General Systems
Division
P. O. Box 2150
Atlanta, Georgia 30301

IBM 5110
Computing System

78-3

September, 1978

IBM 5110 NEWSLETTER

This issue contains tips and techniques to help you save storage in your IBM 5110 Computing System. We hope you will find these suggestions useful in your day to day application, and helpful in your future programming.

Contents

- Ten Ways to Save Storage in Your IBM 5110
  —For BASIC
  —For APL
- Another Look at WRITEFILE USING
  —Examples that illustrate ways of circumventing the length of syntax requirement
- Use of Procedure Files in APL

TEN WAYS TO SAVE STORAGE IN YOUR IBM 5110

Experienced programmers know the value of conserving storage in any system design; you just never know when you're going to need another 500 bytes to accommodate a change or addition to an application. Following are tips for revising your programs to do the same job but requiring fewer bytes.

FOR 5110 BASIC

1 Diskettes initialized to a smaller sector size can save storage. For example, if you are using diskettes initialized to 512 byte sector size, changing to diskettes initialized to 256 bytes for your data will in many cases reduce related open buffers by 50%.

2 RUN without PROC saves core because memory equal to at least the sector size of the procedure file is required by a PROC-driven program.

3 "USING n" instead of "USING A$" saves 428 bytes of core needed as a "scratchpad."

4 CLOSEing files when not needed frees buffer storage for reuse at the time OPEN comes up later.

5 Precise use of arrays can save storage. For example:

   a Reuse arrays when possible rather than DIMension new ones.

   b Character arrays can be DIMensioned for minimum string length.

   c DIMension arrays for minimum dimensions. (Redimensioning after DIM to a smaller array does not release storage.)

6 "RECL=n" where n is a power of 2 can save core. This will sometimes omit the necessity of increasing a buffer size by another sector length.

7 Use RUNS instead of RUN when short precision is possible for a saving of 4 bytes per numeric variable.

8 Using files to minimize USE when CHAINing often reduces the requirement of otherwise occupied storage.

9 Eliminate program characters and save one byte each. For example:

a Minimize REM statements

b Shorten comments

c Use more than one statement with the same FORM or image statement when possible

d Omit repeated program sequences by using GOSUBs and user-defined functions instead

e Delete unused program lines

f
```
A,B,C=12
is better than
A=12
B=12
C=12
```

g
```
DATA 1, 2, 3
is better than
DATA 1
DATA 2
DATA 3
```

h Sometimes in a FORM statement
C can be used instead of C40
Xn can be used instead of POS(m)

i TAB(n) can replace a lot of blanks in a PRINT statement

10 You can free 201 bytes for other assignment if PRINT FLP is never used. Stay with the Display and COPY DISPLAY.

One last comment: A large program can often be rewritten as two smaller ones by CHAINing the first to the second. You achieve the same results while requiring less storage.

FOR 5110 APL

1 Read the section on "Active WS Storage Considerations" in Chapter 4 of the IBM 5110 APL User's Guide, SA21-9302.

2 The APL SYMBOL table, used for storing information about the SYMBOLS used in your workspace, is initialized at 125 SYMBOL names. This can be set as low as 41 SYMBOLS, for a net saving of 672 characters. If your programs require more than 41 SYMBOL names, but fewer than 125, the SYMBOLS can be set between 41 and 125 in increments of 21. The space saved would be somewhat

less than the 672 characters saved if the value were 41, however. To set the SYMBOLS start with a CLEAR workspace. Example:

```
)CLEAR
)SYMBOLS 41
```

Now you can)COPY your workspace into the active workspace or begin writing your functions. Do not use more SYMBOL names than you have allocated with the SYMBOLS command.

3 Do not use the)PROC command unless it is necessary for the operation of your programs. Use of)PROC will take 768 characters from your active workspace until the next power-on sequence or the next RESTART.

