DATA Step Essentials

Neil Howard, Ingenix Pharmaceutical Services, Basking Ridge, NJ

Abstract

The DATA step is the most powerful tool in the SAS system. Understanding the internals of DATA step processing, what is happening and why, is crucial in mastering code an output. Concepts you should understand:

- what is the Logical Program Data Vector (LPDV or PDV),
- what are automatic SAS variables and how are they used,
- what is the SAS Supervisor,
- why it’s important to understand the internals of DATA step processing,
- what happens at program compile time,
- what’s actually happening at execution time,
- how are variable attributes captured and stored.

By understanding DATA step processing, you can debug your programs and interpret your results with confidence.

This paper focuses on techniques that capitalize on the power of the DATA step and working with (and around) the default actions. Topics include:

- compile versus execution time activities;
- organizing your data to maximize execution;
- data defaults, data conversions;
- missing values, formatting values;
- ordering variables;
- functions for: editing data, assigning values, shortening expressions, performing table lookup;
- data management;
- effectively creating SAS data sets;
- the logic of the MERGE;
- efficiency techniques.

‘Real world’ examples are presented to illustrate each topic.

Introduction

SAS procedures are powerful and easy to use, but the DATA step offers the programmer a tool with almost unlimited potential. In the real world, we’re lucky if systems are integrated, data is clean and system interfaces are seamless. The DATA step can help you, your programmers, your program, and your users perform better in the real world – especially when you take advantage of the available advanced features. Given that any of the topics/examples covered in this presentation have more than enough details, idiosyncrasies, and caveats to warrant its own tutorial, we will address selected essential processing tips and a range of “real world” situations that illustrate:

- DATA step compile and execute
- coding efficiencies
- maximizing execution
- data: type conversions and missing values
- other data issues
- data set management
- table lookup

DATA Step Compile vs. Execute

There is a distinct compile action and execution for each DATA and PROC step in a SAS program. Each step is compiled, then executed, independently and sequentially. Understanding the defaults of each activity in DATA step processing is critical to achieving accurate results. During the compilation of a DATA step, the following actions (among others) occur:

- syntax scan
- SAS source code translation to machine language
- definition of input and output files
- creating:
  - input buffer (if reading any non-SAS data),
  - Program Data Vector (PDV),
  - and data set descriptor information
- set variable attributes for output SAS data set
- capture variables to be initialized to missing

Variables are added to the PDV in the order seen by the compiler during parsing and interpretation of source statements. Their attributes are determined at compile time by the first reference to the compiler. For numeric variables, the length is 8 during DATA step processing; length is an output property.

The variables output to the SAS data set are determined at compile time; the automatic variables are never written, unless they have been assigned to SAS data set variables set up in the PDV (_N_, _ERROR_, end=, in=, point=, first., last., and implicit array indices); the variables written are specified by user written DROP and/or KEEP statements or data set options; the default being all non-automatic variables in the PDV. The output routines are also determined at compile time.

The following statements are compile-time only statements. They provide information to the PDV. and cannot by default (except in the macro language) be conditionally executed. Placement of the last six statements (shown below) is critical because the attributes of variables are determined by the first reference to the compiler:

- drop, keep, rename
- label
- retain
- length
- format, informat
- attrib
- array
- by
- where

Once compilation has completed, the DATA step is executed: the I/O engine supervisor optimizes the executable image by controlling looping, handling the initialize-to-missing instruction, and identifying the observations to be read. Variables in the PDV are initialized, the DATA step program is called, the user-controlled DATA step machine code statements are executed, and the default output of observations is handled.

By understanding the default activities of the DATA step, the SAS programmer can make informed and intelligent coding decisions. Code will be more flexible and efficient, debugging will be straightforward and make more sense, and program results can be interpreted readily.

Coding Efficiencies & Maximizing Execution

The SAS system affords the programmer a multitude of choices in coding the DATA step. The key to optimizing your code lies in recognizing the options and understanding the implications. This may not feel like advanced information, but the application of these practices has far-reaching effects.

Permanently store data in SAS data sets. The SET statement is dramatically more efficient for reading data in the DATA step than any form of the INPUT statement (list, column, formatted). SAS data sets offer additional advantages, most notably the self-documenting aspects and the ability to maintain them with
procedures such as DATASETS. And they can be passed directly to other program steps.

A “shell” DATA step can be useful. Code declarative, compile-only statements (LENGTH, RETAIN, ARRAY) grouped, preceding the executable statements. Block-code other non-executables like DROP, KEEP, RENAME, ATTRIB, LABEL statements following the executable statements. Use of this structure will serve as a check list for the housekeeping chores and consistent location of important information. Use consistent case, spaces, indentation, and blank lines liberally for readability and to isolate units of code or to delineate DO-END constructions.

