Paper BtB-07

PEEKing at Roadway Segments

Carol A. Martell, UNC Highway Safety Research Center

ABSTRACT

In this practical application of some special SAS® functions and CALL routines we control the location in memory of variables to be compared from one observation to the next. Forcing the variables to be written adjacent to one another enables us to treat them as a single variable. We use the special functions PEEKC, ADDR, and CALL POKE, along with a DOW loop.

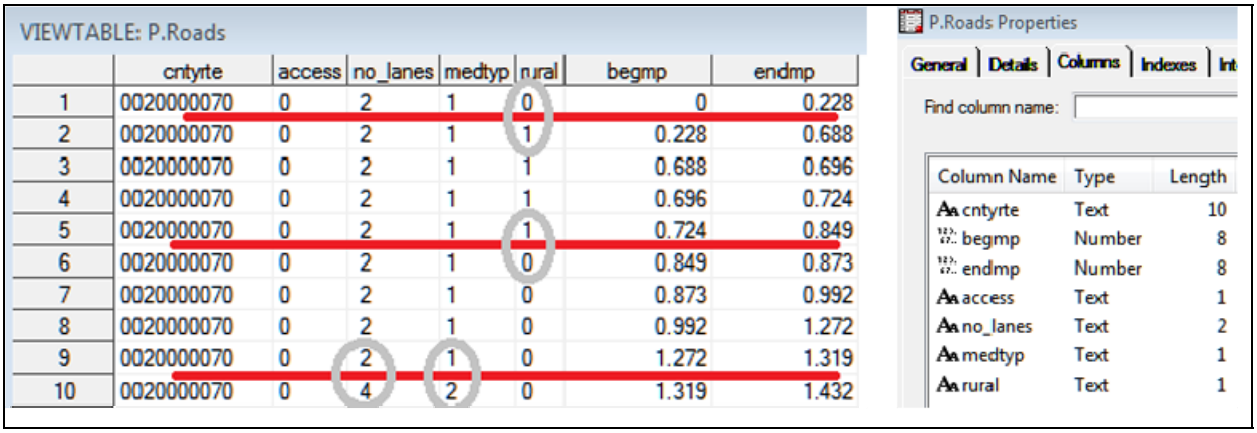
INTRODUCTION

This paper does not attempt to cover all the aspects of the special functions, dubbed the APP functions (for ADDR-PEEK-POKE). Peter Crawford and Paul Dorfman have already done so. This paper demonstrates how we use the functions, which read directly from and write directly to memory, to streamline the process of re-segmenting roadway data.

Roadway traffic records contain descriptions of contiguous, homogeneous stretches of roadway, called segments. The begin- and end-points for the segments are milepost values. The homogeneity encompasses many attributes. There are times when some attributes are irrelevant for a study, and the segments need to be redefined using a smaller subset of variables. For example, if shoulder width is the only difference between two adjacent segments, and shoulder width variance is to be ignored, the two segments are collapsed into one. This paper explores various ways to use the APP functions to facilitate collapsing the data. Our examples are carried out using character variables in a 32-bit environment, so we use PEEKC, ADDR and CALL POKE rather than PEEKCLONG, ADDRLONG and CALL POKELONG.

THE DATA

A partial listing of the input roadway data is shown in Figure 1. The goal is to collapse contiguous records when the variables of interest do not change. The collapsed record has new boundaries, and is given new begin and end milepost values to reflect these new boundaries. We name the new begin and end milepost variables for our collapsed segments *bmp* and *emp*, respectively. The variables *begmp* and *endmp* represent begin and end milepost values in the input data, which is sorted by *cntyrte (*roadway id*)* and *begmp*. Red lines show where homogeneity is lost; the changing values are circled in grey. After collapsing, the first three segments should have the following pairings for *bmp* and *emp*: (0,0.228), (0.228,0.849), and (0.849,1.319). All variables are character except for the milepost values.



VIEWTABLE: P.Roads

	cntyrte	access	no_lanes	medtyp	rural	begmp	endmp
1	0020000070	0	2	1	0	0	0.228
2	0020000070	0	2	1	1	0.228	0.688
3	0020000070	0	2	1	1	0.688	0.696
4	0020000070	0	2	1	1	0.696	0.724
5	0020000070	0	2	1	1	0.724	0.849
6	0020000070	0	2	1	0	0.849	0.873
7	0020000070	0	2	1	0	0.873	0.992
8	0020000070	0	2	1	0	0.992	1.272
9	0020000070	0	2	1	0	1.272	1.319
10	0020000070	0	4	2	0	1.319	1.432

Figure 1. Data listing and variables of interest

The segments shown in Figure 1 are contiguous, meaning that each segment begins where the last segment ends. This is not the case, however, throughout the data. Occasionally, there will be a gap, where *begmp* does not equal the previous record's *endmp.* We begin a new segment after a gap.