4 A physical buffer is allocated for each diskette data set that has been opened. These buffers are the same size as the diskette sector size. A savings of 256 bytes per open data set can be realized by initializing diskettes to sector sizes of 256 rather than 512 bytes.

5 When using direct files in your programs use IOR rather than IORH. IOR releases the buffer after data is read into the workspace, making the space available for program utilization until the next access to that file is made. IOR will then reallocate the buffer when more data is required or when an update is to be made to the file. Thus the total amount of space required for the file operation is the same for IOR and IORH, but the buffer space is available to user programs when I/O operations are not being done.

6 Whenever possible use integer rather than decimal numeric values in your calculations. Integer data requires 4 bytes to store each value while decimal data requires 8 bytes per value. If you have an array of data, one decimal value will cause the entire array to be stored as decimal, doubling the amount of space required to store the array, as well as for all intermediate calculations. If you have to use decimal data, scale it by some appropriate factor so that it can be used as integer, then rescaled to decimal for output. For example, if ARRAY is an array of decimal values for which you wish to keep 5 digits to the right of the decimal point, scale it in the following way:

```
ARRAY ← ⌊ ARRAY x 10 * 5
```

ARRAY will now contain all integer data including the 5 digits to the right of the former decimal point. In the general case, the expression would be:

```
ARRAY ← ⌊ ARRAY x 10 * N
```

where N is the number of digits to the right of the decimal to be retained. To rescale for output do:

```
ARRAY ← ARRAY ÷ 10 * N
```

7 Retract (⎕SVR) or expunge (⎕EX) all shared variables when no longer needed between uses. In effect this will assure that open files will be closed, freeing up buffer and overhead space.

8 Identical names local to more than one user-defined function do not require additional SYMBOL table space for each function. However if one of the functions calls the other, additional storage is required to save the status of all duplicate local names and labels of the calling function.

9 Expunge (⎕EX) any variables no longer required by the program, for the same reason shared variables were expunged in tip 7.

10 The section "Additional Storage Using Diskette Data Files" in Chapter 4 of the IBM 5110 APL User's Guide, SA21-9302, can be extended to include user-defined functions as well as variables in the diskette data file. The functions must first be converted to their canonical (data) representation using ⎕CR and assigning to a variable, then stored on the diskette file. Later when they are read into the workspace they are fixed to function form with the ⎕FX system function. For example, to put function PGM into canonical form do:

```
VARIABLE ← ⎕CR 'PGM'
```

VARIABLE can now be written to a diskette file. Later when it is read back into the workspace, it is put into function form by doing the following:

```
⎕FX VARIABLE
```

Remember to expunge (⎕EX) the variables or programs involved when they are not needed.

ANOTHER LOOK AT WRITEFILE USING

The most common question asked of 5110 Market Support is the use of the WRITEFILE USING family of statements. These include:

```
0240 MAT REWRITEFILE USING 250,FL9,REC=A,B$,C$,D$,E,F,EXIT 8000

0450 REREADFILE USING 1450,FL1,X$,E$,MATD$,J1,E1,MATR,W1,W$

1250 WRITEFILE USING 1480,FL1,A2,C2,C1,Y(1),Y(2),Y(3),Q$,EOF 2220

5110 READFILE USING 5120,FL9,KEY=K$,A$(4),B$,C$,D,E,NOKEY 6000
```

There are others. The problem is the length of syntax required within a 64-character statement line; it sometimes leaves little or inadequate space for a list of variables. Following are some examples that illustrate ways of circumventing the length of syntax requirement by:

1 The repeated use of REWRITEFILE USING statements.

2 The use of arrays in the statement lists.

3 FORM assigned to a character variable.

4 Concatenation of character variables.

The examples are based on an actual payroll program, which uses a 255-byte record length. In each example only the pertinent lines of the program are shown, and it is assumed that the records used were initialized in the format suggested by the FORM statements.

Example 1: Repeated REWRITEFILE Statements.

