the Using Supplied Routines manuals for more information about the syntax of *⎕NA* and *EXP*.

Stopped in a package

Sooner or later, something will go wrong and you will get a SYNTAX ERROR in some function that's in a packaged workspace. As is usual in APL2, execution will stop inside the function at the point of the error. It is *IMPORTANT* to remember that APL2 is now using the name table from the package and not the one from the active WS. So, if you do *)NMS* you will see the names of objects *in the package*! You can edit things, create/change/delete things, run things; in short you can do what you normally do in APL2 - you're just doing it using the name table of the package. *)RESET* will get you back to the active WS. Any changes that were made in the package name scope will remain. If you subsequently *)SAVE* the workspace, the changes will still be there when the workspace is again *)LOAD*ed.

Packages can make things easier (an example)

Suppose that you have two versions of the same workspace and you would like to know what's changed. Without packages it's rather difficult, you have to get around name conflicts somehow.

In the appendix you can see a sample workspace that will compare the objects in two workspaces and tell you which ones are different. It will even work on itself! The COMPARE workspace only has about four simple functions that do the actual comparisons. The rest of the workspace is documentation, some code to allow packaging to work in CMS or TSO, and some code that allows COMPARE to be packaged.

It works by packaging the two workspaces to be compared and then accesses the objects in the two packaged workspaces with *⎕NA*. Name conflicts are avoided by using surrogate names. It will even compare the system variables.

Other interesting tidbits

If you're going to be serious about packaging you'll also want to know about the optional left argument to the PACKAGE function. It's a list of names that are accessible (via *⎕NA*) from outside of the package. If the namelist is missing or empty, then the default is to allow all names to be accessed (including system "quad" names). The external function EXP, because it is reaching back to the previous name scope, ignores this name list.

If you do use a name list on a package, you should allow some room for debugging. If you make the namelist too restrictive, you may find it difficult to isolate a problem.

When using the PACKAGE function in TSO you will need to allocate a SYSPUNCH data set. This is where the PACKAGE function will put its object deck. On both CMS and TSO, you will do well to get the OS/VS Linkage Editor manuals.

Finally, if you access a package that is NOT shared (i.e. in a DCSS in CMS or the LPA in TSO) then it will be loaded into your free space. That means, that your memory requirements may change. For example, you may need to use a smaller workspace and a larger free space.

Summary

Packages are simply APL2 and they do just what you want them to do. Name scopes may be a little confusing (just like the first time you saw indirection or recursion), but are very powerful.

Remember that one customer reported major improvements in elapsed time, CPU time, and programmer productivity when APL2 Release 2 workspaces were converted to APL2 Release 3 packages. The customer attributed the gains to: sharing code among multiple users, the ability to eliminate function files, simplified logic, and the need to maintain only a single copy of a packaged workspace.

Packages allow name scope *isolation*. You have the ability to put out end-user "run only" applications. Users can now *SHARE* the same copy of APL2 code (without function files) for the first time in APL history. Finally, you may find it a great deal easier to maintain utilities functions as packages. You can maintain a single copy of the utility and all workspaces will use the same code.