Checking for gaps requires comparing two different variables - the begin milepost of the current record must be compared with the end milepost of the previous record. Things start getting ugly when we perceive a gap. The values

to be output are those of the previous record, rather than the values in the Program Data Vector (PDV,) which is holding the current record. We examine one way to accomplish the task without APP functions. When a new segment begins, we store the new values in a set of retained variables. The values in each subsequent record are compared with the retained values to look for changes. When a change is found, the retained values are renamed as they are written to the output table. The retained variables are then populated with the values from the new segment. The first and last records need special treatment. For the first record, we must initialize *bmp*, *emp*, and the array we use for comparison. For the last record, we must write the final segment record. The following code collapses the data without the use of APP functions.

```
DATA segs(DROP=cntyrte access no_lanes medtyp rural
    RENAME=
    (cntyrte1=cntyrte access1=access no_lanes1=no_lanes medtyp1=medtyp
    rural1=rural)
    );
SET s.roads END=eof;
LENGTH cntyrte1  $ 10 access1  $ 1 no_lanes1  $ 2 medtyp1 rural1  $ 1;
ARRAY currnt(5) cntyrte access no_lanes medtyp rural;
ARRAY compar(5) cntyrte1 access1 no_lanes1 medtyp1 rural1;
RETAIN bmp emp cntyrte1 access1 no_lanes1 medtyp1 rural1;
IF _n_=1 THEN DO;
    bmp=begmp;
    emp=endmp;
    DO j=1 TO 5;
           compar(j)=currnt(j);
           END;
    RETURN;
    END;
IF cntyrte NE cntyrte1 OR access NE access1 OR no_lanes NE no_lanes1
OR medtyp NE medtyp1 OR rural NE rural1 OR begmp NE LAG(endmp)
    THEN DO;
           segnum+1;
           OUTPUT;
           bmp=begmp;
           DO j=1 TO 5;
               compar(j)=currnt(j);
               END;
           END;
emp=endmp;
IF eof THEN DO;
    segnum+1;
    OUTPUT;
    END;
RUN;
```

CONTROLLING MEMORY LOCATIONS

We begin to examine how the APP functions can be used to compare a single value, rather than five, across observations. If we can cajole SAS into storing these variables adjacent to one another, we will be able to treat them as one long string. We begin by seeing what the default locations are for these variables. We know that SAS stores groups of variables contiguously. The groups are retained numeric, retained character, non-retained numeric, and non-retained character. Because we are reading these character variables from a SAS table, they belong in the retained character storage group. The ADDR function returns the memory address for the first byte of a variable. For each variable of interest, we create a new variable containing that memory address and write it to the SAS log.

```
DATA roads;
SET p.roads(OBS=1);
a_cntyrte=ADDR(cntyrte);       PUT a_cntyrte=;
a_access= ADDR (access);       PUT a_access=;
a_no_lanes= ADDR (no_lanes);   PUT a_no_lanes=;
a_medtyp= ADDR (medtyp);       PUT a_medtyp=;
a_rural= ADDR (rural);         PUT a_rural=;
RUN;
```

Here are the values revealed in the SAS log. We know that *cntyrte* is 10 characters long. The variable stored in memory adjacent to *cntyrte* will begin in position 263653850. We see that our next variable of interest does not begin in that position.

```
a_cntyrte =      263653840
a_access =       263653866
a_no_lanes =     263653875
a_medtyp =       263653880
a_rural =        263653881
```

