From Obscurity to Utility: ADDR, PEEK, POKE as DATA Step Programming Tools

Paul M. Dorfman
Senior ETL Engineer, Red Buffalo, Jacksonville, FL

SUMMARY

APP functions is an (unofficial) collective abbreviation for the SAS® functions ADDR, PEEK, PEEKC, the CALL POKE routine, and their so-called LONG 64-bit counterparts - SAS tools designed to directly read from and write to the physical memory in the Data step and SQL Procedure.

APP functions have long been a SAS user dark horse. Firstly, the examples of APP usage in SAS documentation boil down to a few tidbits in a technical report, all intended for system programming tasks, with no hint how the functions could be used in plain data management SAS programming. Secondly, the note about the CALL POKE routine in the SAS documentation is so intimidating in tone that many a folk may have decided to avoid the potentially precarious route altogether.

However, nothing can stand on the way of a curious SAS programmer daring to take a closer look; and it turns out that APP functions are very simple and useful tools! They can be used to explore how things "really work", make code more concise, implement "en masse" (group) data moves, and, oftentimes, significantly improve execution efficiency.

The authors have been exploring the APP world since 1998, occasionally letting the SAS-L community to peek at their findings. This tutorial is an attempt to the results in a systematic way. Welcome to the APP world! You are in for a few pleasant surprises.

INTRODUCTION

Let us cut to the chase: What are APP functions, anyway? APP functions are the following SAS functions and call routines:

ADDR	ADDRLONG
PEEK	PEEKLONG
PEEKC	PEEKCLONG
CALL POKE	CALL POKELONG

So, strictly speaking, not all of them are functions - two are actually call routines, but we hope the reader will overlook this minor etymological impropriety for the sake of simplicity. We shall refer to them as APP functions, or just APP, throughout the paper.

One of us (*P.D.*) has conducted an informal (and utterly non-scientific) poll of a small sample of about 97.3 SAS programmers with various degrees of expertise (hence the fractional number, the only SAS programmer standing at 1.0 being Ian Whitlock). 70% of them had never heard of APP functions, whilst 15% had heard of them, but actually had never used them. Among the rest 15%, 10% had only *seen* them used, and only 5% - exclusively from the SAS-L crowd had used them for plain data management purposes, i.e. things other than, say, obtaining the address of Communication Vector Table in z/OS.

Nonetheless, an inquisitive SAS programmer, and particularly one not willing to be restricted by the covers of a SAS manual, can make much better and wider use of APP functions - and have some programming fun along with improved performance. However, as it always happens in the world of unavoidable trade-offs, some necessary preliminaries have to be covered first.

SAS VARIABLE ADDRESSING PROPAEDEUTICS

First, a warning: A number of distinguished folks may find this part too elementary and self-explanatory to swallow, in which case they are urged to use their internal GO TO instruction. However, since relatively few SAS software users come from the CS direction, we feel compelled to briefly discuss basic concepts, whose understanding is paramount to the subject of the paper.

Our trial-and-error efforts have revealed that APP can be used both in the Data step and Proc SQL (in the latter, with the understandable exception of the POKE routines, since no call routines can be used in SQL, anyway). However, we shall concentrate exclusively on APP usage inside the Data step, the chief reason being the absence of a clear SQL concept similar to the PDV in the Data step.

When SAS compiles a Data step, it allocated a RAM (or real storage, in z/OS speak) location for every variable and expression it encounters in the step. The location is like a box in memory comprising a number of bytes. For each numeric variable and/or expression, the box contains 8 bytes. For each character variable, it contains just as many bytes as the declared length of the variable is (the value returned by the vlength() function).

Each byte in the box has a number assigned to it, which is the physical address of the byte. The lowest number is the initial address of the variable or expression; the rest of the bytes are assigned consecutive numbers starting from the initial address strictly up by a unity. The software can access these bytes and their content by its physical address only.

Under non-APP circumstances, you never need to use physical addresses, and indeed, why bother? The compiler maintains a rapidly-searched symbol table to look physical addresses up by variable names or, internally, expressions. Given a variable name, for example, SAS finds its initial physical address from the symbol table and, beginning from it, extracts just as many bytes from next consecutive addresses as the variable expression length prescribes. So, the programmer has the luxury to be completely oblivious of what the physical addresses are.

However, the employment of APP functions is predicated on such knowledge, so if we intend to use them we need to be a little bit more aware of the way SAS organises its stuff in memory. This is convenient to do along with the learning the workings of the ADDR and ADDRLONG functions.

ADDR and Variable Addressing Exploration

The purpose of the ADDR function is simple: On 32-bit platforms, it returns the physical address of its argument, which is usually a large integer. Consider the simplest case:

```
16   data _null_ ;
17      retain num 1 chr 'char' ;
18      addr_n = addr (num) ;
19      addr_c = addr (chr) ;
20      put addr_n = / addr_c = ;
21   run ;

addr_n=40826688
addr_c=40826992
```

We give the ADDR function a variable name, and it returns the memory address of its leftmost byte. As noted above, the rest of the bytes in the variable always occupy the consecutive addresses up by one, all the way to the full variable declared (attribute) length. Thus, in this step, the eight bytes of the numeric variable NUM occupy eight consecutive addresses from 40826688 to 40826695, whilst the character string CHR takes up four consecutive addresses from 40826992 to 40826995. To avoid the confusion, let us make it clear that whenever we say that a variable occupies a certain address, it is an equivalent of saying that its leftmost byte begins at the address, or, conversely, that the variable's lowest address points to the leftmost byte.

Usually the argument is a variable name, which must be available at compile time. In fact, SAS documentation lists it as the only possibility, but as we will see later on, there are other possibilities, too. For example, if we combine SAS tokens in an expression the software should have a location in the physical memory to store the result before utilizing it in some way. Hence, the ADDR function must be able to return the address of a SAS expression, and it does:

```
22   data _null_ ;
```

```
23      retain num 1 chr 'char' ;
24      addr_n = addr (num) ;
25      addr_c = addr (chr) ;
26      addr_exp_n = addr (exp(num) * input(chr, pib4.)) ;
27      addr_exp_c = addr (put (num, z2.) || reverse (chr)) ;
28      put addr_n = ;
29      put addr_c = ;
30      put addr_exp_n = ;
31      put addr_exp_c = ;
32   run ;

addr_n=40851456
addr_c=40851792
addr_exp_n=40851512
addr_exp_c=40851802
```

Note that at this point we do not see any orderly manner in which the compiler assigns the addresses for variables and expressions. However, the order exists, and to discover it, the ADDR function is just a tool. It will be immediately apparent if we create a sufficient number of variables with different lengths falling in the four possible classes:

1.non-retained numeric
2.non-retained character
3.retained numeric
4.retained character

and observe their addresses:

```
data _null_ ;
   retain rn1  1  rn2  22  rn3  333  rn4  4444 ;
   retain rc1 '1' rc2 '22' rc3 '333' rc4 '4444';

   n1 =  1 ;  n2 =  22  ; n3 =  333  ; n4 =  4444 ;
   c1 = '1' ; c2 = '22' ; c3 = '333' ; c4 = '4444' ;

   a_rn1 = addr (rn1) ; a_rn2 = addr (rn2) ;
   a_rn3 = addr (rn3) ; a_rn4 = addr (rn4) ;
   a_rc1 = addr (rc1) ; a_rc2 = addr (rc2) ;
   a_rc3 = addr (rc3) ; a_rc4 = addr (rc4) ;

   a_n1 = addr (n1) ; a_n2 = addr (n2) ;
   a_n3 = addr (n3) ; a_n4 = addr (n4) ;
   a_c1 = addr (c1) ; a_c2 = addr (c2) ;
   a_c3 = addr (c3) ; a_c4 = addr (c4) ;

   put  59 * '-' ;
   put 'retained     numeric  : ' a_rn1-a_rn4 ;
   put 'non-retained numeric  : ' a_n1- a_n4  ;
   put  59 * '-' ;
   put 'retained     character: ' a_rc1-a_rc4 ;
   put 'non-retained character: ' a_c1- a_c4  ;
run ;
-----------------------------------------------------------
retained     numeric  : 41054792 41054800 41054808 41054816
non-retained numeric  : 41054968 41054976 41054984 41054992
-----------------------------------------------------------
retained     character: 41055288 41055289 41055291 41055294
non-retained character: 41055664 41055665 41055667 41055670
```

We see that each class of variables occupies a distinctly separate area in the physical memory, where all the variables belonging to a particular class are strung together in a block of addresses in the physical memory, the distance between them being determined by the lengths of the individual variables.

Numeric variables all have the same memory length, 8, since in the physical memory each is nothing more but an 8-byte character string converted to the corresponding double-float number by the algorithm hidden behind the RB8 informat. Indeed, according to the log, all retained numeric variables are placed by the compiler into equidistant addresses spaced 8 bytes apart:

```
addr(rn4) = addr(rn1) + 3*8 = addr(rn2) + 2*8 = addr(rn3) + 8.
```

The same pattern holds for non-retained numeric variables, except that all of them as a block occupy an area in memory set apart from that occupied by the retained numeric variables. It begins at the address 41054968, set 152 bytes apart from the last retained numeric variable - instead of the address 41054816+8=41054824, where the next retained numeric variable would started if it existed.

Now turning our attention to the character variables, we observe a similar rule. However, since character variables may have different lengths, the distance between their initial addresses is not always 8, as in the case of numeric variables, but varies according to the length of adjacent variables. For instance, looking at the addresses occupied by the retained character variables, we plainly see that they are spaced apart as follows:

```
addr(rc4) = addr(rc3) + 3
addr(rc3) = addr(rc2) + 2
addr(rc2) = addr(rc1) + 1
```

as it should be expected, since we have intentionally concocted the variables serially with the lengths of 1, 2, 3, 4.

Just as retained and non-retained numeric variables reside in two separate memory areas, so do retained and non-retained character variables. For instance, if another retained character variable with the lengths 5, say, existed in the step, its location would start at the address 41055294+5=41055299, but because the next character variable belongs to the non-retained class, it instead begins some 370 bytes away, at the address 41055664.

Thus, the first useful function of the ADDR function (no tautology intended) is exploratory: It lets us definitively tell, without the need to speculate, how and where the compiler places variables in the PDV.

Now let us extend the analysis a bit further into the variables coming from SAS data files and/or grouped in arrays.

Addressing Variables Coming from SAS Data Files

Since the variables coming from SAS data sets and views are automatically retained, their addresses belong to two distinct areas only: Retained numeric and retained character. In each group, every two consecutive variables always start at two addresses set apart by the length of the first variable:

```
data a ;
   retain n1  1  n2  22  ;
   retain c1 '1' c2 '22' ;

   n3 =  333  ; n4 =  4444  ;
   c3 = '333' ; c4 = '4444' ;
run ;

data _null_ ;
   set a ;
   a_n1 = addr (n1) ;
   a_n2 = addr (n2) ;
   a_n3 = addr (n3) ;
   a_n4 = addr (n4) ;

   a_c1 = addr (c1) ;
   a_c2 = addr (c2) ;
```

```
   a_c3 = addr (c3) ;
   a_c4 = addr (c4) ;

   put  61 * '-' ;
   put 'numeric   (all retained): ' a_n1-a_n4 ;
   put  61 * '-' ;
   put 'character (all retained): ' a_c1-a_c4 ;
run ;
-------------------------------------------------------------
numeric   (all retained): 41053304 41053312 41053320 41053328
-------------------------------------------------------------
character (all retained): 41053680 41053681 41053683 41053686
```

As we shall see later, for many applications of other APP functions, such as moving data in blocks, it is imperative that the variables in question upon occupy neighboring addresses falling into the above pattern. From this standpoint, it is important that if variables are read from a SAS data file, this rule be always satisfied: Because no such variable can be made non-retained (unless it is dropped and recreated), the compiler will never place it in an address outside of its block. Numeric variables read from a SAS data file are always put at equidistant addresses 8 bytes apart because all of them have length 8. The same will be true for character variables if all of them are defined with equal lengths.

Addressing Variables Grouped in Arrays

Arrays present a special case of interest for this discussion; for it is chiefly the grouped variables where the main thrust of APP data management capabilities is going to apply. There are several reasons:

Array elements always have the same declared (and hence, memory) lengths, so if their elements are all retained or all non-retained, their initial addresses will be always grouped equidistantly. This is always true for arrays initialized at compile time, for their variables are all retained.

The elements of a temporary array are always retained, which guarantees that their addresses are contiguous and equidistant.

