Packaged Workspaces

April 7th, 1988

Michael T. Wheatley

IBM
APL Development
General Products Division
Santa Teresa Laboratory
San Jose, California, USA

Packaged Workspaces

Abstract

Packaged workspaces are a new facility in APL 2 Release 3 and provide an alternative form in which applications can be provided to users. Packaged workspaces allow APL application code to be shared between APL 2 users and provide name scope isolation between portions of APL applications.

This paper describes packaged workspaces, their uses, benefits and pitfalls.

Introduction

The concept of a workspace, a working area in the computer's memory, has been important in every APL implementation since APL\360. A workspace is a very practical thing:

- it is the area of memory which the APL interpreter uses to to obtain and store values, to hold function definitions, and for temporary storage during execution of APL primitives;
- it defines a set of names and associated definitions or values which comprise an APL application, or a set of related APL functions;
- it may be stored on a computer's external medium such as a disk drive or magnetic tape, and is then referred to as a "saved workspace";
- when brought into the computer's memory for processing it is referred to as an "active workspace".

In 1966, when APL\360 was introduced, a 32K workspace was typical and considered adequate. Ten years later, typical workspaces had grown to about 128K; five years after that, 500K and 1MB workspaces were common, and today APL 2 supports 128MB workspaces.

As workspaces and the applications in them grew, the problems involved in managing named objects in the workspace became more and more serious. It is not uncommon to find applications today that consist of thousands of functions and variables. Since the name of each function and variable must be unique at

any given instant in a workspace, large applications become complex, if nothing else, in terms of "name pollution". It is often difficult to extend a large application, or to perform maintenance on it, without introducing errors because of the large number of named objects in the workspace.

Other programming languages have effectively dealt with this problem of name pollution by allowing program segmentation, and the creation of applications by linking separately compiled modules together. Each module need only be known by a few external names or entry points, even though it may contain a large number of internal names. Some similar capabilities are provided by the localization of names in APL functions. But APL functions are often tiny in comparison to the compilation units in other languages, and the sheer number of APL functions in a large application often causes name pollution.

The workspace provides another mechanism for segmentation of large APL applications. By structuring the application as a series of workspaces, the number of objects in each workspace can be substantially reduced. This works well in some applications where the application can be broken down into relatively large phases, with each phase implemented by a separate workspace. For more dynamic applications, however, where there is not a predictable pattern in the use of the functions, the cost of saving the current workspace, loading a new one, and most importantly, transferring data between workspaces, is very high.

Packaged workspaces are designed to provide an answer to this problem of name pollution, an answer that allows large applications to be effectively and simply segmented into manageable components.

As the use of APL for large production applications grows, another problem becomes apparent: each user of the application requires his own copy of the application code in his active workspace. As the number of simultaneous users grows, the requirement for real memory to support them also grows. The situation becomes worse as users begin to contend for real memory in the computer complex and the paging subsystem begins to thrash.

Packaged workspaces also attempt to address this problem by allowing APL application code to be shared between simultaneous users of an application. Only one copy of the application need reside in the computer memory; all users can share it on a read-only basis.

Packaged workspaces offer a number of other benefits in many situations:

- they facilitate the process of central maintenance, because users do not, and often cannot, save their own copy of the application code or utility functions;
- they provide a greater level of control for the application developer by allowing limits to be set on the entry points to applications;
- in some cases substantial performance improvements can be attained for shared applications through the elimination of function files;

- they can provide a greater degree of security than is offered by saved workspaces.

Description of Packaged Workspaces

The following discussion is intended to be illustrative, and does not contain complete tutorial or reference material. For reference material concerning these facilities, see Chapter 24 in the "API 2 Programming: System Services Reference" manual (SH20-9218).

What is a packaged workspace?

Saved workspaces are collections of API names, values and definitions that are stored on a computer's external medium (such as a disk drive). Saved workspaces are created with the API)*SAVE* command and are accessed with the API)*LOAD* and)*COPY* commands.

An API user can have only one "active workspace" (one that has been activated by a)*LOAD* or)*CLEAR* command) at a time. Through the use of AP 101 (the stack processor) it is possible to activate a new saved workspace under program control. But there is never more than one such active workspace for an API user at a given instant in time.

A packaged workspace is simply a reformatted saved workspace. It contains the same information as a saved workspace. It is reformatted so that it can be handled by the same operating system facilities that locate and load compiled program modules. Packaged workspaces are accessed through Processor 11 which uses these operating system facilities to dynamically load or locate packaged workspaces.

Packaged workspaces and objects within them can be accessed dynamically by API applications through the use of □*NA* and Processor 11. Multiple packaged workspaces can be available and in use at the same time in an API user's address space.

How are packaged workspaces created?

Packaged workspaces are created from saved workspaces. A saved workspace can be transformed into a packaged workspace by using the external function *PACKAGE* which is provided with API 2. The *PACKAGE* function reads a saved workspace from disk, converts it to the format of an object module, and stores that object module in another disk file.

The object module produced by the *PACKAGE* function has the same format as object modules produced by compilers such as Fortran or COBOL. The *PACKAGE* function does not compile a saved workspace, rather it simply converts its format to that of an object module. Object modules, whether they are produced by the *PACKAGE* function or by a compiler, look the same to the operating system and can be processed by other operating system facilities such as a linkage editor or loader.

In most situations, the object module file created by the *PACKAGE* function must be processed by a linkage editor before it can be accessed by an APL user as a packaged workspace. The linkage editor again transforms the format of the object module into something called a load module, and saves the resulting load module in a load module library.

The packaged workspace in the form of a load module in a load library can be accessed by APL users. Any APL user who has access to the load library can cause a copy of the packaged workspace to be loaded into his address space. By limiting access to the load library, the application owner can limit access to the packaged workspace in much the same way access is limited for saved workspaces.

If the application is to be used simultaneously by multiple APL users, however, it may be desirable to install a resident shared copy of the packaged workspace in a shared area of the computer's memory (e.g., the Link Pack Area in MVS or a DCSS in VM). This is not a task that can be normally accomplished by an APL programmer - it requires some planning and manual intervention by the authorized personnel who maintain the computer facility.

The process of creating a packaged workspace may sound difficult and complex to APL users because of the number of steps involved and because of the need to use unfamiliar facilities such as the linkage editor. Actually, the whole process is relatively simple and straight forward and can be quickly accomplished by the average APL user without the need for any substantial education in operating system facilities. Only if the packaged workspace is to be installed on a resident shared basis do systems personnel have to be involved, and they must be involved in this case because specialized system knowledge as well as systems level authorization is required to complete this step.

An example will clarify the process and illustrate its simplicity. Because the process differs slightly between the VM/CMS and MVS/TSO environments, two examples in appendices A and B have been developed. It is recommended that the reader review the appropriate appendix.

Once a workspace is packaged, how can it be accessed?

Once a workspace has been transformed into a packaged workspace, objects in it can be accessed through Processor 11. Using ⎕*NA*, an APL user can declare a name to be external to his workspace and to exist in a packaged workspace. For example, the user might declare *SETUP* as an external function which exists in packaged workspace *REPORT* in load library *REPLIB* with the following APL expression:

```
'REPLIB.REPORT' 11 ⎕NA 'SETUP'
```

After this declaration, the SETUP function can be used as if it existed in the user's workspace. It can be called with arguments, and it can be used as part of a larger APL application. The workspace in which it is declared can be saved and

reloaded, and the external function *SETUP* can continue to be used, without having to be redeclared.

This "magic" is accomplished as follows:

- When the name *SETUP* is declared using ⎕*NA*, Processor 11 is contacted. It uses the first item of the left argument of ⎕*NA* to locate the packaged workspace. It loads that packaged workspace and attempts to locate *SETUP* within it.

- If *SETUP* is found in the packaged workspace, it is established as a name in the user's workspace. It's name class and attributes (function, variable, time stamp, etc.) are taken from the packaged workspace.

- Sufficient information is stored in the user's workspace to locate *SETUP* in the packaged workspace.

- When the name *SETUP* is encountered during execution of an APL expression in the user's workspace, the system locates it's definition (or value if it is a variable) from the packaged workspace.

- If the user's workspace is saved, the external name *SETUP* and the control information associated with it is saved along with the user's workspace.

- If the user's workspace is cleared, or replaced with a)*LOAD* command, Processor 11 is contacted by the system and the packaged workspace is deleted.

- If the saved workspace containing the external name *SETUP* is reloaded, the packaged workspace in which *SETUP* actually exists is not reloaded until *SETUP* is first encountered during the execution of an APL expression.

As can be seen from this explanation, a packaged workspace is loaded when a name is declared with ⎕*NA* to exist within it, or when a name that was previously declared to exist within it, is encountered during execution of a)*LOAD*ed application. Thus, unless an external name is localized in an application, there is no need to reissue the ⎕*NA* after reloading an application.

If a resident shared copy of the packaged workspace installed (in the Link Pack Area in MVS, or in a DCSS in VM), it is simply accessed as needed by Processor 11. If it was not installed on a resident shared basis, Processor 11 typically must load the packaged workspace from a load library into free space (defined with the APL2 "FREESPACE" invocation option) in the user's address space.