Appendix

```
                    COMPARE WORKSPACE

                (C) COPYRIGHT IBM CORP. 1987


   ABSTRACT
.+-------------------------------------------------------.
|COMPARE the objects in two workspaces for differences.|
'-------------------------------------------------------'

   CMP_LOAD
.+---------------.
| .+--. .+----. |
| |PKG| |.LOAD| |
| '---' '-----' |
'∈--------------'

   CMP_OBJ
.+---------------.
| .+--. .+---. |
| |PKG| |.OBJ| |
| '---' '----' |
'∈--------------'

   CMPLIB
.+------.
|PKGLIB|
'------'

   DESCRIBE
.+-------------------------------------------------------------.
|The primary purpose of this workspaces is for COMPARING the  |
|objects in two workspaces (most of which will be identical)  |
|in order to see any differences.                             |
|                                                             |
|This workspace should normally be  )LOADed rather than  )COPYed |
|because it may create objects in the workspace which could cause|
|name conflicts and/or loss of data.                          |
|                                                             |
|***** NOTE *****                                             |
|COMPARE creates data sets that may OVERWRITE existing data sets |
|(see warnings in HOW and in PKG).                            |
'-------------------------------------------------------------'

   HOW
.+-------------------------------------------------------------.
|                        ***** COMPARE *****                  |
|To COMPARE two workspaces, the syntax is:                    |
|                                                             |
|      'wsid_a' COMPARE 'wsid_b'                              |
|
```

```
If the workspaces have objects of the same name that do not compare
then it will create a variable F_name or V_name (F for functions or
operators and V for variables) that is a two row matrix of the
form:  F_name←2 1⍴(⎕CR wsid_a.name)(⎕CR wsid_b.name).
For variables, it will just be the value of the variable.
Note also, that for functions/operators, the value for: 2⎕AT name
will be appended as the last line of the ⎕CR.

Objects which are in only one of the WSs are referred to as
ORPHANS; you'll see a list of names (if any) for each WS.

The system variables (except:  ⎕A1 ⎕LC ⎕TS ⎕WA) are also compared.
Any differences are reported and a variable is created with a
prefix of QD_ (for example: QD_10 for ⎕10).  These are the same
as above:  two element variables containing the value from each WS.

                        ***** SHOWDIF *****
The function SHOWDIF will try to show the differences between
two objects in one of the created variables.  For example:
          Z←SHOWDIF F_FN