However, it is possible - and we must point this out as a *potential caveat* - that SAS non-temporary arrays are extremely diverse. They may incorporate both retained and non-retained variables, the same variable more than once, or the same variable can be a part of more than one array. In such cases, the compiler has to establish an internal scheme to cross-reference variables residing in separate blocks of physical memory, and the rule "array items occupy contiguous addresses distanced apart according to the array element declared length" may no longer hold. An example of such a situation is given below:

```
data _null_ ;
   array nt [3] _temporary_ ;
   put 56 * '-' ;
   put "Temporary numeric  : "  @ ;
   a1 = addr (nt[1]) ; put a1 = @ ;
   a2 = addr (nt[2]) ; put a2 = @ ;
   a3 = addr (nt[3]) ; put a3 =   ;

   array ct [3] $ 10 _temporary_ ;
   put "Temporary character: "  @ ;
   a1 = addr (ct[1]) ; put a1 = @ ;
   a2 = addr (ct[2]) ; put a2 = @ ;
   a3 = addr (ct[3]) ; put a3 =   ;

   array nn [*] x y z ;
   put "Permanent numeric  : "  @ ;
   a1 = addr (nn[1]) ; put a1 = @ ;
   a2 = addr (nn[2]) ; put a2 = @ ;
   a3 = addr (nn[3]) ; put a3 =   ;
```

```
   array cc [3] $3 ('111' '222' '333') ;
   put "Permanent character: "  @ ;
   a1 = addr (cc[1]) ; put a1 = @ ;
   a2 = addr (cc[2]) ; put a2 = @ ;
   a3 = addr (cc[3]) ; put a3 =   ;

   retain wday 'work' ;
   wend = 'rest' ;
   array w [*] $4 wend wday wday wday wday wday wend ;
   put "Character irregular: " @ ;
   a1 = addr (w[1]) ; put a1 = @ ;
   a2 = addr (w[2]) ; put a2 = @ ;
   a3 = addr (w[3]) ; put a3 =   ;
run ;
-----------------------------------------------------
Temporary numeric  : a1=40872440 a2=40872448 a3=40872456
Temporary character: a1=40872696 a2=40872706 a3=40872716
Permanent numeric  : a1=65476592 a2=65476600 a3=65476608
Permanent character: a1=65476800 a2=65476803 a3=65476806
Character irregular: a1=65477168 a2=65476809 a3=65476809
```

Fully retained, fully non-retained, compile-initialized, and non-initialized arrays all conform to the "contiguous, equidistant" rule. However, the array W[*] has seven elements incorporating two variables, one initialized at compile time, the other - at run time, hence one is retained, and the other is not retained. As a result, although its two variables WEND and WDAY are the first on the array variable list, their addresses lie distinctly apart.

What Is The LONG Story?

So far, we have been talking about 32-bit machines, where addresses are short enough to fit into the Procrustean bed of the integer precision provided by the SAS double-float numeric data type. Now that SAS is being ported to 64-bit machines, their addresses no longer satisfy this premise, because they become much longer, while the SAS numeric type does not get to be expanded to 128 bits, staying the way it has been. As a consequence, an attempt to store the physical address of a variable on a 64-bit machine in a 32-bit double-float may result in rounding the address off.

So, for one thing, on a 64-bit machine the ADDR function will not be able to help display and explore addresses in the manner shown above. But it is a minor inconvenience compared to the havoc the rounding can wreak when trying to use an address obtained in such a manner with the rest of APP functions. Indeed, if we want them to produce expected and meaningful results, physical addresses given to them as arguments must be supplied exactly as they are, all the more that, as we have seen above, the actual difference between two addresses can lie precisely in the lowest-order digits.

The only way out of this situation is to represent addresses by character strings. Since a string consists of one or more of 256 characters forming the collating sequence, it is essentially a 256-base number, thus capable of representing very large integers with very few characters. For example, an 8-byte character string can represent numbers from 0 to 256**8-1, while a SAS double-float fails to exactly represent an integer as small as 256**7.

This concept underlies the ADDRLONG function, which returns the address of a variable (or expression) as a 20-byte character string. However, of the 20 bytes only the non-blank part is considered as the binary representation of the address. ADDRLONG also works on 32-bit platforms, where addresses are so short that they can be represented only by first 4 bytes of the string returned by ADDRLONG. The step below demonstrates the equivalence of the actual address value returned by ADDR and ADDRLONG on a 32-bit platform (Windows XP Pro, in this case):

```
data _null_ ;
   v = 1 ;
   av = addr (v) ;
   avlong = addrlong (v) ;

   put 39 * '-' ;
   put 'ADDRLONG returns ' avlong = ;
```

```
   put 'ADDR     returns ' av     = ;

   avlong_vlen = vlength (avlong) ;
   avlong_len  =  length (avlong) ;

   ae = inputn (avlong, 'pib' || put (length (avlong), 1.) || '.') ;

   put 'Numeric ADDRLONG equivalent ' ae = ;
run ;
---------------------------------------
ADDRLONG returns avlong=pkr_
ADDR     returns av=41053040
Numeric ADDRLONG equivalent ae=41053040
```

Of course, on a 64-bit platform, this equivalence trick is impossible to pull because of the SAS numeric integer precision limitation, which is a good reason to have ADDRLONG in the first place. Throughout the paper, we will use primarily the "short" APP functions. However, all code examples can be converted to their "long" versions by replacing all APP functions in the step to their "long" brethren.

Note that SAS recommends using the "long" versions of APP functions at all times. This indeed is a sound advice: 32-bit applications developed using the "long" APP functions will not need to be converted if it is eventually moved to a 64-bit platform.

PeekC[LONG], Peek[LONG]

To this point, we have only learned how to determine the physical addresses at which SAS variables reside, and a little bit about the way they are arranged in memory. The next question the curious SAS programmer would ask is, of course: Given a physical address (known beforehand or obtained using ADDR), how to find out what is stored at it and in higher adjacent addresses?

The functions PEEKC (PEEKLONG) and PEEK (PEEKLONG) give the answer, and the step below illustrates the usage:

```
17   data _null_ ;
18      n = 12345 ;
19      n_memory_image = peekc (addr(n) , 8) ;
20      n_autoconvert  = peek  (addr(n) , 8) ;
21      n_img_convert = input (n_memory_image , rb8.) ;
22
23      put 32 * '-' / n_memory_image=$hex16. / n_autoconvert= / n_img_convert= ;
24
25      retain c1 'ENTIRE PHRASE, NOT ' c2 'PART OF ' c3 'WHOLE PHRASE! ' ;
26
27      whole1 = peekc ( addr(c1),                                  40  ) ;
28      whole2 = peekc ( addr(c1), vlength(c1)+vlength(c2)+vlength(c3) ) ;
29      partc2 = peekc ( addr(c2),             vlength(c2)+vlength(c3) ) ;
30
31      put (whole1 whole2 partc2) (/=) ;
32   run ;

--------------------------------
n_memory_image=00000000801CC840
n_autoconvert=12345
n_img_convert=12345

whole1=ENTIRE PHRASE, NOT PART OF WHOLE PHRASE!
whole2=ENTIRE PHRASE, NOT PART OF WHOLE PHRASE!
partc2=PART OF WHOLE PHRASE!
```

Above, PEEKC (addr(n),8) asks SAS to look at the address of the leftmost byte of the numeric variable N and extract the content of the next 8 addresses in physical memory beginning from and including one at ADDR(N). Why eight? Because we know that a numeric variable takes up 8 bytes in memory, PEEKC extracts the bytes from their addresses exactly as they are stored there. Therefore, the result is an 8-byte string exactly mirroring the way N is stored in physical memory, and it is nothing else but the 8-byte, double-precision binary image of the variable N.