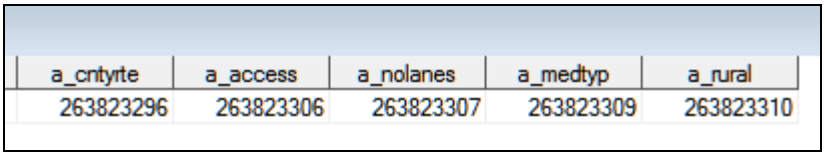
Our variables of interest are only a few of the variables in the source data. When we display the memory address of all the character variables in the input data set, we see that they all are indeed stored adjacent to one another, but our 5 variables are scattered throughout. Although the actual memory addresses change with a new run, the offsets match what we saw above: *a_access* begins 26 bytes after *a_cntyrte*, and *a_no_lanes* begins 9 bytes after *a_access*. The intervening variables occupy the space between our variables of interest. Here is a partial listing of all the retained character variables:

```
a_cntyrte =      263829240
a_improve1 =     263829250
a_town =         263829252
a_func_cls =     263829256
a_rte_type =     263829258
a_aadt_yr =      263829259
a_terrain =      263829263
a_pop_grp =      263829264
a_med_type =     263829265
a_access =       263829266
a_pct_trk1 =     263829267
a_county =       263829269
a_spd_limt =     263829271
a_surf_typ =     263829273
a_no_lanes =     263829275
…
```

Next, we experiment with methods that might keep our subset of 5 variables adjacent to one another. Using PROC SQL and selecting our variables of interest first might force adjacency.

```
PROC SQL;
CREATE TABLE roads AS SELECT cntyrte, access, no_lanes, medtyp, rural, *,
ADDR(cntyrte) AS a_cntyrte,
ADDR(access) AS a_access,
ADDR(no_lanes) AS a_nolanes,
ADDR(medtyp) AS a_medtyp,
ADDR(rural) AS a_rural
FROM p.roads(obs=1);
QUIT;
```

The VIEWTABLE seen in Figure 2 shows that PROC SQL arranges the variables as listed. A warning in the log is triggered for each explicitly named variable, because SELECT…* attempts to load each explicitly named variable a second time. Now *cntyrte* occupies 10 bytes, followed by *access* using 1 byte, and so forth. PROC SQL works, but since we need to process this data sequentially, we return to the DATA step for more experimentation.



a_cntyrte	a_access	a_nolanes	a_medtyp	a_rural
263823296	263823306	263823307	263823309	263823310

Figure 2. PROC SQL provides adjacency

From reading previously published papers about the APP functions, we know that an ARRAY statement should help. We find, however, that if we use the ARRAY statement before the SET statement, SAS makes unfortunate assumptions about the variable lengths.

```
DATA roads;
ARRAY myvars [5] $  cntyrte  access  no_lanes medtyp rural;
SET p.roads(OBS=1);
a_cntyrte= ADDR(cntyrte);      PUT a_cntyrte=;
a_access= ADDR(access);        PUT a_access=;
a_no_lanes= ADDR(no_lanes);    PUT a_no_lanes=;
a_medtyp= ADDR(medtyp);        PUT a_medtyp=;
a_rural= ADDR(rural);          PUT a_rural=;
RUN;
```

We see a warning in the SAS log about multiple length definitions, and learn that *cntyrte* will be truncated. Examining the addresses in the log, we see that our variables are adjacent, but now have lengths of 8 bytes. This is the SAS default with ARRAY definitions that introduce new variables. Here is the log:

```
WARNING: Multiple lengths were specified for the variable cntyrte by
         input data set(s). This may cause truncation of data.
a_cntyrte =      257807392
a_access =       257807400
a_no_lanes =     257807408
a_medtyp =       257807416
a_rural =        257807424
```

We move the ARRAY statement so that it follows, rather than precedes, the SET statement. We hope SAS will use the variable lengths from the input SAS table, and still store the ARRAY variables together.

```
DATA roads;
SET p.roads(OBS=1);
ARRAY x[5] $ cntyrte access no_lanes medtyp rural;
a_cntyrte= ADDR(cntyrte);      PUT a_cntyrte=;
a_access= ADDR(access);        PUT a_access=;
a_no_lanes= ADDR(no_lanes);    PUT a_no_lanes=;
a_medtyp= ADDR(medtyp);        PUT a_medtyp=;
a_rural= ADDR(rural);          PUT a_rural=;
RUN;
```

This approach works! Our variables are not only adjacent, but also have correct lengths.

```
a_cntyrte =      267520424
a_access =       267520434
a_no_lanes =     267520435
a_medtyp =       267520437
a_rural =        267520438
```

Since the ARRAY statement is processed at compilation time and the SET statement at execution time, it seems counter-intuitive that SAS would use information from a SET statement at compilation time. The reader can perform a test, compiling a DATA step that reads from a table. Replacing the table after compilation with one having different variable lengths and examining the results will show that metadata is accessed for a SET statement at compilation time.