will show the lines from the two versions of FN that don't match.
The lines from FN in the left workspace will be the first item
of the result.  SHOWDIF will return the differences side by side
if they will display within ⎕PW (a two element vector), otherwise
it will leave the result as a two row matrix.

An optional left arg on SHOWDIF will cause SHOWDIF to decomment
its argument before the comparison.  Only the existence of
a left argument is checked; its content is irrelevant.


   ***** WARNING ***** WARNING ***** WARNING ***** WARNING *****

The PKG function in this workspace will create data sets that may
OVERWRITE existing data sets.  See the PKG function for details.
```

```
[0]   COIBM
[1]   '(C) COPYRIGHT IBM CORP, 1987'

[0]   L COMPARE R;T;VA;VB;VAB;FA;FB;FAB
[1]  ⍝ GLOBALS: L_WS R_WS ΔNLA ΔNLB ΔCRA ΔCRB ΔATA ΔATB EXP
[2]  ⍝ FIRST PACKAGE THEM AND ⎕NA TO THE ⎕NL, ⎕CR, AND ⎕AT OF EACH
[3]   'Packaging WS''S...'
[4]   →(∨/0=⍷PKG¨(L_WS R_WS)←(⍷¨L R)~¨' ')/0     ⍝ QUIT IF CAN'T PKG
[5]   →0⍴L R ΔNA¨'ΔNLA ⎕NL' 'ΔNLB ⎕NL'           ⍝ GET ⎕NL FROM L R
[6]   →0⍴L R ΔNA¨'ΔCRA ⎕CR' 'ΔCRB ⎕CR'           ⍝   "  ⎕CR  "   " "
[7]   →0⍴L R ΔNA¨'ΔATA ⎕AT' 'ΔATB ⎕AT'           ⍝   "  ⎕AT  "   " "
[8]   →0⍴3 11 ⎕NA 'EXP'                          ⍝ IN CASE I'M A PACKAGE
[9]   (L R)←⍪¨L R       ⍝ TO MAKE DISPLAY LINE UP NICELY
[10] ⍝              DO OPERATORS AND FUNCTIONS
[11]  '..... Comparing Functions/Operators .....'
[12]  T←(FA←(⊂[2]ΔNLA 3 4)~¨' ')∘.≡⍤1 0(FB←(⊂[2]ΔNLB 3 4)~¨' ')
[13]  FAB←(∨/T)/FA          ⍝ COMMON FNS/OPS
[14]  FA←(~∨/T)/FA          ⍝ FNS/OPS FOUND ONLY IN WS_A
[15]  FB←(~∨⌿T)/FB          ⍝ FNS/OPS FOUND ONLY IN WS_B
[16]  ⍞(0<⍴FA)/''''Orphan Fns/Ops in '',L,'': '',⍕FA'
[17]  ⍞(0<⍴FB)/''''Orphan Fns/Ops in '',R,'': '',⍕FB'
[18]  COMPARE_FNS¨FAB       ⍝ SEE IF THEY ARE THE SAME
[19] ⍝              DO VARIABLES
[20]  '..... Comparing Variables .....'
[21]  COMPARE_SYS           ⍝ COMPARE SYSTEM VARS FIRST
[22]  T←(VA←(⊂[2]ΔNLA 2)~¨' ')∘.≡⍤1 0(VB←(⊂[2]ΔNLB 2)~¨' ')
[23]  VAB←(∨/T)/VA          ⍝ COMMON VARIABLES
[24]  VA←(~∨/T)/VA          ⍝ VARS FOUND ONLY IN WS_A
[25]  VB←(~∨⌿T)/VB          ⍝ VARS FOUND ONLY IN WS_B
[26]  ⍞(0<⍴VA)/''''Orphan Variables in '',L,'': '',⍕VA'
[27]  ⍞(0<⍴VB)/''''Orphan Variables in '',R,'': '',⍕VB'
[28]  COMPARE_VAR¨VAB       ⍝ SEE IF THEY ARE THE SAME
```

```
[29] ⍝              CLEANUP
[30]  →0⍴⎕PKGDELETE L R
[31]  →0⍴⎕EX¨'L_WS' 'R_WS' 'ΔNLA' 'ΔNLB' 'ΔCRA' 'ΔCRB' 'ΔATA' 'ΔATB' 'FXP'

[0]   COMPARE_FNS R;A;B
[1]  ⍝ SEE IF COMMON FNS/OPS ARE IDENTICAL, IF NOT, PUT THEM INTO CALLER
[2]   A←ΔCRA R←(∈R)~' '
[3]   B←ΔCRB R
[4]   →(A≡B)/0          ⍝ THEY MATCH
[5]  ⍝ ELSE - CREATE THEM IN THE CALLER'S NAMESCOPE
[6]   '  Mis-match in Common Fn/Op: ',R
