Technical Report

April 1976 07.607

IBM 5100 SHARED VARIABLE PRIMER

Andrew J. Cubbon

Communications General Systems Division Rochester, Minnesota

07.607

IBM 5100 SHARED VARIABLE PRIMER

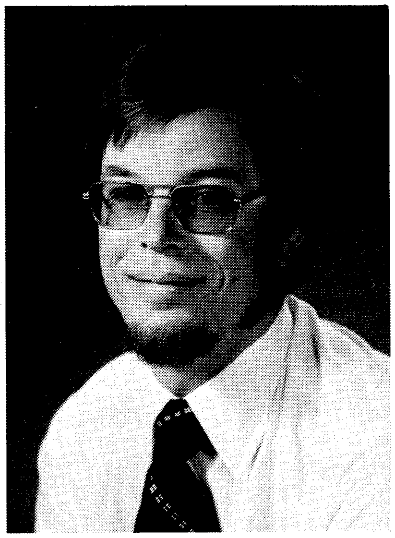
Andrew J. Cubbon

This primer is designed to supplement the IBM 5100 APL Reference Manual (SA21-9213) Chapter 8. It begins with several simple exercises using APL Shared Variables. Following the exercises is a more detailed discussion of I/O using Shared Variables in question and answer format.

Some of the topics covered are:

– How Shared Variables work.
– How to read and write tape.
– How to copy data files.
– How to use the printer.
– How to exchange data with BASIC and Communications.
– How to close input files.
– How to avoid some common errors.

Communications General Systems Division Rochester, Minnesota



Andrew J. Cubbon is a Staff Marketing Support Representative in the 5100 Marketing Support Center. He received a BS degree in 1969 from the University of Illinois and an MS in 1974 from the University of Minnesota.

Cubbon joined IBM in June 1969.

INTRODUCTION FOR IBM 5100 APL I/O USING SHARED VARIABLES

This report is designed to assist the new user of the IBM 5100 with APL Shared Variables. The only way to do device (tape, printer, etc) operations under program control in APL is to use Shared Variables. First, run through some simple exercises using Shared Variables. This should be followed by a detailed series of questions and answers.

Output to Tape

This exercise will write data to the inboard tape unit.

1 Mount a data cartridge.

2 Mark a small file.

```
      )MARK 1 1 1
```

3 Offer a Shared Variable.

```
      1 ⎕SVO 'SV'
2
```

4 Open the file for output.

```
      SV←'OUT  1'
```

5 Check the return code.

```
      SV
0 0
```

6 Write a character record.

```
      SV←'REC 1'
```

7 Check the return code.

```
      SV
0 0
```

8 Write a numeric record.

```
      SV←3 3 ⍴⍳9
```

9 Check the return code.

```
      SV
0 0
```

10 Close the file. The last (only) buffer will be written to tape.

```
      SV←⍳0
```

11 Check the return code.

```
      SV
0 0
```

Input From Tape

Read back the file just created.

1 Using the same Shared Variable and tape file, open the file for input.

```
      SV←'IN 1'
```

2 Check the return code.

```
      SV
0 0
```

3 Read the first record. The tape will read the first (only) block.

```
      SV
REC 1
```

4 Read the second record. There will be no tape movement as value is in the buffer.

```
      SV
1 2 3
4 5 6
7 8 9
```

5 Attempt to read a third record.

```
      SV
      Nothing (empty vector) indicates end-of-data
```

6 Check the close return code.

```
      SV
0 0
```

7 Retract the Shared Variable.

```
      ⎕SVR 'SV'
2
```

8 Erase the Shared Variable to avoid problems with its reuse.

```
      )ERASE SV
```

Copying and Printing a File

This exercise will copy the file you just created to the second tape and print each record.