Now that we can force our variables of interest to be adjacent to one another (contiguous in memory), let's prove that we can read them in as a single string. The combined length of our variables is 15. The PEEKC function in the following code returns the contents in memory for the 15 bytes beginning with byte 1 of the variable *cntyrte*. The value of that string should be the same as a string built by concatenating our 5 variables using the CAT function. The values do match:

```
DATA roads;
SET p.roads(OBS=1);
ARRAY x[5] $ cntyrte access no_lanes medtyp rural;
str=PEEKC(ADDR(cntyrte),15);
catstr=CAT(cntyrte,access,no_lanes,medtyp,rural);
PUT str= / catstr=;
RUN;

str =      00200000700 210
catstr =   00200000700 210
```

BABY-STEPPING WITH APP FUNCTIONS

Feeling confident (but not really) we take a baby step forward. We use what we have learned to perceive changes and assign new segment numbers to our original segments. We are not collapsing the data at this point. We write out every record read, adding the implicitly retained variable *segnum.* This segment number should not increment until a change occurs in any of our variables of interest. We create an explicitly retained variable to hold the value of our five variables (the contents of the 15 bytes in memory,) for comparison with each subsequent record. We call it *str* and assign it a length of 15. We replace the value in *str* only when we perceive a change or find a gap:

```
DATA roads;
LENGTH str $ 15;
RETAIN str;
SET p.roads;
ARRAY x[5] $ cntyrte access no_lanes medtyp rural;
IF PEEKC(addr(cntyrte),15) NE str OR begmp ne LAG(endmp) THEN DO;
    segnum+1;
    str=PEEKC(ADDR(cntyrte),15);
    END;
RUN;
```

A VIEWTABLE of the relevant variables in Figure 3, again marked to show desired break points due to value changes, reveals success. The value for *segnum* changes with each red line.

VIEWTABLE: Work.Roads

	segnum	cntyrte	access	no_lanes	medtyp	rural	begmp	endmp
1	1	0020000070	0	2	1	0	0	0.228
2	2	0020000070	0	2	1	1	0.228	0.688
3	2	0020000070	0	2	1	1	0.688	0.696
4	2	0020000070	0	2	1	1	0.696	0.724
5	2	0020000070	0	2	1	1	0.724	0.849
6	3	0020000070	0	2	1	0	0.849	0.873
7	3	0020000070	0	2	1	0	0.873	0.992
8	3	0020000070	0	2	1	0	0.992	1.272
9	3	0020000070	0	2	1	0	1.272	1.319
10	4	0020000070	0	4	2	0	1.319	1.432

Figure 3. Successful assignment of new segment numbers

Code to accomplish the same thing without using the direct memory addressing might look like this:

```
DATA roads;
SET p.roads;
```

```
IF cntyrte NE lag(cntyrte)
    OR access NE lag(access)
    OR no_lanes NE lag(no_lanes)
    OR medtyp NE lag(medtyp)
    OR rural NE lag(rural)
    OR begmp NE LAG(endmp)
    THEN segnum+1;
RUN;
```

There are many ways to test for differences between records and append a segment number. Here are two more:

Create a variable containing the memory string and compare it with its lag:

```
DATA roads;
LENGTH str $ 15;
SET p.roads;
ARRAY x[5] $ cntyrte access no_lanes medtyp rural;
str=PEEKC(ADDR(cntyrte),15);
IF str NE LAG(str) OR begmp ne LAG(endmp) THEN segnum+1;
RUN;
```

Read the data in a DOW loop:

```
DATA roads;
LENGTH str $ 15;
 DO j=1 BY 1 UNTIL(eof);
        SET p.roads END=eof;
        IF PEEKC(ADDR(cntyrte),15) NE str OR begmp NE lag(endmp)
               THEN segnum=SUM(segnum,1);
        str=PEEKC(ADDR(cntyrte),15);
        OUTPUT;
        END;
ARRAY x[5] $ cntyrte access no_lanes medtyp rural;
RUN;
```

However we go about appending a new segment number to our input records, we can now take the last baby step. We use BY group processing in a DATA step to output one record per *segnum* assigning the new begin and end milepost values *bmp* and *emp*. Figure 4 shows the resulting data, with new milepostings that span the collapsed input segments.

```
DATA segs;
SET roads;
BY segnum;
RETAIN bmp;
IF FIRST.segnum THEN bmp=begmp;
IF LAST.segnum THEN DO;
    emp=endmp;
    OUTPUT;
END;
RUN;
```

