ON DATA TRANSFER

Aileen L. Yam, Clinical Information Analysis, Inc., Plainsboro, NJ

ABSTRACT

In clinical research organization, there are frequently requests from pharmaceutical companies for data transfer, where raw data are to be labeled, stored in different lengths or formats, reordered according to the order of variables in case report form, transposed, combined with other data, and finally output into SAS® data sets or ASCII files. This paper presents tips on preparing data for transfer. The tips include a guide to thorough planning, the use of SAS commands and macro facility to reduce the amount of work needed for some common tasks, examples to resolve general and unusual data situations, and precaution on error-prone programming codes. In light of the popularity of the Windows environment, the tips also contain some easy ways to convert Microsoft Excel files to SAS data sets and vice versa.

I. KEY CONSIDERATIONS IN PLANNING

1. TRANSFER SPECIFICATIONS

a. Specifications of the format and structure of the transfer data, system requirements, SAS® version, as well as frequency and mode of transfer need to be clearly defined.

b. Problems can arise due to lack of system compatibility, conversion failure, data corruption, misunderstanding of data content, structure or format specifications. Suggest test transfer before the actual transfer to identify potential problems.

c. Document specification changes.

2. STANDARDS

a. Standards in terms of documentation, labeling, programming, backup and archiving need to be established.

b. Standards are especially helpful when resource profile changes.

3. DATA

a. Unclean data can cause unexpected results.

b. Find out whether clean data are expected.

c. Even if clean data are not expected, it is a good idea to look for duplicate observations and do range checks on primary variables. It is also helpful to document data problems and send them along with data transfers.

d. For multiple studies of the same drug, check for variations in data structures, formats or values. Programs need to be adjusted to the variations accordingly.

4. PROGRAMMING

a. Variable names, labels, formats, lengths and types, and the values of categorical variables are to be consistent in each study of the same drug so that data can be concatenated with little chance for error.

b. Except for key variables common to all data sets, most of the other variables are unique and it is better not to create a variable name that can have different meanings in different data sets. For example, a variable called duration that can mean duration of study or duration of adverse events or duration of drug exposure depending on which data set duration comes from can cause confusion as well as problems in merging.

c. Document complex logic.

d. Use spaces and indentations to make programs easy to read and debug.

e. Avoid data-driven or patient-specific programming logic.

f. Find out if it is possible to globally transform all input data sets with a macro program.

5. VALIDATION

a. It is important to validate each step during program development and to do the right job the first time. It also helps to have an independent person to do validation.

b. For validation, check on the following major items:

1. SAS log - check log files for warning and error messages.

2. data formats - a good source to validate formats against data transfer specifications is in the PROC CONTENTS output.

3. data structure - check the transfer data against both the data transfer specifications and the input data.

4. derived variables - PROC FREQ summarizes newly derived values and gives a frequency count for comparison with the original values from which the new values are derived.

5. the last variable in each data set – it is the most important variable to check. It is usually the first place that shows whether the destination file reads and writes the right columns from the source file.

6. serious adverse events – check them visually or programmatically as thoroughly as possible.

7. variables with very long comments or with many missing values – they are prone to error and need to be checked against source data.

8. merging and concatenation – verify the number of observations before and after merging or concatenation occurs; eliminate unnecessary variables.

9. total number of observations – do PROC CONTENTS on all transfer data sets and compare the total number of observations with the source files.

10. systematic errors - they usually come from a single source, such as a macro or a program module that is used repeatedly.

II. PROGRAMMING TIPS

The following discusses some convenient and useful features of the SAS software for doing general tasks and for resolving unusual data situations in data transfers.