When the user's workspace in which the external name is declared is cleared or replaced with a)*LOAD* command, or when the user signs off APL, Processor 11 releases its access to the packaged workspace. If the packaged workspace had been loaded by Processor 11, this causes it to be deleted from the user's address space. The packaged workspace will also be released if all of the external names

in the user's workspace which point to the packaged workspace are expunged. So, for example, if an external name is localized:

```
    ∇RUN;SETUP
[1] →(1≠'REPLIB.REPORT' 11 ⎕NA 'SETUP')/ERROR
[2] ....
```

and if that external name is the only one in the user's workspace which is defined to exist in that packaged workspace, then the packaged workspace will be loaded each time the function *RUN* is executed and deleted each time the function completes. Obviously, if the *RUN* function is executed frequently, substantial overhead will be incurred as a result of this localization of the external name.

How does the system locate a packaged workspace?

When an external name is declared, Processor 11 uses the first item in the left argument of ⎕*NA* to locate the packaged workspace. There are a number of different variations in the way this information can be specified,[1] but two are commonly used:

'*LIB.MEMB*' 11 ⎕*NA* '*ROUTINE*'

> specifies that the object *ROUTINE* is located in the packaged workspace which is stored as member *MEMB* in load library *LIB*. In MVS/TSO, *LIB* is the ddname allocated to the dataset in which *MEMB* resides. In VM/CMS, *LIB* specifies the file name of the load library in which *MEMB* resides; that load library must have file type *LOADLIB* and must reside on an accessible minidisk.

'*MODULE*' 11 ⎕*NA* '*ROUTINE*'

> specifies that the object *ROUTINE* is located in the packaged workspace named *MODULE*. In MVS/TSO, *MODULE* will be found using standard OS search order (i.e., LPA, JPA, STEPLIB, etc.). In VM/CMS, a CMS nucleus extension named *MODULE* will be used if it exists, otherwise, Processor 11 will attempt to load a *TEXT* file with file name *MODULE*.

The first alternative ('*LIB.MEMB*') is typically used when the packaged workspace is to be loaded from a load library on disk.

The second alternative ('*MODULE*') is typically used to gain access to a packaged workspace that has been installed on a resident shared basis. This alternative must be used with care because:

---

[1] For a complete explanation of the possible left arguments to ⎕*NA*, see the "API 2 Programming: System Services Reference" manual, SH20-9218.

- Under MVS/TSO, Processor 11 issues a LOAD (SVC 8) against the name *MODULE*. This will cause the operating system to search the following, in order, for a load module with the specified name:

  1. the job pack area (for a previously loaded module with the same name);
  2. any task libraries that are in effect (this sometimes includes the load library from which API.2 is loaded);
  3. the step library (if one exists);
  4. the link pack area (this is the place where shared routines are placed in MVS);
  5. the MVS link libraries.

  Since the shared area is far down in this search order, the possibility exists that a non-shared copy will be located instead. Further, because the system searches through a large number of names in this process, there is a reasonable possibility that it will first find something else with the same name. This is particularly true if the packaged workspace has a common name like *LOAD*, *APL2*, or *IEFBR14*. API users are advised to consult a system administrator or system programmer when attempting to install or name packaged workspaces for resident shared use.

- Under VM/CMS, special initialization (described below) is required to cause the resident shared packaged workspace to be available as a CMS nucleus extension. If a CMS nucleus extension with the specified name is not found, the system attempts to load a *TEXT* file with the same name. If a *TEXT* file is loaded, it will not be shared and may be destroyed by CMS commands issued from the API 2 session. The use of *TEXT* files for packaged workspaces is not recommended.

In the VM/CMS environment, packaged workspaces can be installed on a resident shared basis by link editing them with API 2 (RESPPS option at API.2 installation) and placing API 2 in a DCSS, or they may be saved in separate DCSS's. Either of these options require the involvement of an authorized system administrator. The second alternative (separate DCSS's) provides more flexibility, since updates to a packaged workspace do not require reinstallation of API.2 or other packaged workspaces. Each packaged workspace must be placed in a separate DCSS, but the DCSS's may be overlapped. Processor 11 will take care of making sure that the correct DCSS is available at the appropriate times.

If, under VM/CMS, resident shared packaged workspaces are loaded into separate DCSS's, the user must issue the following CMS command before attempting to use ⎕*NA* to access objects in a packaged workspace:

```
AP2VUTIL DCSS LOADFW dcss_name pkg_name
```

where *dcss_name* is the name of the DCSS in which the packaged workspace is saved, and *pkg_name* is the name to be assigned to the packaged workspace. This *pkg_name* must correspond to the *MODULE* name specified in the left argument of ⎕*NA*, viz.:

```
      )CLEAR
CLEAR WS
      100 ⎕SVO 'CMD'
2
      CMD←'AP2VUTIL DCSS LOADFW REPDCSS REPORT'
      CMD
0
      'REPORT' 11 ⎕NA 'SETUP'
```

This *AP2VUTIL* command causes the *REPDCSS* to be accessed and set up as a CMS nucleus extension named *REPORT*. The subsequent ⎕*NA* directs Processor 11 to that CMS nucleus extension.

Finally, while it is not essential, if the VM/CMS API 2 user wishes to delete access to the resident shared packaged workspace when he is finished with it, the following CMS command can be issued:

```
AP2VUTIL DCSS PURGEFW dcss_name pkg_name
```

Can more than one packaged workspace be loaded at a time?

Yes, if there is sufficient space in the user's address space, many packaged workspaces can be loaded simultaneously. The number is limited simply by workspace size and free space size. Some space is required in the user's workspace and in free space to hold control information about a packaged workspace, and of course, if the workspace is not accessed on a resident shared basis, it will require space into the user's address space.

Names can be declared in the user's workspace to be external and to exist in one or more packaged workspaces. But names can also be declared to be external in a packaged workspace and to exist in another packaged workspace. Thus, the user's workspace can point to a packaged workspace and that packaged workspace can point to another packaged workspace, and so on. Packages can point to each other, and can even point back to the user's workspace.

As can be imagined, such structures can be quite complex. To attempt to clarify terminology in such a structure, the term "user's active workspace", or "active workspace" is used to refer to the workspace which the user controls with *)CLEAR*, *)LOAD*, or *)SAVE* commands. This is the primary workspace; it is available automatically on a read-write basis when the user invokes API.2 and its contents are managed with system commands. For a given API. user, there is only one active workspace. The same user, however, may have access to many saved workspaces and many packaged workspaces. As described above, many packaged workspaces may be accessed simultaneously for a given API user as a result of external names declared in the active workspace or external names declared in other packaged workspaces which are accessed by the user.

What sorts of things can be accessed in a packaged workspace?

The user who creates a packaged workspace can specify limits on just what can be accessed in that packaged workspace. The *PACKAGE* function, used to create a packaged workspace, is ambivalent. If a left argument is supplied, it specifies a list of names that are to be accessible in the packaged workspace. If such a list is provided when the packaged workspace is created, only those names can be accessed in the packaged workspace. Attempts to use *⎕NA* to access a name not on the list will fail.

If a left argument is not provided to the *PACKAGE* function when the packaged workspace is created, all names in the packaged workspace will be accessible as external names from the user's active workspace or from another packaged workspace.

Within the constraints imposed by the left argument of *PACKAGE*, any name that exists in the packaged workspace is accessible as an external name through the use of *⎕NA*. This includes:

- names of defined functions and defined operators;
- names of variables;
- names of labels (if there were suspended functions in the saved workspace when it was packaged);
- names of system functions or system variables;
- names of shared variables (see explanation below);
- names of external variables, functions or operators.

Shared variables do not exist as such in a saved workspace. To be shared, a variable must be offered using *⎕SVO* after the workspace has been loaded or accessed. Since packaged workspaces are created from saved workspaces, to have a shared variable in a packaged workspace, it must have been offered after the package was accessed. This is possible in two situations:

- the user invokes an external function which exists in a packaged workspace and that function, as part of its processing, shares a variable;

- the user accesses *⎕SVO* in a packaged workspace and and uses it to share a variable. This is possible using a surrogate name in the right argument of *⎕NA*, viz.:

```
      'REPLIB.REPORT' 11 ⎕NA 'ΔSVO ⎕SVO'
1
      100 ΔSVO 'VAR'
2
      'REPLIB.REPORT' 11 ⎕NA 'VAR'
1
      VAR←'COMMAND'
      VAR
0
```

Unlike shared variables, external variables retain their characteristics across *)LOAD* and *)SAVE*, and across packaging. Thus a name that was external

when a workspace was saved, will remain an external name when the workspace is reloaded, or if the workspace is packaged and subsequently accessed. Names of such external objects in packaged workspaces may themselves be the target of ⎕NA. Thus, for example, if X was a variable in saved workspace WSX, and that workspace was packaged, then another workspace WSY could define it as external with the expression:

```
'LIB.WSX' 11 ⎕NA 'X'
```

If workspace WSY was subsequently packaged, an APL user could declare X to be external to his active workspace and to exist in packaged workspace WSY, viz.:

```
'LIB.WSY' 11 ⎕NA 'X'
```

With these declarations the user would actually access X in WSX.

The limit for such indirection (external names pointing to external names) is 181 levels of indirection. As the reader can imagine, declaration of such indirection could be extended to a structure that eventually creates a loop and points to itself. If such a structure is created, the system will detect it when the external name is encountered during execution and a *SYSTEM LIMIT* (interface capacity) error will be generated.

How are global references resolved in external functions?

Each workspace, packaged or saved, defines a set of named objects - variables, functions and operators. Defined functions and defined operators can refer to their arguments and operands, can create and use local names, and can create and use global names. When a global reference is made from within a defined function or operator, that name is expected to be found in the workspace. If the name is not found, a *VALUE ERROR* results.

