TR

MULTI-USER SQL APPLICATIONS IN APL2

by Dr. James A. Brown

September 1985

TR 03.274

September 1985
TR 03.274

MULTI-USER SQL APPLICATIONS IN APL2

BY
Dr. James A. Brown

International Business Machines Corporation
General Products Division
Santa Teresa Laboratory
San Jose, California

ABSTRACT

This paper presents a method for quick implementation of small multi-user database applications. The advantages of having a single access to a database for multiple users are discussed. Some of the unique features of *APL2* are introduced and used to show a sample implementation of the multi-user server. A general method for domain checking of relational tables is presented.

1: Introduction

This paper discusses a method for quick implementation of small multi-user database applications. These applications may be given to users without distribution of the actual code of the application and without granting the user authority to use the database.

The discussion will include the representations of relational data in the programming language *APL*2, some unique features of the language that make multi-user applications easy, and some techniques for checking data destined for relational tables. An earlier paper, (1) discussed the actual mechanisms for communication with the relational database products DB2 and SQL/DS. It showed that *APL*2, unlike other programming languages, can access whole tables in a single operation.

To put on a professional appearance, you would need to add a full screen, panel driven front end to make the application user-friendly. This aspect of the application is no different than for a single user application and is not discussed here.

The reader does not need intimate knowledge of *APL* to understand the algorithms presented. The early sections assume very little knowledge of *APL*2 Later sections show the actual code that implements a multi-user server and complete understanding of these programs does assume *APL* familiarity. Someone without *APL* knowledge can still appreciate the style and brevity of the programs.

Appendices 2, 3, and 4 give some practical information about running and using the shared variable processor and are presented for completeness.

2: Objectives

There are numerous small applications that never get implemented because they would take up time needed for more critical matters. The purpose of this paper is to show a way to get a multi-user database application running quickly. The reduced development time makes it practical to write the smaller applications which, although they sometimes receive only casual use, increase the availability of information to users and thereby increase their productivity.

The surprise is that the method described, in addition to being fast to implement, increases the security of the application, tightens control on access to databases, and provides additional function to the database products in a general way.

The database products by themselves provide a secure multi-user environment. However, the database products are passive reacting only to requests. The server presented here is normally passive but may be self activating. It may wake itself up to take a backup, to monitor its own usage, to contact its users, or to apply maintenance to itself.

The following sections will present the relevant features of *APL2*, the implementation of a multi-user server, and a method for additional checking of relational data.

3: Features of APL

This section introduces the data structures of *APL* and shows how they are used to represent relational data. Then three somewhat unique features of *APL* are presented that make the implementation of the multi-user algorithms easy.

3.1: APL2 Data

This section will describe how *APL2* represents collections of data. A collection of data in *APL2* is called an array. An array can be used to represent almost any arrangement of data.

There are only two kinds of data in *APL* -- numbers and characters. A number may be logical (0 or 1), integer (1234), real (3.86), scaled (1*E*10), or complex (2*J*3). A character may be an ordinary character ('a') from the set of 256 EBCDIC characters, or an extended character (like a Japanese or Hebrew character) from a set of 2,147,483,648 extended characters.

An array in *APL2* is a rectangular collection where at each point in the rectangle you find a single number, a single character, or another array.

Here's a 3 by 3 array of numbers (a matrix):

```
      3 3⍴ 23 1 123E20 1 0 124E15 ¯1 1 1E11
23 1 1.23E22
 1 0 1.24E17
¯1 1 1.00E11
```

The symbol ⍴ is the "reshape" function. It means "reshape the numbers on the right into a collection having three rows and three columns.

Here's a 3 by 3 array with numbers and characters:

```
      3 3⍴'A' 'B' 'TITLE' 'C' 'D' 55 'E' 'F' 66
AB TITLE
CD    55
EF    66
```

Here's a 3 by 3 array with a matrix at each spot:

```
      3 3⍴ ⊂2 2⍴1 0 0 1
 1 0   1 0   1 0
 0 1   0 1   0 1

 1 0   1 0   1 0
 0 1   0 1   0 1

 1 0   1 0   1 0
 0 1   0 1   0 1
```

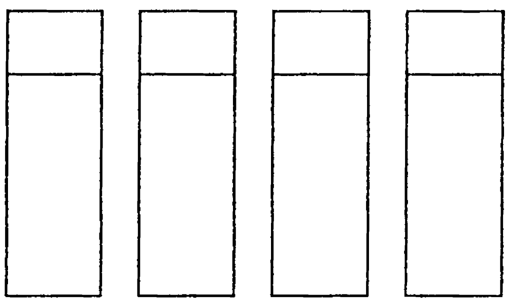
The symbol ⊂ is the "enclose" function. It means package the 2 by 2 array into a scalar array (an atom) which is then repeated nine times to get the three by three array.

In general, at any spot in an *APL2* array, it is OK to have any other array.

3.2: Representing relational data using APL2 arrays

In an *APL2*, anything can be at any spot. Real data, however, tends to be organized. In a relational table, you can only have numbers (of various formats) and character strings (of various lengths). A relation, in *APL2* terms, is a matrix with some discipline applied to what kind of data may occupy the spots.

Here's a stylized representation of a relation:



The top set of boxes represents column titles each of which is a character string. Each vertical box is one column of the relational table. Columns may be numeric or character. A numeric column has a single number in each row. A character column has a character string in each row.

Here's a real 4 by 4 *APL2* matrix that represents the employee table for a (very) small company:

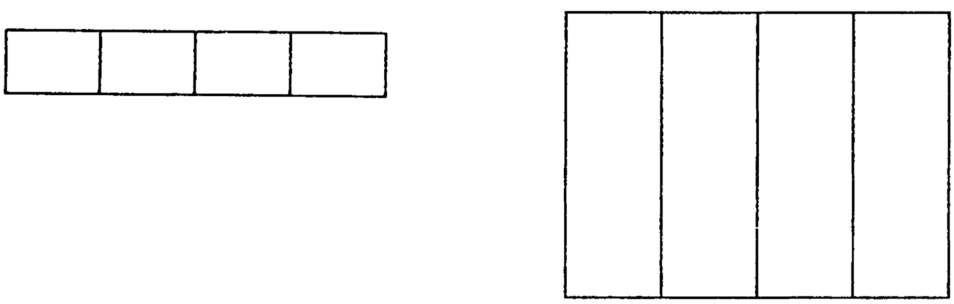
```
      WHO        ⍝ display the table
 EMPLOYEE NAME            ID   SALARY  DEPT
 DOE, JOHN            314159   25000  M75
 SMITH, JOHN          271828   22026  J88
 SHAKESPEARE, WILLIAM     14     250  Q25

      ⍴WHO       ⍝ compute shape of the table
4 4
```

The first and last columns contain character strings, the middle columns contain numbers, and the first row is character strings representing titles.