1 Mount another tape in the second tape unit.

2 Mark a small file on the second tape.

```
      )MARK 1 1 1 2
```

3 Offer the Shared Variables, one each for input, output and the printer.

```
      1 ⎕SVO 3 3 ⍴ 'SV OSVPNT'
2 2 2
```

4 Open the printer.

```
      PNT←'PRT'
```

5 Open the input device.

```
      SV←'IN 1'
```

6 Open the output device with an ID.

```
      OSV←'OUT 2001 ID=(COPY)'
```

7 Check return codes.

```
      PNT
0 0
      SV
0 0
      OSV
0 0
```

8 Get first record.

```
      DATA←SV
```

9 Print it and check return code.

```
      PNT←DATA
      PNT
0 0
```

The printer will print:

```
REC 1
```

10 Write it to tape and check return code.

```
      OSV←DATA
      OSV
0 0
```

11 Get second record.

```
      DATA←SV
```

12 Print it and check return code.

```
      PNT←,(⍕ DATA),⎕AV[157]
      PNT
0 0
```

The printer will print:

```
1 2 3
4 5 6
7 8 9
```

13 Write it to tape and check return code.

```
      OSV←DATA
      OSV
0 0
```

14 Attempt to get third value.

```
      DATA←SV
```

15 Check shape of DATA.

```
      ⍴DATA
0               the zero (empty vector) indicates end-of-data
```

16 Erase the Shared Variables. Note that this will close them also.

```
      )ERASE SV OSV PNT
```

QUESTIONS AND ANSWERS ABOUT SHARED VARIABLES

Q: <u>Why Shared Variables?</u>

A: This is a frequently asked question. To put it another way, why did APL not use READ/WRITE or GET/PUT? The answer: because this would require the protocol for the various devices to become part of the APL language. Any new devices or changes to existing devices protocol would require a redefinition of that portion of the APL language. Also, APL has functions with only one or two arguments. I/O devices usually require many more, which would destroy the simplicity of APL's syntax.

Thus, the Shared Variable, a simple link to the outside world. None of the requirements of the I/O device are included in the language. This facilitates additions or changes in I/O configurations.

Q: <u>What is a Shared Variable?</u>

A: It is an ordinary variable with connections. That is, all the rules of ordinary variables apply: 1 to 77 character identifier, first character is alphabetic, can be used in APL expressions, etc. However, it is unlike ordinary variables when it is linked to the 5100's I/O devices. Values assigned to a Shared Variable are passed out to the selected device. When referenced, the value is provided by the selected device. <u>Each</u> separate reference gets a <u>new</u> value. If a value <u>is to be used more than once, it should be assigned to an ordinary variable</u>.

Examples:

The vector 3 4 5 is passed out to an I/O device via the Shared Variable SV:

```
      SV←3 4 5
```

The Shared Variable SV is referenced to get two records from an I/O device.

```
      SV
  1 2 3
      SV
 10 11 12
```

Q: <u>Are there any restrictions on their use?</u>

A: Although assigning and referencing are similar to ordinary variables, there are times when one or the other operation is not permitted. Depending on how you are accessing the I/O device, doing the wrong thing can cause an INVALID OPERATION error which causes an INTERRUPT to occur in the statement performing the incorrect operation. An example of this would be trying to assign a value (other than an empty vector) to a Shared Variable established for an input operation.

Q: <u>What about errors?</u>

A: If the data assigned to a Shared Variable is invalid, the 5100 does two things: It displays an error message, and puts a return code in the Shared Variable. The return code can be checked by the APL function doing the I/O operations. If no error is detected, a return code of 0 0 is placed in the Shared Variable. It is always good procedure to check the return code after an assignment operation. Remember, only one reference per value. Thus, the return code is only available for one reference unless it is assigned to an ordinary variable. The following example of checking return codes could be used:

If SV is the Shared Variable and has just been assigned a value,

```
[n] →(0≠+/SV)/0          exits the function if the
                         return code is not 0 0.

[n] →(0≠+/RT←SV)/ER      prints the return code if
                         not 0 0.
      .
      .
      .
      .
[m] ER:'ERROR CODE ',⍕RT
```

ESTABLISHING SHARED VARIABLES

Q: <u>How is a variable shared?</u>

A: As you have seen, you use Shared Variables to exchange data with the I/O devices of the 5100. Now we will discuss how to establish the connection. This is done by offering a variable to be shared. The offer is done with the Shared Variable Offer system function ⎕SVO. ⎕SVO requires two arguments: the number 1 and the name of the variable to offer. The 1 indicates the variable is to be used for I/O operations*. The variable's name is a character argument to ⎕SVO. For example:

```
          1 ⎕SVO 'SV'
```

offers SV for I/O operations.