Similarly, when a function in a packaged workspace refers to global names, those names must be found in the packaged workspace, or more precisely in the name scope of the packaged workspace. Thus, in a simple case, if a user executes an external function that exists in a packaged workspace, and that function refers to a global variable, that variable had better exist in the packaged workspace, or the function will be suspended with a *VALUE ERROR*. No attempt is made to look for the name in the caller's active workspace if it cannot be found in the packaged workspace.

In a sense, having access to a packaged workspace is like having a second workspace loaded simultaneously. When an external function is called from the user's active workspace, the system switches from the name scope of the active workspace to the name scope of the packaged workspace. Until the external function completes execution, all names will be resolved in the packaged workspace name scope, and no reference will be made to names in the user's active workspace. When the external function completes execution, the system switches back to the name scope of the user's active workspace.

If, while executing in the name scope of a packaged workspace, a function creates new global variables or changes the values of global variables, those changes remain in effect in the packaged workspace name scope as long as the packaged workspace is accessed - even if the user's workspace is saved and reloaded at a later date. Thus, if the user calls an external function that creates a global variable, that variable is created in the packaged workspace's name scope and is available to other functions in the packaged workspace, or for that matter as the target of a *⎕NA* from another name scope.

How do name scopes work?

To understand how this works, one has to understand a little bit about name management in an APL workspace. Every APL workspace - the active workspace, saved workspaces and packaged workspaces - contains a name table which is used to catalog the names in the workspace and to locate their values or definitions. This name table is accessed by the *)FNS*, *)VARS*, *)OPRS*, and *)NMS* commands and by system functions like *⎕NC* and *⎕NL*. The name table is a volatile thing: new entries are added as names (including local names) are created, and entries are deleted as names are expunged or de-localized. In many APL systems the name table is referred to as the "symbol table".

When a name is declared to be external and to exist in a packaged workspace, the system locates and accesses the packaged workspace and copies the name table from the packaged workspace into the user's active workspace. This second name table is not directly accessible to the user, but is used by the system as the definition of the packaged workspace's name scope. When the user calls external functions or references external variables in the packaged workspace, this second name table is used to locate their definitions or values. If the user specifies an external variable, this second name table is updated to point to the new value. The same thing occurs if an external function creates or changes the value of variables during it's execution - the second name table is updated to reflect the changes.

The second name table is also significant insofar as it represents this APL user's instance of the packaged workspace. If the packaged workspace is installed on a resident shared basis, and if another APL user simultaneously accesses it, both users get individual copies of the packaged workspace's name table in their active workspaces. Each user's modifications to the packaged workspace name scope are reflected in the user's private copy of its name table. Thus, one user's modifications are not available to another user.

When and if the user saves his active workspace with a *)SAVE* command, the second name table is saved along with it. When the saved workspace is reactivated with a *)LOAD* command, the second name table comes in with it and the status of that name scope when the active workspace was saved is preserved.

Does each name scope have its own copy of system variables?

The system variables that can be set by the user can be divided into two categories: those that are workspace related and those that are session related. Workspace related system variables are restored to their default values when the workspace is cleared. They may also be saved with a workspace and will retain their values when that workspace is reloaded. Session related system variables retain their values for the duration of an API session or until reset. Those values are not reset by)*CLEAR* or)*LOAD*. The session related system variables defined in API.2 are: □*FW*, □*NLT*, and □*TZ*.

The system maintains only one copy of session related system variables. The values of those variables are not affected by moving from one name scope to another. If their values are set in one name scope, those values remain in effect when the system enters another name scope.

Each workspace - active, saved or packaged - has its own copy of the workspace related system variables. As the system moves from one name scope to another, the values of these variable may change. Setting the values of these variables in one name scope does not not affect their values in other name scopes.

How are updated values in a packaged workspace handled?

If a user updates an external variable, or if while executing in an external function, the function updates a global variable, the new value is placed in the user's active workspace and the private copy of the packaged workspace's name table is updated to point to it. Thus, values created or modified during execution in a packaged workspace consume space in the user's workspace. These new values, although they exist in the user's workspace, are not accessible to the user, except through external names. The new values physically exist in the user's active workspace, but are part of the packaged workspace's name scope and not of the active workspace's name scope. They are pointed to by the private copy of the packaged workspace's name table, not by the active workspace's name table.

The same principles hold true for changes in the definition of functions or operators, or for defined functions or operators that are dynamically created or read from a function file while executing in a packaged workspace's name scope. Similarly, if values or definitions are expunged from the packaged workspace's name scope, the associated name table is updated to reflect these changes.

Just as the updated packaged workspace name table is saved with the user's active workspace when a)*SAVE* command is issued, new or modified values or definitions are also saved and are preserved when the workspace is reloaded.

One final important piece of information before completing this topic. When an unmodified defined function or defined operator is accessed (for execution, or by functions like □*CR*), its definition is accessed directly from the packaged workspace, without having to make a copy of the definition in the user's active workspace. Thus, the definitions of functions or operators from a packaged

workspace do not consume space in the user's active workspace unless they have been modified, or dynamically created. The same thing is not true for variables. When a variable from a packaged workspace is accessed for the first time, even for a read-only reference, or as the argument to a function, a copy of the value of the variable is made in the user's active workspace before execution proceeds. The reasons for this are complex, and are due to practical engineering issues rather than any theoretical reasons why it must be done this way. Nonetheless, in APL2 Release 3, all variables in packaged workspaces that are referenced as external variables, or accessed in the course of executing in the packaged workspace's name scope, consume space in the user's active workspace. They are "faulted in" from the packaged workspace as they are accessed.

What is the effect of a name class change in a packaged workspace?

When a name is declared to be external through the use of *⎕NA*, the system determines the name class (function, variable, operator) from its class in the packaged workspace. That same name class is used for the external object. *⎕NC* applied to an external name will yield the correct result.

As a result of execution in the packaged workspace's name scope, however, the name class of the external object could be changed. This could happen, for example, if one function in the packaged workspace expunged another function and then created a global variable of the same name. If this happened, a mismatch between the name class of the external object and the actual object in the packaged workspace would result. The system does not attempt to resolve or correct such mismatches, since the overhead to do so would be too high. Instead, it will detect such a mismatch when and if the external name is subsequently referenced and issue an error message (*VALUE ERROR* for an external variable, or *VALENCE ERROR* for an external function).

The only way to correct such a name class mismatch is to expunge the external name and reestablish it with *⎕NA*. In practice, this behavior does not normally cause a problem, since name class changes of this sort are rare in most applications.

Can an external function be suspended?

An external function in a packaged workspace is just like a function in the user's active workspace. If it does not carry the non-suspendable attribute, it can be suspended as the result of an error or an interrupt, just like any such function in the user's active workspace.

When an APL user is in immediate execution mode with no pendant functions on the stack, he is in the name scope of the active workspace. That is to say, the names available to him (and reported by *)FNS*, *)VARS*, etc.) are defined in the name table of the user's active workspace. When an external function is executed by the user, the system switches to the name scope of the external function's packaged workspace. If, while the external function is executing, it is suspended (as a result of an error or an interrupt), the user is left in the name scope of the packaged workspace. *)FNS*, *)VAR*, *⎕NL*, etc. will report

names from the packaged workspace's name table. Names from the active workspace's name table will not be accessible until the pendant function is removed from the stack.

While suspended in a packaged workspace's name scope, there is no restriction on what the user can do. For example, the user can display the state indicator (the stack), display and modify defined functions, modify local or global variables, execute functions, and resume execution. All of these operations are carried out in the name scope of the packaged workspace.

This ability to be suspended in a packaged workspace is an extremely useful characteristic of the system. It allows normal debugging techniques to be used for applications that are structured to use packaged workspaces. Unfortunately, however, getting an application to suspend in a packaged workspace is not always that easy. Typically, to cause suspension, a developer simply sets a stop vector on the desired function. For a stop vector to be effective on an external function, however, it must be set in the name scope of the packaged workspace. This can be done using $\square EC$ in the packaged workspace's name scope, viz.:

```
      'REPLIB.RFPORT' ⎕NA 'SETUP'
1
      'REPLIB.RFPORT' ⎕NA 'ΔEC ⎕EC'
1
      ΔEC 'SΔSETUP←1'
 2  0 0  1
      SETUP 'INITIAL'
SETUP[1]
      )SI
SETUP[1]
*
```

If the workspace was packaged with a left argument to *PACKAGE*, and if $\square EC$ was not included in that list, this technique will not be effective. Developers, therefore, should plan to incorporate debugging hooks in applications that are to be packaged with a left argument to *PACKAGE*. Such debugging hooks can be as simple as including $\square EC$ in the left argument of the *PACKAGE* function.

What if a packaged workspace is replaced with a new version?

When an external name is declared in the user's active workspace, the name table is copied from the packaged workspace into the user's active workspace. As execution proceeds, changes to this name table are made to reflect new and updated variables and other changes in the packaged workspace's name scope. If the user then issues a *)SAVE* command, these changes and the modified packaged workspace's name table are saved along with the user's active workspace. The packaged workspace itself, however, is not saved.

If the user reloads the saved workspace at some future date, the modified packaged workspace's name table and associated changes are reloaded along with it. The actual packaged workspace will be reaccessed by the system when the first unmodified external name in it is accessed. Execution proceeds as it would have prior to the *)SAVE* command. That is to say, the state of the active

workspace and the packaged workspace are the same as they were when the user issued the)*SAVE* command.

If, however, the packaged workspace had been modified and repackaged by its owner between the time the user saved and reloaded his workspace, a mismatch would occur. The modified copy of the packaged workspace's name table would not necessarily correspond to the actual contents of the new version of the packaged workspace.