Use meaningful names for data sets and variables, and use labels to enhance the output. Comment as you code; titles and footnotes enhance an audit trail. Based on your understanding of the data, code IF-THEN-ELSE or SELECT statements in order of probability of execution. Execute only the statements you need, in the order that you need them. Read and write data (variables and observations) selectively, reading selection fields first, using DROP/KEEP, creating indexes. Prevent unnecessary processing. Avoid GOTOs. Simplify logical expressions and complex calculations, using parentheses to highlight precedence and for clarification. Use DATA step functions for their simplicity and arrays for their ability to compact code and data references.

Data Conversions

Character to numeric, and numeric to character, conversions occur when:
- incorrect argument types passed to function
- comparisons of unlike type variables occur
- performing type-specific operations (arithmetic) or concatenation (character)

SAS will perform default conversions where necessary and possible, but the programmer should handle all conversions to insure accuracy. The following code illustrates:
- default conversion,
- numeric-to-character conversion using PUT function,
- character-to-numeric conversion with INPUT function:

```
data convert1;
    length x $ 2 y $ 1;
    set insas; *contains numeric variables flag and code;
    x = flag;
    y = code;
    run;

data convert2;
    length x $ 2 y 8
    set insas; *contains numeric variables flag and code;
    x = put(flag, 2.);
    y = input(put(code, 1.), 8.);
    run;

data convert3;
    length x 2;
    set insas; *contains character variable status;
    z = input(status, 2);
    run;
```

Missing Data

The DATA step provides many opportunities for serious editing of data and handling unknown, unexpected, or missing values. When a programmer is anticipating these conditions, it is straightforward to detect and avoid missing data; treat missing data as acceptable within the scope of an application; and even capitalize on the presence of missing data. When a value is stored as “missing” in a SAS data set, its value is the equivalent of negative infinity, less than any other value that could be present. Numeric missings are represented by a “.” (a period); character by “ ” (blank). Remember this in range checking and recoding. Explicitly handle missing data in IF-THEN-ELSE constructions; in PROC FORMATs used for recoding; and in calculations. The first statement in the following example:

```
if age < 8 then agegroup = "child";
if agegroup = " " then delete;
```

will include any observations where age is missing in the agegroup “child”. This may or may not be appropriate for your application. A better statement might be:

```
if (. < age < 8) then agegroup = "child";
```

Depending on the user’s application, it may be appropriate to distinguish between different types of missing values encountered in the data. Take advantage of the twenty-eight special missing values:

```
_ _ _ _ A - Z
```

All these missing values test the same. Once a missing value has resulted or been assigned, it stays with the data, unless otherwise changed during some stage of processing. It is possible to test for the presence of missing data with the N and NMISS functions:

```
y = nmiss(age, height, weight, name);
** y contains the number of nonmissing arguments;

z = n(a,b,c,d);
** z contains the number of missings in the list;
```

Within the DATA step, the programmer can encounter missing data in arithmetic operations. Remember that in simple assignment statements, missing values propagate from the right side of the equal sign to the left; if any argument in the expression on right is missing, the result on the left will be missing. Watch for the “missing values generated” messages in the SAS log. Although DATA step functions assist in handling missing values, it is important to understand their defaults as well. Both the SUM and MEAN functions ignore missing values in calculations: SUM will add all the non-missing arguments and MEANS will add the nonmissings and divide by the number of nonmissings. If all the arguments to SUM or MEANS are missing, the result of the calculations will be missing. This works for MEAN, but not for SUM, particularly if the intention is to use the result in a later calculation:

```
x = a + b + c; * if any argument is missing, x = . ;
x = SUM(a,b,c); * with missing argument, x is sum of nonmissings;
x = SUM(a,b,c,0); * if a,b,c are missing, result will be zero;
y = (d + e + f + g) / 4; *number of nonmissings is divided by 4;
y = MEAN(d,e,f,g); * if all argument s are missing, y = . ;
```

Since there are 90+ DATA step functions, the moral of the function story is to research how each handles missing values. New variables created in the DATA step are by default initialized to missing at the beginning of each iteration of execution. Declare a RETAIN statement to override the default:

```
retain total 0;
total = total + add_it;
* this will work as long as add_it is never missing;
```
The SUM statement combines all the best features of the RETAIN statement and the SUM function:

```
total + add_it;
```

*total is automatically RETAINed;
* add_it is added as if using the SUM function;
* missings will not wipe out the accumulating total;

Missing values in the right-most data set coded on a MERGE or UPDATE statement have different effects on the left-most data set. When there are common variables in the MERGE data sets, missings coming from the right will overwrite. However, UPDATE protects the variables in the master file (left-most) from missings coming from the transaction file. (See Real World 7 example.)