This is the most intuitive way to represent a relation. The column titles are over the columns where you would expect them. This is good if all you are going to do is format and display the data. Just mention the name of the array that represents the relation, and you get a simple report.

If, however, you are going to be doing computation, you normally do not want to do computations on the titles -- only on the data. Therefore, here is another representation of the same relation more convenient for computational purposes:



This, in *APL2* terms, is a two item vector consisting of the column titles and the data. If the employee table *WHO* were represented like this, then selecting the second item would select only the data portion. In *APL* notation this is written 2⊃*WHO* where ⊃ is called "pick".

There are other representations that you could choose for representing relational tables. For example, the *APL* interface to SQL supports a vector form of a relation which is often more efficient in storage (2). *APL2* does not impose a representation on you.

There is one more difference between a relation and an *APL2* matrix -- columns in a relation are governed by strict formatting rules. A numeric column is either integer, short integer, real, etc. A character column is either fixed length or or variable length where variable means no longer than some maximum length.

One way that *APL2* reduces development time for an application is its insensitivity to the declared column formats. When retrieving a table, however it is defined, you just get an array and do not need to know the format with which the table was defined. (You can determine the format by using a DESCRIBE operation, but you never have to.)

When writing data into a table, formats are more important. You'd better put numbers into numeric columns and characters into character columns. *APL*s interface to SQL will reject inappropriate data, but a better way to control the contents of tables is discussed in the section of Domains of Data.

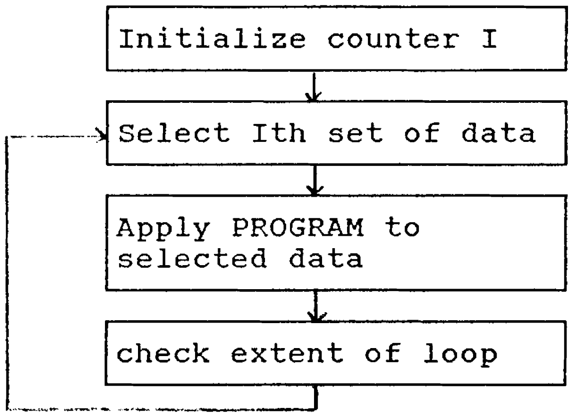
Another way that *APL2* reduces development time is the array orientation of *APL* processing. Access to relational data is on an array basis. Large parts of tables or even whole tables can be accessed and updated in a single operation. There is no need to do operations one row at a time. In this respect, *APL2* almost looks like an end user application -- yet it is a general purpose programming language.

3.3: Unique APL Facilities

There are many very powerful facilities of *APL2* which make writing applications fast. Three facilities, in particular, are discussed now and used later in the implementation. The facilities are normally not found as parts of a programming language. They will be discussed primarily by example.

The EACH operator

You often write a program with the intention that it will be applied to one set of data. When the need arises to apply it to many sets of data, you write a loop that causes the program to be called many times. This may be pictured as follows:



This might be written as a DO loop, a DO WHILE, or any of a number of programming constructs. An *APL* loop can be written in this style but there is a more elegant solution. Using the data structures of *APL*2, the sets of data are represented as a vector of arrays pictured as follows:

1st set	2nd set	3rd set

The new operator "each" (¨) takes a program or expression and applies it to each item of a collection. This may be pictured as follows:

PROGRAM ¨	1st set	2nd set	3rd set

is the same as:

PROGRAM 1st set	PROGRAM 2nd set	PROGRAM 3rd set

Here's a real example using a trivial program (in fact, just one primitive function). The function "interval" applies to one integer and produces the list of integers 1 to that number:

```
      ⍳4
1 2 3 4
      ⍳2
1 2
```

Using "each", the function can be applied to a whole collection:

```
      ⍳¨2 3 4
 1 2  1 2 3  1 2 3 4
```

giving a three item vector of vectors.

PROGRAM could be a simple computation as in ⍳¨ or a complete application program. The program could be written in *APL2*, FORTRAN, ASSEMBLY language, etc. When we write PROGRAM¨ the computation is probably more significant than ⍳¨, but the style is the same -- apply PROGRAM repeatedly to different data.

The EXECUTE function

APL has a way to treat character data as part of a program. For example:

```
      '2+3'
2+3
```

This is just a string of three characters.

The "execute" function treats the character data as though it were an expression in the program:

```
      ⍎'2+3'
5
```

Thus, "execute" is like taking off the quotes and just entering 2+3 which, of course, evaluates to 5.

While this is interesting, it doesn't look particularly significant. It could be statically compiled.

Here's another example of a character string which is not a constant but rather the result of a computation:

```
      A←'2+'      ⍝ define A as two characters
      A,'3'       ⍝ join A to the character '3'
2+3
```

Now we can apply the "execute" function to this:

```
      ⍎A,'3'
5
```

This is significant. The character string is dynamically computed as part of program execution, and then treated as a line in the program. There is no possibility of such a concept in a compiled language. You cannot compile such expressions because the value of *A* cannot, in general, be pre-determined.

This is a very powerful concept and it must be used with caution. "Execute" should not be used where other techniques will suffice because it can be inefficient. Later, in the discussion of checking relational data, "execute" will be seen as a most general facility. It is also used to implement multiple applications under a single multi-user server.

Shared names

A variable is a name which at different times has different array values. Normally, a variable is associated with a single user and holds data associated with his private application.

APL has the ability to process two independently running programs which have a name in common. Both programs can see the value of the variable and set it even though they are running in different virtual machines (in CMS) or different TSO address spaces (in MVS). Such a variable is called a <u>shared variable</u> because access to it is shared between two users.

Here's a possible session between two users. (Users in *APL* are identified by a number.) The vertical axis represents time:

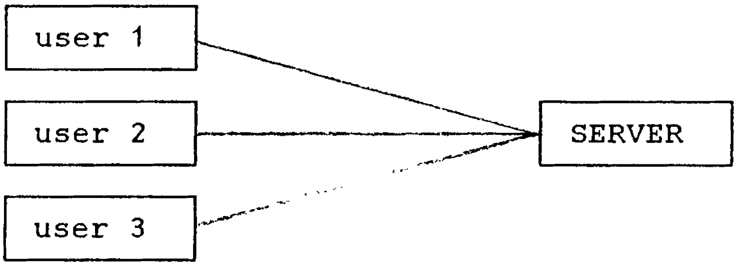
```
        user 1234              |         user 5678
_______________________________+_______________________________
                               |
      5678 ⎕SVO 'NAME'         |
1                              |
                               |        1234 ⎕SVO 'NAME'
                               | 2
                               |
             now NAME is a shared variable
                               |        NAME←5
                               |
      NAME                     |
5                              |
           value set by 5678 is seen by 1234
      NAME←NAME+1              |
                               |
                               |       NAME
                               | 6
           value set by 1234 is seen by 5678
                               |
```

⎕*SVO* is the way one program identifies a name it wishes to share with another user (it stands for Shared Variable Offer). The response of 1 on the left says 1234 has offered the name but his partner has not accepted it. But when 5678 says ⎕*SVO*, he gets a 2 and now the programs have a name in common. Thus, sharing a variable is a cooperative venture requiring the conscious intention of both partners. Now as the session proceeds, any value set by one of the partners is available to both partners.