The system detects such potential mismatches by maintaining a creation date in each name table in the system. Thus the modified copy of the packaged workspace's name table, which was saved along with the user's workspace, would have one creation date, and the new version of the packaged workspace would have a different one. The system detects such a mismatch when the user attempts to access any external object in the packaged workspace. Each time the mismatch is detected, the system will issue a warning message:

PACKAGED WORKSPACE name MODIFIED SINCE LAST ACCESS

and will attempt to access the external object in the new version of the packaged workspace, using its new name table. Since the name table from the new version of the packaged workspace is used, modifications that were recorded in the old version of that name table will not be available.

In most situations, this behavior will not cause problems. If, however, the application depends on the modified state of the packaged workspace name table across)*SAVE* and)*LOAD*, a new version of the underlying packaged workspace will cause this modified state (and all associated data) to be lost. To avoid such loss of data, owners of packaged workspaces are advised to maintain backup copies of previous versions of their packaged workspaces that can be used when this situation occurs. If the user does not resave his active workspace after receiving the warning message, the modified state and data can be recovered by reverting to use of the previous version of the packaged workspace.

How do external operators work?

In APL2, defined operators may take functional operands which may be specified to vary the behavior of the derived function, viz.:

```
      ∇Z←(FN COMPUTE)R
[1]   Z←'THE RESULT IS: '.,⍕FN R
      ∇

      -COMPUTE 1 2 3
THE RESULT IS: ¯1 ¯2 ¯3
      ⌽COMPUTE 3
THE RESULT IS: 1 2 3
```

When a defined operator is placed in a packaged workspace and declared as external from the user's active workspace, its operands and arguments come from the user's active workspace, as would be expected. If an operand is a defined function, any names it refers to during its execution come from the user's active

workspace, rather than the packaged workspace. Similarly, if a functional operand requires implicit arguments (like ⍺), those implicit arguments must come from the user's active workspace, viz.:

```
      'REPLIB.REPORT' 11 ⎕NA 'COMPUTE'
1
      ⎕IO←0
      ⍺COMPUTE 3
THE RESULT IS: 0 1 2
      ⎕IO←1
      ⍺COMPUTE 3
THE RESULT IS: 1 2 3
```

When a functional operand is passed to an external operator, the system passes an identification of its name scope along with the operand. Later, if that operand is executed in the packaged workspace, the system switches back to the correct name scope to carry out its execution. If the functional operand passed to an external operator is passed along as an operand to another external operator in a different packaged workspace, it continues to carry its original name scope identifier with it.

Does the user have access to the name scope identifier?

A name scope is identified by the left argument to ⎕NA for the function or operator used to enter that name scope. In the example shown above, the name scope identifier for the external operator *COMPUTE* is `'REPLIB.REPORT' 11`. As we will see below, the name scope identifier for the user's active workspace is `'' 11`.

An application can determine its own name scope identifier through the use of the external function *QNS* which is distributed with API.2, viz.:

```
      3 11 ⎕NA 'QNS'
1
      QNS 0
 REPLIB.REP 11
```

Can a packaged workspace access its caller?

We have seen that ⎕NA can be used to create a link from the active workspace to a packaged workspace, or between packaged workspaces. It can also be used to create a link from a packaged workspace to the user's active workspace.

```
      '' 11 ⎕NA 'ROUTINE'
```

specifies that the external object *ROUTINE* is to be located in the user's active workspace and not in a packaged workspace. This variant of ⎕NA can only be successfully issued from a packaged workspace, since it is not possible to declare an external name to exist in your own name scope.

This use of ⎕NA provides a valuable facility that allows applications in packaged workspaces to "reach back" into the user's active workspace to access objects there. For example, using this facility, a packaged application, designed to print

the functions in the user's active workspace, could reach back to use ⎕NL to obtain the necessary list of function names, and reach back to use ⎕CR to obtain the definitions, viz.:

```
      '' 11 ⎕NA '∧NL ⎕NL'
1
      '' 11 ⎕NA '∧CR ⎕CR'
1
      ,[⍳0]ΔCR¨⊂[2]ΔNL 3 4
```

While this technique will work well for many applications, there are other situations where the application in the packaged workspace might be called from the user's active workspace or from another packaged workspace. In such situations, it may be important to be able to reach back into the caller's environment rather than all the way back into the active workspace. There are a number of ways in which this can be done:

- If, when entering the packaged workspace, the caller had provided his own name scope identifier (found through the use of ⎕NS 0) as an argument to the external function, the external function could then use that information as the left argument to ⎕NA to access objects in its caller's environment.

- If the packaged workspace was entered via an external operator, a functional operand could be provided which would execute in the caller's name scope. This functional operand could be used by the external operator as the vehicle by which access to the caller's name scope could be gained.

- An external function, *EXP*, is provided with API 2 to facilitate this process of reaching back into the caller's name scope. This function, described in the "API 2 Programming: Using the Supplied Routines" manual (SH20-9233), provides the ability to execute functions and to reference or specify variables in the previous name scope, viz.:

```
      3 11 ⎕NA 'EXP'
1
      ⍝ Reference caller's ⎕IO
      EXP ⊂'⎕IO'
0

      ⍝ Specify caller's ⎕IO
      EXP '⎕IO' '←' 1
1

      ⍝ Execute ⎕NL in caller's environment
      EXP '⎕NL' 2
ONE
TWO
```

  EXP will execute in the caller's active workspace if there is no previous name scope to reach back to. Consequently, it can be used in applications during the debugging phase, before they are packaged.

These three different techniques are all valuable in different situations, for example:

- The first technique uses $\square NA$ to specifically identify the target name scope. It is particularly useful where an application wants to ensure it is reaching back into the user's active workspace, as opposed to any intervening packaged workspaces.

- The operand to an external operator approach is valuable because it is a relatively transparent mechanism to use, and because the name scope identifier is passed along to any other packaged workspace the callee might itself call.

- The third technique, using EXP, is simple and general, but requires special coding in the application program.

Packaged Workspace Applications

Packaged workspaces represent an important advance for APL systems and APL application developers. They can be used to advantage in a wide variety of different applications. This section attempts to describe some of the common uses and associated benefits.

Application segmentation

Packaged workspaces provide a simple, effective way of segmenting large applications. For example, self contained portions of an application could be placed in separate packaged workspaces and thus isolated from one another. Each portion could be considered as a "black box" capable of performing certain function and accessible through entry points defined with $\square NA$. The application developer could then access these "black boxes" as required without concern for name conflicts or undesired side effects with other portions of his application. Each "black box" portion could be developed and tested separately, and in fact could be replaced with updated technology or algorithms at a later date, without affecting the operation of the rest of the application system.

Such a notion should not be foreign to APL developers. Defined functions, with their ability to localize names, provide the same sort of facility on a smaller scale. With packaged workspaces, however, it is possible to isolate groups of functions and variables, including global variables whose values can be updated and used in subsequent accesses to the packaged workspace.

Name scope isolation

There are many APL workspaces which are designed to be used in conjunction with other arbitrary APL applications. For example, workspace analysis tools which can be used to analyze a user's application, producing call trees or cross reference reports, are common. Normally, tools like these are copied into the user's workspace and must co-reside there with the user's application. This inevitably leads to name conflicts. Tools like these often go to extraordinary lengths to avoid name conflicts, only to be done in by other tools which use a similar technique.

Packaged workspaces provide an effective solution to this problem. Tools that must work in conjunction with arbitrary APL applications can be implemented as packaged workspaces, without concern for name conflicts. Only the entry points must be declared (using ⎕*NA*) as names in the user's application, and those name conflicts can be resolved by specifying a surrogate name in the right argument to ⎕*NA*, viz:

```
'TOOLS.XREF' 11 ⎕NA 'MYREF XREF'
```

Dynamically accessed applications

Very large APL applications are sometimes implemented as a series of workspaces which are accessed by stacking)*LOAD* commands. It is not uncommon, for example, to see applications which present a menu of subapplications to the user and which implement access to those subapplications using this sort of technique.

Threading one's way through a series of workspaces using stacked)*LOAD* commands, however, is not simple and is rarely robust. Unexpected errors or interrupts can leave the naive user stranded in a subapplication. Error trapping facilities such as ⎕*EC* are not effective across workspace transitions.

Again, packaged workspaces provide a simple and effective alternative for such applications. Subapplications, implemented as packaged workspaces, can be dynamically accessed and deleted. Execution of functions within the packaged workspaces can be implemented under control of ⎕*EC* at the external function level. If an error or unexpected event occurs during subapplication execution, control will be returned to the master application which can then recover in a graceful fashion.

The use of packaged workspaces for such applications considerably reduces the complexity of the control mechanisms for subapplication transition, while at the same time improving the reliability and integrity of the overall application.

Shared applications

An obvious benefit of packaged workspaces is the ability to share APL application code between simultaneous users. Instead of requiring each user to)*LOAD* a separate copy of the application into his address space, a single copy of a packaged workspace can be made resident in a shared area of the computer's memory (PPA in MVS, DCSS in VM), and accessed by multiple simultaneous APL users. Resident shared access to a packaged workspace can dramatically reduce the total real memory required to support a set of simultaneous users running the same applications. Further, if there are a sufficient number of users active, large portions of the application code which would otherwise have to be paged in will remain resident in the computer's real memory. This can lead to improved response times, and of course, a significantly reduced paging load on the system.

