Paper 203

A Simple Framework for Sequentially Processing Hierarchical Data Sets for Large Surveys

Richard L. Downs, Jr. and Pura A. Peréz
U.S. Bureau of the Census, Washington, D.C.

ABSTRACT

This paper explains a simple framework for applying traditional sequential processes (reformat, edits, imputation, etc.) to a hierarchical group of SAS® data sets. The framework merges the data sets using a series of layered DATA step views. It then shows how users can write DATA step code that processes the merged data sets and outputs updated versions of one or more of those data sets. The DATA step code uses a macro to help dynamically create a keep option for each output data set. The DATA step then uses a combination of checking for last. variables and missing values to determine the proper time to output to each data set. This framework largely isolates the complexity of relating the data sets from the complexity of the actual process, it maintains the data set hierarchy, and eliminates the need for any post-process processing (all output data sets are complete at the end of the process data step).

INTRODUCTION

Most U.S. Bureau of the Census demographic personal or telephone interview surveys have switched from paper questionnaires to electronic instruments run using the Computer-Assisted Survey Execution System (CASES). Post collection, we extract data from CASES and, for most surveys, convert the output into one or more SAS data sets for processing.

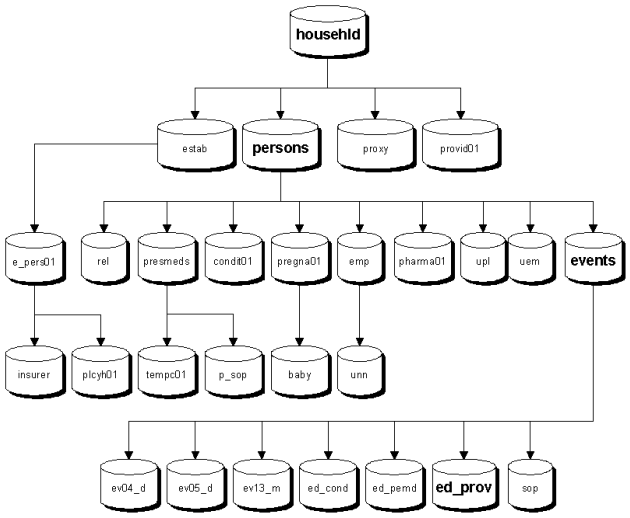

Figure 1

Processing varies from survey to survey, but usually it involves taking the CASES output data through a series of sequential steps: reformat, edits, imputation, weighting, recodes, table production, and internal/external user file production. Traditionally, we process on a mainframe environment running 3GL (FORTRAN and COBOL) programs against one or more flat or hierarchical files. Redesigned processing is on UNIX workstations running SAS programs against a hierarchical series of data sets. The memory and storage resources available are much more limited than those in the mainframe environment. Consequently, we need to process the data while maintaining the data set hierarchy through each process.

In this paper we explain and demonstrate a simple framework for applying traditional sequential processing steps to hierarchical data sets while largely isolating the complexity of the data set relationships from the complexity of the actual processes.

CASES OUTPUT DATA

CASES organizes data at the case level and in rosters. Case level information is usually information about the household. A roster is a repeating group of data items. Each roster is a "child" of either the case level or another roster; instruments have a case level and may have up to three roster levels. For example, if we have a survey that collects information on jobs, each interview has a case level with information about the entire household. Each household can have several persons, so we have information about each person stored in a roster at a second level. Each person could have several jobs, so we have information about each job stored in another roster at a third level, etc.

When we translate CASES output to SAS data sets each repeating group becomes a separate data set. Depending on the complexity and size of a survey, the results can be upwards of 50 data sets in a hierarchy up to four levels deep. An example data set hierarchy taken from the Medical Expenditure Panel Survey (MEPS) is shown in figure 1.

The data sets are related and uniquely identified by common variables; for the purposes of this paper we will call them relationship variables. We can see this by looking at a subset of the MEPS data sets from figure 1, shown in figure 2.