A database server works on this same principle except that the data passed is more meaningful. The server is like user 1234 and each of the users of the service is like 5678.

4: A multi-user Application

Now it is time to fit these concepts into the implementation of a multi-user server. Here is a block diagram of the running application:



The lines of communication in this diagram are represented in the program by shared variables.

The server would normally run as a disconnected VM server machine or as a TSO batch session. Each user would normally be logged on and interacting with the server, but they could be batch programs as well.

Once sharing is established, one of the users puts a request in his variable and the server receives the value and processes it.

As a simple example, let the server be a teacher and each user be a student. The students submit answers to homework and test problems and the teacher/server records them in a database. A student sends a request like this:

```
A←'ASSIGN' 1 5 (2.71282 3.14159)
```

where *A* is a variable shared with the server. The teacher has probably set up some full screen panel which prompts for answers. The above expression would not normally be typed by the student but rather would be a line in the small program running the panels. The values given to the variable are, of course, entirely up to the application. The teacher has decided, when he designed the application, that four pieces of information will be passed to the server: the word 'ASSIGN' to say that this is a homework problem (as opposed to a test), 1 meaning the first assignment, 5 meaning problem number 5, and finally the answer to the fifth problem -- the two numbers 2.71828 and 3.14159.

When the server receives this value, it can save it in a database where the teacher can later retrieve and grade it.

If there is only one correct answer to the exercise, the program could even check the answer and do the grading.

As a courtesy, the server makes a response to the student:

```
A←'OK'
```

meaning that the request was logged in the database. Thus the student sends his answers and gets an acknowledgement that it was received.

Later the teacher can use all the power of the SQL language to make selections from the database; select all homework for one student, select all of assignment 1 and order by student, select all of problem 5 and compare how different students did on the same problem, etc.

Another example might be a company whose personnel records are on line in a database. A manager could request to see the salary for each person in his department by entering:

```
SHV←'SALARY' 'DEPT' 'J88'
```

the server would do a selection from the salary database and set the resulting values into the shared variable. The manager would then get the data he requested:

```
SMITH, JOHN    22026
BROWN, JOE     31000
WILLIAMS, BILL 19560
```

Since the application can tell who's asking, it can deny access to this same information if the person asking is not a manager:

```
   SHV←'SALARY' 'DEPT' 'J88'
   SHV
NOT AUTHORIZED TO SEE DATA
```

As a last example, suppose that the server accepts reminder requests. A user tells the server to send him a message at a specified day and time:

```
REQ←'REMIND' 'BROWN' 'STLVM20'(1986 2 1) 'VACATION'
```

The server receives this value and stores it in a 'REMIND' database and sends back an acknowledgement that the message was received and understood:

```
REQ←'OK'
```

On February 1, 1986, the server must wake up and send a message to the user. This will be discussed in more detail later.

Notice that each of these three examples used a different shared variable: *A*, *SHV*, and *REQ*. This is possible because the server which will be shown below does not care what name is used. It will accept a share under any name.

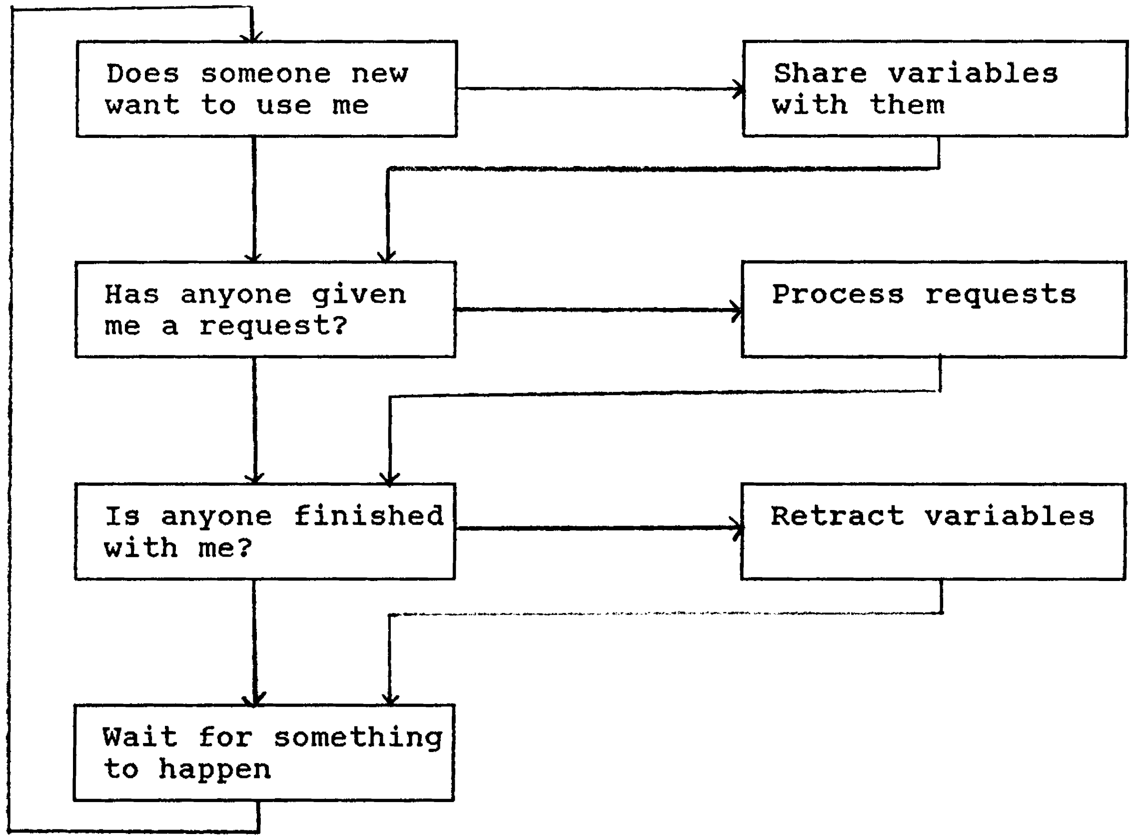
The shared variable is used to pass requests and responses. The bulk of the code for the application resides with the server and cannot be seen by the user of the application. In the examples in this paper, there is no code in the user's workspace at all. In a real application, there would be code primarily involved with prompting or menus for requests and formatting of results. There would <u>not</u> be any code in the user's area for accessing the database.