Understandably, printing it as would not convey a whole lot of information, while printing it in hex, as shown above, just shows the bytes comprising the image in hexadecimal notation. In order to make sense out of its contents, it has to be interpreted as a number, which is done using the RB8 informat resulting in the N_IMG_CONVERT numeric variable above. To save the effort, SAS offers a special function PEEK, which automatically applies the RB8 informat to the 8-byte string extracted from the physical memory starting at ADDR(N). Not surprisingly, the results are identical.

Again, if all we need to extract from the physical memory at ADDR(C1) and higher text just the way it is, without any need to interpret it, the PEEKC (or PEEKCLONG) function is all we need. As an example, in the step above, the compiler, parsing the step in the top-down manner, blocks the variables C1-C3 by giving them addresses spaced according to the attribute lengths of variables C1 and C2, as discussed in the previous section. Effectively, it means that the characters constituting the phrase

```
ENTIRE PHRASE, NOT PART OF WHOLE PHRASE!
```

whose chunks are hidden in the variables C1-C3, are stored in forty consecutive addresses starting from the address pointing to the initial byte of C1, i.e. ADDR(C1). Therefore, it should be no surprise that the expression assigned to WHOLE1 copies the contents of C1-C3 into WHOLE1 - the result equivalent to the concatenation of C1||C2||C3. One should bear in mind that by default, SAS assigns the memory length of 200 to a PEEKC expression. A LENGTH statement, a character (in)format, etc., can be used to change that.

The capability of PEEKC (PEEKCLONG) to extract long strings of consecutive bytes into a character string (expression) directly from the physical memory and without the need to concatenate them piece by piece makes for a powerful data management tool. It is particularly useful in conjunction with arrays because they guarantee that all elements have equal lengths.

Because the expression receiving the result returned by PEEKC is always a character expression, its length is naturally limited to 32767. Therefore, using the second PEEKC argument greater than 32767 is futile, although it would not result in a compile or run-time error. SAS documentation indicates the range from 1 to 32767, 8 being the default. However, we have very good reasons to strongly recommend that the argument indicating the range of addresses accessed by APP must be always coded explicitly (but of course not necessarily hard-coded).

Using PEEKC for Concatenation

Suppose that each observation in a data set A has 10 variables C1-C10, $4 each, which need to be concatenated into a single character variable C. This step does it without looping over each variable and concatenation operator:

```
data pieces ;
   input (c1-c10) (: $4.) ;
cards ;
0000 1111 2222 3333 4444 5555 6666 7777 8888 9999
  a    b    c    d    e    f    g    h    i    j
 zz   yy   xx   ww   vv   uu   tt   ss   rr   qq
 ***  ///  ###  !!!  ???  $$$  ***  +++  \\\  ***
;
run ;

data _null_ ;
   set pieces ;
   length whole $ 40 ;
   whole = peekc (addr(c1), 40) ;
   put 40 * '-' / whole ;
run ;
----------------------------------------
```

```
000011112222333344445555666677778888999
----------------------------------------
a   b   c   d   e   f   g   h   i   j
----------------------------------------
zz  yy  xx  ww  vv  uu  tt  ss  rr  qq
----------------------------------------
*** /// ### !!! ??? $$$ *** +++ \\\ ***
```

The LENGTH statement prevents WHOLE from being given the default length of 200. The number of items in the C-list and the length of each item having been selected pretty much arbitrarily, keep in mind that the second PEEKC argument is capped at

vlength (C1) * (number of C-items) <= 32767.

In Version 9, the same result can be achieved using the CAT function as:

whole = CAT (of C1-C10) ;

An advantage of PEEKC, though, is that if the variables being concatenated do not form or cannot form a prefixed list, the OF keyword cannot be used, and the arguments have to be listed separately. For example, if instead of C1-C10 we had variables named as single letters from A to J, PEEKC would still work in the same manner:

whole = peekc (addr(A), 40) ;

while CAT would need to have all 10 variables explicitly listed. Finally, we have tested PEEKC to execute about 20 per cent faster than CAT. Of course, in the versions earlier than V9, PEEKC reigns alone.

Stringing out an Array

Another apparent application of the same functionality is *stringing out an array*:

```
data _null_ ;
   array cc [ 9999 ] $3 _temporary_ ( 3333 * ('111' '222' '333') ) ;

   all_cc = put ( peekc (addr(cc[1]), 3 * dim(cc)), $30000.) ;

   put 'Check: ' / 8 * '-' ;
   beg = substr (all_cc,    1, 3) ; put beg = ;
   mid = substr (all_cc, 3334, 3) ; put mid = ;
   end = substr (all_cc, 9997, 3) ; put end = ;
run ;

Check:
--------
beg=111
mid=222
end=333
```

Above, the format was used to size ALL_CC up instead of the LENGTH statement, and the substringing was but a quick dirty was to check if PEEKC worked as intended. Instead of hard-coding the second PEEKC argument, the DIM function was used instead. Note that since CC is a temporary array, its elements cannot be used as an OF-list, and thus in the example above, CAT could not be used as an alternative - unless all 9999 appropriately subscripted elements were explicitly referenced as its arguments.

Using PEEKC for Applying String Search Functions Across Variables

The ability of PEEKC to help represent a variable list or an array as a contiguous string lends itself directly to the application of the powerful set of SAS string searching functions to whole collections of variables or array items. For instance, suppose that each row in a data set LOOKUP contains nine character variables from A to I, whose lengths

vary from $4 to $9, and populated with a kind of codes. We need to search each of the variables A--I for the value of another character variable called SRCHFOR and return 1 if it is found in any of them, and 0 otherwise. The standard approach is to group the variables into an array, organise a DO-loop, and iterate through each item. The step below uses PEEKC conjugated with INDEXW:

```
data lookup ;
   informat srchfor $6. a b c d $4. e f g $7. h i $8. ;
   input (srchfor a b c d e f g h i) ($) ;
cards ;
666666 1 22 333 4444 55555 666666 7777777 8888888 9999999
888888 1 22 333 4444 55555 666666 7777777 8888888 9999999
55555  1 22 333 4444 55555 666666 7777777 8888888 9999999
****** 1 22 333 4444 55555 666666 7777777 8888888 9999999
;
run ;

data _null_ ;
   set lookup ;
   array arr [*] $9 a -- i ;

   found = ^^ indexw (peekc (addr(a), 81 ), srchfor) ; * ^^ normalizes to std boolean ;

   put found= srchfor $6. +1 (a--i) (~) ;
run ;
-------------------------------------------------------------------
found=1 666666 1 22 333 4444 55555 666666 7777777 8888888 9999999
found=0 888888 1 22 333 4444 55555 666666 7777777 8888888 9999999
found=1 55555  1 22 333 4444 55555 666666 7777777 8888888 9999999
found=0 ****** 1 22 333 4444 55555 666666 7777777 8888888 9999999
```

Note that the array is declared with the character length one byte greater than that of the longest variable it incorporates. This is needed to ensure that the INDEXW function will always have a blank between words to rely upon.