### Other Data Issues

#### Re-Ordering Variables

SAS-L users periodically carry on the discussion of re-ordering variables as they appear in a SAS data set. Remember that as the compiler is creating the PDV, variables are added in the order they are encountered in the DATA step by the compiler. This becomes their default position order in the PDV and data set descriptor. The best way to force a specific order is with a RETAIN statement, with attention to placement. Make sure it is the first reference to the variable and the attributes are correct. It is possible to use a LENGTH statement to accomplish this, but a variable attribute could be inadvertently altered.

```
data new;
retain c a v;
* first reference to a b c;
set indata;
* incoming position order is a b c;
x = a || b || c;
run;
```

```
data new;
length x $ 35 a $ 10 b $ 7 c $ 12;
* first reference to x a b c;
set indata; *contains a b c, in that position order;
x = a || b || c;
run;
```

#### Handling Character Data

Character-handling DATA step functions can simplify string manipulation. Understand the defaults and how each function handles missing data for optimal use.

#### Length of target variables

Target refers to the variable on the left of the equal sign in an assignment statement where a function is used on the right to produce a result. The default length for a numeric target is 8; however, for some character functions the default is 200, or the length of the source argument. The SCAN function operates unexpectedly:

```
data _null_;  
x = 'abcdefghijklmnopqrstuvwxyz';  
y = scan(x,1,'k');  
put y=;  
run;
```

```
y=abcdefghij
```

The results from SUBSTR are different:

```
data old;  
a='abcdefghijklmnopqrstuvwxyz';  
b=2;  
c=9;  
run;
```

```
data new;  
set old;  
x=substr(a,23,4);  
y=substr(a,b,3);  
z=substr(a,9,c);  
put a= b= c= x= y= z=;  
* a is length $ 26; * x y z have length $ 26;  
run;
```

```
data old;  
length idnum $ 10 name $ 25 age 8;  
idnum=substr(var1_200,1,10);  
name=substr(var1_200,11,25);  
age=substr(var1_200,36,2);  
* length statement overrides default of 200  
* for idnum, name, and age;  
run;
```

#### SUBSTR as pseudo-variable

Another SAS-L discussion involved the use of SUBSTR as a pseudo-variable. Note that when the function appears to the left of the equal sign in the assignment statement, text replacement occurs in the source argument:

```
data fixit;  
source = 'abcdefghijklmnopqrstuvwxyz';  
substr(source, 12, 10) = '##########';  
put source=;  
run;
```

```
source=abcdefghijk##########vwxyz
```

#### Numeric substring

A similar function to SUBSTR if often desired for numerics. One cumbersome solution involves: 1) performing numeric to character conversion, 2) using SUBSTR to parse the string, and 3) converting the found string back to numeric. SAS would also do such conversions for you if you reference a numeric as an argument to a character function or include a character variable in a numeric calculation. See section on data conversions.

A simpler and less error-prone solution (trick) is the use of the numeric MOD and INT functions:

```
data new;  
a=123456;  
x = int(a/1000);  
y = mod(a,1000);  
z = mod(int(a/10),100);  
put a= x= y= z=;  
run;
```

```
a=123456  
x=123  
y=456  
z=34
```

#### Handling imbedded blanks
The TRIM and TRIMN functions are used to remove embedded blanks. Notice the different results:

```
data _null_;  
  string1='trimmed ';  
  string2='?';  
  string3='!';  
  string4=' ';  
  w=string1||string2;  
  x=trim(string1)||string3;  
  y=string2||trim(string4)||string3;  
  z=string2||trimn(string4)||string3;  
  put w= x= y= z=;  
run;```

```
w = trimmed ?  
x = trimmed!  
y = ? !  
z = ?!```

### Table Lookup

Recoding is a common programming challenge, and can be accomplished in several ways:

- hard-coded IF statements
- MERGE
- PROC FORMAT with the PUT function
- data driven FORMATS.

#### Hard-Coded IF Statements

For this example, we know the DEPTINFO (descriptive information for each DEPTNAME) for each sort code (SORTCD):

- 1001 Operations
- 1002 Hardware
- 1003 Software (IBM)
- 1004 Software (MAC)
- 1005 LAN/WAN
- 1006 Technical Support
- 1007 Help Desk 1008 Administrative Support
- 1009 Documentation Library
- 1010 Incentive Program
- 1011 Unassigned
- 1012 Unassigned
- 1013 Unassigned
- 1014 Unassigned
- 1015 Unassigned