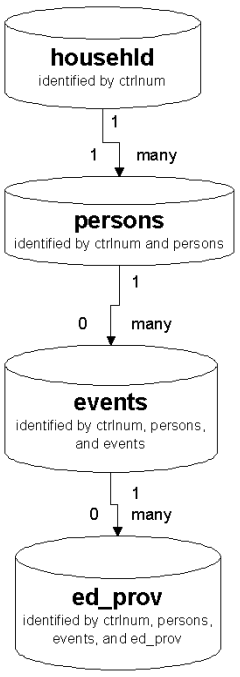

Figure 2

The *household* data set contains a variable ctrlnum that uniquely identifies each household.

Persons has a variable ctrlnum that links the person to the household; it also has a variable persons that uniquely identifies that person within that household.

Events (in this example an event is a health care event, such as a doctor visit) has variables ctrlnum and person that link that event to a person; it also has a variable events that uniquely identifies the event for that person.

Finally, *ed_prov* (SBD provider information concerning health care providers a person initially saw at one facility and later saw at outside that facility) has variables ctrlnum, persons, and events that link the SBD provider to the event; it also has a variable ed_prov that uniquely identifies the SBD provider for that event.

Relationships between the data sets are either one-to-one or one-to-many where many is either one or zero based. The MEPS example has the following relationships:

① Each household relates to one or more persons.
② Each person relates to zero or more events.
③ Each event relates to zero or more SBD providers.

REQUIREMENTS

Our issue is how do we implement the traditional sequential survey processes using SAS given the data hierarchy? Our solution should allow us to maintain the hierarchy, largely isolate the complexity of relating the data sets from the complexity of the survey processing, and eliminate the need for any post-processing at the completion of each processing step.

By maintaining the hierarchy, we require that users do not collapse the hierarchy into one "big file" by either amalgamating the data or creating temporary data sets. Amalgamating the data (creating a household level representation) introduces more chances for errors because roster variables must be renamed since they will repeat per observation. It is also wasteful because of the number of blank values in each observation. Creating a temporary data set based on the lowest level of the hierarchy, in our example *ed_prov*, is wasteful because of the number of values that repeat over multiple observations. Also, for larger surveys, amalgamating or creating a temporary data set to represent the input relationships may be impossible due to resource constraints.

By isolating the complexity of relating the data sets, we want users to have to build minimal relationship logic into their process step code. Ideally users can reference the related data sets as if they are one "big file." Also, users should have to build minimal output control logic into their process step code to create new versions of the appropriate input data sets.

Finally, we want to eliminate the need for any post-processing. This means that all data sets produced by the processing step have the correct number of observations; usually this is the same number of observations as the corresponding input data set. Consequently, the way we build the input relationships usually cannot restrict the universe of the relationships. If we do restrict the input universe on any data set that we output, then we may have to run additional steps after the process step to add back the excluded observations. These additional steps unnecessarily complicate the process and introduce more chances for errors.

EXPRESSING HIERARCHICAL INPUT RELATIONSHIPS

A solution to creating a "big file" feel is to create a series of layered SAS views. We relate two or more data sets with each view depending on the relationships between data sets. We need to have one view for each instance where data sets being related do not have a one-to-one relationship. For example, to relate the MEPS data sets shown in figure 2 we need three views because the relationships between data sets (househld to persons, persons to events, and events to ed_prov) are all one-to-many.

Let us look at how we create these views by setting up the relationships among the four MEPS data sets shown in figure 2. Please note that all the related data sets need to be sorted by their appropriate relationship variables. Please also note that in our examples we create our views in the work library; we can make our views permanent and reuse them if appropriate by creating them in another library.