1. INPUT DATA

a. DOUBLE HYPHENS (--)

The double hyphens (--) refer to a list of variables beginning with the variable before the hyphens and ending with the variable after the hyphens. The beginning variable must come before the ending variable in the internal storage position. PROC CONTENTS shows the internal storage positions of variables. For example, the following is an example to refer to a group of variables, *doe, ray, me, far, sew, la, tea*:

```
doe - - tea
```

It is not necessary to have the beginning and ending variable names in alphabetical or numeric order.

b. SPECIAL VARIABLES WITH UNDERSCORES

The special variables, _NUMERIC_, _CHARACTER_, and _ALL_, refer to all the numeric variables, all the character variables, and all the variables, respectively, defined in the current DATA step. They are handy, especially when used in conjunction with arrays. The following is an example to check scores with values greater than 5:

```
data over5;
      set qoldata;

array scores(*) _numeric_;
do i=1 to dim(scores);
     if scores(i)<=5 then continue;
     put investig= patient= visit= scores(i)= ;
end;
```

```
run;
```

If any element of the *scores* array is less than or equal to 5, then the next iteration of the DO loop continues, otherwise, the information is written out in the log file.

c. VARYING RECORD LENGTH ASCII FILES

One way of reading ASCII files with varying record lengths due to varying length data fields is to use LRECL= and PAD options on the INFILE statement. The LRECL= option defines a fixed maximum logical record length for all records. The PAD option puts blanks to records that are shorter than the logical record length defined in the LRECL= option.

d. RECORDS WITH MISSING FIELDS

If the varying record lengths are due to missing fields rather than varying length data fields, the INFILE statement needs to include the MISSOVER or the TRUNCOVER option. The MISSOVER option sets missing numeric fields to dots and missing character fields to blank. The TRUNCOVER option retains all the readable fields and truncates records with missing fields.

2. DATA ERRORS

a. FLAGGING DATA ERRORS IN LOG FILES

Data errors can be flagged in the SAS log files and then grouped by kind:

```
data _null_;
      if cond1 then put "??DUPLICATES:  &count
duplicate obs - &ds";
      if cond2 then put "??DATA ERROR:  out-of-range
data - &ds";
      if cond3 then put "??CAUTION:  data structure
different - &ds";
run;
```

The macro variable reference, *&ds*, indicates the data set name where problem occurs.

Most operating systems have utility programs and commands to search for text patterns. For example, in UNIX, use the grep command to search all the log files for all the lines beginning with the string ??:

```
grep "^??" *.log | sort > summary
```

The data problems are then sorted in alphabetical order by kind, and are listed in a file called summary.

b. CHECKING DUPLICATE RECORDS

The NODUP option in PROC SORT eliminates observations with all duplicate variable values. The number of observations in the log file before and after the sorting indicates whether any observation is eliminated. The drawbacks are: it is hard to tell which observations are eliminated, or to monitor at the end of all programming which data sets have duplicate observations.

A good way to keep track of duplicate observations in the data is to print out the duplicate data and to output a message in the log file if duplicates exist, as in the following macro program:

```
%macro finddup(ds=,unikdata=,dupdata=,prt=yes);

proc sql;
      create table &unikdata as
      ❶select distinct *
      from &ds;

proc sql;
      create table &dupdata as
      select * from &ds
      ❷except all
      select * from &unikdata;

%if %upcase(&prt)=YES %then %do;
      title "Duplicate Data from &ds";
      ❸select * from &dupdata;
      ❹title;
%end;

❺%if &sqlobs>0 %then %put
      ??DUPLICATES:  &sqlobs duplicate obs - &ds;

%mend finddup;
```

❶The DISTINCT keyword and an asterisk (*) in the first PROC SQL eliminates duplicate rows of data.

❷The EXCEPT and ALL keywords in the second PROC SQL display all the rows found in *&ds* but not in *&unikdata*. ❸The last SELECT statement in the second PROC SQL prints out duplicate data if the *prt* option is *yes*. ❹The TITLE statement resets the title to null.

❺If duplicates exist, the SQL automatic macro variable *&sqlobs* has a value greater than zero and a message is output in the log file.

3. REORGANIZING AND REFORMATTING DATA

a. WITH THE ATTRIB STATEMENT

Putting an ATTRIB statement before a SET statement changes the position and certain specified attributes of variables in an output data set when using an existing data set as input. In fact, any statement, such as the LENGTH, KEEP, or RETAIN statement, before the SET statement can also rearrange data according to the order that the variables appear in the statement. THE ATTRIB statement has the added advantage of reformatting and labeling the data at the same time.