The most common circumvention is the repeated use of REREADFILE USING and REWRITEFILE USING statements. To illustrate this point, unchanged lines from the payroll program are shown below:

```
          .         .
          .         .
          .         .
0060 DIM K$6,P(2),D(3),D$6(3),Y(3)
          .         .
          .         .
          .         .
0110 OPEN FILE FL1,'D80',11,'WORK.DATA',ALL
0120 OPEN FILE FL1,'D80',12,'WORK.INDEX',ALL,KEY
          .         .
          .         .
          .         .
0440 READFILE USING 1440,FL1,KEY=K$,Z$,N$,L$,F$,M$,S$,NOKEY 1640
0450 REREADFILE USING 1450,FL1,X$,E$,D$,J1,E1,R1,W1,W$
0460 REREADFILE USING 1460,FL1,H2,P(1),P(2),D$(1),D$(2)
0470 REREADFILE USING 1470,FL1,D$(3),D(1),D(2),D(3),W2,F2
0480 REREADFILE USING 1480,FL1,A2,C2,C1,Y(1),Y(2),Y(3),Q$
          .         .
          .         .
          .         .
1210 REWRITEFILE USING 1440,FL1,KEY=K$,Z$,N$,L$,F$,M$,S$
1220 REWRITEFILE USING 1450,FL1,KEY=K$,X$,E$,D$,J1,E1,R1,W1,W$
1230 REWRITEFILE USING 1460,FL1,KEY=K$,H2,P(1),P(2),D$(1),D$(2)
1240 REWRITEFILE USING 1470,FL1,KEY=K$,D$(3),D(1),D(2),D(3),W2,F2
1250 REWRITEFILE USING 1480,FL1,KEY=K$,A2,C2,C1,Y(1),Y(2),Y(3),Q$
          .         .
          .         .
          .         .
1440 FORM POS1,C1,C6,C15,C14,C1,C11
1450 FORM POS49,C1,C1,C3,2*NC2,NC6.2,NC5.3,C8
1460 FORM POS77,NC3.1,2*NC5.2,2*C6
1470 FORM POS102,C6,5*NC7.2
1480 FORM POS143,6*NC7.2,C6
          .         .
          .         .
          .         .
```

The KEY field occupies bytes 2 through 7. Numeric and character parts of the record are intermixed, and not all of the 255-byte record is used as can be seen by counting through the FORM statements: only 190 bytes were used. .

This technique is self-explanatory. Any questions can be answered by reading the appropriate sections in the IBM 5110 BASIC Reference Manual (SA21-9308-0) and the IBM 5110 BASIC User's Guide (SA21-9307-0), especially pages 94-114.

Example 2: Use of Arrays.

In Example 1, a READFILE USING, four REREADFILE USING, five REWRITEFILE USING, and five FORM statements were used. This

does the job. However, time to access the file is required for each REWRITEFILE USING line. In programs where time is a concern, it is possible to write an equivalent program to that of Example 1 with fewer statements and sometimes with even fewer characters. The objective of this example is to do what was done in Example 1, but with only one MAT READFILE USING, one MAT REWRITEFILE USING, and one FORM statement, while staying within the allotted 255-byte record.

The technique is to "gather" like variables in one fashion or another forming arrays, and economizing FORM. To illustrate, the variables of Example 1 are renamed and repositioned in the record, such as using character variables Z$(1), Z$(2), Z$(3), Z$(4), and Z$(5), instead of Z$, M$, X$, E$, and D$. Likewise, all numerics are gathered into an array B, each element requiring eight bytes in the file.