We start at the bottom by relating the lowest two data sets in the hierarchy; here, we relate *events* and *ed_prov*. As shown earlier, these data sets are related by the variables ctrlnum, persons, and events. The code for the first view (we will call it *view1*) is shown below:

```
data view1 / view=view1;
merge datalib.events(in=ininner)
      datalib.ed_prov;
by ctrlnum persons events;
if ininner;
```

View1 gives us all observations that match both data sets and all observations from *events* that did not match to *ed_prov*. In the latter case the variables contributed by the second, or outer, data set are set to missing. Please note because of the nature of the data, we will not have a situation where an observation on the outer data set does not match to the inner data set, but we allow for such a situation in the data step. This is the logical equivalent of a SQL left outer join.

We then move up the hierarchy by relating the next data set (*persons*) with the view we just created (*view1*). The relationship is by the variables ctrlnum and persons. The code for the second view (*view2*) is shown below:

```
data view2 / view=view2;
merge datalib.persons(in=ininner) view1;
by ctrlnum persons;
if ininner;
```

Finally we get to the top of the hierarchy by relating the top data set (*househld*) with the previous view (*view2*). The relationship is by the variable ctrlnum. Remember that each household has to have at least one person, so we add that restriction to the DATA step. The code for the final view (*view3*) is shown below:

```
data view3 / view=view3;
merge datalib.househld(in=ininner)
      view2(in=inouter);
by ctrlnum;
if ininner and inouter;
```

We need to be very careful when/if we apply process-specific restrictions to the views. In the above examples we did not apply any process-specific restrictions; we applied restrictions that express the relationships between the data sets. Usually we apply any necessary process-specific restrictions in the process code. We should only apply process-specific restrictions to the views when both:

① The data set being restricted is not output.

② The restrictions do not result in a subset of data sets lower in the hierarchy and any of those data sets are output.

Additionally, if we have a situation where we relate just two input data sets then we have an option of using a MERGE statement in the process DATA step instead of defining a view that merges the two data sets and setting that view in the process DATA step. At first glance it appears that these approaches produce the same result, however, there is a difference in the way SAS initializes contributed variables in the PDV. We discuss initialization of contributed variables in the final section of the paper: writing process-specific DATA step code.

Finally, some of you may wonder why we did not use PROC SQL to create our views. We found that using PROC SQL created several issues:

① Due to the nature of the select statement, we found it necessary to use the coalesce function to compensate for duplicate variables in the SQL views. The PROC SQL code that is the equivalent of the DATA step views described above is shown below.

```
proc sql;
create view view1 as
select *,
coalesce(events.events, ed_prov.events) as events
from datalib.events left join datalib.ed_prov
on events.ctrlnum=ed_prov.ctrlnum and
events.persons=ed_prov.persons and
events.events=ed_prov.events;

create view view2 as
select *,
coalesce(persons.persons, view1.persons) as
persons
from datalib.persons left join view1
```

```
on persons.ctrlnum=view1.ctrlnum and
persons.persons=view1.persons;

create view view3 as
select *,
coalesce(household.ctrlnum, view2.ctrlnum) as
ctrlnum
from datalib.household, view2
where household.ctrlnum=view1.ctrlnum;
```

② Because SQL processes an outer join by forming the Cartesian product of the data sets then selecting the observations, we found that performance severely degraded as the size of the data sets increased.

RELATING DATA SETS ACROSS BRANCHES OF THE HIERARCHY

In our example the input data sets are on one branch of the hierarchy shown in figure 2. Because of the way these data sets are related this is usually a requirement. However, in some situations relating data sets across branches of the hierarchy is possible. Working with CASES data as instruments become larger and more complex, relating data sets across branches will become more common because CASES currently limits the hierarchy to just four levels. To relate data sets across branches, the data sets at a peer level must meet the following criteria:

① The data sets must have a one-to-one or one-to-many logical relationship; the data sets cannot have a many-to-many relationship.

② The data sets must have the appropriate variables to uniquely relate to each other. The relationship variables described earlier are not sufficient to relate across branches.