VIEWTABLE: Work.Segs

	segnum	bmp	emp	cntyrte	access	no_lanes	medtyp	rural
1	1	0	0.228	0020000070	0	2	1	0
2	2	0.228	0.849	0020000070	0	2	1	1
3	3	0.849	1.319	0020000070	0	2	1	0
4	4	1.319	4.393	0020000070	0	4	2	0
5	5	4.393	5.098	0020000070	0	4	1	0

Figure 4. One record per new segment

STRIDING WITH APP FUNCTIONS

Feeling even more confident, we abandon baby steps. To skip creation of an intermediate table we must juggle some variables, and introduce CALL POKE, which writes to a memory location. We read a record and immediately store the relevant memory string in variable *str*. We use the variable *str2* to hold values from the previous record for comparison. A non-iterative DO group, executed only for the first input record, copies *str* to *str2,* initializes *bmp* and *emp,* and, since there is no previous record for comparison, skips to the end of the DOW loop via CONTINUE. For all the remaining records, we compare *str (*new values*)* with *str2 (*old values*)* and we check for milepost gaps. If either condition is met, we increment *segnum* and perform a juggling trick. The PDV contains new roadway values, but we want to write out the previous record's values. CALL POKE takes those previous values, stored in *str2*, and writes them to the memory location in the PDV where our variables are stored. We output the record. The current values, held in *str,* are copied to *str2*, which is now ready to be compared with subsequent records. At end of file, we write the final record. Since the last segment read may constitute an entirely new segment, we plug *str,* rather than *str2,* back into the PDV before writing the record:

```
DATA segs;
LENGTH str str2 $ 15;
DO j=1 BY 1 UNTIL(eof);
    SET p.roads END=eof;
    str=PEEKC(ADDR(cntyrte),15);
    IF j=1 THEN DO;
           str2=str;
           bmp=begmp;
           emp=endmp;
           CONTINUE;
    END;
    IF str NE str2 OR begmp NE emp THEN DO;
           segnum=SUM(segnum,1);
           CALL POKE(str2,ADDR(cntyrte),15);
           OUTPUT;
           bmp=begmp;
           str2=str;
    END;
    emp=endmp;
    IF eof THEN DO;
           segnum=SUM(segnum,1);
           CALL POKE(str,ADDR(cntyrte),15);
           OUTPUT;
    END;
END;
ARRAY v (*) $ cntyrte access no_lanes medtyp rural;
RUN;
```

In real life, more would be required than simply collapsing the data into new segments. Segment accumulators would be initialized first in the j=1 DO group, and again when each new segment begins. Another CALL POKE can write *str* back into place if current variable values feed segment calculations. LAG values can be used for end-of-segment calculations involving variables not in *str*. These are shown in red, italicized, below:

```
DATA segs;
LENGTH str2 $ 15;
DO j=1 BY 1 UNTIL(eof);
    SET p.roads END=eof;
    str=PEEKC(ADDR(cntyrte),15);
    IF j=1 THEN DO;
           str2=str;
           bmp=begmp;
           emp=endmp;
           *initialize seg accumulators;
           CONTINUE;
    END;
    IF str NE str2 OR begmp NE emp THEN DO;
           segnum=SUM(segnum,1);
           *calc seg summaries – possibly use LAG functions;
```

```
            CALL POKE(str2,ADDR(cntyrte),15);
            OUTPUT;
            bmp=begmp;
            CALL POKE(str,ADDR(cntyrte),15);
            *initialize seg accumulators;
            str2=str;
    END;
    *accumulate seg stuff;
    emp=endmp;
    IF eof THEN DO;
            segnum=SUM(segnum,1);
            CALL POKE(str,ADDR(cntyrte),15);
            calc seg summaries – possibly use LAG functions;
            OUTPUT;
    END;
END;
ARRAY v (*) $ cntyrte access no_lanes medtyp rural;
RUN;
```

POKING AROUND

So far, we have been comparing variable values, looking for exact matches. Suppose that one of the variables used to define segments needs to be grouped. As a very simple example, suppose the values 0 and 1 for the variable *access* should be considered equivalent. In other words, all other things being equal, if one input segment has the value 1 and the next has the value 0, they will be treated as homogeneous segments. We can add one statement (in red, italicized, below) to accomplish this. CALL POKE replaces the value '0' with '1' in the 11th position of *str*

```
…
DO j=1 BY 1 UNTIL(eof);
    SET p.roads END=eof;
    str=PEEKC(ADDR(cntyrte),15);
    if access='0' then CALL POKE('1',ADDR(str)+10,1);
    IF j=1 THEN DO;
…
```