4.1: The Multi-User Application Server

The heart of the multi-user application is the server. This section describes a general and obvious logical scheme for the flow of a server and then shows that the scheme maps directly into a simple *APL2* program.

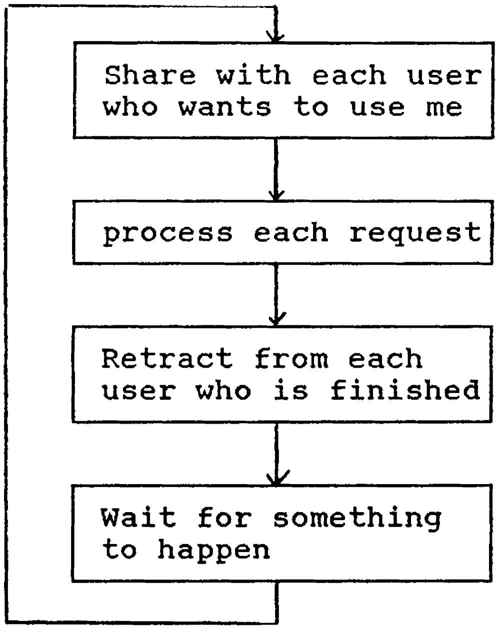
Here is a general block diagram of a multi-user server. Fundamentally, it waits for demands and then responds to them.

There are three kinds of demands: new users, requests from old users, and termination of users.



Notice that each demand is phrased in the plural. Each time a block on the left is visited, there could be zero or more affirmative answers. Thus, each block on the right may process many independent requests.

This program can be realized in *APL2* by a four line program where each line implements one row of the block diagram. The program is organized as follows:



Notice that almost all of the looping structure has been removed accept for the essential loop that repeats the program when something happens. There are now no tests such as "Does someone ...", "Has anyone ...", and "Is anyone ...". These are all replaced with array logic of the form "Get list of ..." followed by "share each", "process each", and "retract each". The program becomes very simple and straight-forward.

Here is the *APL*2 function that implements this server:

```
     ∇SERVER1 INTERVAL
[1]  RUN:SHARE¨⎕SVQ⍳0                      ⍝ share if anyone is ready
[2]   SETS←(⎕SVS¨VARS)∊⊂0 1 0 1            ⍝ isolate requests
[3]   PROCESS¨SETS/VARS                    ⍝ process requests
[4]   RETRACT¨(1=⎕SVO¨VARS)/VARS           ⍝ retract those that are done
[5]   →RUN ⎕SVE ⎕SVE←INTERVAL              ⍝ wait for something to happen
     ∇
```

and the function is called as follows:

```
      SERVER1 1E6
```

Lines [2] and [3] could have been written on one line given wider paper and except for this, *SERVER*1 has one line for each block on the left in the diagram.

Line [1] shares variables with any new users. ⎕*SVQ* (Share Query) returns the account numbers of anyone who has offered a variable to the server which has not yet been accepted. It calls the *SHARE* function for each new offer. Line [2] checks for requests from existing users. ⎕*SVS* is Shared Variable State and an answer of 0 1 0 1 is the state reported for a variable which has been set by the partner but not yet used by the server. Line [3] calls *PROCESS* for each request. Line [4] checks for users who are finished and calls *RETRACT* to terminate sharing. Since the server is sharing the variable, if ⎕*SVO* returns a 1 it can only mean that the user went away. Line [5] sets a timer for 1*E*6 seconds (⎕*SVE*←1*E*6) which will terminate after 1*E*6 seconds or when someone does something to a shared variable (⎕*SVE*). On termination of the the wait, the main loop is started again (→*RUN*).

Notice that each of the checks could produce multiple responses: several new shares, several requests, several terminations. This is the classic use of the "each" operator -- apply a program to a set of data when the number of items in the set is not predictable. The routines are written to operate on only one thing at a time. "Each" takes care of applying these routines one at a time to each piece of data. Thus the iterative application of the programs is accomplished without a loop.

> It is possible to write a server function with no loop at all by using a *REPEAT* operator to apply the server function over and over again. Leave off the →RUN and call the server like this:
>
> *SERVER*1 *REPEAT* 1*E*6
>
> See Appendix 1 for the details of the *REPEAT* operator and a discussion of how operators serve as structured programming constructs.

This server function represents structured programming at its best. Several sub-functions are called and in *APL2* they are separate entities. That means that this server, once written, can be treated like a primitive and applied in other situations to become the core of very different applications. Just write a different *PROCESS* function for each application.

The next section shows the details of a *PROCESS* function which can be used to implement many multi-user applications at once. This is useful if you have many low activity applications and do not wish to tie up many VM server machines or TSO batch sessions. The *SHARE* and *RETRACT* functions are not discussed in the paper but are listed in appendix 1.

4.2: The PROCESS Function

The function *PROCESS* is given the name of a shared variable, and its purpose is to process the one request represented by the value of that variable. If a single application were being implemented, *PROCESS* would be the main program of the application and would honor the request and send back a response.

When you want to write a new multi-user application, you can copy *SERVER* and the functions it calls and write a new *PROCESS* function. The new PROCESS function is, again, written to process one request from one user. The common code makes it work for many requests and many users.

Each small application runs in its own server machine or batch partition. If you have many small applications, it quickly becomes impractical to run each of them independently. If each application is relatively low activity, it may be better to write one *PROCESS* function that can run more than one application. Here is one way to write a general *PROCESS* function:

```
    ∇PROCESS V;ARG;RES
[1]  ARG←⍎V              ⍝ get value of shared variable
[2]  RES←⍎↑ARG           ⍝ execute requested application
[3]  ⍎V,'←RES'           ⍝ send response back to user
    ∇
```

This function assumes that the first item in the value of the variable *V* is the name of the application to be run. Therefore the user selects which of many possible applications he wants by making the name of the application the first item in the vector he sends.

This *PROCESS* function makes heavy use of the "execute" function. This can make the program hard to follow because what gets evaluated is in character strings and not written as part of the program. Here's an analysis of what actually gets evaluated in a specific case: Let's suppose that the following request is made:

```
      REQ←'REMIND' (1986 2 1) 'VACATION'
```

PROCESS would be called with the name of the shared variable *'REQ'* and so *V* would have that character string as its value.

```
      V
REQ
```

Thus in line [1] *ARG←⍎V*, if we substitute for *V* its value, this line becomes [1] *ARG←⍎'REQ'*; and since "execute" just removes the quotes (loosely speaking), this becomes [1] *ARG←REQ*. Therefore, line [1] gets the value from the shared variable and puts it into the variable *ARG*. The application can then find the value in a predictable name no matter what name the user used.