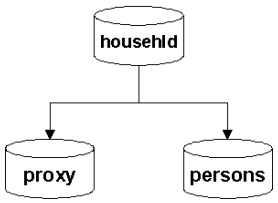

Figure 3

Figure 3 shows *household* and *persons* from the previous example related to *proxy* (*proxy* contains information on a person providing information about one or more other people). *Proxy* and *persons* are related to *household* by the variable ctrlnum. *Proxy* and *persons* are related by the variable proxy_ln; proxy_ln is not a relationship variable. We can sum up the relationships between the data sets as:

① Each household relates to zero or more proxies.
② Each household relates to one or more people.
③ Each proxy relates to one or more persons.
④ Each person can relate to no more than one proxy.

The code for the views that setup these relationships is shown below:

```
data view1 / view=view1;
merge datalib.proxy
      datalib.persons(in=inouter);
by ctrlnum proxy_ln;
if inouter;

data view2 / view=view2;
                    merge
                    datalib.household(in=ininne
                    r)
                          view1(in=inouter);
                    by ctrlnum;
                    if ininner and inouter;
```

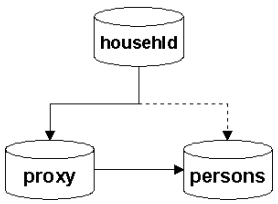

Figure 4

View1 gives us all observations that match from *proxy* and *persons* and all observations from *persons* that do not match to *proxy*. This is the logical equivalent of a SQL right outer join. *View2* is similar to *view3* from the previous example.

These views create the relationship shown in figure 4. We get from *household* to *persons* through *proxy*. However, the views make this connection even if a *proxy* observation does not exist for a *persons* observation.

EXPRESSING HIERARCHICAL OUTPUT RELATIONSHIPS

Since our processing might update values or create new variables at any level of the hierarchy, the processing step needs to be able to create new versions of each input data set. This brings up two issues: first, how do we assign the correct variables to the different data sets; and second, when do we output to the different data sets.

We handle the first issue by using a keep data set option and dynamically creating a list of variables from the corresponding input data set with a macro. Generating the list dynamically is crucial to maintaining data integrity. The macro (%VARLST) is listed below:

```
%macro varlst(dataset);
%local dsid i rc;
%let dsid = %sysfunc(open(&dataset));
%do i = 1 %to
   %sysfunc(attrn(&dsid,nvars));
   %sysfunc(varname(&dsid,&i))
%end;
%let rc = %sysfunc(close(&dsid));
%mend;
```

%VARLST goes through the following steps:

① Open the specified data set.
② Looping from one to the number of variables in the data set, retrieve and display the name of each variable.
③ Close the data set.

Please note that this macro requires at least SAS 6.12 or 6.09e. If you are running an older version of SAS, an alternative macro approach is described in Appendix A.

Going back to our first MEPS example, the data statement that produces the new versions of all four input data sets is listed below:

```
data household(keep=%varlst(datalib.household))
     persons(keep=%varlst(datalib.persons))
     events(keep=%varlst(datalib.events))
     ed_prov(keep=%varlst(datalib.ed_prov));
```

We handle the second issue, when to output, through a combination of last. variables and checking for missing values. The set statement for our example is shown below; note that we set our last view (*view3*) by all of the relationship variables, though the last view was defined only by the first relationship variable:

```
set view3;
by ctrlnum persons events;
```

We do the actual output handling at the end of the process data step. Output to any of the data sets is based on two conditions: first, is it the right time to output to that data set; and second, does data for that data set exist in the PDV.

We determine proper time for output by testing whether or not the last. variable associated with a data set is true. For example, in our MEPS processing the time is right to output to *persons* when last.persons is true. A proper time test is not necessary for the bottom data set in the hierarchy.

We determine whether the data are missing by testing whether or not the relationship variable associated with a data set is missing. For example, in our MEPS processing we do not output to *persons* if persons is missing. The missing test is not necessary for the top data set in the hierarchy.