[7]   A←⊃(⊂[2]A),(⊂'⍝    ∇',⍕⎕TS ΔATA R)       ⍝ PUT TIME-STAMP AT THE END
[8]   B←⊃(⊂[2]B),(⊂'⍝    ∇',⍕⎕TS ΔATB R)
[9]   →0⍴EXP('F_',R)'←'(2 1⍴A B)

[0]   COMPARE_SYS;ΔECA;ΔECB;SYSVARS;A;B
[1]  ⍝ SEE IF SYSTEM VARS ARE SAME (EXCEPT:  ⎕AI ⎕LC ⎕TS ⎕WA)
[2]   SYSVARS←'⎕AV⎕CT⎕EM⎕ET⎕FC⎕IO⎕LX⎕ML⎕NLT⎕PP⎕PW⎕PR⎕RL⎕RS⎕SVE⎕TC⎕TZ⎕UL'
[3]   SYSVARS←(1+SYSVARS='⎕')⊂SYSVARS
[4]   →0⍴L_WS ΔNA 'ΔECA ⎕EC'
[5]   →0⍴R_WS ΔNA 'ΔECB ⎕EC'
[6]   A←ΔECA¨SYSVARS
[7]   B←ΔECB¨SYSVARS
[8]   →(A≡B)/0          ⍝ THEY ALL MATCH
[9]  ⍝ ELSE SOME ARE DIFFERENT
[10]  (SYSVARS A B)←(~∈A≡B)/¨SYSVARS A B
[11]  'System Vars that don''t match:',⍕SYSVARS
[12] LOOP:→(0=⍴SYSVARS)/0
[13]  →0⍴EXP('QD_',(1↓∈↓SYSVARS)~' ')'←'(2 1⍴3⊃¨↓¨A B)
[14]  (SYSVARS A B)←1↓¨SYSVARS A B
[15]  →LOOP

[0]   COMPARE_VAR R;A;B
[1]  ⍝ SEE IF COMMON VARS ARE IDENTICAL, IF NOT, PUT THEM INTO CALLER
[2]   →0⍴L_WS ΔNA 'A ',R←(∈R)~' '
[3]   →0⍴R_WS ΔNA 'B ',R
[4]   →(A≡B)/0          ⍝ THEY MATCH
[5]  ⍝ ELSE - CREATE THEM IN THE CALLER'S NAMESCOPE
[6]   '  Mis-match in Common Var: ',R
[7]   →0⍴EXP('V_',R)'←'(2 1⍴A B)

[0]   Z←L SHOWDIF R;A;B;C;⎕IO
[1]  ⍝ SHOW DIFFERENCES BETWEEN THE TWO (SIDE BY SIDE IF CAN)
[2]   ⎕IO←1
[3]   L←2=⎕NC 'L'                  ⍝ LEFT ARG PRESENT
[4]   C←⍴¨R←⍕¨,R                   ⍝ ENSURE THAT EACH PART OF R
[5]   R←(¯2↑¨1,¨C)⍴¨R              ⍝    IS A MATRIX
[6]   (A B)←⊂[2]¨R                 ⍝ SEPARATE AND FORMAT INTO SIMPLE VEC OF VECS
[7]   ⍝L/'A←(+/¨∧\¨''⍝''≠A)↓¨A'    ⍝ ELIMINATE COMMENTS IF LEFT ARG
[8]   ⍝L/'B←(+/¨∧\¨''⍝''≠B)↓¨B'
[9]   C←(A←DLT¨A)∘.≡__(B←DLT¨B)     ⍝ COMPARE THEM (W/O LEAD/TRAIL BLANKS)
[10]  Z←⊃(~∨/C)/A                  ⍝ LINES IN A THAT DIDN'T COMPARE
[11]  Z←Z(⊃(~∨⌿C)/B)               ⍝ LINES IN B THAT DIDN'T COMPARE
[12]  ⍝(⎕PW<¯1+⍴⍕Z)/'Z←2 1⍴Z'      ⍝ SHOW SIDE BY SIDE, ELSE 2 1 MATRIX

[0]   Z←DLT R
[1]  ⍝ DELETE LEADING/TRAILING BLANKS
[2]   Z←((∨\Z)∧⌽∨\⌽Z←R≠' ')/R

[0]   Z←L ΔNA R
[1]  ⍝ PROVIDES ⎕NA COMPATIBLY IN CMS AND TSO
[2]   Z←(('TSO'≡__∈HOST)/CMPLIB,'.'),L)11 ⎕NA R

[0]   R←PKG WS;CMD;PACKAGE;CTL;DAT;XA;T
[1]  ⍝ CONVERTS WS TO A PACKAGED WORKSPACE
[2]  ⍝ TSO LEAVES SYSPUNCH (CMP_OBJ) AND CMPLIB (CMP_LOAD) ALLOCATED
[3]  ⍝ NAMING CONVENTION USED FOR WORKSPACES IS:  V.wsname
[4]  ⍝ CMPLIB IS A SIMPLE CHAR VEC (I.E. CMPLIB←'PKGLIB')
[5]  ⍝     IT IS THE FIlename OF THE LOAD LIBRARY
[6]  ⍝ CMP_OBJ IS A TWO ELEMENT VEC OF VECS (I.E. CMP_OBJ←'PKG' '.OBJ')
[7]  ⍝     'SYSPUNCH' WILL BE ALLOCATED TO CMP_OBJ
```

```
[8]   ⍝     IT WILL CONTAIN THE OBJECT DECK CREATED BY "PACKAGE"
[9]   ⍝ CMP_LOAD IS A TWO ELEMENT VEC OF VECS (I.E. CMP_LOAD←'PKG' '.LOAD')
[10]  ⍝     CMPLIB WILL BE ALLOCATED TO CMP_LOAD
[11]  ⍝     ITS MEMBERS WILL BE THE PACKAGED WORKSPACES
[12]  ⍝ *** NOTE ***
[13]  ⍝                     CMS