Q: <u>What is the result of ⎕SVO?</u>

A: The <u>result</u>, not return code, of this system function is:

```
     0 - invalid variable name,
         invalid left argument, or
         already eight variables being shared

     1 - 1 was not the left argument of ⎕SVO

     2 - the variable was accepted.
```

Remember, this is a result of the ⎕SVO function just like 7 is the result of the + function if 3 and 4 are its arguments. It can be tested or used the same as the result of the +.

Q: <u>Can we share more than one variable?</u>

A: Yes, up to eight. They may be offered one at a time or several variables may be offered at the same time. To do the latter, the right argument of ⎕SVO is a character matrix whose rows are the names of the variables to be offered. The result is a vector of numbers, one for each variable offered. For example:

```
          1 ⎕SVO 2 3 ρ'SV PNT'
     2 2
```

* For purists, the 1 is the "processor number" used on larger systems with multiple users.

Offers SV and PNT which were both accepted. This could also have been done as two separate offers:

```
      1 ⎕SVO 'SV'
2
      1 ⎕SVO 'PNT'
2
```

Q: <u>Why offer more than one variable?</u>

A: If you want to operate more than one device at a time, you need one Shared Variable for each device. For example, if you are reading records from the tape and printing them on the printer, you would have one Shared Variable for the tape and one for the printer.

Q: <u>How do we check if a variable is shared?</u>

A: If you have been doing a lot of experimenting with Shared Variables, you may get to the point where you don't remember if a variable is shared or not. This can be checked using ⎕SVO with only one argument (monadic). The result is the status of the right argument which is a variable name or a matrix of names. For example:

```
      ⎕SVO 'SV'
2
```

tells you that SV is currently being shared.

Q: <u>What if we don't know the variable's name?</u>

A:
```
      ⎕SVO ⎕NL 2
```

Checks all the variables in the workspace. If one of the elements of the result vector are not equal to 0, the corresponding row in the ⎕NL 2 matrix of variable names is the name of a Shared Variable.

Q: <u>What happens if an existing variable is offered?</u>

A: If a variable already exists in the ordinary sense and is then offered, its value is used by the 5100 to open the I/O device (discussed in the next section). Therefore, if the variable that is offered is something other than a valid Openlist, an error message will occur during the offer. For example:

```
          A←3
          1 ⎕SVO 'A'
     INVALID DATA TYPE
     2
```

The share is successful (the 2) but the open attempt is not, so an error message is displayed. The 5100 is still expecting a valid Openlist. If one is now assigned to A, the I/O operations will proceed normally.

This error may be avoided by erasing the variable prior to sharing it. Further measures will be discussed in a later section.

OPENING I/O DEVICES

Q: <u>How are the I/O devices opened?</u>

A: After the Shared Variable has been accepted by the 5100, we must communicate our I/O requirements. For example, do we want to do input or output, to what file, shall we assign an ID, what type should the file be, etc. These and other characteristics are established with an Openlist. An Openlist is a character string (vector) that the 5100 is expecting <u>first</u> and only once.

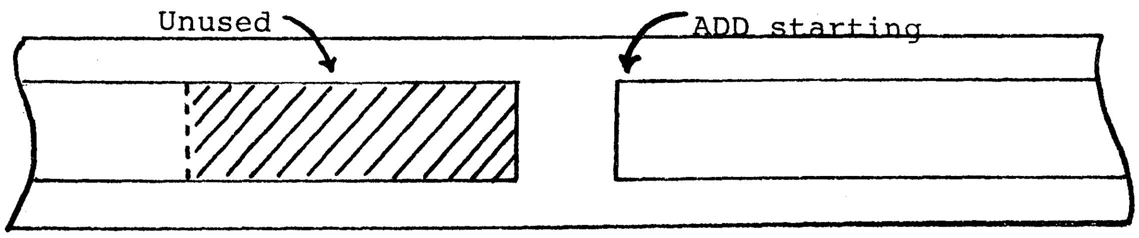
Q: <u>What do we do first?</u>

A: The first parameter tells the 5100 what we want to do:

IN	- Input from the I/O device
OUT	- Output to the I/O device
ADD	- Add records to an existing tape file
PRT	- Output to the 5103 Printer

Q: <u>Does ADD use the same amount of tape as OUT?</u>

A: No, generally it will use more! The more times you open a file for ADD, the more extra tape is used. The cause is ADD starting a new physical block when the file is opened, skipping the unused part of the previous ending block.