Data set EXPENSES contains the expense data with only SORTCD as an identifier. It is required that all reports must display the lengthy department description.

```
1002 12 94 150000  
1005 12 94 200000  
1003 12 94 500000  
1006 12 94 329500  
1010 12 94 975200  
1007 12 94 150000  
1011 12 94 500000  
2004 12 94 230500```

The users want a listing and separate SAS data set with valid expense data (an “unassigned” sort code with expenses is considered an error). The deliverables will be an error report and an error file to facilitate corrections.

#### Table Lookup using MERGE

EXAMPLE 2: Table Lookup with MERGE (assuming a data set with department descriptions (DEPTINFO) for each SORTCD has been created:

```
proc sort data=expenses;  
by sortcd;  
run;  
proc sort data=deptinfo;  
by sortcd;  
run;  
data expens2 errdept missmnth;  
merge expenses(in=inexp) deptinfo(in=indpt);  
by sortcd;  
if inexp and indpt then  
do;  
  if index(upcase(deptname),'UNASSIGNED') > 0 then  
    output errdept;  
  else output expens2;  
  end;  
else if inexp and not indpt then output errdept;  
else if (indpt and not inexp) and  
  index(upcase(deptname),'UNASSIGNED') = 0  
then output missmnth;  
run;```

The MERGE provides the users with a “bonus” file by coding multiple data set names on the DATA statement and using the IN= option on the MERGE statement. Data set EXPENS2 contains the valid expense data; ERRDEPT holds the incorrect expense data; and MISSMnth (optional) shows which sortcodes have no expense data for the month.

#### PROC FORMAT with the PUT function

EXAMPLE 3: PROC FORMAT with PUT function:

```
proc format;  
  value regfmt 100-200 = "NE"  
    201-300 = "NW"  
    301-400 = "SE"  
    401-500 = "SW";  
run;  
data new;```
set indata; *contains numeric variable city;!ncontains numeric variable city;
region = put(city, regfmt.);
*creates a new variable region based on values of city;
runt;

data sw;
set indata;
if put(city, regfmt.) = "SW";
* creates a subset based on value of city;
* does not create a new variable to do this;
runt;

Accomplishing data recoding using PROC FORMAT with the PUT function provides several benefits to the users and programmer: it is readable; easy to maintain -- the list of values need only be changed in one location; the formats can be permanently stored in a format library; the DATA step code itself is shorter and easier to follow.

Data Driven PROC FORMAT Generation

EXAMPLE 4: Table Lookup using a SAS data set to generate the PROC FORMAT:

data fdnm(keep=start end label fmtname);
set deptinfo end=eofdept;
length label $32;
start = sortcd;
end = sortcd;
label = deptname;
fmtname = 'convdept';
output fdnm;
if eofdept then
do;
   start = .;
   end = .;
   label = "ERROR";
   fmtname = 'convdept';
   output fdnm;
end;
run;
proc format cntlin=work.fdnm;
runt;

The overhead associated with this solution comes from reading the DEPTINFO dataset and using it to make a CNTLIN dataset for PROC FORMAT (see SAS log). The temporary dataset, WORK.FDNM, is passed to PROC FORMAT with the CNTLIN= options to create the SAS format CONVDEPT:

<table>
<thead>
<tr>
<th>FORMAT NAME</th>
<th>CONVDEPT LENGTH: 22 NUMBER OF VALUES: 18 MIN LENGTH: 1 MAX LENGTH: 40 DEFAULT LENGTH 22 FUZZ: STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>START END LABEL (VER. 6.11)</td>
<td></td>
</tr>
<tr>
<td>. . ERROR</td>
<td>1002 150000 12 94 Hardware</td>
</tr>
<tr>
<td>. . ERROR</td>
<td>1003 500000 12 94 Software (IBM)</td>
</tr>
<tr>
<td>. . ERROR</td>
<td>1005 200000 12 94 LAN/WAN</td>
</tr>
<tr>
<td>. . ERROR</td>
<td>1006 329500 12 94 Technical Support</td>
</tr>
<tr>
<td>. . ERROR</td>
<td>1007 150000 12 94 Help Desk</td>
</tr>
<tr>
<td>. . ERROR</td>
<td>1010 975200 12 94 Incentive Program</td>
</tr>
<tr>
<td>. . ERROR</td>
<td>1011 50000 12 94 Unassigned 1011</td>
</tr>
<tr>
<td>. . ERROR</td>
<td>2004 230500 12 94 Unassigned 1011</td>
</tr>
</tbody>
</table>

Using format from CNTLIN with PROC PRINT:

<table>
<thead>
<tr>
<th>OBS</th>
<th>SORTCD</th>
<th>EXPENSES</th>
<th>MONTH</th>
<th>YEAR</th>
<th>DEPTNAME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1002</td>
<td>150000</td>
<td>12</td>
<td>94</td>
<td>Hardware</td>
</tr>
<tr>
<td>2</td>
<td>1003</td>
<td>500000</td>
<td>12</td>
<td>94</td>
<td>Software (IBM)</td>
</tr>
<tr>
<td>3</td>
<td>1006</td>
<td>329500</td>
<td>12</td>
<td>94</td>
<td>Technical Support</td>
</tr>
<tr>
<td>4</td>
<td>1007</td>
<td>150000</td>
<td>12</td>
<td>94</td>
<td>Help Desk</td>
</tr>
<tr>
<td>5</td>
<td>1010</td>
<td>975200</td>
<td>12</td>
<td>94</td>
<td>Incentive Program</td>
</tr>
<tr>
<td>6</td>
<td>1011</td>
<td>50000</td>
<td>12</td>
<td>94</td>
<td>Unassigned 1011</td>
</tr>
<tr>
<td>7</td>
<td>2004</td>
<td>230500</td>
<td>12</td>
<td>94</td>
<td>Unassigned 1011</td>
</tr>
</tbody>
</table>

Table Lookup Conclusions

For small lists and table lookups against small lists on relatively static data, the MERGE example is preferable to IF/ELSE. Where data are volatile, or the lookup list is very large, it will prove more efficient to use the PROC FORMAT with the PUT function and/or create the formats from data that drives the list. The formats are easily maintained, excellent documentation, and provide a mechanism for making changes in only one location in the program.

Notice from the example that other applications of PROC FORMAT with the PUT function become apparent. The table lookup can re-code variables, assign values, range-check values, and shorten expressions.

Data Set Management

Here's where the rubber meets the road -- the odd challenges encountered in user applications, like 'em or not. And this is where the power of the DATA step can be the most effective -- in handling “real world” situations:

- referencing a data set at compile time
- oddly located “bad” records
- writing for word processing packages
- variable-length raw data records
- deleting observations based on last in a series
- optimizing first, processing
- manipulating sort order
- choosing MERGE or UPDATE
Real World 1: Referencing a Data Set at Compile Time

It is often necessary to capture the number of observations in a data set at compile time:

```sas
data _null_;  
call SYMPUT('n_obs', put(n_obs, 5.));  
stop;  
set indata nob = n_obs;  
run;
```

The `SYMPUT` function in the example above will capture the number of observations from the data set descriptor at compile time, without processing any data. The value of the macro variable `&n_obs` becomes available to reference from another program step.

Real World 2: Other People’s Data

Other people’s data entry programs can cause unexpected problems. Suppose there’s a bug in the CICS/COBOL program that collects sales data. The first record for each city and each hour is known to be “bad” data. The COBOL programmers get rid of the record when they pass the data to the General Ledger system. However, other departments can only read the raw data for ad hoc reports.

This input data shows which lines should be deleted (note: you can’t delete the first observation and every third observation because there aren’t always three people working in an hour, nor do they necessarily occur in the same order):

<table>
<thead>
<tr>
<th>CITY</th>
<th>NAME</th>
<th>HOUR</th>
<th>DOLLARS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOSTON BILL</td>
<td>107000</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>BOSTON DAVE</td>
<td>345998</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>BOSTON JEAN</td>
<td>357654</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>BOSTON BILL</td>
<td>20</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>BOSTON BILL</td>
<td>25</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>LONDON JEAN</td>
<td>7936</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>LONDON JEAN</td>
<td>7936</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>LONDON MONTY</td>
<td>357654</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>LONDON MONTY</td>
<td>357654</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>LONDON MONTY</td>
<td>357654</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>LONDON MONTY</td>
<td>357654</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>LONDON MONTY</td>
<td>357654</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>PARIS PIERRE</td>
<td>7936</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>PARIS PIERRE</td>
<td>7936</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

A simple way to solve this problem uses the features of PROC SORT. After you read the raw data, sort the data by CITY and HOUR (not by name, so SAS will retain the names in the order they appeared in original data set). This sort assures there will never be HOUR 7 for LONDON occurring immediately after HOUR 7 for BOSTON. Use the first.hour automatic SAS variable to delete bad data.