It goes without saying that practically all string-searching functions, such as INDEX, INDEXW, INDEXC, VERIFY et al., can be used in the manner outlined above. Because this method avoids (explicit) looping and replaces a serial array search with a string-searching function with a much faster underlying algorithm, it can be particularly valuable when more than one look-up variable needs to be searched.

Even more intriguing is the fact that although the method uses the PEEKC function - returning a character string - it can be also used in the same manner to search a list of numeric variables without ever converting them to character strings. However, it is not surprising. After all, in the physical memory, numeric variables are nothing but 8-byte character strings representing their RB8 images. Consider the example similar to the above, but with all numeric variables:

```
data a ;
   input srchfor a b c d e f g h i ;
cards ;
666666 1 22 333 4444 55555 666666 7777777 8888888 9999999
888888 1 22 333 4444 55555 666666 7777777 8888888 9999999
 55555 1 22 333 4444 55555 666666 7777777 8888888 9999999
     . 1 22 333 4444 55555 666666 7777777 8888888 9999999
;
run ;

data _null_ ;
   set a ;

   found = ^^ index (peekc (addr(a), 72), put (srchfor, rb8.)) ;
```

```
   put found= srchfor z6. +1 (a--i) (~) ;
run ;
-------------------------------------------------------------------
found=1 666666 1 22 333 4444 55555 666666 7777777 8888888 9999999
found=0 888888 1 22 333 4444 55555 666666 7777777 8888888 9999999
found=1 055555 1 22 333 4444 55555 666666 7777777 8888888 9999999
found=0      . 1 22 333 4444 55555 666666 7777777 8888888 9999999
```

Note how this code goes about its business:

1.The variables A--I coming from A are all retained and hence positioned at equidistant contiguous memory addresses spaced 8 addresses apart.
2.PEEKC extracts all of them at once starting from the address of the first variable ADDR(A) for the length of (9 variables)*(8 bytes each)=72 into the expression serving as the first argument to INDEX.
3.At this point, the 72 bytes of the expression represent nothing more than

```
      put (a, rb8.) || put (b, rb8.) ... || ... put (h, rb8.) || put (i, rb8.)
```

So, in order to find whether the SRCHFOR value is present in this string, all we need to do is search for put(srchfor,rb8.), because as stored in memory, that is what SRCHFOR is.

Still more curious tricks can be pulled by applying this technique to looking for a series of numbers in an array. Imagine that we have a large temporary numeric array and want to establish whether the sequence of numbers (2.72, -3.14, 9.81) is present among its elements. It is not difficult to do in a loop, but definitely requires a certain skill. Using PEEKC requires only a proper sizing and application of the INDEX function:

```
data _null_ ;
   srchmem = put (2.72, rb8.) || put (-3.14, rb8.) || put (9.81, rb8.) ;

   array NP [-500 : 499] _temporary_ (1 2 3 2.72 -3.14 9.81 4 5 6) ;
   lkup = put (peekc (addr (NP[lbound(NP)]), 8 * dim(NP)), $8000.) ;
   found = ^^ index (lkup, srchmem) ;
   put 'Found in NP? ' found ;

   array NA [0 : 999] _temporary_ (1 2 3 4 5 6 7 8 9 -3.14 2.72 9.81) ;
   lkup = put (peekc (addr (NA[lbound(NA)]), 8 * dim(NA)), $8000.) ;
   found = ^^ index (lkup, srchmem) ;
   put 'Found in NA? ' found ;
run ;
--------------
Found in NP? 1
Found in NA? 0
```

Having established a firm mental picture of a numeric array as a contiguous sequence of RB8 images in the physical memory starting at the initial array address, we can let our programming imagination fly. This step, for example, counts the number of times the sequence of the values 1, 2, 3 is encountered in the array NN by determining the difference between the length of the original array string image and one with the "search string" put(1,rb8.) || put(2,rb8.) || put(3,rb8.) translated to blanks and compressed.

```
data _null_ ;
   array nn [23] (0 0 0 , 1 2 3, 0 0 , 1 2 3, 0 0 0 , 1 2 3, 4 5, 1 2 3, 0) ;

   str_all = put (peekc (addr(nn[1]), 23*8), $184.) ;
   srchfor = put(1,rb8.) || put(2,rb8.) || put(3,rb8.) ;
   str_twd = tranwrd (str_all, srchfor, '') ;
   len_all = length (str_all) ;
   len_twd = length (compress(str_twd)) ;
```

```
   n_123 = (len_all - len_twd) / 24 ;
   put n_123 = ;

  *-------- Same using a single (convoluted!) expression --------*;

   n_123 = (length (put (peekc (addr(nn[1]), 23*8), $184.)) -
            length (compress(tranwrd
                   (put (peekc (addr(nn[1]), 23*8), $184.),
                    put(1,rb8.)||put(2,rb8.)||put(3,rb8.), '')))
           ) / 24 ;

   put n_123 = ;
run ;
----------------------
n_123=4
n_123=4
```

The part following the comment line shows how that can be achieved, if desired, using a single expression, which can be used to create a function-style macro directly returning the count value (and, given the convoluted nature of the expression, for job security).

Perhaps a less unusual problem, which can be readily helped by PEEKC called in the above fashion, is counting the number of missing values in a temporary array or large array whose variables do not form a prefixed list (and OF <variable list> syntax cannot be used). The example below shows how it can be macroized function-style, as hinted above:

```
%Macro aNmiss (aname, lbound, hbound) ;
   %local alen ;
   %let   alen = %eval (8 * (&hbound-&lbound+1)) ;

   (  length (put (peekc (addr(&aname[&lbound]), &alen), $&alen..)) -
      length (compress(tranwrd
     (put (peekc (addr(&aname[&lbound]), &alen), $&alen..),
      put(.,rb8.), "")))
    ) / 8
%mEnd aNmiss ;

data _null_ ;
   array nn [0 : 2999] _temporary_ ( 3000 * 1 ) ;

   nn [   2] = . ;
   nn [  22] = . ;
   nn [ 222] = . ;
   nn [2222] = . ;

   nmiss = %aNmiss (nn , 0 , 2999) ;

   put nmiss = ;
run ;
-------
nmiss=4
```

PEEKC Length Limitations

It must be reiterated that the techniques described in this section are still limited to arrays whose overall length in bytes (or the number of consecutive addresses they occupy in the physical memory) is capped at 32767. For a

numeric array (or character $8 array) the upper limit is thus 4095 elements. Of course, this limitation can be easily circumvented, if need be, by breaking the array in manageable chunks and calling PEEKC more than once.