Q: <u>How do we specify which device and file?</u>

A: The next parameter specifies the device and file number (not used with PRT):

ddfff	dd - device number
	fff - file number

For example:

2001	tape 2, file 1
3	(tape 1), file 3

Thus:

SV←'IN 3' opens tape 1 file 3 for input.
SV←'OUT 2001' opens tape 2 file 1 for output.
SV←'ADD 17' opens tape 1 file 17 to add records to end of existing file.

<u>The remaining parameters are optional when they are valid</u>.

Q: <u>How about file identifiers?</u>

A: ID=(identifier) — the identifier can include any 0 to 17 characters except a right parenthesis.

For example:

```
ID=(FILE-1)
ID=(10/12/75)
ID=(SALES MAR)
```

The ID parameter has different results when used with the different I/O operations:

IN & ADD — The value specified is checked against file header ID. If not the same, the open attempt fails--INVALID FILE. If the ID parameter is not used, no checking occurs.

OUT- Case 1: a marked but unused file (type 0). The specified ID is written in the file header. If the ID parameter is not used, an ID of DATA is used.

Case 2: A previously used data file (types 1, 2, or 8). The value specified is checked against the file header ID. If not the same, the open attempt fails--INVALID FILE.

Case 3: A previously used file (not types 1, 2, or 8). Open fails--INVALID FILE.

Note: In order to overwrite a file with a new ID or nondata file type, it has to be dropped via the)DROP command.

PRT- The ID parameter is invalid.

For example:

> SV←'IN 1 ID=(FILE-1)' opens file 1 for input and checks to see if the header ID is FILE-1.

Q: <u>What kinds of data can be transferred to or from an I/O device?</u>

A: The kinds of data that can be written out to or read from I/O devices depends on the I/O device and (if applicable) the file type specified. The tape will accept all types of data. The printer (PRT) will only accept single characters or strings of characters. More details will be covered in a later section.

Q: <u>What about specifying file type?</u>

A: TYPE=t where t is:

```
 I - for data exchange (type 1)
I1 - for data exchange (type 1)
I2 - for general exchange (type 2)
 A - for APL internal (type 8) this is the default
     value.
```

For example:

```
TYPE=I
TYPE=I2
```

This parameter is only valid for OUT operations. Other operations get the type from the I/O device.

For example:

> 'OUT 3 TYPE=I' opens tape 1 file 3 as a data exchange file

<u>For TYPE=I, I1, or I2 files</u>, data must be single characters or character strings. Other forms are invalid and will cause an INVALID DATA TYPE error. The Ravel , and Format ⍕ functions can be used to convert other forms into character strings (vectors). For example, if:

```
A ← 3 4 5  it can be outputted as  ⍕A
A ← 2 3 ⍴ 'ABC'  it can be outputted as  ,A
```

When reading numeric data in from an exchange file, it can be converted back to numeric with the Execute ⍎ function. For example, if SV is the Shared Variable then:

```
A←⍎SV  will cause A to get the numeric values of the
       input characters.
```

TYPE=A files will accept any form of data in any shape. No conversion is necessary.

Q: What about BASIC source (type 3) files?

A: Type 3 files can be read by APL but cannot be created. To create a BASIC source file, use TYPE=I2 which can be loaded as either source or data by BASIC.

Q: Can we get rid of the error messages?

A: MSG=OFF This parameter may be used with IN, OUT, ADD, and PRT. It (once processed) prevents the displaying of error messages.

If this parameter is used, it is more important that the return codes from the I/O operations be checked to prevent errors from going unnoticed.

For example:

```
SV←'OUT 1 MSG=OFF' opens tape 1 file 1 for output and
                   inhibits error messages
```

Q: How do we know it is opened?

A: Once the open is complete you should check the return code. To do this, you have only to reference the Shared Variable. For example:

```
[n]   SV←'OUT 3'
[n+1] →(0≠+/SV)/0   exits function if an error occurred.
```

TRANSFERRING DATA

Q: How do we get data from the I/O device?