```sas
proc sort data=citysale;  
by city hour;  
run;
```

```sas
data dropit keepit;  
set citysale;  
by city hour;  
if first.hour then  
do;  
output dropit;  
delete;  
end;  
output keepit;  
run;
```

Using `first.hour` would drop these:

<table>
<thead>
<tr>
<th>CITY</th>
<th>NAME</th>
<th>HOUR</th>
<th>DOLLARS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOSTON BILL</td>
<td>107000</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>BOSTON DAVE</td>
<td>345998</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>BOSTON JEAN</td>
<td>357654</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>BOSTON BILL</td>
<td>20</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>BOSTON BILL</td>
<td>25</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>LONDON JEAN</td>
<td>7936</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>LONDON JEAN</td>
<td>7936</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>LONDON MONTY</td>
<td>357654</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>LONDON MONTY</td>
<td>357654</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>LONDON MONTY</td>
<td>357654</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>LONDON MONTY</td>
<td>357654</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>LONDON MONTY</td>
<td>357654</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>PARIS PIERRE</td>
<td>7936</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>PARIS PIERRE</td>
<td>7936</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

Real World 3: VP’s Admin likes WORD

The user wants mailing labels from their SAS data set in WORD format. The text strings ‘NAME’ and ‘LOCATION’ must appear on the first line of the file; on the subsequent data lines, each field must be separated by the WORD tab character (hex value =09).

After the OUTLABEL file (ASCII) is written, it is “pulled” into WORD and merged with the WORD label document:

```sas
libname newlabl 'c:\saspaper';  
filename outlabel 'c:\saspaper\barbnew.txt';  
*** the note-1 field to stuff the envelopes with the right document;  
*** note-1 appears on the checklist but not the labels;  
proc sort data=newlabl.barb;  
by note1 lastname name1;  
run;
```

```sas
proc print data=newlabl.barb;  
title 'SAS dataset: newlabl.barb -- do labels with WORD';  
title2 'Envelope checklist';  
id name1;  
var lastname deptloc note1 note2;  
run;
```