For a more complicated comparison, we consider using value ranges in a variable as homogeneous. We use the same input data, but ignore changes in the variable *rural.* Instead, we use the numeric variable *aadt.* For this variable, we consider the following ranges of values, reflected in this PROC FORMAT, to be homogeneous:

```
PROC FORMAT;
VALUE adtf 0='0'
1-999='1-999'
1000-4999='1,000-4,999'
5000-9999='5,000-9,999'
10000-19999='10,000-19,999'
20000-high='20,000+';
RUN;
```

VIEWTABLE: P.Roads

	cntyrte	access	no_lanes	medtyp	aadt	begmp	endmp
1	0020000070	0	2	1	11000	0	0.228
2	0020000070	0	2	1	11000	0.228	0.688
3	0020000070	0	2	1	11000	0.688	0.696
4	0020000070	0	2	1	18000	0.696	0.724
5	0020000070	0	2	1	18000	0.724	0.849
6	0020000070	0	2	1	18000	0.849	0.873
7	0020000070	0	2	1	18000	0.873	0.992
8	0020000070	0	2	1	18000	0.992	1.272
9	0020000070	0	2	1	18000	1.272	1.319
10	0020000070	0	4	2	18000	1.319	1.432
11	0020000070	0	4	2	21000	1.432	1.982
12	0020000070	0	4	2	24000	1.982	2.469
13	0020000070	0	4	2	24000	2.469	2.588
14	0020000070	0	4	2	24000	2.589	2.686

Figure 5. Data listing with new segment changes

The input data is seen in Figure 5 with lines indicating where the new segment breaks should belong. Now, our first two segment *bmp* and *emp* values should be (0,1.319) and (1.319,1.432). In the PROC FORMAT code seen above, the resulting format, *adtf,* supplies value labels with a maximum length of 13. An inefficient but illustrative way to incorporate this range comparison is to create a new variable, assigning it the value of PUT(aadt,adtf.). If we do so, we have a new character variable of length 13. It will not be, however, a retained character variable, so adding our new variable to the ARRAY statement does not place it adjacent to the other retained character variables:

```
DATA roads;
LENGTH c_aadt $ 13;
SET p.roads(OBS=1);
c_aadt=PUT(aadt,adtf.);
ARRAY v(*) $ cntyrte access no_lanes medtyp c_aadt;
a_cntyrte=addr(cntyrte);       PUT a_cntyrte=;
a_access=addr(access);         PUT a_access=;
a_no_lanes=addr(no_lanes);     PUT a_no_lanes=;
a_medtyp=addr(medtyp);         PUT a_medtyp=;
a_c_aadt=addr(c_aadt);         PUT a_c_aadt=;
RUN;;

a_cntyrte =      234957384
a_access =       234957394
a_no_lanes =     234957395
a_medtyp =       234957397
a_c_aadt =       234957520
```

The new variable, *c_aadt*, is not located 1 byte after *rural,* but is instead located 123 bytes later where, apparently, the non-retained character variables are stored; our new variable is in that storage group. We can bring it back into the fold by retaining it. Using a RETAIN statement brings the variable into the retained character group. Adding *c_aadt* to the ARRAY statement places it adjacent to the other variables in the array.

```
DATA roads;
LENGTH c_aadt $ 13;
RETAIN c_aadt;
SET s.roads(OBS=1);
c_aadt=PUT(aadt,adtf.);
ARRAY v(*) $ cntyrte access no_lanes medtyp c_aadt;
a_cntyrte=addr(cntyrte);       PUT a_cntyrte=;
```

```
a_access=addr(access);          PUT a_access=;
a_no_lanes=addr(no_lanes);      PUT a_no_lanes=;
a_medtyp=addr(medtyp);          PUT a_medtyp=;
a_c_aadt=addr(c_aadt);          PUT a_c_aadt=;
RUN;
```

```
a_cntyrte =          234971448
a_access =           234971458
a_no_lanes =         234971459
a_medtyp =           234971461
a_c_aadt =           234971462
```

We use the string in memory that starts in ADDR(*cntyrte)* and extends for a length of 27 to find changes:

```
DATA roads;
LENGTH c_aadt $ 13 str $ 27;
RETAIN c_aadt str;
SET s.roads;
c_aadt=PUT(aadt,adtf.);
ARRAY x[5] $ cntyrte access no_lanes medtyp c_aadt;
IF PEEKC(addr(cntyrte),27) NE str OR begmp ne LAG(endmp) THEN segnum+1;
str=PEEKC(ADDR(cntyrte),27);
RUN;
```

The results, seen in Figure 6 show the segment breaks due to *aadt* changes as well as another due to a gap.