A: Now that you have established the link to the I/O device and have told it what you want to do, how do you do it? Let us discuss getting data from the I/O device.

Before you can get any records, you will always get the return code from the open.

```
      SV
0 0
```

There are several techniques you can use to get the data from the I/O device. You could simply key in the Shared Variable and press execute:

```
      SV
record 1
      SV
record 2
```

This method is useful if you are just checking the first few records to see, for example, if they are the correct format. However, for larger amounts of data, this becomes tedious.

Q: What about an input function?

A: This leads to another method. You could write a function to repetitively reference the input variable and perhaps process, display, and/or print each input record. For example:

```
[n]    LOOP: SV
[n+1]  →LOOP
```

The problem with the above function segment is that it never stops. When there is no more input data, the I/O device will be closed. An Openlist will be expected, not getting it, it will repeatedly display:

```
DEVICE NOT OPEN
8 0
```

Q: How can we prevent this error?

A: To prevent this, a check should be made for end-of-data. End-of-data is signalled by an empty vector. The above segment could be rewritten:

```
[n]    LOOP: →(0≠⍴SV)/LOOP
[n+1]   'END-OF-DATA'
```

Here line n is executed until the shape of the Shared Variable is 0, indicating end-of-data. Of course, the sample does not do anything with the values read. They could also be assigned to ⎕ to display them:

```
[n] LOOP: →(0≠⍴⎕←SV)/LOOP
```

Q: <u>What if we want to process the input values?</u>

A: They should be assigned to an ordinary variable for processing.

```
[n]    LOOP: →(0=⍴ORD←SV)/0
[n+1]   ORD + 12
[n+2]   →LOOP
```

Q: <u>How can we copy data to another device?</u>

A: Input records could be assigned to an output Shared Variable (OSV) to copy data from one device to another:

```
[n] LOOP: →(0≠⍴OSV←SV)/LOOP
```

The problem with the above function segment is that the return codes from the output operations are never checked. This could be done:

```
[n]    LOOP: →(0=⍴OSV←SV)/ENDOFDATA
[n+1]   →(0≠+/OSV)/0
[n+2]   →LOOP
[n+3]  ENDOFDATA:
```

Where line n+1 exits the function if the return code indicates the output operation was not successful.

Q: <u>Does an empty vector always indicate end-of-data?</u>

A: No, if the file is an exchange file (file types 1, 2, or 3), it is possible to get an empty vector that is not at end-of-data. This occurs if two new-line characters (⎕AV[157]) were written on the media when it was created. This condition can be distinguished from an end-of-file by checking the Close return code. The Close return code is available at the next reference after an empty vector. In this case, its 9 0 instead of 0 0. If the Close return code is 9 0, you can continue processing data. Note: If you are copying, writing out any empty vector to an output Shared Variable closes it.

Q: <u>What about errors?</u>

A: Errors during input are also indicated by an empty vector which indicates that the device is closed. The return code specifying the error is, like the real empty vector return code just discussed, available on the next reference to the Shared Variable. A value of 0 0 indicates a true end-of-data has been reached. Error messages will also be displayed.

Q: <u>How do we close an input device?</u>

A: It is not necessary to explicitly close input devices if end-of-data is reached. In fact, doing so will cause an error. However, if the file is to be reopened, or another file on the same device is to be opened, before end-of-data is reached, an explicit close must be done. To do this you assign an empty vector to the Shared Variable:

```
SV←⍳0
```

There are other implicit ways of closing which will be discussed in later sections.

Q: <u>How do we write data to an I/O device?</u>

A: You saw how to do output with the copy operation. In that example, the input Shared Variable, SV, was assigned to the output Shared Variable, OSV. Remember also, that we checked the return code to insure that the output operation occurred successfully:

```
[n]    LOOP: →(0=⍴OSV←SV)/ENDOFDATA
[n+1]   →(0≠+/OSV)/0
[n+2]   →LOOP
[n+3] ENDOFDATA:
```

It is important that you <u>close</u> output devices. This ensures any partial blocks will be written to the I/O device. This can be done by assigning an empty vector to the Shared Variable as you did for input devices:

```
        OSV←⍳0
or
        OSV←''
or
        OSV←0⍴1
```

Q: <u>What other ways are there?</u>

A: If you retract (⎕SVR) the Shared Variable before it is closed, it will be closed. If you)ERASE or expunge (⎕EX) a Shared Variable, it will be retracted and closed. If you localize the variable, it will be closed when the function is exited. However, with these techniques, you cannot check if the close was successful, since the return code is not available.