[14]  ⍝  "PACKAGE" CREATES A FILE:  wsname TEXT A.  A PRE-EXISTING FILE OF
[15]  ⍝  THAT NAME WILL BE OVERWRITTEN.  "PKGDELETE" WILL ERASE THIS FILE.
[16]  ⍝
[17]  ⍝                     TSO
[18]  ⍝     THIS FUNCTION ONLY CHECKS TO SEE THAT 'SYSPUNCH' AND CMPLIB
[19]  ⍝     ARE ALLOCATED, IT DOES NOT VERIFY THAT THEY ARE ALLOCATED
[20]  ⍝     TO CMP_OBJ AND CMP_LOAD RESPECTIVELY.
[21]  ⍝
[22]  ⍝     THIS FUNCTION DOES NOT PROTECT ANY PRE-EXISTING DATA IN
[23]  ⍝     CMP_OBJ AND CMP_LOAD.  HENCE, YOU MUST PROTECT AGAINST DESTROYING
[24]  ⍝     DATA SETS.  ADDITIONALLY, THE FUNCTION "PKGDELETE" DELETES
[25]  ⍝     THESE DATA SETS.
[26]   →('TSO'≡__∈HOST)/TSO
[27]   →(1=R←~3 11 ⎕NA 'PACKAGE')/ERROR
[28]   →(0=⍴R←PACKAGE WS,' APLWSV2')/ERROR   ⍝ PACKAGE THE WORKSPACE
[29]   →R←0                                  ⍝ EXIT
[30]  ⍝
[31]  TSO:→0⍴102 ⎕SVO 2 3⍴'CTLDAT'           ⍝ SEE IF WE ARE IN MVS OR MVS/XA
[32]   CTL←0                                 ⍝ CHECK THE WORKSPACE ADDRESS
[33]   XA←(16×102⍬×102⍬)≤DAT                 ⍝ IN XA IF WORKSPACE ABOVE THE LINE
[34]   →0⍴⎕EX 2 3⍴'CTLDAT'
[35]   →0⍴100 ⎕SVO 'CMD'
[36]   CMD←'APL DDI SYSPUNCH'                ⍝ SEE IF SYSPUNCH ALLOCATED
[37]   →(~8=__↑CMD)/L2                       ⍝ BRANCH IF ALLOCATED
[38]   CMD←'APL DSI ',∈CMP_OBJ               ⍝ SEE IF DATA SET EXISTS
[39]   →(~8=__↑CMD)/L1                       ⍝ BRANCH IF IT EXISTS
[40]  ⍝ ALLOC NEW DATA SET
[41]   T←'ALLOC FI(SYSPUNCH) DSN(',∈CMP_OBJ,') NEW SPACE(5,5) '
[42]   CMD←T,'TRACKS LRECL(80) BLKSIZE(3120) RECFM(F B) DSORG(PS)'
[43]   →(0=R←CMD)/L2
[44]   →ERROR
[45]  L1:CMD←'ALLOC FI(SYSPUNCH) DSN(',∈CMP_OBJ,') SHR'   ⍝ ALLOC OLD DATA SET
[46]   →(0≠R←CMD)/ERROR
[47]  L2:→(1=R←~3 11 ⎕NA 'PACKAGE')/ERROR
[48]   →(0=⍴R←PACKAGE 'V.',WS)/ERROR         ⍝ PACKAGE THE WORKSPACE
[49]   CMD←'APL DDI ',CMPLIB                 ⍝ SEE IF CMPLIB ALLOCATED
[50]   →(~8=__↑CMD)/L4                       ⍝ BRANCH IF ALLOCATED
[51]   CMD←'APL DSI ',∈CMP_LOAD              ⍝ SEE IF DATA SET EXISTS
[52]   →(~8=__↑CMD)/L3                       ⍝ BRANCH IF IT EXISTS
[53]  ⍝ ALLOC NEW DATA SET
[54]   T←'ALLOC FI(',CMPLIB,') DSN(',∈CMP_LOAD,') NEW SPACE(5,5) '
[55]   CMD←T,'TRACKS DIR(2) BLKSIZE(4096) RECFM(U) DSORG(PO)'
[56]   →(0=R←CMD)/L4
[57]   →ERROR
[58]  L3:CMD←'ALLOC FI(',CMPLIB,') DSN(',∈CMP_LOAD,') SHR'  ⍝ ALLOC OLD DATA SET
[59]   →(0≠R←CMD)/ERROR
[60]  L4:                                    ⍝ OKAY, NOW LINKEDIT THE MESS
[61]   T←'LINK ',(↑CMP_OBJ),' LOAD(',(↑CMP_LOAD)
[62]   CMD←T,'(',WS,')) NOTERM NOPRINT ',XA/'RMODE(ANY)'
[63]   →(0=R←CMD)/0
[64]  ⍝
[65]  ERROR:'PKG ERROR ',⍕R

[0]   Z←PKGDELETE NAMES;CHK;CMD;R
[1]   ⍝ Z←0 IF EVERYTHING WAS PROPERLY DELETED OR NEVER THERE