In the ATTRIB statement below, the dates are reformatted to *date9*, the length of *la* is changed from a default of 8 to 3, and *doe* is given a new format. The variables are reordered from *doe--tea* to *tea--doe*.

```
data after;
attrib tea label='A Drink with Jam and Bread'
       la length=3 label='A Note to Follow Sew'
       sew label='A Needle Pulling Thread'
       far label='A Long, Long Way to Run'
       me label='A Name I Call Myself'
       ray label='A Drop of Golden Sun'
       doe format=doefmt. label='A Female Deer'
       date1 format=date9. label='Date Composed'
       date2 format=date9. label='Date Simplified';
       set before;
run;
```

b. WITH PROC SQL

The following SQL statements produce the same results as the ATTRIB statements. With PROC SQL, however, each variable in the SELECT statement has to be listed individually, a from-to list such as *date1-dateN* is not allowed. The results from either the ATTRIB statement or PROC SQL are identical.

The data are reorganized according to the order of variables in the SELECT statement. The labels, formats, and lengths are defined in the same manner as with the ATTRIB statement.

```
proc sql;
      create table withsql as
      select tea label='A Drink with Jam and Bread'
          la length=3 label='A Note to Follow Sew'
          sew label='A Needle Pulling Thread'
          far label='A Long, Long Way to Run'
          me label='A Name I Call Myself'
          ray label='A Drop of Golden Sun'
          doe format=doefmt. label='A Female Deer'
          date1 format=date9. label='Date Composed'
          date2 format=date9. label='Date Simplified'
      from before;
```

4. MULTIPLE STUDIES OF THE SAME DRUG

Usually, data transfers are done on several studies of the same drug. The data transfer programs therefore need to be able to accommodate all studies with as little modification as possible.

The number of visits may vary from study to study. The following is an example to keep track of the maximum number of visits in a study. The maximum number of visits can then be put into an array in an example in the next section of this paper. The purpose of putting the maximum number of visits into a macro variable is to make the program flexible in handling any number of visits so that the same program can be used for any study of the same drug.

A simplified data set with regular and unscheduled visits is created for illustration. The unscheduled visits are visits with non-integer values.

```
data maxvis;
      input investig patient visit dosvalue;
cards;
23    101    1      3
23    101    2      7
23    101    2.1    6
23    101    2.2    3
23    101    3      3
23    102    1      3
23    102    2      3
23    102    3      2
23    102    4      3
run;

proc sql;
      ❶reset noprint;
      select ❷max(total) into: ❸nmaxvis
      from
          (select count(distinct visit) as total
          from maxvis
          group by patient);

%let nmaxvis=❹%left(&nmaxvis);
```

❶The RESET statement is used so that the result is not printed. ❷A variable, *total*, is created to find out the total number of visits for each patient. The *max* function is used to obtain the maximum visit number. ❸The maximum number of visits among all patients is put into a macro variable called *nmaxvis*. ❹The *%left* function is used to remove the leading blank in the macro variable *nmaxvis*.

5. TRANSPOSING DATA WITH UNSCHEDULED VISITS

The following examples use the data set *maxvis* and the macro variable *nmaxvis* from the previous example.

a. VERTICAL TO HORIZONTAL DATA FORMAT

The output transfer data will have one observation per patient. The dose values *(dosvalue)* at each visit become variables, *dose1-dose&nmaxvis*. The actual visit *(realvis)* including unscheduled visits are captured in variables, *realvis1-realvis&nmaxvis*.

```
proc sort data=maxvis;
      by investig patient visit;
run;

data maxvis;
      set maxvis;
      by investig patient visit;
❶if first.patient then visnum=0;
      visnum+1;
run;

data v2h(drop=visit dosvalue visnum);
      set maxvis;
      by investig patient visit;

retain dose1-dose&nmaxvis realvis1-realvis&nmaxvis;
❷array dose{*} dose1-dose&nmaxvis;
  array realvis{*} realvis1-realvis&nmaxvis;

❸if first.patient then do;
      do i=1 to dim(dose);
         dose{i}=.; realvis{i}=.;
      end;
end;

❹dose{visnum}=dosvalue;
  realvis{visnum}=visit;

❺if last.patient then output;
run;
```

❶The variable *visnum* is created for tallying the number of visits. ❷The ARRAY statement is made flexible for any number of visits by the macro variable *nmaxvis* obtained from the previous example. ❸The doses and the actual visit numbers are first initialized to missing. ❹Then each time an observation is read, the value of the array index, *visnum*, determines the element of the array that takes the *dosvalue*. The same logic is applied to *realvis*.