Q: <u>What about errors during a Close?</u>

A: As with input files, after the explicit close (assigning an empty vector), the next reference of the Shared Variable will yield the return code of the Close. A 0 0 indicate a successful close. It is more important to check output closes because a close failure may make the file <u>unusable</u>. Of course, error message will occur (unless MSG=OFF was specified) during the implicit closes even though no return codes are available.

DATA TYPES

Q: <u>What kinds of data can be output?</u>

A: This usually depends on the type of I/O device being addressed. For example, the tape cartridge will accept any type of data when writing to an APL internal (type 8) file. The printer and exchange files (types 1 or 2) require the data be a single character or strings of characters.

Q: <u>Why use exchange files?</u>

A: Exchange files, as the name implies, are for data exchanging (type 1) with BASIC on the 5100 or the Communications feature for transmission to a remote system. The general exchange file (type 2) is for exchanging BASIC source. APL will read a BASIC source file (type 3) but cannot create one.

Q: <u>How is the printer different?</u>

A: The printer is very similar to the exchange files in that it accepts only single characters and strings of characters. It differs though, in that it prints the data when the assignment occurs, rather than waiting for a full buffer. Because of the implicit new line, an actual new line at the end of the character string has no effect. To get multiple spacing, just add an extra new line character. For example, the following will do double spacing:

```
P←'ABC',2⍴⎕AV[157]
```

Q: <u>What if we want to exchange (type 1 or 2) or print numeric data?</u>

A: This problem can be solved using the Format ⍕ function which converts its right argument into characters. If we wanted to exchange or print the numbers in the variable A.

```
SV←⍕A
```

would convert it to an appropriate character string (vector).

Q: What if we want to exchange (type 1 or 2) a matrix?

A: This can be accomplished with the Ravel , function. This function will convert any right argument into a string (vector). For example, the following will output the character matrix variable A:

```
SV←,A
```

Note, however, that the shape (number of rows and columns) is not retained. We could output the shape separately as follows:

```
SV←⍕⍴A
```

Q: What about printing a matrix?

A: You could use Ravel here but this will cause the matrix to be printed as a string of values. To retain the matrix shape when it is printed, you could print it row at a time. However, row at a time does not take advantage of the matrix handling capabilities of APL. A better method involves catenating new-line characters to the end of each row prior to raveling it. If A is a numeric matrix, the following will cause it to print correctly.

```
A←3 4 ⍴⍳12
P←,(⍕A),⎕AV[157]
```

Q: What about data for BASIC?

A: Some important rules you must keep in mind for data exchanged with BASIC on the 5100. First, all data items must be separated by commas--not blanks. Character data must be enclosed in quotes and cannot exceed 18 characters per item. For example:

```
'THIS CHARACTER ITE','M IS TOO LONG'
```

The above item was broken into two parts. Negative signs have to be converted to minus signs. Finally, the length of the entire record should be kept under 54 characters including commas and quotes if it is to be LOADed in BASIC. If it is only to be accessed via 'GET' there is no arbitrary limit to its size.

Q: <u>What about reading exchange data?</u>

A: Since we are discussing exchange data, a few points about reading exchange files should be made. First, the data will come back as character vectors. Numeric data can be reconverted using the Execute ⍎ function:

```
A←⍎SV
```

If the file was created by BASIC, the Execute function can be used to remove the commas (which are treated as the Catenate function) and quotes if the data is <u>not mixed</u> numeric and character. Remember to translate the minus signs to negative signs before converting numeric data.

```
[n]     ORD←SV
[n+1]   ORD[(ORD='-')/⍳⍴ORD]←'¯'
[n+2]   ORD←⍎ORD
```

Q: <u>Will exchange data read the same as it was written?</u>

A: Not always! If you included any new line characters in the character strings, when they were written, they will be interpreted as end-of-record marks. This means they will not be read back and the original record will be segmented. For example, if this string was written on an exchange file:

```
SV←'ABC',⎕AV[157],'CDE'
```

will be read back as two separate logical records:

ABC and CDE

Note: The ⎕AV[157] was not read.