There are many sorts of applications that might be installed on a resident shared basis in a particular installation. Obvious candidates include applications like

IC/1 which are large and often used by more than one user simultaneously. Highly volatile applications (where the application code changes frequently) are less attractive, if only because of the necessity to involve systems personnel to install the packaged workspaces on a resident shared basis. Frequently accessed utility programs and tools, however, often make other good candidates for resident shared packaged workspaces. For example, in an environment with a number of concurrent API developers active, it might be worthwhile considering implementing the public workspace `1 DISPLAY` as a resident shared packaged workspace.

Alternative to function files

Many large applications resort to implementing "function files" in an attempt to reduce their real memory (or "working set") requirement. Rather than requiring all functions to be resident in the workspace at the same time, the application stores many of the less frequently used functions on an external file and reads them in as required for execution. When execution of such a function is complete the function is expunged to make room for the next function to be called.

While this technique is effective at reducing the working set for a large API application, it imposes an overhead on the application to manage this dynamic accessing of functions. This overhead is measurable in two ways: CPU utilization and complexity. It is not uncommon to find that up to 30% of the CPU time incurred in such an application is devoted to this process of dynamically accessing functions. Further, an application using such techniques is often larger and more complex, if only because these techniques must be imbedded in the applications itself. This, of course, increases the working set of the application - the original problem we were attempting to solve.

If the application is to be used by multiple simultaneous API users, it could be installed as a resident shared packaged workspace. Some of the resulting real memory reduction could be used to make more or all of the dynamically accessed functions from the function file resident in the packaged workspace. This can, in many situations, lead to substantial reductions in CPU utilization and response times.

Central maintenance control

One problem with public workspaces is that users often save their own copies of them, or `)COPY` functions from them into their own applications. Later, if a new version of the public workspace is installed, users may be unaware that their own copies are down level. This can lead to all sorts of problems.

Worse, there are applications, like `ADRS` whose operation is dependent on each user saving his own private copy. Such applications must implement complex, confusing and often unreliable mechanisms which attempt to ensure that the code is at the correct maintenance level.

If instead, the application was installed as a packaged workspace, a fresh copy of the code would be accessed each time the user accessed the application. Further, if the user had saved a workspace which accessed the packaged workspace, he would be notified automatically by the system if a new version of the packaged workspace had been installed when he subsequently attempted to use his saved workspace.

Object code only applications

One flaw sometimes cited by those who do not espouse the use of APL is that it is difficult if not impossible to protect proprietary application code. They point to compiled languages where it is possible to ship the application in object code ("OCO") form and thus prevent user's access to the proprietary details imbedded in the source code.

It is generally acknowledged that object code does not afford complete protection of proprietary information, particularly against disassembler tools and aggressive "hackers". It does, nonetheless, make it considerably more difficult for normal and non-malicious users to gain access to algorithms used in the product.

The ability to lock functions in APL is not considered to be the same level of security as is afforded by object code. Packaged workspace, however, do provide some additional facilities when combined with other techniques can in fact provide about the same level of protection as object code. By:

1. removing comments from the workspace,
2. changing the names to non-meaningful names,
3. locking the functions,
4. packaging the workspace,
5. limiting access to functions in it (through the use of the left argument to *PACKAGE*),
6. protecting the packaged workspace load modules with operating system security facilities,

the application developer can further limit access to the algorithms embodied in the code.

As noted in the list above, packaged workspaces provide two of the facilities which aid in this process. First, by providing a left argument to the *PACKAGE* function when a packaged workspace is created, the developer can limit the entry points to the packaged workspace. Second, because a packaged workspace is a load module which is accessed by operating system facilities on a read only basis, security facilities like RACF can be used to limit access to it, just as they can be used to limit access to compiled programs.

Access to data

Many APL applications use a considerable amount of CPU resource reading data from files into the APL workspace. Often such data is static and does not have to be written back out. With larger and larger workspaces, it is possible in many situations to read an entire file into the workspace and reap the resulting benefits of application simplicity and performance. In such cases, however, the application must still read the data, one or a few records at a time into the workspace.

If the data was placed in a packaged workspace instead of in a file, it could be accessed as an external variable. While it would still consume space in the user's workspace, it could take considerably less time to load it, since the load of a packaged workspace is accomplished by operating system techniques rather than an APL application that involves looping. In fact, if the packaged workspace was installed on a resident shared basis, it would take almost no time to access it!

Appendix A: Creating Packaged Workspaces in VM/CMS

The process of creating a packaged workspace is simple and relatively straight forward. It does, however, involve the use of some operating system facilities which may be unfamiliar to the APL user. For additional information on VM/CMS commands such as *FILEDEF* or *LKED*, consult the "VM/SP CMS Command Reference" manual (SC19-6209).

Creation of a packaged workspace from a saved workspace requires three steps:

1. Copying and resaving the workspace to be packaged.

2. Conversion of the saved workspace to an object module. This process is performed with the function *PACKAGE* which is provided with APL2.

3. Conversion of the object module to a load module. This process is performed with the "linkage editor" - a program that is provided as a part of VM/CMS.

For the purposes of illustration, we will assume that the following simple functions and variables have been defined and saved in a workspace called *REPORT*.

```
      )CLEAR

      ∇Z←SETUP R
[1]   Z←R,MESSAGE
      ∇

      ∇Z←PROCESS R
[1]   GLOBAL←R
[2]   Z←'PROCESSING COMPLETE'
      ∇

      MESSAGE←' SETUP COMPLETE'

      ⎕IO←0

      )SAVE REPORT
```

Step 1 - Copying and resaving the workspace

Before beginning the packaging process, it is recommended that the user *)COPY* and resave the workspace to be packaged. This accomplishes two things:

1. it compacts the workspace and cleans out any unnecessary garbage in it;

2. it ensures that the workspace is at the correct level. The *PACKAGE* function will not process workspaces saved in prior releases of APL2.

For our simple sample workspace, this can by accomplished as follows:

```
)CLEAR
)COPY REPORT
⎕IO←0
)WSID REPORT
)SAVE
```

Note that any system variables with non-default values must be reset in this process, since the)*COPY* command does not copy system variables.

Step 2 - Creating the object module

An object module is a file with a specific internal format that differs from the format of the file produced by the)*SAVE* command. The external function *PACKAGE*, provided with API.2, will convert the file produced by the)*SAVE* command to one in object module format. This process is performed under control of API.2:

```
    )CLEAR
CLEAR WS

    3 11 ⎕NA 'PACKAGE'
1

    PACKAGE 'REPORT APLWSV2 A'
REPORT TEXT A
```

The right argument to the *PACKAGE* function is the VM CMS file identifier of the saved workspace. VM/CMS file identifiers consist of 3 fields: filename, filetype and filemode. For an API.2 saved workspace, the filename is the workspace name as specified on the)*SAVE* command. The filetype is *APLWSV2* for private workspaces or *V000000n* for public workspaces (n specifies the library number). The filemode is the identification of the CMS minidisk on which the saved workspace resides and is defined by the library number from the)*SAVE* command and the *LIBTAB APL2* file which defines those libraries.

The saved workspace to be packaged must reside on an accessible CMS minidisk. Note that the)*SAVE* command has the ability to dynamically link to and access CMS minidisks. The *PACKAGE* function does not have this ability - the CMS minidisk must be already accessed before the *PACKAGE* function is run.

If the filetype and filemode are not specified in the right argument to *PACKAGE*, *APLWSV2* * will be defaulted. Hence, in the example above,

```
PACKAGE 'REPORT'
```

could have been substituted.

The *PACKAGE* function is ambivalent. If no left argument is specified, as in the example above, all names in the packaged workspace will be accessible to users. By specifying a list of names as a left argument to *PACKAGE*, the developer can limit access to objects in the packaged workspace. For example,

```
'SETUP' 'PROCESS' PACKAGE 'REPORT'
```

would cause a packaged workspace to be built in which only the *SETUP* and *PROCESS* functions would be accessible (via □*NA*) to users. Other names like *MESSAGE*, □*IO* or □*EC* in the packaged workspace could not be accessed.

The *PACKAGE* function produces an output file in which the resulting object module is placed. This file has a filename which matches the filename of the saved workspace and a filetype and filemode of *TEXT A*. The result of the *PACKAGE* function is the VM/CMS identifier of the object module file produced. This file identifier will be required for the next step in the process.

If the *PACKAGE* function fails in this process of creating an object module file, it will return a null vector result. Normally, a message indicating the error will also be produced. Typical errors include:

- *WORKSPACE DATASET OPEN FAILURE*: the saved workspace specified cannot be found
- *WS INVALID*: the input file specified is not an API 2 workspace, or is an API 2 workspace saved under a prior release of API 2
- *AN I/O ERROR HAS OCCURRED WHILE WRITING THE WORKSPACE*: insufficient space on disk for the object module file.

Step 3 - Creating the load module

The object module file created in step 2 by the *PACKAGE* function can be converted to a load module using the linkage editor provided as a part of VM/CMS. The following CMS commands will accomplish this task for our sample workspace:

```
FILDEF SYSLMOD DISK REPLIB LOADLIB A (RECFM U
LKED REPORT (NAME REPORT
FILEDEF SYSLMOD CLEAR
ERASE REPORT TEXT A
```

These VM/CMS commands can be issued outside the API 2 environment or from within the API 2 environment by means of the)*HOST* command or via AP 100.

The first command above specifies the name of the load library into which the resulting load module will be placed. In this example, a load library named *REPLIB LOADLIB A* will be created. Except for the filename, *REPLIB*, and filemode, *A*, this command should be specified as shown. Changing other command arguments could lead to undesirable results. The filename (*REPLIB*) can be specified by the user as the filename of an existing or new load library, and the filetype can be specified as * or as the CMS minidisk on which the load library resides.