Here is a complete list of the new variable names (all in array form) replacing the old names that were used in Example 1:

```
B(1)←J1, B(2)←E1, B(3)←R1, B(4)←W1, B(5)←H2
B(6)←P(1), B(7)←P(2), B(8)←D(1), B(9)←D(2), B(10)←D(3)
B(11)←W2, B(12)←F2, B(13)←A2, B(14)←C2, B(15)←C1
B(16)←Y(1), B(17)←Y(2), B(18)←Y(3)


Z$(1)←Z$, Z$(2)←M$, Z$(3)←X$, Z$(4)←E$, Z$(5)←D$, B$(1)←N$
B$(2)←D$(1), B$(3)←D$(2), B$(4)←D$(3), B$(5)←Q$, B$(6)←W$
L$(1)←S$, L$(2)←F$, L$(3)←L$
```

While the program lines equivalent to those of Example 1 now appear more simply written, they do the same job.

```
          .          .
          .          .
          .          .
0060 DIM K$6,B(18),B$8(6),Z$3(5),L$15(3)
          .          .
          .          .
          .          .
0110 OPEN FILE FL1,'D80',11,'WORK.DATA',ALL
0120 OPEN FILE FL1,'D80',12,'WORK.INDEX',ALL,KEY
          .          .
          .          .
          .          .
0440 MAT READFILE USING 1440,FL1,KEY=K$,B$,L$,Z$,B,NOKEY 1640
          .          .
          .          .
          .          .
1210 MAT REWRITEFILE USING 1440,FL1,KEY=K$,B$,L$,Z$,B
          .          .
          .          .
          .          .
1440 FORM POS1,6*C8,3*C15,5*C3,18*NC8.3
          .          .
          .          .
          .          .
```

The records used of course were initialized with a different arrangement of fields than those of Example 1. The KEY field, for instance, now occupies bytes 1 through 6, and the character and numeric fields are not intermixed.

Only one MAT READFILE, one MAT REWRITEFILE, and one FORM statement were used. By counting you can see that the total record requirement is 252 bytes, which is within the 255 bytes of the record length. Every statement was written on a 64-character line. Therefore, the objective of Example 2 has been met!

Example 3: FORM Assigned to a Variable.

While the objectives of Example 2 were met, the use of bytes in the 255-byte record was extravagant. The records could have been initialized with smaller fields.

The objective of this example is to use fewer bytes and still meet all the objectives of Example 2. This is achieved by placing the specifics into the FORM statement to keep each field in the record as small as possible and all fields contiguous. FORM may be longer than 64 characters by assigning it to a character variable, as is the case in this example:

```
          °         °
          °         °
          °         °
0060 DIM K$6,B(18),B$8(6),Z$3(5),L$15(3),U$40,T$38,V$78
0061 T$='FORM POS1,5*C6,C8,C11,C14,C15,4*C1,C3,'
0062 U$='2*NC2,NC6.2,NC5.3,NC3.1,2*NC5.2,11*NC7.2'
0063 STR(V$,1,38)=STR(T$,1,38)
0064 STR(V$,39,40)=STR(U$,1,40)
          °         °
          °         °
          °         °
0110 OPEN FILE FL1,'D80',11,'WORK.DATA',ALL
0120 OPEN FILE FL1,'D80',12,'WORK.INDEX',ALL,KEY
          °         °
          °         °
          °         °
0440 MAT READFILE USING V$,FL1,KEY=K$,B$,L$,Z$,B,NOKEY 1640
          °         °
          °         °
          °         °
1210 MAT REWRITEFILE USING V$,FL1,KEY=K$,B$,L$,Z$,B
          °         °
          °         °
          °         °
```

Only 190 bytes of the 255-byte record were used!

Example 4: Use of Concatenation.

In Examples 2 and 3, arrays and MAT REWRITEFILE USING were used to gain the objectives. Still another way of meeting these objectives is illustrated. The records used are initialized with fields with as few bytes as possible, but the KEY occupies bytes 2 through 7 as it did in Example 1. Also, character fields and numeric fields are not intermixed as they were in Example 1.

The technique is to concatenate (see BASIC Reference Manual) the character variables into fewer variables before going to file. The resulting variables must be less than 128 characters each. This same technique can also be extended to include numerics by first converting them to characters, but, instead, the method of Examples 2 and 3 for numerics is retained by using MAT B inside the statement list when concatenation is used.