AFTERWARDS

Q: What happens when we are done?

A: Several things can occur when the I/O operation is complete. You'll remember that if end-of-data occurs on input, the file closes. Or, we can close a file by assigning an empty vector to the Shared Variable.

Q: What happens to the Shared Variable after the file is closed?

A: It remains in existence as a Shared Variable. It is as if no I/O had occurred. It is waiting for an Openlist. That is, we can use it again for another I/O operation.

Q: What if we do not want to share anymore?

A: If the Shared Variable is no longer needed, it can of course be)ERASE'ed for expunged (⎕EX). If you wish to keep the variable but return it to an ordinary variable, you can Retract it:

```
      ⎕SVR 'SV'
2
```

The result is the same as the result of ⎕SVO when the variable was offered. Just like ⎕SVO, ⎕SVR will accept a character matrix as an argument:

```
      ⎕SVR 2 3 ⍴'SV OSV'
2 2
```

resulting in a numeric vector.

Q: What if we do not know the variable's name?

A:
```
      ⎕SVR ⎕NL 2
```

will retract all the variables in the workspace. If a variable is not shared, it's status will not be affected.

Q: What happens if the I/O device is still open?

A: It will be closed. However, the success of the close cannot be checked, because the return code is not available.

Q: <u>What is the value of a retracted Shared Variable?</u>

A: The last value it had. This value is left in the variable and will change only when a new value is assigned.

Q: <u>What happens if we)ERASE a Shared Variable?</u>

A: If a Shared Variable is erased or expunged, it is retracted. If the I/O device is still open it is closed. However, the success of the close cannot be checked, because the return code is not available.

Q: <u>What about ⎕SVC and ⎕SVQ?</u>

A: These functions are of limited meaning on the 5100. They are more applicable to multiple, independently controlled processors.

They have been retained only to maintain consistency with APL/SV on larger systems. Their results are covered in Appendix D of the IBM 5100 APL Reference Manual. Their meaning on larger systems is discussed in the APL Shared Variable User's Guide (SH20-1460).

PROBLEMS

Q: <u>What are some common SV problems?</u>

A: The following questions will cover some of the common Shared Variable problems. These problems are easily avoided and will cease to bother you once you get used to using Shared Variables.

Q: <u>What causes an INVALID DATA TYPE error during the offer?</u>

A: Sometimes you will get INVALID DATA TYPE or INVALID PARAMETER errors when you offer a variable (⎕SVO). This is caused by offering an existing variable that already has a value, if that value is not a character vector or valid Openlist.

To avoid these errors, it is good procedure to offer only new variables. To assure this, you should erase all Shared Variables when they are no longer needed. The best way to accomplish this is to localize them in the functions where they are used.

Q: <u>What causes the NOT WITH OPEN DEVICE error?</u>

A: You may get this error even though you cannot find any outstanding Shared Variables. This is caused by a Shared Variable being local to a suspended function that is buried in the execution stack. To check this, do a)SI and execute enough right arrows (→) to clear the stack (one for each *'ed entry). Remember, when developing functions, to keep the execution stack clear and localize the Shared Variables.

Q: <u>How can we reuse an output file?</u>

A: A problem that occurs during development of a function is the INVALID FILE error when you try to open an output file the second time. To prevent this error, you could)DROP the file each time, but this is tedious. To avoid the error, specify ID=(DATA) to overwrite the file.

Q: <u>How come we get a VALUE ERROR on input?</u>

A: This error indicates an incomplete data file. This is usually caused by failing to close the file when it was created. Be sure the file is closed before removing the tape. Again, localization of the Shared Variable can help here. If the variable is localized, the file will be closed whenever the function is exited.

Q: <u>What about)COPY,)SAVE, etc?</u>

A: To avoid confusion and problems, library functions that will alter or put the workspace on tape (except)CONTINUE) should only be done <u>without</u> Shared Variables in the workspace. Again, localization will help. Since)COPY,)SAVE, etc, cannot be done with suspend functions, all local Shared Variables will not exist when they are executed. If you are not sure, the following expression will insure a workspace free of Shared Variables:

```
⎕SVR ⎕NL 2
```