```sas
data _null_;  
set newlabl.barb;  
length hextab $1;  
retain hextab '09'x;  
file outlabel;  
if _n_ = 1 then put @1 hextab 'NAME' hextab 'LOCATION';  
put hextab : $1. name1 : $15.-l lastname : $15.-l hextab : $1. deptloc : $8.-l ;  
run;
```

The resulting ASCII file is ready to bring into WORD (tab characters do not display):
Real World 4: Hinky Data

Transferring ASCII files between various software packages and platforms can also cause problems. When an ASCII file was transferred between a MACINTOSH mail program to a PC, the text lines were written as variable length records (vs. fixed on the MAC), and many apostrophes became represented by hex code 12 (shows as a '.' in the SAS LOG). In addition, the MAC tab character became '>' in the translated file.

If a 40 page story is late for a publishing deadline: you can:
- beg secretary to make changes in WP package;
- make changes in the word processor yourself; or
- write program using character manipulation functions.

The “damaged” file (note periods instead of apostrophes and >):

>>I'll never see another loss like that. No more soldiers, no more blood.
She hadn't ever tried to talk to the ghosts; it was hard to tell which one was
>The travel clerk didn't have to remind her not to approach the time
tourists,
said to the uncaring sky. "What's the good of all these dead custers,
anyway?"

filename hinky 'c:\saspaper\hinky.txt';
filename fixed 'c:\saspaper\fixed.txt';
data _null_; infile hinky missover length=lg;
    input @1 textline $varying200. lg;
    length badchar apos parachar $1;
    retain badchar '12'x apos parachar '>'; if index(textline,badchar) gt 0 then do;
        list;
        textline = translate(textline,apos,badchar);
    end;
    file fixed;
    if index(textline,parachar) gt 0 then do;
        textline = translate(textline,' ','parachar');
        put @4 textline;
    end;
    else put @1 textline;
run;

(undamaged text deleted)

Real World 6: They Want WHAT??!!!
The creation of a “super-sort” variable can allow you to minimize the number of first variables used to successfully process a data set. In this example, trouble tickets (TICKET) can be assigned to multiple directors (DIRECTOR) and multiple reporting areas (AREA) for investigation of system outages (DURATION). The system outages (OUTAGES) can affect multiple lines of business (LINEBUSN):

<table>
<thead>
<tr>
<th>TICKET</th>
<th>LINEBUSN</th>
<th>DIRECTOR</th>
<th>AREA</th>
<th>DURATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>321</td>
<td>NET</td>
<td>TURNER</td>
<td>OPERATIONS</td>
<td>35</td>
</tr>
<tr>
<td>565</td>
<td>CRP</td>
<td>MILLER</td>
<td>SOFTWARE</td>
<td>12</td>
</tr>
<tr>
<td>565</td>
<td>CRP</td>
<td>MILLER</td>
<td>LAN/WAN</td>
<td>12</td>
</tr>
<tr>
<td>565</td>
<td>NET</td>
<td>MILLER</td>
<td>SOFTWARE</td>
<td>15</td>
</tr>
<tr>
<td>565</td>
<td>NET</td>
<td>MILLER</td>
<td>LAN/WAN</td>
<td>30</td>
</tr>
<tr>
<td>565</td>
<td>BUS</td>
<td>MILLER</td>
<td>SOFTWARE</td>
<td>30</td>
</tr>
<tr>
<td>565</td>
<td>BUS</td>
<td>MILLER</td>
<td>LAN/WAN</td>
<td>30</td>
</tr>
<tr>
<td>436</td>
<td>CRP</td>
<td>MILLER</td>
<td>HARDWARE</td>
<td>80</td>
</tr>
<tr>
<td>436</td>
<td>CRP</td>
<td>MILLER</td>
<td>SOFTWARE</td>
<td>25</td>
</tr>
<tr>
<td>436</td>
<td>NET</td>
<td>JONES</td>
<td>HARDWARE</td>
<td>75</td>
</tr>
<tr>
<td>436</td>
<td>NET</td>
<td>MILLER</td>
<td>SOFTWARE</td>
<td>25</td>
</tr>
<tr>
<td>436</td>
<td>BUS</td>
<td>MILLER</td>
<td>SOFTWARE</td>
<td>75</td>
</tr>
<tr>
<td>436</td>
<td>BUS</td>
<td>JONES</td>
<td>HARDWARE</td>
<td>25</td>
</tr>
</tbody>
</table>

Real World 6: Need to delete last in a series

A health care worker has a data set with unequal lines of data per person for different years with the same variables per line. The objectives:
- keep all lines for person if last yr is 1991 or less
- delete all lines if year on last record is GE 1992:

data yrinfo;
    length id yr info $10;
    input id yr info info;
    run;

data keepite deleteit;
    set yrinfo;
    by id;
    length delflag 3;
    retain delflag ;
    if first.id then delflag = 0;
    if first.id and yr ge 92 then delflag = 1;
    if delflag = 1 then do;
        output deleteit;
        delete;
    end;
    if delflag = 0 then output keepit;
run;

I'll never see another loss like that. No more soldiers, no more blood.
The travel clerk didn't have to remind her not to approach the time
tourists,
said to the uncaring sky. "What's the good of all these dead custers,
anyway?"
To complete the tracking process, users want two reports: 1) a summary by line of business and director showing the total number of minutes for each ticket and the number of areas affected; 2) a list indicating line of business at the top of each page for every unique line of business/director/ticket combination.

A “super-sort” variable can be created (using character concatenation) to simplify processing, replacing the more tedious first processing for all the combinations of LINEBUSN, DIRECTOR and TICKET (though NOT AREA):

\[
data sortexmp;
set outages;
length suprsort $11;
suprsort =linebusn||substr(director,1,5)||put(ticket,3.0);
run;
*(Note numeric variable ticket converted to character for substring);
\]

Subsequent processing uses the SUPRSORT variable to produce the detail report and summary report file in one data step:

\[
proc sort data=sortexmp;
bysuprsort linebusn director area ticket;
run;
filename detail 'c:saspaper\suprdetl.prn';
data dirtotl(keep=linebusn director ticket numarea dirtot);
set sortexmp;
bysuprsort;
retain dirtot numarea;
file detail print;
if first.suprsort then
  do;
dirtot=0;
umarea = 0;
  put _page_;
title 'detail listing by line of business';
  put @5 'line of business: ' linebusn;
end;
dirtot + duration;
umarea + 1;
  put @5 'line of business: ' linebusn;
if last.suprsort then output dirtotl;
run;
proc sort data=dirtotl;
byleague director;
run;
proc print data=dirtotl;
byleague;
sum dirtot;
run;
\]

The SUPRSORT variable can also be used in SAS procedures, like PROC MEANS or PROC FREQ, to minimize the unnecessary _TYPE_s (PROC MEANS) or TABLEs (PROC FREQ) produced by using multiple BY statements. The summary report from data set DIURTOL:

\[
LINEBUSN=BUS
OBS DIRECTOR TICKET DIURTOT NUMAREA
1  JONES 436  25  1
2  MILLER 436  75  1
3  MILLER 565  60  2
   ----
LINEBUSN  160
\]

And the detail report looks like:

\[
LINE OF BUSINESS: BUS
BUS  JONES  HARDWARE  436  25

LINE OF BUSINESS: BUS
BUS  MILLER  SOFTWARE  436  75

LINE OF BUSINESS: BUS
BUS  MILLER  LAN/WAN  565  30
BUS  MILLER  SOFTWARE  565  30
\]

**Real World 7: Manipulating Sort Order**

When a Performance Tracking system was coded, three-character codes were used for line of business. However, the users rejected the report because the lines of business printed in alphabetical order, not in the order that the customers expected.

The first report generated appeared as follows:

**Listing by Line of Business in Alpha Order**

<table>
<thead>
<tr>
<th>Line of Business=ACT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>ACCOUNTS RECEIVABLE</td>
</tr>
<tr>
<td>CONTROLLER</td>
</tr>
<tr>
<td>GENERAL LEDGER</td>
</tr>
<tr>
<td>PAYROLL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line of Business=CRP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>BENEFITS</td>
</tr>
<tr>
<td>GROUNDS</td>
</tr>
<tr>
<td>HUMAN RESOURCES</td>
</tr>
<tr>
<td>MEDICAL</td>
</tr>
<tr>
<td>PURCHASING</td>
</tr>
<tr>
<td>RECEIVING</td>
</tr>
<tr>
<td>TRAVEL</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line of Business=NET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>LOCAL AREA NETWORK</td>
</tr>
<tr>
<td>ROLM EQUIPMENT</td>
</tr>
<tr>
<td>SITE LICENSE</td>
</tr>
<tr>
<td>SYSTEM SOFTWARE</td>
</tr>
<tr>
<td>TECHNICAL SUPPORT</td>
</tr>
<tr>
<td>WIDE AREA NETWORK</td>
</tr>
</tbody>
</table>

Using PROC FORMAT, the system designer can code the line of business and force the specific expected order on the report:

```
proc format;
  value $lobord 'NET' = 1
  'CRP' = 2
  'OPR' = 3
  'ACT' = 4;
  value nicename 1 = 'NETWORK'
  2 = 'CORPORATE'
  3 = 'OPERATIONS'
  4 = 'ACCOUNTING';
run;
```

```
data neword;
set applinfo;
length ordlob 3.;
ordlob = put(linebusn,$lobord.);
run;
```

And the detail report looks like:

\[
LINE OF BUSINESS: BUS
BUS  JONES  HARDWARE  436  25

LINE OF BUSINESS: BUS
BUS  MILLER  SOFTWARE  436  75

LINE OF BUSINESS: BUS
BUS  MILLER  LAN/WAN  565  30
BUS  MILLER  SOFTWARE  565  30
\]
An UPDATE application is actually called for. Create an update transaction, using named input and the special missing option (_) to change only the variables requiring update. Use the same variables on the transaction file as on the master file. Variables in the transaction file with missing values will NOT overwrite the fields in the master file. (LIMIT for Morgan has been explicitly coded to "._" to demonstrate this feature). Only those changes with the special missing character underscore (_) will update a master file field to missing (see Jody’s color):

```
data updttrns;
  length bdate 8 interest $15 limit 8;
  informat bdate mmddyy8.;
  input name= $ & bdate= $ size= $ color= $ interest= $& what= $ limit=;
  missing _ _ ;
  cards;
  name=morgan bdate=12-02-83 color=orange limit=:
  name=jody bdate=08-23-84 interest=Goose Bumps color=_
  name=suzanne bdate=11-15-50 size=na color=na interest=mystery series limit=5
  what=coworker
  ;
run;
```

The UPDATE statement produces the desired result:

```
NAME     BDATE       SIZE   COLOR   INTEREST    WHAT LIMIT
jody     08-23-84    g14     purple   Nancy Drew  niece       20
john      10-14-93    t4       red      Lion King  nephew      20
meghan    12-02-83    j5     green    music      godchild    50
morgan    12-02-83    j5     teal     theater    godchild    50
sal       04-11-45    mxl      none     hang gliding  college      5
mary ann  10-17-95    b18      pink     rattles    daughter 100
```

Using a MERGE to add a new person is fine. But the merge will produce unreliable results when changing values of any of the variables (Morgan’s favorite color to orange or Jody’s interest to Goose Bumps books). This application might suggest a file of change transactions (UPDTTRNS) and a merge by NAME and BDAT:

```
data newmstr2;
  merge master(in=inmast) updttrns(in=intran);
  if (inmast and intran) or (inmast and not intran)
     then output newmstr2;
  if intran and not inmast then output newmstr2;
run;
```

The resulting data set added Suzanne, but lost all of Jody's information except INTEREST. Morgan's color changed, but all of other information was lost:

```
NAME   BDATE       SIZE   COLOR  INTEREST    WHAT LIMIT
jody    08-23-84    g14     purple  Nancy Drew niece       20
john    10-14-93    t4      red     Lion King  nephew      20
mary ann 10-17-95  b18       pink   rattles   daughter 100
meghan  12-02-83    j5      green   music     godchild    50
morgan  12-02-83    j5      orange theater    godchild    50
sal     04-11-45    mxl     none    hang gliding college      5
suzanne 11-15-50    na     na      mystery series coworker 5
```

### References

Update, 1992

- Repole, W: Avoiding, Accepting, and Taking Advantage of Missing Data, 1994


SAS and SAS/GRAPH are registered trademarks of SAS Institute, Inc., Cary, NC.