VIEWTABLE: Work.Roads

	segnum	cntyrte	access	no_lanes	medtyp	aadt	begmp	endmp
1	1	0020000070	0	2	1	11000	0	0.228
2	1	0020000070	0	2	1	11000	0.228	0.688
3	1	0020000070	0	2	1	11000	0.688	0.696
4	1	0020000070	0	2	1	18000	0.696	0.724
5	1	0020000070	0	2	1	18000	0.724	0.849
6	1	0020000070	0	2	1	18000	0.849	0.873
7	1	0020000070	0	2	1	18000	0.873	0.992
8	1	0020000070	0	2	1	18000	0.992	1.272
9	1	0020000070	0	2	1	18000	1.272	1.319
10	2	0020000070	0	4	2	18000	1.319	1.432
11	3	0020000070	0	4	2	21000	1.432	1.982
12	3	0020000070	0	4	2	24000	1.982	2.469
13	3	0020000070	0	4	2	24000	2.469	2.588
14	4	0020000070	0	4	2	24000	2.589	2.686

Figure 6. Expected breaks plus a gap break

ANOTHER VIEW

Suppose we need not look for gaps. Without the APP functions, our code might look like this:

```
DATA segs(KEEP=segnum cntyrte bmp emp  access no_lanes medtyp rural);
SET p.roads;
BY cntyrte access no_lanes medtyp rural NOTSORTED;
RETAIN bmp;
IF FIRST.rural THEN bmp=begmp;
IF LAST.rural THEN DO;
    segnum+1;
    emp=endmp;
    OUTPUT;
    END;
RUN;
```

To take a similar approach using APP functions, we can create a view or table using SQL or a DATA step that will provide a single BY variable. These two approaches create views:

```
PROC SQL;
CREATE VIEW roads AS
    SELECT cntyrte, access, no_lanes, medtyp, rural,
    *,
    PEEKC(ADDR(cntyrte),15) AS str
FROM p.roads;
QUIT;


DATA roads/VIEW=roads;
SET p.roads;
ARRAY v(*) $ cntyrte access no_lanes medtyp rural;
str=PEEKC(ADDR(cntyrte),15);
RUN;
```

Having created the view or table, we can use a DATA step to both assign the new segment number and write one record per segment. BY group processing is much faster with one BY variable rather than five BY variables.

```
DATA segs;
SET roads;
BY str NOTSORTED;
RETAIN bmp;
IF FIRST.str THEN bmp=begmp;
IF LAST.str THEN DO;
       segnum+1;
       emp=endmp;
       OUTPUT;
       END;
RUN;
```

CONCLUSION

The APP functions provide an opportunity to take shortcuts. The careful coder will drive with caution, testing along the way. In this paper, we have experimented with a single storage group – retained character variables. Extending the example herein to include numeric variables from the input roadway segments would require PEEKing at a different string variable. Although it would be read from a different starting point in memory, and all the variable lengths would be 8 bytes, the technique would remain the same. The author strongly recommends studying the meticulous information available in previously published papers about the APP functions.

ACKNOWLEDGMENTS

The author offers many thanks to Paul Dorfman and Eric Rodgman for reviewing this paper. Any errors remaining herein are strictly the property of the author!

RECOMMENDED READING

In SAS Documentation:

- *Base SAS® Language Reference: Concepts*

In Conference Proceedings:

- A-P-P Advanced Data Management Functions, Peter Crawford and Paul Dorfman
- From Obscurity to Utility: ADDR, PEEK, POKE as DATA Step Programming Tools, Paul Dorfman
- The ADDR-PEEK-POKE Capsule: Transporting Data Within Memory and Between Memory and the PDV, Paul Dorfman
- HOW to DOW, Paul Dorfman

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Carol Martell
UNC Highway Safety Research Center
Chapel Hill, NC
carol_martell@unc.edu