Here's what the rest of the program would look like after similar analysis:

```
[1]  ARG←REQ          ⍝ get value of shared variable
[2]  RES←REMIND       ⍝ execute requested application
[3]  SHV←RES          ⍝ send response back to user
```

[1] gets the value that the user gave to the shared variable. [2] calls the *REMIND* program which uses the name *ARG* to get the other parameters of the application. In [3], the result returned by *REMIND* is sent back to the user as his response.

This is a simple minded *PROCESS* function. A more practical one would do some error checking. For example, it would check that the application name was a legal one.

Let *APPL* be the list of applications supported

```
      APPL←'REMIND' 'SALARY' 'ASSIGN' 'TEST'
```

then the following expression will check that the requested application is one that is supported:

```
      ARG[1]∊APPL
```

Here's what the improved process function looks like:

```
    ∇PROCESS1 V;ARG;RES
[1]  ARG←⍎V               ⍝ get value of shared variable
[2]  →(ARG[1]∊APPL)/OK ⍝ branch if legal application
[3]  RES←'ILLEGAL APPLICATION' ⍝ set message for user
[4]  →DONE                ⍝ go send response to user
[5] OK:RES←⍎↑ARG          ⍝ execute requested application
[6] DONE:⍎V,'←RES'        ⍝ send response back to user
    ∇
```

A responsible *PROCESS* function will also protect itself against any failures in the applications it runs. A simple way to do this is to use a controlled execution which will not terminate if an error occurs. *⎕EC* is like "execute". It evaluates its character right argument and returns a return code and and error code along with the result. If the executed expression had an error, the return code is zero. If the expression is just an ordinary evaluation, as it should be for our *PROCESS* function, the return code is one.

Here is a *PROCESS* function with complete error trapping:

```
    ∇PROCESS2 V;ARG;RES;RC;ET
[1]  ARG←⍎V                  ⍝ get value of shared variable
[2]  →(ARG[1]∊APPL)/OK ⍝ branch if legal application
[3]  RES←'ILLEGAL APPLICATION' ⍝ set message for user
[4]  →DONE                   ⍝ go send response to user
[5] OK:(RC ET RES)←⎕EC↑ARG  ⍝ execute requested application
[6]  →(1=RC)/DONE            ⍝ go send result of successful call
[7]  RES←'ERROR IN APPLICATION'  ⍝ set message for user
[8] DONE:⍎V,'←RES'          ⍝ send response back to user
    ∇
```

Thus, by adding a little more mechanism (the function is still only eight lines long), a general *PROCESS* function is developed which is safe from errors made by the user and by the implementer of the application.

Now to add a new application, you only need add the name of the application to the list of legal applications and write a function with that name.

A more professional function could send more error information back to the user or perhaps log the information in a file. You can write more code to do whatever you want, but the style has been established.

5: Adding a REMIND application

Up to now, the server has only responded to external demands. A *REMIND* facility would require the server to wake up when it was time to send out a reminder. This is accomplished by using a modified server function:

```
     ∇SERVER2 INTERVAL;WAIT
[1]  RUN:WAIT←INTERVAL                  ⍝ set maximum waiting time
[2]   SHARE¨⎕SVQ⍳0                      ⍝ share if anyone is ready
[3]   SETS←(⎕SVS¨VARS)∊⊂0 1 0 1         ⍝ isolate requests
[4]   PROCESS¨SETS/VARS                 ⍝ process requests
[5]   RETRACT¨(1=⎕SVO¨VARS)/VARS        ⍝ retract those finished
[6]   ⎕SVE←DOREMIND WAIT                ⍝ send any REMIND messages
[7]   →RUN ⎕SVE                         ⍝ wait for next event
     ∇
```

[1] sets the variable *WAIT* to the longest time. [6] processes any reminders past due and returns the earliest time to the next reminder. [7] waits for the time interval set in ⎕*SVE* to elapse. Now the application will wake up at that time (or sooner if anyone makes some other request first). The functions that implement *DOREMIND* are shown in Appendix 1.

6: Adding a Maintenence application

Since the *APL* server is just a program and is running in real time, it is possible for it to apply maintenance to itself to fix problems or enhance function -- even add a new application to itself.

There are two ways to do this. First, the server could wake up periodically and check for the existence of a file of updates. If the file exists, the server could run an *UPDATE* program to read the file and establish the new functions and variables in the workspace. This could potentially involve spooled files and could be a complicated procedure.

The second way is to merely have an application (perhaps again called *UPDATE*) which gets as its *ARG*s the transfer forms of objects to be added or updated. Of course, only a small subset of users would be authorized to update the functions in the server.

Using one of these schemes means that the server need never be made unavailable for the purpose of doing maintenance. If you want to add a new application, just send the updated *APPL* variable and the new definitions to the server. Everyone can then immediately begin using the new application. If you want to send a new version of an existing function, just send it. The next time the function is needed the new version will be called. (This will not work for the *SERVER* function itself because it never stops running.)

7: Checking Relational data

For a given column of a relational table, there is a certain set of legal potential values. For example, a column of department names may contain one of a set of legal department names. The set of legal potential values is called the domain of the column. When a value is to be inserted into a column, several classes of domain checking may be done:

- class 1 - data type - numeric or character
- class 2 - data length - the number of values
- class 3 - data range - the set of legal values

In general, the database products take care of type and length considerations and the application must take care of data range.

7.1: Checking expressions

This section will show how the database products enforce type and length and show a general scheme for an application to enforce ranges.

Here is the *WHO* table presented before:

```
     WHO          ⍝ display the table
EMPLOYEE NAME               ID   SALARY  DEPT
DOE, JOHN               314159   25000   M75
SMITH, JOHN             271828   22026   J88
SHAKESPEARE, WILLIAM        14     250   Q25
```

The database products enforce formats on columns. The DESCRIBE SQL operation is used to fetch the following format description:

```
     FORM
NAME ID SALARY DEPT
V 32 I  I      C 3
```

FORM is a 2 by 4 matrix where the first row gives the titles of each column and the second row gives the format of each column. This information can be extracted from the database for any table or view desired. It tells you what checking the database will allow in each column. Thus, column 1 is variable length character with a maximum length of 32. Columns 2 and 3 are integer. Column 4 is fixed length character of length three.

The following four variables represent four rows that are candidates for new rows in the *WHO* table:

```
NEW1←'MORTON,J'      3270    'BEAUCOUP' 'J88'

NEW2←'LATTERMANN,D'  5150.95 31000 'HSC'

NEW3←'POLGAR,E'      2741    10 'OWN'

NEW4←'WINTON,S'      3775    48500   'J88'
```

When you attempt to put a new row into a relational table (An INSERT SQL operation) the database checks the proposed data against the formats and either rejects improper data or converts it to the correct form.