The output handling code for our MEPS example is shown below:

```
if last.ctrlnum then output househld;
if last.persons and persons ne . then
   output persons;
if last.events and events ne . then
   output events;
if ed_prov ne . then output ed_prov;
```

Of course there are processing situations that require deviating from this output handling logic. However, changing the output requirements requires minimal changes in the basic output handling code.

DEMONSTRATION

Let us show the way SAS processes these views; the complete code for this demonstration is listed in the unabridged version of this paper available at the URL given in the Contact Information. The following data steps setup simplified versions of our MEPS data sets for two households. Our sample data are depicted in figure 5.

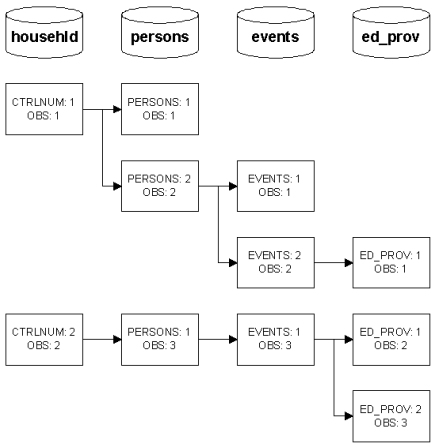

Figure 5

```
data datalib.househld;
input ctrlnum;
cards;
001
002
;
data datalib.persons;
input ctrlnum persons;
cards;
001 001
001 002
002 001
;
data datalib.events;
input ctrlnum persons events;
cards;
001 002 001
001 002 002
002 001 001
;
data datalib.ed_prov;
input ctrlnum persons events ed_prov;
cards;
001 002 002 001
002 001 001 001
002 001 001 002
;
```

We set up the views as described earlier and set *view3* in the data step shown below. Note that this DATA step creates new versions of all four input data sets.

```
data househld(keep=%varlst(datalib.househld))
   persons(keep=%varlst(datalib.persons))
   events(keep=%varlst(datalib.events))
   ed_prov(keep=%varlst(datalib.ed_prov));
set view3;
by ctrlnum persons events;
put ctrlnum= last.ctrlnum= /
    persons= last.persons= /
    events= last.events= /
    ed_prov= last.ed_prov= /;
if last.ctrlnum then output househld;
if last.persons and persons ne . then
   output persons;
if last.events and events ne . then
   output events;
if ed_prov ne . then output ed_prov;
```

Running the DATA step, we get the following values for each iteration in the PDV; periods represent missing data:

```
CTRLNUM=1 LAST.CTRLNUM=0
PERSONS=1 LAST.PERSONS=1
EVENTS=. LAST.EVENTS=1
ED_PROV=.

CTRLNUM=1 LAST.CTRLNUM=0
PERSONS=2 LAST.PERSONS=0
EVENTS=1 LAST.EVENTS=1
ED_PROV=.

CTRLNUM=1 LAST.CTRLNUM=1
PERSONS=2 LAST.PERSONS=1
EVENTS=2 LAST.EVENTS=1
ED_PROV=1

CTRLNUM=2 LAST.CTRLNUM=0
PERSONS=1 LAST.PERSONS=0
EVENTS=1 LAST.EVENTS=0
ED_PROV=1

CTRLNUM=2 LAST.CTRLNUM=1
PERSONS=1 LAST.PERSONS=1
EVENTS=1 LAST.EVENTS=1
ED_PROV=2
```

Comparing this with the data representation in figure 5, we see that the PDV contents mirror figure 5.

The relationship variables tell us which data sets contributed to each iteration and with their last. variables tell us when we output. For example, look at iteration two. The PDV tells us we are looking at the first event for the second person in household one. We also know there is no SBD provider for this event (ed_prov is missing). Finally, at the end of this iteration we will only output to *events*, since we are not in the last iteration for this household or person (last.ctrlnum and last.persons are false), and there is no data in the SBD provider variables.