❺After all the observations for each patient are read, the last observation of each patient is output.

b. HORIZONTAL TO VERTICAL DATA FORMAT

The following program transposes the data back to vertical format without losing the original regular and interim visit values.

```
data h2v(keep=investig patient visit dosvalue);
      set v2h;
      by investig patient;

array _d{*} dose1-dose&nmaxvis;
array _v{*} realvis1-realvis&nmaxvis;

do i=1 to dim(_d);
      dosvalue=_d{i};
      visit=_v{i};
      if visit>.z then output;
end;
run;
```

6. COMBINING DATA

Use the KEEP, DROP or RENAME statements to keep the DATA steps and the data sets clean. Save only the variables that are necessary at each step.

Use the "OPTIONS MSGLEVEL=I;" to check for variables being overwritten from the MERGE statement. If two variables have the same name in more than one data set, the following message will appear in the log file:

INFO: The variable xx on data set A1 will be overwritten by data set B1.

If the variable mentioned in the INFO message is identical in the data sets and if it appears in the BY statement after the MERGE statement, the INFO message can be ignored.

The variables in the latter data sets in a MERGE statement overwrite those in the previous ones if the same variables exist in more than one data set.

Check data structures before putting data sets together. Sometimes one of the data sets needs to be transposed.

7. DERIVED VARIABLES

Do a PROC FREQ on the original variables by the derived variables to see if variables are

properly derived. If the derived variables are created with formats or IF statements, make sure that no category is missing.

```
proc freq data=chkfmt;
      tables origvar*derivvar/list missing;
```

Check to see if the newly formatted or created variables are truncated. Use the LENGTH statement or the ATTRIB statement to define length and avoid truncation.

8. FIRST. AND LAST. SELECTIONS

Combining the FIRST. statement or the LAST. statement with the WHERE or the IF conditions in the same data step may cause unexpected results. The conditional statements may have already eliminated observations that are the FIRST. or the LAST. observations.

9. SUMMING VARIABLES

The SUM function:

```
x = sum (a, b, c);
```

is different from adding all variables together:

```
x = a + b + c;
```

If there are no missing arguments, both of the above statements produce the same results. But if there are missing arguments, the second statement returns a missing total and cause the message "NOTE: Missing values were generated as a result of performing an operation on missing values" to appear in the log file. The first statement returns the sum of all non-missing arguments, and the message about missing values will not appear in the log file.

10. AUTOMATIC DATA TYPE CONVERSIONS

Most SAS programmers are probably familiar with the statement that to eliminate automatic data type conversion messages in SAS log, use the PUT function to convert numeric variables to character variables, and use the INPUT function to convert character variables to numeric variables.

However, the following code generates an error message, *ERROR: 228-185: Informat $VARFMT is unknown*, and the conversion from character to numeric does not take place:

```
c_to_n=input(varname,$varfmt.);
```

One way to do the conversion is:

```
c_to_n=put(varname,$varfmt.)*1;
```

The character variable is converted to numeric variable by multiplying the result with one. But it will still generate the message, *NOTE: Character values have been converted to numeric values at the places given by: (Line): (Column).*

A better way to do the conversion is:

```
c_to_n=input(left(put(varname,$varfmt.)),8.);
```

The *left* function removes leading blanks from the character variable.