CALL POKE(): THE INVASIVE OPERATION

So far, we have been in the read-only mode as far as APP functions are concerned. The ADDR and PEEK functions only allow for extracting data from the physical memory. Naturally, this limits the universe of tasks they can serve, even though, as we have seen above, the read-only APP functions are quite useful in their own right.

Now we shall turn our attention to the CALL POKE (or POKELONG) routine, which is intended to write data directly into a physical memory address. As we shall see later, this action alone can engender a certain degree of controversy, but for the time being let us concentrate on its capabilities.

In principle, using the CALL POKE routine is simple. Give it:

1.physical address from where to begin to write to memory (it can be an expression)
2.character expression that needs to be written
3.number of consecutive addresses including and above the starting address as a receptacle

and the routine will stick the string into the physical memory exactly as specified. Note that for #2, a numeric expression will not work, and, contrary to its habit, SAS will not even attempt an automatic conversion. The simplest way to introduce the workings of CALL POKE is, as always, by the force of example. This step shows a few:

```
data _null_ ;
 *-call poke character literal----------------*;
   c_into = '.....................................';

   call poke ('call poke a literal', addr(c_into), 20) ;
   put c_into = ;

 *-call poke character expression-------------*;
   c_into = '.....................................';
   c_from = 'call poke ' || 'a character expression' ;

   call poke (c_from , addr(c_into), vlength(c_from)) ;
   put c_into = ;

 *-poke numeric literal as RB8. image---------*;
   n_into = 3.1416 ;

   call poke ( put (2.71828, rb8.), addr(n_into), 8) ;
   put n_into = ;

 *-poke numeric expression as RB8. image------*;
   n_into = 1000 ;
   n_from = n_into * 1000 ;

   call poke ( put (n_from, rb8.), addr(n_into), 8) ;
   put n_into = ;

 *-making 5 consecutive numeric vars missing---*;
   retain a 1 b 2 c 3 d e 5 ;

   plug5 = put ( repeat (put(.,rb8.), 4), $40.) ;

   call poke ( plug5, addr(a), 5 * 8) ;
   put (a -- e) (=) ;
run ;
```

```
-----------------------------------------
c_into=call poke a literal..................
c_into=call poke a character expression.....
n_into=2.71828
n_into=1000000
a=. b=. c=. d=. e=.
```

A few minutes of going over these examples and giving them some thought should suffice to get a good feel of the routine's modus operandi. Now let us proceed to more specialized examples. Most of them originated from applying APP techniques to answer questions asked on SAS-L, and some are taken from the authors' real-world consulting practices.

Using CALL POKE to Explode a String into an Array

Imagine that we need to process a large number of records from a SAS data set UNPROCESSED containing a long (say $1000) character variable STR, and suppose that we want to manipulate its individual bytes as 1-byte elements of an array ASTR - for instance, because doing the same with the notoriously slow SUBSTR function kills performance. In some languages (notably, COBOL) it is possible to define a contiguous portion of memory simultaneously as a string and an array. The Data step compiler cannot place STR and ASTR at the same address, which means that the bytes of STR have to be moved to the elements of ASTR at the run-time. However, to do this, we first need to extract each byte from the string in a loop using the same SUBSTR function, so any potential performance gain from using the array in the first place may be lost. Using CALL POKE as shown below to move all bytes of STR to ASTR *en masse* is a more efficient approach to the problem:

```
%let len = 1000 ;

data _null_ ;
   length str $ &len ;                        * define STR ;
   str = repeat ('*', &len - 1) ;             * test non-blank characters ;

   array astr [1 : &len ] $ 1 _temporary_ ;   * define array ASTR ;

   addr_astr_1 = addr (astr[1]) ;             * get astr[1] address just once ;

   do test_repeat = 1 to 4.32e6 ;             * repeat mass move to evaluate performance ;
      call poke (str , addr_astr_1 , &len) ;  * move STR into ASTR en masse ;
   end ;
 * do test_repeat = 1 to 1e4 ;                * repeat loop move to evaluate performance ;
 *    do j = 1 to &len ;
 *       astr[j] = substr (str, 1) ;          * move STR bytes to ASTR one at a time ;
 *    end ;
 * end ;
run ;
```

Note that above, CALL POKE needed to be called only once. The only reason it is enveloped in a loop was just to see how much faster this en masse move works compared to the traditional looping technique. On our test machine (SAS version 9.1 TS Level 1B0 running under XP Pro, 2-way 1 GHz PIII, 1 GB), the standard one-by-one move (commented out in the step above) executes the complete string-to-array explosion 10,000 times in 2.50 seconds. Adjusting the number of test iterations reveals that during the same time, CALL POKE does the same 4,320,000 times, i.e. it runs approximately 400 times faster. Adjusting the string (and, accordingly, array) length shows that the longer the string, the greater the advantage. For example, at the maximum of 32767, the en masse explosion occurs about 550 times faster than when moving bytes one at a time.

Of course, the method is not limited to the exploding of a string into an array in 1-byte chunks. The array element length can be arbitrary, the only obvious requirement being that (a) the length of the string to be exploded must be a multiple of the array element length, and (b) the total number of bytes in the string must not exceed that in the array. The following step explodes the 256-byte collating sequence into 16 pieces, 16 characters each, and then displays their contents in the hex notation:

```
data _null_ ;
   array hexa [ 1 : 8 ] $ 32 ;

   call poke ( collate (0, 255), addr (hexa[1]) , 256) ;

   put 70 * '-' ;
   do j = 1 to 8 ;
      put hexa [j] = $hex64. ;
   end ;
run ;
----------------------------------------------------------------------
hexa1=000102030405060708090A0B0C0D0E0F101112131415161718191A1B1C1D1E1F
hexa2=202122232425262728292A2B2C2D2E2F303132333435363738393A3B3C3D3E3F
hexa3=404142434445464748494A4B4C4D4E4F505152535455565758595A5B5C5D5E5F
hexa4=606162636465666768696A6B6C6D6E6F707172737475767778797A7B7C7D7E7F
hexa5=808182838485868788898A8B8C8D8E8F909192939495969798999A9B9C9D9E9F
hexa6=A0A1A2A3A4A5A6A7A8A9AAABACADAEAFB0B1B2B3B4B5B6B7B8B9BABBBCBDBEBF
hexa7=C0C1C2C3C4C5C6C7C8C9CACBCCCDCECFD0D1D2D3D4D5D6D7D8D9DADBDCDDDEDF
hexa8=E0E1E2E3E4E5E6E7E8E9EAEBECEDEEEFF0F1F2F3F4F5F6F7F8F9FAFBFCFDFEFF
```

Run-Time Array Initialization

The most efficient way to initialize an array is to do it at compile time. Oftentimes, however, there is a need to do it at the run-time. A typical situation of this kind arises when it is necessary to aggregate a large array of data within each of a very large number of (usually short) BY-groups, and thus the array needs to be initialized every time a new BY-group starts - potentially millions of times.