Notice that the character variable names are the same as those in Example 1, and numeric variables are handled the same as those in Examples 2 and 3.

In the program below, lines 1210 through 1213 concatenate all character variables into a simple character variable R$ (alternatively, STR could have been used to accomplish this). Also, lines 441 through 454 use the STR function to redistribute the characters of R$ back into their original, smaller variables.

```
          .          .
          .          .
          .          .
0060 DIM K$6,B(18),D$6(3)
0061 DIM Z$1,N$6,L$15,F$14,M$1,S$11,P$48,R$85
0062 DIM Q$6,D$6(3),X$1,E$1,D$3,W$8
          .          .
          .          .
          .          .
0110 OPEN FILE FL1,'D80',11,'WORK.DATA',ALL
0120 OPEN FILE FL1,'D80',12,'WORK.INDEX',ALL,KEY
          .          .
          .          .
          .          .
0440 READFILE USING 1440,FL1,KEY=K$,R$,MATB,NOKEY 1640
0441 Z$=STR(R$,1,1)
0442 N$=STR(R$,2,6)
0443 L$=STR(R$,8,15)
0444 F$=STR(R$,23,14)
0445 M$=STR(R$,37,1)
0446 S$=STR(R$,38,11)
0447 Q$=STR(R$,49,6)
0448 D$(1)=STR(R$,55,6)
0449 D$(2)=STR(R$,61,6)
0450 D$(3)=STR(R$,67,6)
0451 X$=STR(R$,73,1)
0452 E$=STR(R$,74,1)
0453 D$=STR(R$,75,3)
0454 W$=STR(R$,78,8)
          .          .
          .          .
          .          .
1210 P$=Z$!!N$!!L$!!F$!!M$!!S$
1211 R$=P$!!Q$!!D$(1)!!D$(2)!!D$(3)!!X$!!E$!!D$!!W$
1213 REWRITEFILE USING 1440,FL1,KEY=K$,R$,MATB
          .          .
          .          .
          .          .
1440 FORM POS1,C85,2*NC2,NC6.2,NC5.3,NC3.1,2*NC5.2,11*NC7.2
          .          .
          .          .
          .          .
```

Only one READFILE USING, one REWRITEFILE USING, and one FORM statement were used; every statement was written within a 64-character line; and the total record requirement was only 190 out of the 255 allowed. Thus, again the objectives have been met.

USE OF PROCEDURE FILES IN APL

Procedure files in APL cause input that would otherwise come from the keyboard to be read directly from the procedure file. Any input to a request from □ or ⍞ must thus be placed as records on the procedure file at the time it is created. If you wish to enter data directly from the keyboard when using a procedure file, you may do so by replacing uses of □ or ⍞ with calls to appropriate screen input routines. See the IBM 5110 APL Reference Manual and the IBM 5110 APL User's Guide for specifics.

You may initiate programs with procedure files that create data for follow-on programs. If you do and if errors occur that either terminate the programs abnormally or if the intended operations do not successfully complete, you must determine this before executing any follow-on programs. To do otherwise could give erroneous results.

Following are two methods for determining whether or not a prior job has completed satisfactorily.

Method 1

You create a procedure file that)LOADs a WS. The next record in the procedure file names a program to be run. Upon completion of this program, the procedure file issues a)COPY for a second WS or set of programs to be brought into the existing active WS. You must check for the following:

1 The program you just ran terminated abnormally with some sort of INTERRUPT. Examples would be a DOMAIN ERROR, a RANK ERROR or someone may have pressed the ATTN key. If any of these occurred, a check of □LC in a follow-on program would show that ⍴□LC > 1.