11. VARIABLE FORMATS

Sometimes a SAS data set contains variables with permanently associated formats. The format program and the format catalog may not have been sent to you, or the format catalog may not be compatible with your version of SAS; hence the SAS data set generates error messages when processed. To dissociate a permanently associated format from a variable called *fmtvar* in order that the data can be processed, create a DATA step and remove the format as follows:

```
data remove;
      set perm.fmtds;
format fmtvar;
```

The advantage of associating formats with variables in a FORMAT statement alone, or in a FORMAT statement within an ATTRIB statement or within PROC SQL is that the actual values of the data remain intact during calculation, regardless of the formats attached. The disadvantage is that if the formats are not available in permanent SAS format catalog, the SAS system will issue an error message and stops processing when the SAS data sets are put to use.

One solution is to create additional variables by using the PUT statement to put out the formats permanently as values.

Another solution is to store the formats in a permanent SAS data set. For example:

```
proc format library=library cntlout=perm.fmtsd2;
      select A-Z $A-$Z;
run;
```

The SELECT statement A-Z $A-$Z is a shortcut to refer to all the numeric, character and picture formats instead of itemizing them.

Before release 6.11 of the SAS software, backward compatibility in transporting SAS formats was a problem. Before then, moving the formats from lower to higher releases was fine, but moving from higher to lower releases was not fully supported.

By saving the formats as a permanent SAS data set, the data set can be made a transport file and the transport file is portable across platforms.

It is helpful to send the SAS codes for the formats and PROC CONTENTS outputs associating the formats with variables along with data transfers.

12. OUTPUT ASCII FILES

The following program is adapted from the %FLATFILE program in SAS Communications and in the SUGI 21 Proceedings to automate the process of writing out ASCII files. The %FLATFILE program outputs an ASCII file from a SAS data set one at a time. The following program adds to the %FLATFILE program by converting every SAS data set in the same directory into ASCII files all at once. The additional codes and modifications are highlighted in bold. Let's call this program %MASCII.

```
%macro mascii(lib=);

%let lib=%upcase(&lib);

proc sql;
     create view temp as
     select *
      from dictionary.columns
     ❶where libname="&lib";

proc sort data=temp out=datainfo;
     by ❷memname ❸npos;
run;

data _null_;
     set datainfo end=eof;
     by memname npos;
if last.memname;
     n+1;
if eof then call symput('numds',put(n,8.));
call symput('ds'||left(put(n,8.)),trim(memname));
run;


%do j=1 %to &numds;
     data &&ds&j;
          set datainfo;
          where memname="&&ds&j";
     run;
     data _null_;
          set &&ds&j end=last;
     ❹call symput ('var'||left(put(_n_,3.)),name);
     ❺ if upcase(type)='CHAR' then
     ❻    call symput('format'||left(put(_n_,3.)),'$'||
     ❼                 put(length,3.)||'.');
     else if format^=' ' then
          call symput('format'||left(put(_n_,3.)),format);
      else call symput('format'||left(put(_n_,3.)),
          'best10.');

     if last then call symput('numvar',left(put(_n_,3.)));
     run;

     data _null_;
          set &lib..&&ds&j;
     file "&&ds&j..dat" ls=100;
     put
          %do i=1 %to &numvar;
               &&var&i &&format&i +1
          %end;;
      run;
%end;

%mend mascii;

%mascii(lib=abc123);
```

Since PROC SQL contains information on all the variables in a data set, it is used to print out the variables in an ASCII file. Specifically, the information used in the program are: LIBNAME, MEMNAME, NPOS, NAME, TYPE, FORMAT and LENGTH. ❶LIBNAME describes the libref used for the data library. ❷MEMNAME contains the data set names. ❸NPOS indicate the relative position of the variable in the data set. ❹NAME contains variable names. ❺TYPE refers to the type of variables: *num* is for numeric, *char* is for character. ❻FORMAT gives the variable format and is blank if no format is given. ❼LENGTH is for variable length.

The additional codes find out the maximum number of data sets in a directory and use a DO loop to process all the data sets.

A simpler way to create a fixed column, space delimited ASCII file is:

```
%macro mascii2(lib=,ds=);

%let lib=%upcase(&lib);
%let ds=%upcase(&ds);

data _null_;
      set &lib..&ds;
file "&ds.dat" ls=100;
put (_all_) ( ' ');
run;

%mend mascii2;
%mascii2(lib=abc123,ds=demo);
```

The space within the single quotes can be replaced with other special character as delimiters. The order of the variables from the input file will be preserved in the output ASCII file.