For the following example, suppose that a data set SCORES contains a number of numeric scores, say score1-score11; and that the observations are grouped by a variable KEY:

```
data scores ;
   input key $ score1 - score11 ;
cards ;
A  1 5 8 3 8 5 7 3 8 5 2
A  2 3 8 4 6 3 9 9 3 5 9
B  2 3 8 4 2 3 9 9 3 5 9
B  8 3 5 8 2 8 2 7 5 2 3
B  1 3 5 5 9 5 1 9 5 5 5
C  7 3 5 8 2 8 2 7 5 2 5
C  1 3 5 5 9 5 1 9 5 3 5
run ;
```

For each distinct KEY, we need to produce an aggregate score AGG_SCORE1 as a product of some preset initial value and SCORE1 values within the KEY group, then do the same for SCORE2, and so on, the preset initial values being specific for each score variable. It means that before each group starts, the initial values have to be moved to AGG_SCORE1-AGG_SCORE11 overwriting their contents, and at the end of the group, they have to be written out. The step below does just that:

```
data agg_scores ( keep = key agg_: ) ;
   array init_nums [11] _temporary_ (01 03 05 07 11 13 17 19 23 29 31) ;

   array agg_score agg_score1-agg_score11 ;
   array     score     score1-    score11 ;

   retain init_rb8_str init_agg_addr ;

   if _n_ = 1 then do ;
      init_rb8_str  = put (peekc (addr (init_nums[1]), 88), $88.) ;
```

```
      init_agg_addr = addr (agg_score1) ;
   end ;

   call poke (init_rb8_str, init_agg_addr, 88) ;

   do until ( last.key ) ;
      set scores ;
      by  key ;
      do over score ;
         agg_score = agg_score * score ;
      end ;
   end ;
run ;
```

Note that the initial values are *not* moved from INIT_NUMS to AGG_SCORE one by one. Instead, the following occurs:

1.At the beginning of the step, (a) the PEEKC function snatches the whole 88 bytes of INIT_NUMS array, and this RB8 image of the array is stored in a retained string INIT_RB8_STR, and (b) the starting address of the array AGG_SCORE is placed in the retained variable INIT_AGG_ADDR.
2.Before each BY-group (i.e. before the DoW-loop begins to iterate) INIT_RB8_STR is moved en masse into the array AGG_SCORE, thus initializing it.
3.The initial values are multiplied by corresponding SCORE values for each observation within the group.
4.After the group is processed, the aggregated array elements are output by the implicit OUTPUT statement.

Admittedly, this technique is overly fancy for dealing with an 11-element array and a dozen of records, but in the real life, where the number of both array elements and BY-groups often happens to be vastly greater, it can make a real performance difference.

Copying Arrays En Masse

An alert reader must have already noticed that the combination of PEEK and POKE as outlined above in effect provides a means of copying arrays without the need to copy their elements one by one. As an icing on the cake, it can be done in one fell swoop using a single expression. Of course, for the step below to work properly, the array must be equivalent, i.e. have the same types and element lengths, plus the dimension of the array being copied cannot exceed that of the target.

```
data _null_ ;
   array tocopy [-5 : 4] _temporary_ (0 1 2 3 4 5 6 7 8 9) ;
   array target [ 0 : 9]             (9 8 7 6 5 4 3 2 1 0) ;

   call poke (put (peekc (addr(tocopy[-5]), 80), $80.), addr(target[0]), 80) ;

   put 19 * '-' / (target[*]) (~) ;
run ;
-------------------
0 1 2 3 4 5 6 7 8 9
```

En Masse Array Modification

On the other hand, the technique can be very useful when there is a need to alter the array in some way. For example, suppose that we have a $1 array, and we want to have (a) all consecutive elements holding '1, '2', '3', '4' values to be replaced with the sequence 'A', 'B', 'C', 'D', and then (b) the entire array reversed. Without APP functions, it would amount to a fair chunk of Do-loop array programming; with APP functions, it boils down to:

```
data _null_ ;
   array cc [14] $ 1 ('0' '1' '2' '3' '4' '5' '6' '7' '1' '2' '3' '4' '5' '1') ;

** Using intermediate variable to hold array image ;
```

```
   addr1  = addr (cc[1]) ;
   cc_img = put( peekc (addr1, 14), $14.) ;
   cc_img = tranwrd (cc_img, '1234', 'ABCD') ;
   cc_img = reverse (cc_img) ;
   call poke (cc_img, addr1, 14) ;
   put 41 * '-' / (cc[*]) (+1) ;

** Using single convoluted expression to restore status-quo ;

   call poke (reverse (tranwrd (put ( peekc (addr(cc[1]), 14), $14.), 'DCBA', '4321')),
                  addr(cc[1]), 14) ;

   put 41 * '-' / (cc[*]) (+1) ;
run ;
----------------------------------------
 1  5  D  C  B  A  7  6  5  D  C  B  A  0
----------------------------------------
 0  1  2  3  4  5  6  7  1  2  3  4  5  1
```

Moving Data in Blocks within an Array

So far, we have been using the power of APP functions to move data en masse between different arrays or from and/or to character strings. However, there is nothing that prevents us from using the same exact concept for moving array elements in whole blocks within a single array.