We look at the output data sets from our demo to verify we output the correct variables at the correct time. Since our demo did not update or add any variables in any data sets, using PROC COMPARE we can verify that all our output data sets are exact copies of the input data sets. An excerpt from the PROC COMPARE output for the input and output *persons* data sets is show below:

Data Set Summary

Dataset	NVar	NObs
DATALIB.PERSONS	2	3
WORK.PERSONS	2	3

Variables Summary

Number of Variables in Common: 2.

```
Observation Summary

Number of Observations in Common: 3.
Total Number of Observations Read from DATALIB.PERSONS: 3.
Total Number of Observations Read from WORK.PERSONS: 3.

Number of Observations with Some Compared Variables Unequal: 0.
Number of Observations with All Compared Variables Equal: 3.

NOTE: No unequal values were found. All values compared are
exactly equal.
```

WRITING PROCESS-SPECIFIC DATA STEP CODE

		ITERATION	
	_N_	INPUT	OUTPUT
househld	1	1	3
	2	4	5
persons	1	1	1
	2	2	3
	3	4	5

Table 1

The series of layered SAS views create a "big file" feel from a series of merges. Any time we merge data sets we need to be aware of the timing of when each data set is input and output.

Table 1 shows the DATA step iterations where each *househld* and *persons* observation from our demonstration is input and output. For example, we read the first *househld* observation in iteration one, but do not output the observation until iteration three. Similarly, we input the second *persons* observation in iteration two, but do not output until iteration three. Also, SAS resets all data set variables to their input values each time it executes the set statement at the beginning of each iteration. We can generalize two implications of this common situation:

① The values contributed by a single data set observation may be in the PDV for multiple iterations.

② We should only update a data set variable's value in the same iteration where that data set is output.

For example, if we update a variable from the *househld* data set in iteration one we will lose that update because *househld* is not output in iteration one and SAS resets all *househld* variables to their input values at the beginning of iteration two. Consequently, we should restrict updates to *househld* variables to iterations where the last. variable associated with *househld* (last.ctrlnum) is true. This is shown below:

```
if last.ctrlnum then do;
/* update househld variable(s) here */
end;
```

Recall that if we have a situation where we relate just two input data sets we can use a MERGE statement instead of creating a view and setting the view. If we used a MERGE statement then the way SAS initializes contributed variables in the PDV changes; SAS initializes contributed variables only when it reads a new observation from a particular data set. This means that the second implication above does not apply. Also, we can generalize an additional implication from using a MERGE statement in this situation:

② Depending on the application, if we have data sets in a one to many relationship, an observation of the outer data set (in our example *persons*) will not necessarily be processed with the same information contributed by the inner data set (in our example *househld*) as subsequent observations from the outer data set.

For example, if we use a MERGE statement and update a variable from the *househld* data set in iteration one we will not lose that update because SAS does not reinitialize the variables contributed by *househld* until it reads the next *househld* observation. Consequently, we do not have to restrict updates to *househld* to iterations where last.ctrlnum is true as before. Also, if we do update a *househld* variable, then any subsequent *persons* in that household will be processed with the modified *househld* data, not the original *househld* data.

We do not go into detailed examples of writing process-specific data step code here because that is beyond the scope of this paper. Also, we want to stress that these implications are not unique to this framework; they need to be considered when ever we merge data sets that have a one-to-many relationship. Simply keep in mind which variables come from which data sets, when those data sets are output, and time the references and updates to those variables appropriately when designing process-specific code.

CONCLUSION

Although our example is CASES output specific, we can easily apply this framework in similar situations. We can apply this framework in situations where we match-merge input data sets that meet the criteria listed below and output one or more of those data sets:

① The data sets must be a hierarchy of one-to-one or one-to-many relationships.