[2]   ⍝ IN CMS, DELETES TEXT DECKS SPECIFIED BY NAMES VECTOR
[3]   ⍝ IN TSO, FORCES PROCESSOR 11 TO CLOSE THE OPEN LOADLIB
[4]   ⍝ THEN DEALLOCATES AND DELETES SYSPUNCH AND CMPLIB
[5]    →0⍴⎕EX 'Z←CHK C' 'CMD←C' 'Z←CMD'      ⍝ FOR READABILITY
[6]    Z←~2=__100 ⎕SVO 'CMD'
[7]    →('CMS'≡__∈HOST)/CMS
[8]    R←'OTHER.OTHER' 11 ⎕NA 'OTHER'
[9]    Z←Z+~∨/0 12=CHK 'FREE FI(SYSPUNCH)'
[10]   Z←Z+~∨/0 12=CHK 'FREE FI(',CMPLIB,')'
[11]   Z←Z+~∨/0 8=CHK 'DELETE ',∈CMP_OBJ
```

```
[12]  Z←Z+~∨/0 8=CHK 'DELETE ',∈CMP_LOAD
[13]  →0
[14] CMS:⍎(2>=__NAMES)/'NAMES←⊂,NAMES'
[15] L1:Z←Z+~∨/0 28=CHK 'ERASE ',(↑NAMES),' TEXT A'
[16]  →(0≠⍴NAMES←1↓NAMES)/L1

[0]   R←HOST;CTL;DAT                  ⍝ OBTAIN HOST ID VIA AP102.
[1]   ⍝  WARNING: THE CONTENTS OF THIS FUNCTION ARE SUBJECT TO CHANGE
[2]   ⍝           BETWEEN APL2 RELEASES.
[3]   ⍝
[4]   ⍝  RESULT: NORMAL  - ENCLOSED CHARACTER VECTOR OF HOST NAME
[5]   ⍝          ERROR   - DEGREE OF COUPLING OR AP102 RETURN CODE
[6]   ⍝
[7]   ⍝  NOTE: (DAT↑240) = (X'F0' IN WS) = (HOST INFO IN APL2 1.1.00)
[8]   ⍝
[9]   ⎕SVE←5                          ⍝ SET EVENT TIMER TO 5 SECONDS.
[10] L1:→(2^.=R←102 ⎕SVO 2 3⍴'CTLDAT')/L2     ⍝ → IF SHARE WORKS.
[11]  →(0≠⎕SVE)/L1                    ⍝ → IF EVENT OCCURS.
[12]  →0                              ⍝ RETURN WITH DEGREE OF COUPLING.
[13] L2:→(0≠R←CTL)/CTL←⎕SVE←0         ⍝ → IF AP102 ERROR.
[14]  CTL←1,(DAT↑240),4               ⍝ GET HOST SYSTEM DATA.
[15]  R←,(8⍴2)⊤DAT                    ⍝ CONVERT LOW-ORDER BYTE TO BITS
[16]  R←R/'TSO' 'CMS' '?32' '?16' '?8' '?4' '?2' '?1'


⍝ A SAMPLE RUN COMPARING TWO VERSIONS OF COMPARE

      )LOAD COMPARE
SAVED 1987-11-04 14.20.07 (GMT-8)
(C) COPYRIGHT IBM CORP. 1987

      'COMPARE' COMPARE 'COMPB'
Packaging WS'S...
..... Comparing Functions/Operators .....
Orphan Fns/Ops in COMPARE :  COIBM DLT HOST SHOWDIF
Orphan Fns/Ops in COMPB   :  COMARE_FNS HOWUSE SAVWS SYSTEM
  Mis-match in Common Fn/Op: ΔNA
  Mis-match in Common Fn/Op: COMPARE
  Mis-match in Common Fn/Op: COMPARE_FNS
  Mis-match in Common Fn/Op: COMPARE_SYS
  Mis-match in Common Fn/Op: COMPARE_VAR
  Mis-match in Common Fn/Op: PKG
  Mis-match in Common Fn/Op: PKGDELETE
..... Comparing Variables .....
System Vars that don't match: ⎕LX
Orphan Variables in COMPARE :  ABSTRACT CMP_LOAD CMP_OBJ
Orphan Variables in COMPB   :  WSID

⍝ SHOW BOTH VERSIONS OF "ΔNA" FUNCTION
      F_ΔNA
 Z←L ΔNA R
 ⍝ PROVIDES ⎕NA COMPATIBLY IN CMS AND TSO
 Z←(('TSO'=__∈HOST)/CMPLIB,'.'),L)11 ⎕NA R
 ⍝   ∇1987 11 3 10 16 24 502

 Z←L ΔNA R
 ⍝ PROVIDES ⎕NA COMPATIBLY IN CMS AND TSO
 Z←(('TSO  '=__SYSTEM)/'PKGLIB.'),L)11 ⎕NA R
 ⍝   ∇1987 3 13 10 5 31 0

⍝ SHOW BOTH VERSIONS OF "⎕LX" SYSTEM VARIABLE
      QD_LX                   ⍝ THE ONE FROM "COMPB" IS EMPTY
 COIBM
```