As in the example on %MASCII, codes can be added to convert every SAS data set in the same directory into ASCII files all at once.

III. MICROSOFT EXCEL FILES

With the Windows environment becoming more and more popular, very often the data files are in Excel format. There are different techniques available in the SAS system for data transfer between Excel and SAS. The Dynamic Data Exchange (DDE) facility is commonly used. The references at the end of this paper cite a very detailed article on DDE in the SUGI 20 proceedings.

The following covers a method using SAS/ACCESS software in the Windows environment with SAS version 6.12.

Here is a caveat for preparing Excel files. With PROC ACCESS in version 6.12 of the SAS software, each Excel file needs to be saved as the type: Microsoft Excel 97 & 5.0/95 workbook. Otherwise, there will be an error message when the Excel file is imported into SAS. The error message is:

error: database error
unexpected OLE2 file manipulation error
file work._imex_.data does not exist
import unsuccessful

1. EXCEL TO SAS

Below is an example to convert every Microsoft Excel file in the same directory to SAS data sets all at once with one program.

The example is adapted from SAS usage note TS020. The original codes convert one file at a time. I have added to the original codes the option to convert one or more than one file in the same directory and the option to identify the range of the Excel data to be converted. The modifications are highlighted in bold.

```
%macro xls_sas(lib=,pathname=);

❶%let j=%words(&string);

%do i=1 %to &j;

    %let oneds=%upcase(%scan(&string,&j));

   %let lib=%upcase(&lib);
   %let pathname=%upcase(&pathname);

    proc access dbms=xls;
        create work.&oneds..access;
        path="&pathname\&oneds..xls";
   ❷scantype=yes;
   ❸getnames=yes;

   %if %upcase(&oneds)=AE %then %do;
        ❹range='a1..y338';
    %end;
    %else %if %upcase(&oneds)=PE %then %do;
        range='a1..ag30';
    %end;

   ❺assign=yes;
   ❻mixed=yes;

   ❼create work.&oneds..view;
        select all;
   run;

   ❽proc access viewdesc=work.&oneds
        out=&lib..&oneds;
   run;

%end;

%mend xls_sas;

❾%let string=ae dose drug demo ecg  medhx pe vs;
%xls_sas(lib=abc123,
        pathname=%str(c:\task1\exceldat));
```

The macro WORDS❶ is available on page 256 of the SAS Guide to Macro Processing. WORDS counts the number of words listed in the %LET *string*= statement, and returns the number of words. ❷SCANTYPE scans entire column to determine variable type. ❸GETNAMES reads the first row of data for variable names. The default for GETNAMES is NO and the default SAS variable names are VAR0, VAR1, VAR2, etc. ❹Sometimes Excel files contain missing values in all fields at the end of the files. RANGE restricts the reading of specific rows and columns so that rows with missing values in all fields are not included in the conversion. The RANGE specification is optional. ❺ASSIGN assigns SAS variables the names read from GETNAMES. ❻MIXED allows numeric-to-character conversion in mixed data columns so that numeric values are displayed as characters instead of missing values. ❼The view and descriptor must be created even though the goal is to create a SAS data set. ❽The output data set prefix or name needs to be different from the view descriptor prefix or name. The

same prefixes and the same names will result in an error message: *ERROR: Unable to create PREFIX.XXX.DATA because PREFIX.XXX.VIEW already exists.* The prefixes in the above program are made different, one of the prefixes is *work* and the other is *&lib*, so that the data set names can remain the same.

❾The Excel files can also be converted to SAS data sets one at a time by supplying one data set name in the %LET *string*= statement, or by pulling down the FILE menu and choosing the option IMPORT.

2. SAS TO EXCEL

Likewise, SAS data sets can be converted to Excel files one at a time by pulling down the FILE menu and choosing the option EXPORT.