In the four examples, the database will reject *NEW*1 because *'BEAUCOUP'* is not a proper integer. The other three will be accepted because the data type and lengths are correct.

The checking done by the database may not be as strict as your application requires. For example, given the *ID* number 5150.95, as in the second example, the database will truncate and use 5150 as the integer. But the fractional part is indicative that someone entered bad data (maybe it should be a salary). Nothing in the database checking will prevent a negative salary or invalid department ID from being accepted by the database. If you want this kind of data rejected, your program must reject it.

General and complete error checking can be achieved by associating with each column an expression which validates a proposed value for that column. The expressions may be saved as a third row of the DESCRIBE matrix.

Here is the DESCRIBE table for *WHO* with the checking expressions where *X* is the proposed new value for the particular column:

```
     FORM
NAME ID              SALARY          DEPT
V 32 I               I               C 3
1    (X=⌊X)∧(X>0)   (X=⌊X)∧(X>0)   (⊂X)∊DEPTS
```

Each checking expression is defined so that it returns a 1 if the proposed data for the column is valid and 0 otherwise. The designer of the table and the application can decide how extensive this check will be. In this example, it is assumed that any value for *NAME* is correct so the checking expression is 1 (which, of course, evaluates to 1). The database will catch errors for this column (like name too long or non-characters). The *ID* and *SALARY* columns must be positive (*X*>0) and integer (*X*=⌊*X*). If either numeric column is given character values, the checking expressions will fail and generate an error. Finally, the expression to check for a legal department assumes that the *DEPTS* variable has previously been defined containing all the legal department names.

```
      DEPTS←'J88' 'R42' 'M75' 'Q25' 'HSC'
```

These are, of course, only sample expressions. The checking may be as extensive or special purpose as desired. If you require integers in a certain range, you merely write a function that checks the range:

```
      ∇Z←RANGE N
[1] Z←(N≤UPPER)∧(N≥LOWER)∧(I N)
```

This gives a 1 if *N* is within limits and an integer and a 0 otherwise. The checking expression that would use this function would look like this:

```
      RANGE X
```

7.2: Evaluation of checking expressions

Given the checking expressions and the proposed values for a new row, application of an expression to its corresponding value will return either a 0 or a 1. If all applications return 1, the row is acceptable and may be INSERTED into the database.

Here's a general function (*OK*) which, given a checking expression as left argument and a value as right argument, will apply the expression to the value

```
     ∇Z←EXP OK X;R;E
[1]  (R E Z)←⎕EC EXP   ⍝ apply the expression
[2]  Z←(0 Z)[1+1=R]    ⍝ answer is result or zero
[3]  ⎕ES (1=↑E)/E      ⍝ propagate resource errors
     ∇
```

Line [1] executes the expression. Recall that ⎕*EC* is like "execute" accept that it returns a return code, error code, and result. Here each of the three arrays is given a name. A 1 is expected as the return code (meaning ordinary expression with a result) and line [2] returns the computed result if the return code is 1 and returns 0 otherwise meaning that the data is unacceptable. Line [3] makes sure that an error caused by lack of some system resource is not interpreted as bad data. Notice that the right argument of the function is *X* which is the correct name for the checking expression.

The third row of *FORM* (*FORM*[3;]) contains the four checking expressions for a row of *WHO*. To check the data in *NEW*1 you could enter the four expressions:

```
      FORM[3;1] OK 'MORTON,J'
1
      FORM[3;2] OK 3270
1
      FORM[3;3] OK 'BEAUCOUP'
0
      FORM[3;2] OK 'J88'
1
```

But there are four checking expressions and four values to be checked so the "each" operator may be used to apply between corresponding checking expressions and values:

```
   NEW1←'MORTON,J'  3270 'BEAUCOUP' 'J88'

   FORM[3;] OK¨ NEW1
1 1 0 1
```

This operation may be pictured like this:

```
  ┌──────────────────┐
  │'1' OK 'MORTON,J' │
  └──────────────────┘
   1
          ┌───────────────────────┐
          │'(X=⌊X)∧(X>0)' OK 3270 │
          └───────────────────────┘
           1
                 ┌─────────────────────────────┐
                 │ '(X=⌊X)∧(X>0)' OK 'BEAUCOUP'│
                 └─────────────────────────────┘
                   0
                             ┌───────────────────────┐
                             │'(⊂X)∊DEPTS' OK 'J88'  │
                             └───────────────────────┘
                              1
```

In this example, *NAME*, *ID*, and *DEPT* were OK but *SALARY* was not. Because *SALARY* was character data, the expression (*X*=⌊*X*)∧(*X*>0) got a *DOMAIN ERROR* which was trapped by ⎕*EC* which returned a return code of zero (meaning error). Therefore, *OK* returned a zero for that item.

If you only want a single answer saying "yes" or "no" for the whole row, apply the "and" function between the four values with the "reduction" operator:

```
      ∧/FORM[3;] OK¨ NEW1
0
```

∧/ will return 1 only if every item of its argument is a 1. *NEW*1, as previously discussed, would be rejected by the database as well because the error is a class 1 error. This is not true of *NEW*2.

```
   NEW2←'LATTERMANN,D'  5150.95 31000 'HSC'

   FORM[3;] OK¨ NEW2
1 0 1 1
   ∧/FORM[3;] OK¨ NEW2
0
```

This time the database would allow the row and truncate the user ID to an integer. The application, however, rejects it because only integers will be accepted.

```
   NEW3←'POLGAR,E' 2741  10 'OWN'

   FORM[3;] OK¨ NEW3
1 1 1 0
   ∧/FORM[3;] OK¨ NEW3
0
```

Here there are no bad data types and no conversions that go wrong so the database would surely accept the data. The application still rejects it this time because the *DEPT* is not one of the valid departments.

```
   NEW4←'WINTON,S' 8775  48500  'J88'

   FORM[3;] OK¨ NEW4
1 1 1 1
   ∧/FORM[3;] OK¨ NEW4
1
```

Finally, here's someone who's got all the data correct and this data is acceptable to the database and to the application.

Thus, by using character data as representations of checking expressions and the "each" operator to apply them in parallel to sets of values, the application has a trivial, yet completely general, way to apply any desired degree of domain checking to relational data before it gets into the tables.

8: Conclusion

This paper has shown the design and implementation of a multiple user server which may be used as the core of a multiple user application. The program can be used as is or modified to suit a particular need.

Using these concepts, the *APL2* application writer who has authorization to use a database, may distribute an application which makes use of the database without requiring that each user also have database authorization. The application can easily provide any level of authority or range checking on users and data. His application is safe in that users never have access to the code -- it's never in their virtual machine or address space.