The second command causes the object module file *REPORT TEXT **, produced in step 2, to be converted to load module format and placed as a

member named *REPORT* in the load library defined above. This command has the following format:

```
LKED filename (NAME membername
```

where *filename* is the filename of the object module file produced in step 2, and *membername* is the name of the resulting load module produced. It is recommended, but not required that the two names be the same.

The final two commands clear the ddname *SYSLMOD* and delete the object module file created in step 2, which is no longer needed. These commands are optional, but are recommended as part of normal cleanup. Finally, it is recommended that the original saved workspace be retained, since it may be required for maintenance or other purposes in future.

Appendix B: Creating Packaged Workspaces in MVS/TSO

The process of creating a packaged workspace is simple and relatively straight forward. It does, however, involve the use of some operating system facilities which may be unfamiliar to the API user. For additional information on MVS/TSO commands such as *ALLOCATE*, *FREE*, *LINK* and *DELETE*, consult the "OS/VS2 TSO Command Language Reference" manual (SC28-0646).

Creation of a packaged workspace from a saved workspace requires three steps:

1. Copying and resaving the workspace to be packaged.

2. Conversion of the saved workspace to an object module. This process is performed with the function *PACKAGE* which is provided with API.2.

3. Conversion of the object module to a load module. This process is provided with the "linkage editor" - a program that is provided with the MVS operating system.

For the purposes of illustration, we will assume that the following simple functions and variables have been defined and saved in a workspace called *REPORT*.

```
      )CLEAR

      ∇Z←SETUP R
[1]   Z←R,MESSAGE
      ∇

      ∇Z←PROCESS R
[1]   GLOBAL←R
[2]   Z←'PROCESSING COMPLETE'
      ∇

      MESSAGE←' SETUP COMPLETE'

      ⎕IO←0

      )SAVE REPORT
```

Step 1 - Copying and resaving the workspace

Before beginning the packaging process, it is recommended that the user *)COPY* and resave the workspace to be packaged. This accomplishes two things:

1. it compacts the workspace and cleans out any unnecessary garbage in it;

2. it ensures that the workspace is at the correct level. The *PACKAGE* function will not process workspaces saved in prior releases of API.2.

For our simple sample workspace, this can by accomplished as follows:

```
)CLEAR
)COPY REPORT
⎕IO←0
)WSID REPORT
)SAVE
```

Note that any system variables with non-default values must be reset in this process, since the)*COPY* command does not copy system variables.

In the MVS/TSO environment, workspaces may be saved in SAM or VSAM libraries depending on the library number specified on the)*SAVE* command (for details see "API 2 Programming: System Services Reference", SH20-9218). The *PACKAGE* function used in the next step will only process workspaces which have been saved in a SAM library. If the workspace to be packaged exists in a VSAM library, it should be resaved in this step in a SAM library by specifying an appropriate library number on the)*SAVE* command.

Step 2 - Creating the object module

An object module is a file with a specific internal format that differs from the format of the file produced by the)*SAVE* command. The external function *PACKAGE*, provided with API 2, will convert the file produced by the)*SAVE* command to one in object module format.

The object module produced by the *PACKAGE* function will be written into a sequential data set allocated to the ddname *SYSPUNCH*. This allocation must be performed before the *PACKAGE* function is executed. The data set allocated should have *LRECL*(80) and *RECFM*(F) or *RECFM*(F B), and should be approximately 1.5 times the size of the saved workspace in bytes. For our example, we will allocate *SYSPUNCH* to a new data set named *REPORT.OBJ* with the following TSO command:

```
ALLOCATE FILE(SYSPUNCH) DSN(REPORT.OBJ) NEW +
         DSORG(PS) SPACE(5 5) TRACKS +
         RECFM(F B) LRECL(80) BLKSIZE(3120)
```

Once this allocation is complete, the *PACKAGE* function can be run:

```
    )CLEAR
CLEAR WS

    3 11 ⎕NA 'PACKAGE'
1

    PACKAGE 'REPORT'
USER.REPORT.OBJ
```

The right argument to the *PACKAGE* function is the name of the workspace to be packaged, as specified in the)*SAVE* command that created it. As noted above, that workspace must be saved in a sequential library.

The *PACKAGE* function is ambivalent. If no left argument is specified, as in the example above, all names in the packaged workspace will be accessible to users.

By specifying a list of names as a left argument to *PACKAGE*, the developer can limit access to objects in the packaged workspace. For example,

```
'SETUP' 'PROCESS' PACKAGE 'REPORT'
```

would cause a packaged workspace to be built in which only the *SETUP* and *PROCESS* functions would be accessible (via □*NA*) to users. Other names like *MESSAGE*, □*IO* or □*FC* in the packaged workspace could not be accessed.

The result of the *PACKAGE* function is the fully qualified data set name of file into which the object module is written (i.e., the file allocated to ddname *SYSPUNCH*). If the *PACKAGE* fails in this process of creating an object module, it will return a null vector result. Normally, a message indicating the error will also be produced. Typical errors include:

- *ALLOCATION ERROR - SYSPUNCH*: the *SYSPUNCH* ddname for the input saved workspace has not been allocated
- *WS INVALID*: the input data set is not an APL2 workspace, or is an APL2 workspace saved under a prior release of APL2, or saved in a VSAM library
- *AN I/O ERROR HAS OCCURRED WHILE WRITING THE WORKSPACE*: insufficient space in the output data set.

Step 3 - Creating the load module

The object module created by the *PACKAGE* function in step 2 can be converted into a load module using the linkage editor provided with the MVS operating system. The linkage editor, invoked with the TSO command *LINK*, reads the object module file and creates a load module as a member of a load library. The load module created will be approximately the same size in bytes as the original saved workspace. The load library in which it will be created by the linkage editor:

- must be a partitioned data set,
- must exist before the the *LINK* command is executed,
- normally has a data set name whose lowest level qualifier is *LOAD*,
- must have *RECFM(U)*, but may have any *BLKSIZE* less than 32K, and
- need not be allocated to a ddname when the *LINK* command is executed.

The following TSO commands can be used to create the necessary load library for our example:

```
ALLOCATE FILE(REPLIB) DSN(PKGLIB.LOAD) NEW +
         DSORG(PO) SPACE(5,5) TRACKS +
         BLKSIZE(4096) RECFM(U) DIR(2)
```

The TSO *LINK* command takes the following format:

```
LINK object LOAD(library(member) options
```

where:

object is the data set name of the object module to be link edited;

library is the data set name of the load library into which the resulting load module will be placed;

member is the member name to be assigned to the resulting load module;

options are various options supported by the *LINK* command. The only option that is required to create a packaged workspace load module is *RMODE(ANY)*, and that is required only in the MVS/XA environment.

The following *LINK* command can be used to transform the sample object module *REPORT.OBJ*, created in step 2, into a load module stored as member *REPORT* in the load library *PKGLIB.LOAD* allocated above:

```
LINK REPORT.OBJ LOAD(PKGLIB.LOAD(REPORT)) RMODE(ANY)
```

The TSO *LINK* command will assume low level qualifiers of *OBJ* and *LOAD* for the input and output data sets if they are not specified. Consequently, in our example, the *LINK* command can be simplified to:

```
LINK REPORT LOAD(PKGLIB(REPORT)) RMODE(ANY)
```

Once the *LINK* command has executed successfully, the object module created in step 1 can be deleted. It is recommended, however, that the original saved workspace be retained for maintenance purposes. The following commands accomplish this cleanup and free the two ddnames that remain allocated in our example:

```
FREE FILE(SYSPUNCH)
FREE FILE(REPLIB)
DELETE REPORT.OBJ
```

Packaged Workspaces

M. T. Wheatley
IBM Santa Teresa Laboratory
San Jose, California, USA

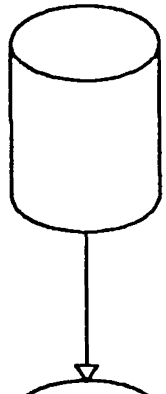
Packaged Workspaces

* APL workspaces formatted as load modules
* May be shared between APL users
* Dynamically accessible via □NA
* Provide name scope isolation
* Improved system performance when shared
* Alternative to function files
* Central maintenance control
* Allow OCO-like applications

Creating Packaged Workspaces (CMS)

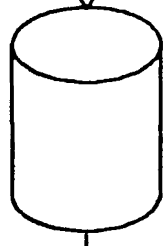
```
)SAVE REPORT
)CLEAR
)COPY REPORT
⎕IO←0
)WSID REPORT
)SAVE
```

Saved workspace:
REPORT APLWSV2 A



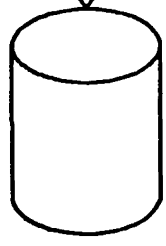
```
)CLEAR
3 11 ⎕NA 'PACKAGE'
PACKAGE 'REPORT'
REPORT
```

Object module:
REPORT TEXT A



```
FILEDEF SYSLMOD DISK REPLIB LOADLIB A
        (RECFM U
LKED REPORT (NAME REPORT
FILEDEF SYSLMOD CLEAR
ERASE REPORT TEXT A
```

Load module:
REPLIB LOADLIB A(REPORT)

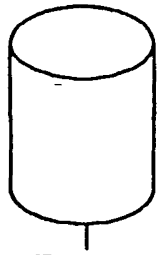


```
'REPLIB.REPORT' 11 ⎕NA 'SETUP'
```

Creating Packaged Workspaces (TSO)