The following example for converting SAS files to Excel is adapted from SAS usage note TS020. Modifications, highlighted in bold, are made so that the program can have the option to handle one or more than one file at a time, and can optionally rename or reformat variables.

```
%macro sas_xls(lib=,pathname=);

%let j=%words(&string);

%do i=1 %to &j;

    %let oneds=%upcase(%scan(&string,&j));

   %let lib=%upcase(&lib);
   %let pathname=%upcase(&pathname);

   ❶options noxwait;
     x "del &pathname\&oneds..xls";

   proc dbload dbms=xls data=&lib..&oneds;
   path="&pathname\&oneds..xls";

   ❷version=5;
   ❸putnames=yes;

   %if %upcase(&oneds)=MEDHX %then %do;
        ❹limit=0;
        ❺rename oldname=newname;
    %end;
    %else %if %upcase(&oneds)=VS %then %do;
        ❻where …where condition…;
        ❼format varname 4.1;
        ❽label;
    %end;

   ❾reset all;

   load;
   run;

%end;

%mend sas_xls;

%let string=ae dose drug demo ecg medhx pe vs;
%sas_xls(lib=abc123,
        pathname=%str(c:\task2\sasdat));
```

❶ Existing Excel file cannot be overwritten by DBLOAD and needs to be deleted beforehand. The X statement temporarily exits the SAS session to execute the command of deleting an Excel file. The NOWAIT option reactivates the SAS session after the command in the X statement is executed without having to type EXIT. ❷ The version of Excel is optional. ❸The PUTNAMES option writes column names to the first row of a SAS data set. ❹The LIMIT statement is required if there are more than 5000 observations. ❺The RENAME statement redefines names for columns in the Excel file. ❻The WHERE statement subsets output observations. ❼Some variables may appear in the Excel file with leading zeroes, and can be formatted to remove the leading zeroes before outputting to Excel file. ❽The LABEL statement uses SAS variable labels as column names. ❾RESET ALL is required if LABEL or DELETE is used.

IV. TRANSPORTING FILES TO DIFFERENT OPERATING SYSTEMS OR RELEASES OF THE SAS SOFTWARE

If the sending and the receiving host environments are different, SAS files need to be put in transport format for transfer.

There are several ways to create transport files. The selection of which method to use depends on the characteristics of the host environments.

SAS/CONNECT software is the best choice if the software is available on both host systems, since the software is dedicated to this function.

Otherwise, PROC CPORT and PROC CIMPORT are better choices for two major reasons: First, they can import and export SAS data sets, SAS catalogs or SAS data libraries. Second, they are compatible with PROC UPLOAD and PROC DOWNLOAD of the SAS/CONNECT software. However, the transport files created by PROC CPORT cannot

be processed or unpacked by PROC COPY or PROC XCOPY.

PROC COPY or PROC XCOPY can import and export SAS data sets only. The transport files created by PROC COPY or PROC XCOPY cannot be processed or unpacked by PROC CIMPORT.

V. SUMMARY

Data transfers need careful preparation. Data are critical groundwork. If the data are not correct, the analyses would not lead to the right conclusions.

The key for doing data transfers is to have thorough planning, to check each step in the program development process, to be consistent in naming and formatting, to develop error-detecting tools, to find patterns in data, and to write programs to automate tasks performed frequently.

REFERENCES

Buchecker, M. Michelle (1996), "%Flatfile, and Make Your Life Easier," Proceedings of the Twenty-First Annual SAS Users Group Conference, 21, 178-180.

Didier, Cros (1995), "Use of Dynamic Data Exchange (DDE) for Data Transfer Between SAS (Versions 6.08 and 6.10 under Windows) and Excel," Proceedings of the Twentieth Annual SAS Users Group Conference, 20, 110-118.

"Counting the Number of Words in a String," SAS® Guide to Macro Processing, Version 6, Second Edition, Cary, NC: SAS Institute Inc., 1990, 256.

"Tips and Techniques," SAS Communications, Vol. XX, No. 1, First Quarter 1994, 7-8.

For additional information, contact:

Aileen L. Yam
Clinical Information Analysis, Inc.
40 Linden Lane
Plainsboro, NJ 08536