The result is the ability to build new applications or add function to existing applications with little investment of time and effort.

9: Appendix 1: Related defined functions

9.1: A Main function for the server

The function *MAIN* defines the global variables that the rest of the functions in the server uses then starts the server.

```
    ∇ MAIN
[1]   MYNODE←'STLVM20'       ⍝ define node where server runs
[2]   RLIST←0 4⍴''           ⍝ empty REMIND list
[3]   OFFNO←1                ⍝ initial offer number
[4]   T←3 10 ⎕NA '∆FV'       ⍝ access file writing function
[5]   APPL←⊂'REMIND'         ⍝ for now one application
[6]   SERVER2 60×60          ⍝ wait for one hour
    ∇
```

MYNODE is used to determine if a REMIND message should be sent to a user as a message or a file. *RLIST* is the initial REMIND list. In a real implementation, the REMIND list would need to be a file so reminders are not lost over restarts of the system. *∆FV* is a file reading and writing primitive accessed via the external name facility (⎕*NA*). *APPL* is the list of supported applications. For the example, only *REMIND* is implemented. To add another application, make *APPL* a two item (or longer) vector containing the new application name then write a niladic defined function with that name. The arguments to the application will be found in the global variable *ARG*.

9.2: RETRACT and SHARE

The *SHARE* function is called for each account number returned by the ⎕*SVQ* in the *SERVER* function. Thus the argument is a single account number.

```
    ∇ SHARE PROC;HISNAMES
[1]  ⍝  share all variables offered by user PROC
[2]   PROC DOSHARE¨⊂[2] ⎕SVQ PROC
    ∇
```

⎕*SVQ* gets the list of names offered to me by user *PROC* and calls *DOSHARE* for each of them.

Therefore the argument to *DOSHARE* is a single name.

```
    ∇ PROC DOSHARE HISNAME;T  ⍝ share one name
[1]   HISNAME←HISNAME~' '      ⍝ remove extraneous blanks
[2]   MYNAME←'⍙',⍕OFFNO        ⍝ construct a unique name
[3]   T←PROC ⎕SVO MYNAME,' ',HISNAME   ⍝ accept the share
[4]   T←0 0 1 1 ⎕SVC MYNAME    ⍝ set access control
[5]   OFFNO←OFFNO+1            ⍝ update offer sequence number
    ∇
```

This server application will accept any name that the user wishes to offer. A unique name is constructed (on lines [2] and [3]) of the form ⍙51 where ⍙ is a unique character that this application only uses in the names of shared variables and 51 means that this is the 51st offer accepted. Thus even if several users offer the same name, the server will always have a unique name.

The function *VARS* returns, as a vector of vectors, the list of all names that begin with the letter '⍙' and therefore the names of all shared variables.

```
    ∇ Z←VARS      ⍝ return the names of all shared variables
[1]  Z←⊂[2] '⍙' ⎕NL 2
    ∇
```

The *RETRACT* function erases all shared variables. Since this is called only when the partner has retracted on his side, it will effectively terminate all sharing.

```
    ∇ RETRACT VAR    ⍝ erase shared variable
[1]   VAR←⎕EX VAR
    ∇
```

9.3: REMIND

The previous functions presented are general and will run on any *APL2* system. The functions in the *REMIND* facility are designed to run with APL2 Release 2 in CMS and would need to be modified to run in TSO.

The value given to the shared variable should be the word remind, USERID, NODEID, a *⎕TS* style timestamp, and a character vector message (see the text for an example).

The *REMIND* facility is composed of two main functions. *REMIND* gets control when someone sends a request to be reminded at a given time. *DOREMIND* gets control when it is time to send the reminder.

Here the *REMIND* function saves the information about the message in a nested array in the workspace. The array is ordered in time so that the first row will be the first message to be sent. A real application would save the data on a file or in a relational table.

```
    ∇ Z←REMIND R        ⍝ save a remind request
[1]   ⍝ R is  userid nodeid timestamp message
[2]    (3⊃R)←CODETIME 3⊃R              ⍝ change time to minutes
[3]    RLIST←RLIST,[⎕IO]R              ⍝ add request to queue
[4]    RLIST←RLIST[⍋RLIST[;3];]        ⍝ put closest on top
[5]    WAIT←WAIT⌊RLIST[1;3]            ⍝ get time to next event
[7]    Z←'OK'                           ⍝ acknowledge receipt
    ∇
```

DOREMIND checks for messages which are due to be sent, and calls *SENDMESSAGE* for each of them. These messages are then removed from the list.

```
    ∇ Z←DOREMIND TIME;CURRENT;READY;MASK;T
[1]   ⍝ send message to anyone whose time has come
[2]    CURRENT←CODETIME ⎕TS          ⍝ find current time
[3]    MASK←∊CURRENT≥RLIST[;3]       ⍝ locate messages ready
[4]    READY←MASK⌿RLIST              ⍝ select ready messages
[5]    SENDMESSAGE¨⊂[2]READY         ⍝ send ready messages
[6]    RLIST←(~MASK)⌿RLIST           ⍝ delete messages just sent
[7]    →(0=↑⍴RLIST)/0  Z←TIME        ⍝ return if no reminders
[8]    Z←TIME⌊(RLIST[1;3]-CURRENT)×60
    ∇
```

SENDMESSAGE tries to send a message to a signed on user. If this fails, then a reader file is sent instead.

```
    ∇ SENDMESSAGE UM;USERID;NODEID;MESSAGE;T
[1]   ⍝ send MESSAGE to USERID at NODEID
[2]    (USERID NODEID T MESSAGE)←UM
[3]    →(NODEID≡MYNODE)/REMOTE
[4]    T←TOHOST 'TELL ' USERID ' AT ' NODEID ' ' MESSAGE
[5]    →(T=0)/0    ⍝ exit if message was sent
[6]   ⍝
[7]   ⍝ user is not signed on
[8]   ⍝ write a file and send it to him
[9]   REMOTE:T←WRITEONE MESSAGE  ⍝ write one record file
[10]   T←TOHOST 'SENDFILE ARB MESSAGE A ' USERID ' AT ' NODEID
    ∇
```

For simplicity, all timestamps are kept as a simple integer.

```
    ∇ Z←CODETIME T        ⍝ build a compact timestamp
[1]   Z←0 12 31 24 60⊥(100|↑T),1↓5↑T
    ∇
```

WRITEONE writes the character vector *M* to a one record file named ARB MESSAGE. This file is then sent to a user not currently signed on. The function *ΔFV* is an external function defined by entering 3 10 ⎕*NA* '*ΔFV*' and is a function that will take a whole array and write it as a file.

```
    ∇ Z←WRITEONE M                    ⍝ write a one record file
[1]   Z←M ΔFV 'ARB MESSAGE A'
    ∇
```

TOHOST sends a command to the operating system.

```
    ∇ Z←TOHOST R;AP100     ⍝ send command R to CMS
[1]   AP100←'(EBCD'
[2]   Z←100 ⎕SVO 'AP100'
[3]   AP100←∊R
[4]   Z←AP100
    ∇
```

9.4: REPEAT

```
    ∇ (F REPEAT) R
[1]  ⍝ REPEAT FUNCTION F FOREVER
[2]  L:F R       ⍝ call the function
[3]   →L         ⍝ loop back and call it again
    ∇
```

This defined operator calls monadic function *F* with any argument *R*. When *F* completes, it is called again. It is like a structured programming construct DO FOREVER. Because this is a defined operator, you can see that it contains a loop. If it were a primitive operator, like "each", the loop would be buried in its definition. Most structured programming constructs are methods for phrasing loops.