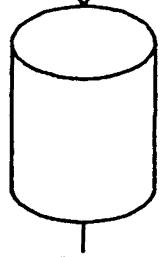
```
)SAVE REPORT
)CLEAR
)COPY REPORT
⎕IO←0
)WSID REPORT
)SAVE
```

Saved workspace:
V.REPORT



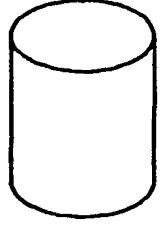
```
ALLOCATE FILE(SYSPUNCH) DSN(REPORT.OBJ)
)CLEAR
3 11 ⎕NA 'PACKAGE'
PACKAGE 'REPORT'
USER.REPORT.OBJ
```

Object module:
REPORT.OBJ



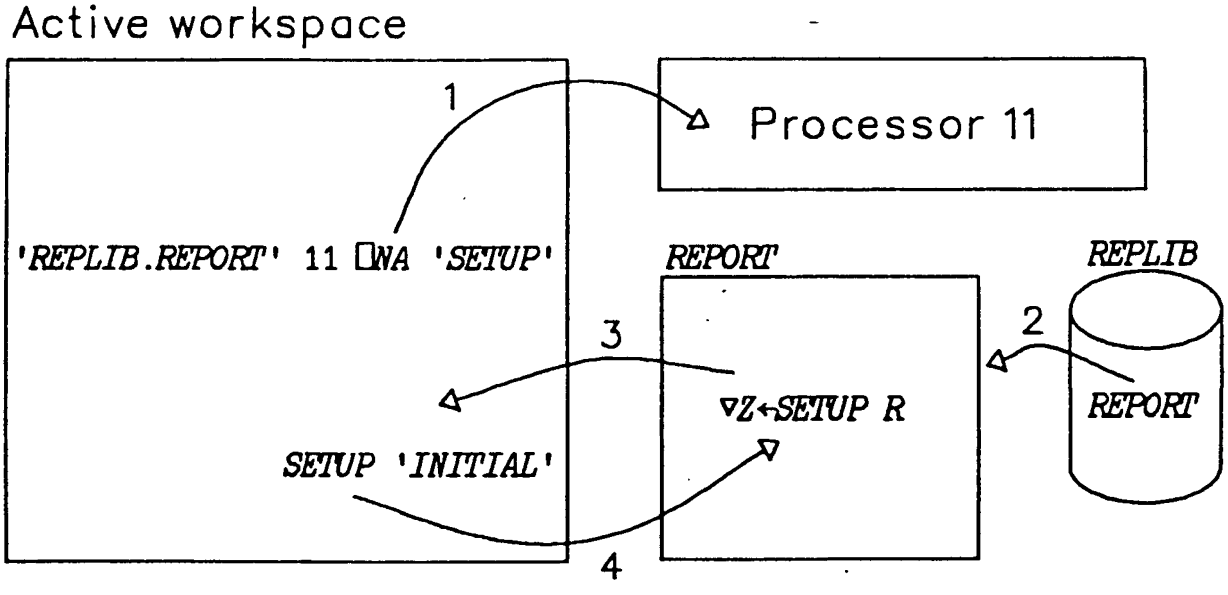
```
ALLOCATE FILE(REPLIB) DSN(PKGLIB.LOAD)
LINK REPORT LOAD(PKGLIB(REPORT)) RMODE(ANY)
FREE FILE(SYSPUNCH)
DELETE REPORT.OBJ
```

Load module:
REPLIB.LOAD(REPORT)

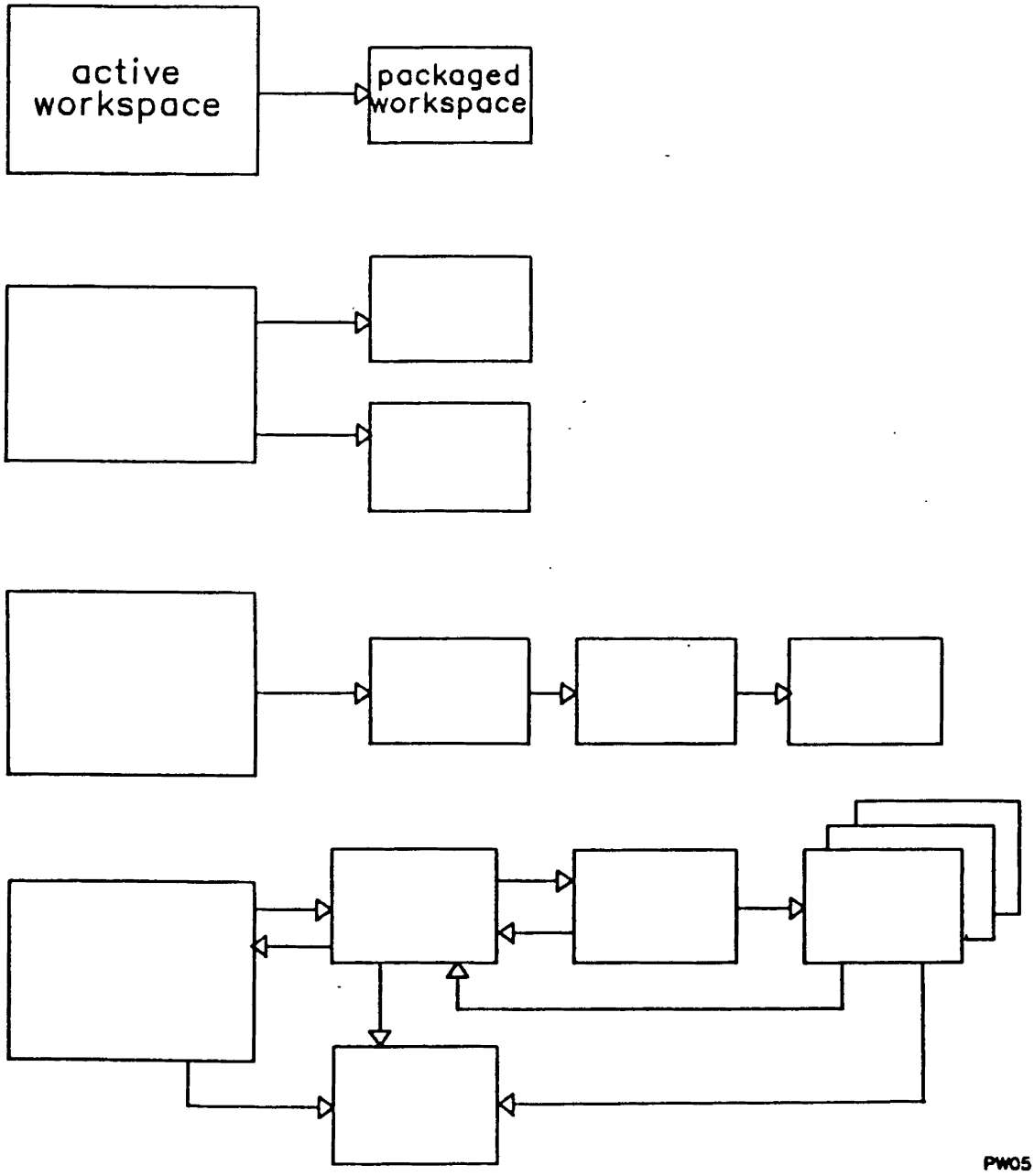


```
'REPLIB.REPORT' 11 ⎕NA 'SETUP'
```