For example, sometimes the elements of an array need to be shifted up or down stream because of either bizarre business requirements or a poor initial data model. One of the authors (PD) once had to "fix" code written years ago, where a "monthly history" array H(3600) containing 3 years worth of some monthly data had to be "receded" 1200 buckets down stream (the originator programmer assumed that the program would not run for more than 3 years). Now the trailing year had to be dropped, and the data from the next year would start trickling in. The buckets at the top of the array whence the data were shifted had to be emptied. So it was either to move H(i) down one at a time repeating it ad nauseam, or to come up with something else. Something else went along the (now familiar to the reader) lines:

```
%let n = 3600 ;
%let m = 1200 ;

data _null_ ;
   array h ( 1:&n ) ;
   length miss $%eval(&m*8) move $%eval((&n-&m)*8) ;

   addr0 = addr (h(        +1)) ;
   addr1 = addr (h(     &m +1)) ;
   addr2 = addr (h(&n - &m +1)) ;
   miss = repeat (put(. , rb8.), &m - 1) ;
   move = peekc  (addr1, (&n-&m)*8) ;

   do until (eof) ;
      set hfile end = eof ;
      call poke (move, addr0, %eval((&n-&m)*8)) ;
      call poke (miss, addr2, %eval((   &m)*8)) ;
      output ;
   end ;
run ;
```

Above, the necessary starting addresses are computed first, and a replacement block consisting of 1200 missing elements is prepared. Then, for each record, the array block H1200-H3600 is extracted into MOVE, the latter is slotted into the array from H1 upwards, and the block containing missing values plugs the gap from H2401 to H3600.

The Great SAS Doc Controversy

The part of SAS documentation describing the CALL POKE routine contains the following grave warning:

> *The CALL POKE routine is intended only for experienced programmers in specific cases. If you plan to use this routine, use extreme care both in your programming and in your typing. Writing directly into memory can cause devastating problems. This routine bypasses the normal safeguards that prevent you from destroying a vital element in your SAS session or in another piece of software that is active at the time.*

Holy smoked prunes! It is enough to discourage even the bravest SAS programmer from using APP functions altogether and begs the question: Is CALL POKE really *that* bad? Well, not in our opinion, but different folks hold different views, so it had been only the matter of time before a (civilized) discussion on the subject erupted on SAS-L. The exchange between one of the authors (P.D.) and renowned SAS guru Mike Rhoads (M.R.) touches on the most important facets of the issue. Here is the gist (marginally edited, and only in the parts marked P.D.):

M.R.: *I feel compelled to put my two cents' in about PEEK and POKE. These routines can be great fun to play with and figure out how SAS is storing data internally. I am sure there are specialized circumstances where their use can be justified in production programs.*

P.D.: *The circumstances when APP functions can and should be used in SAS programs (both ad hoc and production) are the same as for any other SAS tool: When they are the best for the job, i.e. provide the optimum combination of short, clear, correct, and well-performing code.*

M.R.: *HOWEVER, I would strongly recommend that the SAS community not go "PEEK and POKE crazy"...*

P.D.: *I do not think any real danger of going "APP crazy" exists. I have to admit, however, that judging from the hashing experience, the infection may prove to be contagious, and some vaccination may be indeed warranted.*

M.R.: *... and use these routines in other than exceptional circumstances.*

That is kind of hard to swallow. There is no more reason to assign the APP function the "exceptional circumstances" status than to any other SAS tool. They are legitimate Base SAS functions, behave precisely as documented, and what they do is quite simple. Many other SAS tools much more complex, have more caveats, require deeper understanding, may produce unexpected results (think of MERGE as of a quintessence of all of the above), or are version and/or platform dependent. However, they are used under all kinds of circumstances, not necessarily exceptional. The only real concern is POKE. SI caution to that effect can be interpreted this way: Do not write to foreign memory or use POKE if you do not know what you are doing. Not writing to foreign memory is very easy to avoid, for this will never happen as long as the area of memory POKE writes to already belongs to the domain allocated to existing variables. Knowing what one is doing is not unique to using APP, either. In this respect, APP functions are no different from any other SAS routine, each requiring from the programmer to know where and how it should be used.

M.R.: *I am particularly concerned about programs (such as the "COMPRESS array" thread) that make assumptions about how variables are stored in the Program Data Vector.*

P.D.: *I emphatically agree that programs making assumptions about the software are unacceptable. But the program in question makes no such assumptions whatsoever! On the contrary, it is based on, and works because of, the documented knowledge about the way variables are stored in PDV.*

M.R.: *SAS Institute does not guarantee that these decisions and algorithms will not change between versions of the software, nor should they. If they did, we would be stuck forever with less efficient storage techniques.*

P.D.: *Actually, there is no guarantee than anything will behave exactly the same in the versions to come. However, it is highly unlikely that in a future version, TRIM will be used to round numbers, and ROUND - to advance dates. By the same token, it is just as unlikely that the underlying architecture will have changed to the point where temporary*

bytes are scattered haphazardly all over the memory instead of occupying consecutive addresses. This would make no sense and fly in the face of the basic principles of sound software engineering.

CONCLUSION

APP functions (PEEK, PEEKC, CALL POKE routine, and their "LONG" counterparts) are SAS tools for directly reading from, and writing to, the physical memory, primarily in the SAS Data step. Although it is possible to use them in SQL, we have not explored the possibility enough to recommend such usage. However, we do recommend that you try it yourself and tell us about the results! We will be all ears.

APP functions are most useful in organizing en masse data movement between large collections of variables in whole blocks, and hence the most natural target for their application is SAS arrays and variables forming contiguous lists in memory (all retained or not).

Although in our practice, none of APP functions has ever given us any grief, not even in spite of our concerted efforts to break the threshold by intentionally, the caution about the CALL POKE routine in the SAS documentation ought to be heeded. One bulletproof guarantee is this: If you never write outside of the memory areas already allocated by the SAS compiler for named variables and temporary arrays, things will never get out of kilter.

REFERENCES

Except for the SAS Documentation, this paper and all its predecessors co-authored by P.D. Are the only APP sources in the SAS literature. In addition to the examples given in the paper, many more, together with fine discussions of relative merits of the APP techniques, can be found in SAS-L archives starting as early as the beginning of 1998 by following the link

http://www.listserv@listserv.uga.edu/archives/sas-l.html

and searching for such key words as PEEK and POKE.

ACKNOWLEDGEMENTS

Gratitude to all SAS-L participants - both inquirers and responders - who civilly tolerated the audacity of using the threads as an experimental APP setup. The most valuable contributions came from those who expressed more than casual interest in the topic - both publicly on SAS-L and in private communications. Cordial thanks go to (in no particular order) Peter Crawford, Rob Workman, Ian Whitlock, Gregg Snell, Mike Rhoads, Howard Schreier, Sigurd Hermansen, William Viergever, Koen Vyverman, Nat Wooding, Joe Kelley, David Cassell, Venky Chakravarthy, Dale McLerran, Mike Raithel, David Johnson, Ya Huang, Tim Berryhill, Karsten Self, Ray Pass, Gerry Pauline, Jack Hamilton, Pete Lund, Rick Aster, Mark Terjeson, Ron Fehd, Paul Kent, Warren Sarle.

AUTHOR CONTACT INFORMATION

Paul Dorfman
4437 Summer Walk Court
Jacksonville, FL 32258
(904) 260-6509
sashole@bellsouth.net