10: Appendix 2: Shared variable considerations for TSO

The global shared variable processor (GSVP) is an MVS subsystem. Values of shared variables are passed between users by means of shared memory which is allocated in the Common Service Area (CSA).

Access to the GSVP is based on a user's account number (↑⎕*AI*) This number is selected by the *APL2* user when he starts the *APL2* session. To insure security, it is required that an installation control access to the GSVP by providing an exit module whose name is specified in the ISECNAME parameter in the GSVP start up parameter file. This module should be used to grant or deny access to the GSVP services. This module is invoked once at the start of the session. A second exit module whose name is specified in the GSECNAME parameter in the GSVP start up parameter file is invoked every time a user signs on to the GSVP.

Shared memory is secure and one user cannot see values intended for other users.

See APL2 Installation and Customization under TSO (SH20-9222) for more information (3).

11: Appendix 3: Shared variable considerations for CMS

The global shared variable processor (GSVP) runs as a CMS service machine. Values of shared variables are passed between users in writable shared segments (DCSS). Synchronization signals are passed via Virtual Machine Communication Facility (VMCF).

Access to the GSVP is based on a user's account number (↑⎕*AI*). This number is selected by the *APL2* user when he starts the *APL2* session. It is recommend that an installation control access to the GSVP by providing an exit to be invoked by the service machine in the form of a CMS command named AP2SVPEX. This command may be used to grant or deny access to the SVP services for this session. If access is granted, the DCSS is made available. In addition, the exit may provide or alter the account number to be used for this session. This will also provide the number reported in ⎕*AI*.

Even if the GSVP chooses to deny access to the DCSS, it can still respond with the account number to be used. Thus, even if you do not intend to use global shared variables, you could use the GSVP to enforce user numbers. In this case no DCSS need even be defined.

Note that, in CMS, once the DCSS is available, a user has free access to any part of the shared memory. If one set of users will be cooperating among themselves, you may want to provide a DCSS only for them. Then other users sharing with each other through another DCSS cannot see this one. Any number of DCSS's may be defined at the same address with different names. This provides security at the maximum level provided by CMS -- 8 character passwords.

See APL2 Installation and Customization under CMS (SH20-9221) for more information (4).

11.1: GSVP failures

The GSCP is easy to install and under normal conditions does not require any special attention. Occasionally, a problem in installation or in the exits can cause the GSVP to fail. This section discusses some common causes for these failures.

Two kinds of failures in the GSVP can happen:

- failure trapped successfully by CMS leaving CMS active.
  In this case the service machine will enter VM READ and be subject to force off after a fixed period of time (normally 30 minutes).

- failure that causes CMS to fail
  In this case the service machine will enter CP READ and be forced off.

Except for these failures the service machine should always be running.

The GSVP service machine only runs when a new user tries to sign on or to respond to an operator command. If there is no installation exit, then almost no code is executed and a failure is unlikely. If there is an installation exit then a failure in this exit could cause either of the above two failures.

If you have such a problem, it is recommended that you spool the console of the service machine. You may also want to set CP TRACE on for external interrupts and program checks (CP TRACE EXT PROG RUN). After the service machine has issued "AP2CSVP START ..." issue the following commands to establish the base configuration:

```
Q CMSLEVEL
NUCXMAP ALL
AP2CSVP QUERY
```

Looking at the console log should help you discover the source of the failure.

If you get the message "SHARED PAGE ALTERED" or a protection exception, it may be because of incorrect installation. Make sure that in building the DCSS that PROTECT=OFF is coded on the NAMESYS MACRO. This is not the normal default.

Don't let the GSVP server machine get forced off by any automatic monitoring schemes in use.

Be sure the following SETs are in affect:

```
SET AUTOREAD OFF
SET RUN ON
```

12: Appendix 4: Some Shared Variable usage hints

The use of shared variables between sessions assumes that the global shared variable is available. A program can check that the GSVP is running by issuing the following:

```
      3 10 ⎕NA 'SVI'
1
      SVI 0
AP2SVP
```

⎕*NA* is used to associate the name *SVI* with an external routine that returns information about shared variable. A response of 1 means that the name has been associated. The function *SVI* with argument 0 is a request to return the ID of the GSVP. If it returns a non-empty character vector, then the GSVP is active and sharing is allowed. If the result is empty, then an application may choose to wait for a while (⎕*DL* 5×60) and then check again. This can be the case if the GSVP and applications are brought up automatically with an IPL. It is possible that the application will be ready to run before the GSVP is ready. A short wait is probably enough to ensure that the GSVP is ready.

*APL*2 uses account numbers to identify users. The installation can enforce the assignment of account numbers to userids (see Appendices 2 and 3). An application can determine the logon id of a user given his *APL*2 account number as follows:

```
      SVI 33586
BROWN
```

Thus, given the account number as reported by ⎕*SVQ*, the application can determine the system logon ID. This is useful in applications like *REMIND* where the logon ID is needed to send a message to a user.

13: Acknowledgements

The original *SERVER* function was written by Mike Wheatley in preparation for an IBM Internal class in *APL2*. Modifications of the programs were implemented by David McClanahan as part of a working multi-user application in use by *APL* development.

14: References

(1) Brown, J.A., Crowder, H.P., "APL2: Exploiting DB2 and SQL/DS", IBM Santa Teresa Technical Report TR 03.267, July 1985.
(2) APL2 Programming: Using Structured Query Language (SQL), IBM Corp., 1984, SH20-9217
(3) APL2 Installation and Customization under TSO, IBM Corp., 1984, SH20-9222
(4) APL2 Installation and Customization under CMS, IBM Corp., 1984, SH20-9221

The following references are general references for *APL*2.
(5) APL2 General Information, IBM Corp., 1984, GH20-9214
(6) An Introduction to APL2, IBM Corp., 1984, SH20-9229