Accessing Packaged Workspaces

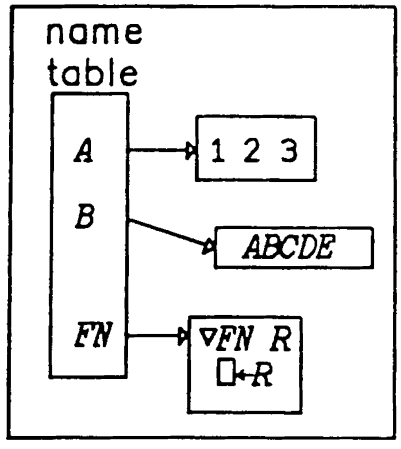


Accessing Packaged Workspaces

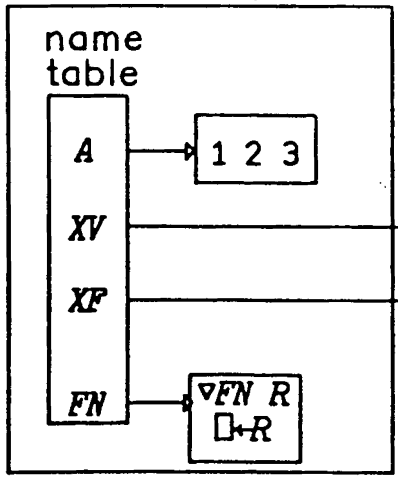


Name Scopes

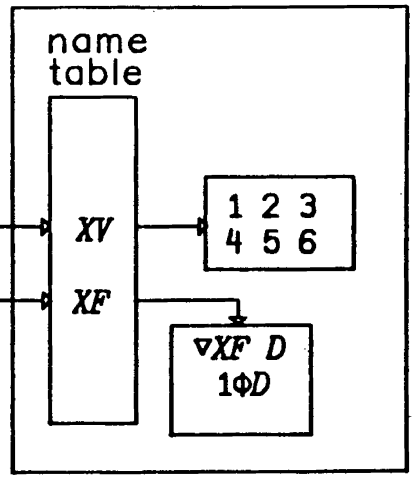
workspace



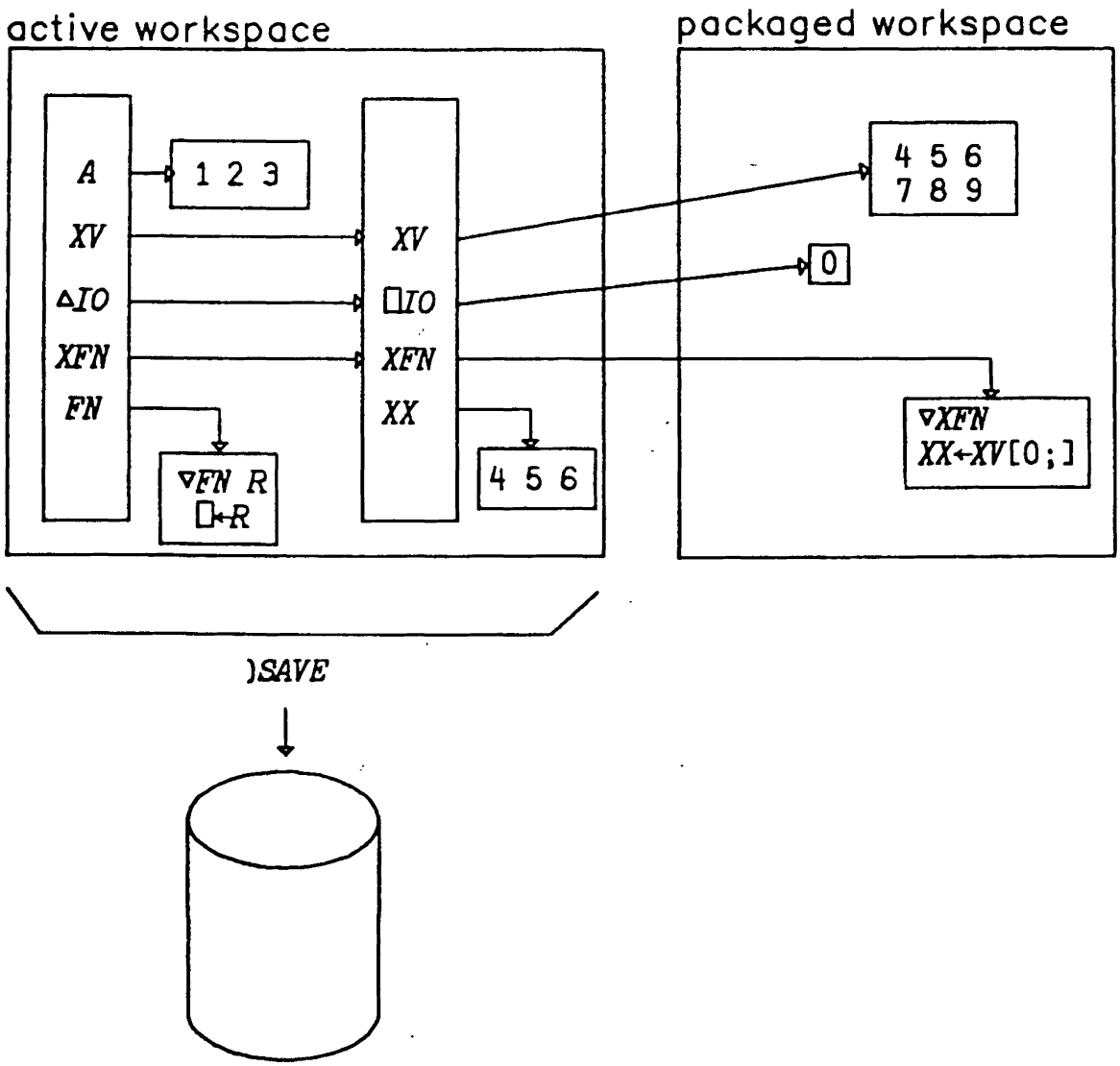
active workspace



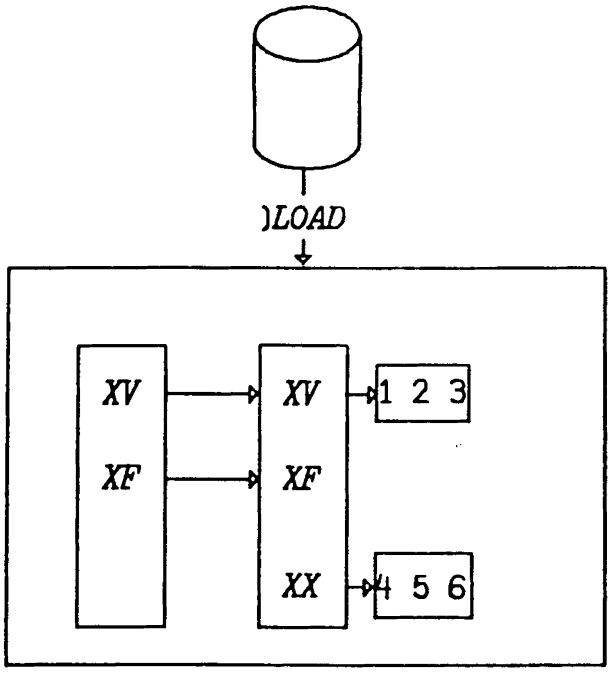
packaged workspace



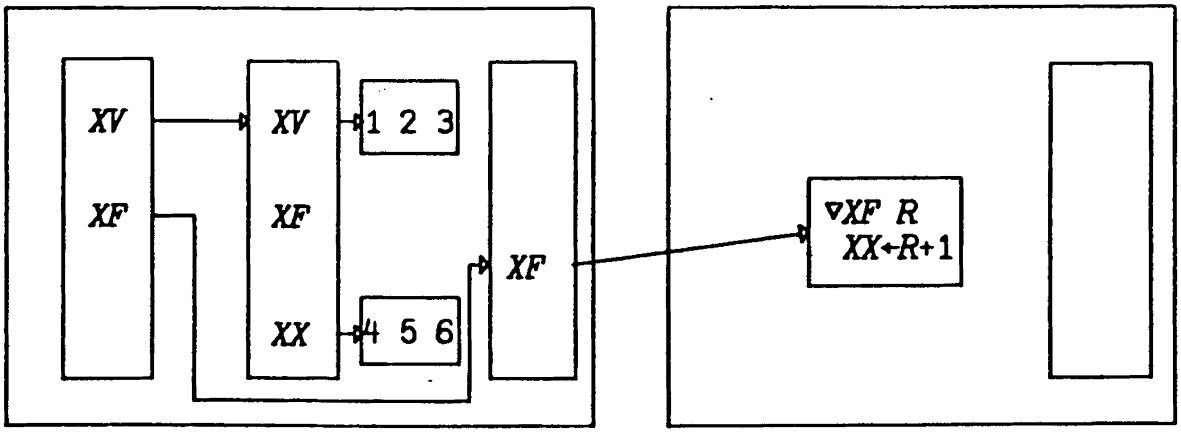
Name Scopes


active workspace
packaged workspace
A
XV
⍙IO
XFN
FN
1 2 3
∇FN R
⎕←R
XV
⎕IO
XFN
XX
4 5 6
4 5 6
7 8 9
0
∇XFN
XX←XV[0;]
)SAVE

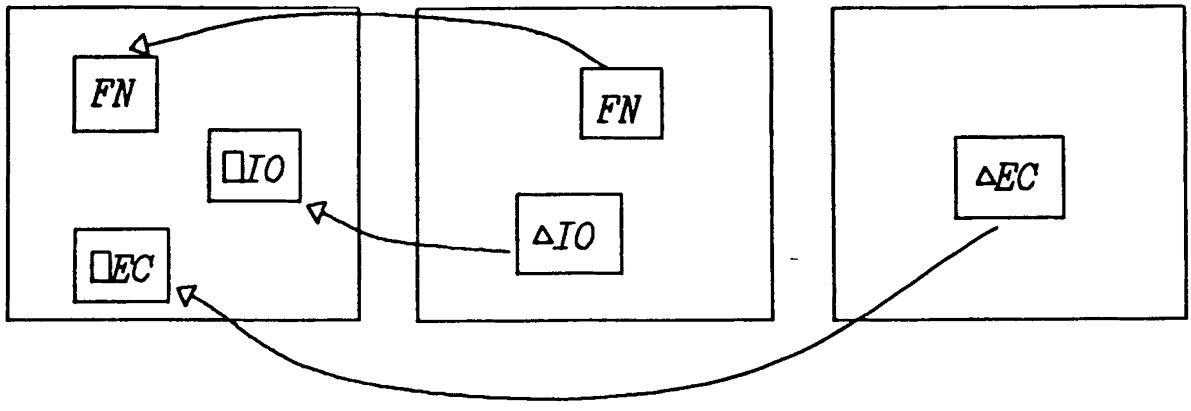
New Version of Packaged Workspace



PACKAGED WORKSPACE xxxx MODIFIED SINCE LAST ACCESS



Reach Back



1. ⎕NA to active workspace

```
'' 11 ⎕NA '⍙EC ⎕EC'
```

2. Pass name scope identifier

```
3 11 ⎕NA 'QNS'
QNS 0
LIB.MEMB  11
```

3. Operand to external operator

```
'LIB.MEMB' 11 ⎕NA 'OP'
⍎ OP DATA
```

4. Use *EXP*

```
3 11 ⎕NA 'EXP'
EXP c'⎕IO'
EXP '⎕IO' '←' 0
EXP 'FN' (1 2 3)
EXP (1 2 3) 'FN' (4 5 6)
```

Uses for Packaged Workspaces

* application segmentation
* name scope isolation
* dynamically accessed applications
* shared applications
* alternative to function files
* central maintenance control
* object code only applications
* access to data