② Each data set must contain relationship variables to uniquely identify each observation and associate it with its parent, grandparent, or great-grandparent observation as appropriate.

③ Each data set must be sorted by its appropriate relationship variables.

④ The data set relationships must either be down one path of the hierarchy or be across branches of the hierarch and meet the criteria discussed in the input relationships section.

The combination of input and output relationships described here fulfill our requirements. Specifically, we preserved the hierarchy by not amalgamating or physically combining data sets during processing. We removed most of the complexity of handling input and output data sets from processing by creating views before the processing step and building a process step skeleton that handles the appropriate output. Finally, we ensured the output is complete by not restricting the input or output universe.

DISCLAIMER

This paper reports the results of research and analysis undertaken by Census Bureau staff. It has undergone a more limited review than official Census Bureau publications. This report is released to inform interested parties of research and encourage discussion.

ACKNOWLEDGMENTS

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

CASES is a registered trademark or trademark of University of California, Berkeley.

CONTACT INFORMATION

Richard's e-mail address is: Richard.L.Downs.Jr@census.gov.

Pura's e-mail address is: Pura.A.Perez@census.gov.

An unabridged copy of this paper is available on the Internet at the

URL: hometown.aol.com/dusia/sug98.htm.

APPENDIX A

Without %SYSFUNC, getting data set information on the fly is a little more involved. The framework described above still applies, however the macro(s) and their calls change slightly.

We create the variable list in two parts using two macros. Part one is before the processing step. The first macro requires a data set name and a prefix. It uses the prefix to form a series of global macro variables; the prefix should be no more than five characters and should be unique for the SAS session. The macro goes through the following steps:

① Run PROC CONTENTS storing the output to a temporary data set (varlst).

② Declare a global macro variable with the name based on the given prefix with a zero suffix. For example, given the prefix hh the macro variable is hh0.

③ Run a DATA step that constructs strings that contain the data set's variable names separated by a space (var1 var2 var3). Store these strings in global macro variables with the names based on the given prefix with a numeric suffix. For our example, given the prefix hh the macro variables would be hh1, hh2, hh3, etc.

Also, store a global macro variable with the name base on the given prefix and a zero suffix. This variable contains the number of macro variables used to store the data set's variable names.

④ Delete the temporary data set.

The macro code is listed below:

```
%macro mkvarlst(dataset,prefix);
%global &prefix.0;
proc contents data=&dataset noprint
   out=varlst;
data _null_;
length temp $200;
retain count 0 temp ' ';
set varlst(keep=name) end=done;
temp = trim(temp) || ' ' || name;
if length(temp) > 190 or done then do;
   count +1;
   if done then
      call symput("&prefix.0",
         compress(put(count,4.)));
   call execute('%global ' ||
      "&prefix" ||
      compress(put(count,4.)) ||
      '; %let ' || "&prefix" ||
      compress(put(count,4.)) ||
      '=' || trim(temp) || ';');
   temp = ' ';
end;
run;
proc datasets library=work nolist;
   delete varlst;
run;
%mend;
```

The second macro requires the prefix specified to the first macro. Based on that prefix, it loops through from 1 to the count of macro variables with that prefix and displays the macro variables. The code is shown below:

```
%macro ptvarlst(prefix);
%local i;
%do i = 1 %to &&&prefix.0;
   &&&prefix&i
%end;
%mend;
```

Lastly, the macro calls are shown below. Please note that %MKVARLST is invoked before %PTVARLST for each data set and that %MKVARLST is called outside the DATA step.

```
%mkvarlst(datalib.househld,hh)
%mkvarlst(datalib.persons,per)
%mkvarlst(datalib.events,ev)
%mkvarlst(datalib.ed_prov,ed)

data househld(keep = %ptvarlst(hh))
     persons(keep = %ptvarlst(per))
     events(keep = %ptvarlst(ev))
     ed_prov(keep = %ptvarlst(ed));
```