2 An intended operation was initiated (such as an I/O update to a transaction file) but it did not terminate correctly. Perhaps there was an unrecoverable I/O error or the file was not)MARKed large enough to contain the data. It is your responsibility to check the return codes in your program so that proper action may be taken. In this case your program's error recovery procedure could set a GLOBAL VARIABLE (such as GV) in the WS that could be tested by a follow-on program. Let GV←1 mean that the operation completed successfully and GV←0 indicate that the operation failed. A test of GV by the follow-on program would enable you to decide whether to allow the procedure to continue or not.

As an example of how to accomplish these tests, place the following lines of code at the beginning of each follow-on program that depends on the successful completion of any preceding program:

```
    ∇ 'your function name
[1]  →(1=⍴⎕LC)/F1                    Was prior function interrupted?
[2]  →0,0⍴⎕←'PRIOR FUNCTION INTERRUPTED - TERMINATE'
[3]  F1:→GV/F2                       Did operation complete OK?
[4]   'FUNCTION DID NOT COMPLETE REQUIRED OPERATION'
[5]   'TERMINATE'
[6]  →0
[7]  F2: continue with your function
```

With this approach you can control program-to-program transition within a procedure file that does not overlay the original active WS.

Method 2

In this case the procedure file)LOADs a WS and then causes a program to be executed. After the program terminates, a second WS is)LOADed, overlaying the existing WS. If the follow-on program depends on the successful completion of the preceding program, it cannot check the status of ⎕LC or the setting of a global variable since they are no longer in the active WS.

The solution is to create a small utility program (for this example, called CLEANUP) which is called by the procedure file just prior to)LOADing the follow-on WS. The CLEANUP program can then check the status of ⎕LC and the setting of a global variable, GV, write the status to a previously)MARKed file, and then terminate.

The steps are:

1)MARK a 1K file on tape or disk. This file will be used to pass ⎕LC and GV status to follow-on WSs.

2 Create your procedure file. It should look something like:

```
)LOAD 'first WS'          First record
'program name'            Second record
CLEANUP                   Third record
)LOAD 'second WS'         Fourth record
'program name'            Fifth record
```

Note that CLEANUP is used after the required program has terminated and just prior to)LOADing the follow-on dependent WS.

A sample CLEANUP program could be:

```
    ∇CLEANUP;SV
[1]  ⎕WA←1 ⎕SVO 'SV'                    Offered shared variable
[2]  SV←'OUT file #  ID=(filename)'     Open file
[3]  SV←((1<⍴⎕LC),1=⍴⎕LC)/0 1           Check ⎕LC status
[4]  SV←GV                              Assign global variable
                                        (Should be 0 (failed)
                                        or 1 (OK))
[5]  SV←\0                              Close file
    ∇
```

3 In each follow-on program that depends on the successful completion of previous programs, insert code similar to the following at the beginning of the program:

```
      ∇'your function';SV               Localize shared variable
[1]    ⎕WA←1 ⎕SVO 'SV'                  Offer shared variable
[2]    SV←'IN file # ID=(filename)'      Open file
[3]    ⎕WA←SV                           Open return code
[4]   →SV/F1                            First read gets ⎕LC status;
                                        go to F1 if OK
[5]   →0,0⍴⎕←'PRIOR FUNCTION INTERRUPTED - TERMINATE'
[6]   F1:→SV/F2                         Second read gets GV status;
                                        go to F2 if OK
[7]    'FUNCTION DID NOT COMPLETE REQUIRED OPERATION'
[8]    'TERMINATE'
[9]   →0                                Quit procedure
[10]  F2: SV←\0                         All OK; close file
[11]   continue with your function
```

Using this approach you can assure that any required data has been created or that any necessary operations have been completed before continuing within an APL procedure.

In summary, since APL procedures offer the possibility of unattended operation because each record of the procedure file takes the place of keyboard generated input up to the procedure end of file, you have the responsibility to assure all required operations have been performed before continuing with a procedure at each step. The two methods outlined are just two ways you might use to accomplish this.