Advanced DATA Step Topics and Tips

Neil Howard, MCI Communications
Cynthia Zender, MCI Communications

Abstract
Understanding the intricacies of the DATA step can make all the difference in your SAS programs. Since arrays, SAS dates, and testing SAS programs will be addressed in other papers, this talk will focus on the use of more advanced techniques to capitalize on the power of the DATA step and working with (and around) the default actions. Topics include: compile vs. execute, and organizing your data manipulations to maximize execution; data defaults and dealing with conversions, missing values, order and formatting; using functions for editing data, assigning values, shortening expressions, and performing table lookup; managing variable-length and odd input data; data management and effectively creating SAS data sets; the efficiency implications of different coding options; updating and merging; and more.

Handout
It should be noted that an additional handout will be available at the presentation that include more SAS code examples, the log and output from the test runs, as well as copies of the presentation transparencies.

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 programs, and your users perform better in the real world — especially when you take advantage of the available advanced features.

Any of the topics/examples covered in this presentation have enough information, idiosyncrasies, and caveats to warrant its own tutorial. The situations that follow address essential processing tips and a broad range of “real world” situations:
- DATA step compile and execute
- coding efficiencies and 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 in compiled, then executed, independently and sequentially. Understanding the defaults of each activity is critical to comprehending DATA step processing, in particular.

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
- creation of input buffer, Program Data Vector (PDV), and data set descriptor information
- setting of variable attributes for output SAS data set
- capture of variables to be initialized to missing

Compile time is when the output variables are specified; automatic variables are never written, unless they have been assigned to SAS data set variables (N, _ERROR_, end, in, point, first, last, and implicit array indices); the variables written are determined by user specified DROP and/or KEEP statements. The output routines are also determined at compile time.

The following are compile-time only statements. They provide information to the PDV, and cannot by default (except in macro) be conditionally executed. Placement of the last four is critical because the attributes of variables are determined by the first reference to the compiler:
- deep, keep, rename
- label
- retain
- length
- format, informat
- attrins
- array

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
Advanced Tutorials

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 and 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 are passed to a 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 the PUT function, and character to numeric conversion using the INPUT function:

```sas
data convert1;
  length x 5 y 1;
  set inas; *contains numeric variables flag and code;
  x = flag;
  y = code;
run;

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

data convert3;
  length x 2;
  set inas; *contains character variable status;
  x = 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:
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:

\[
\text{if} \ (< \text{age} < 8) \ \text{then agegroup = "child";}
\]

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 special missing values:

\[
A::z
\]

All these missing values are handled the same, and can be easily collapsed according to varying requirements:

\[
\begin{align*}
\text{if} \ \text{comment = "unknown"} & \ \text{then age} = A; \\
\text{else if} \ \text{comment = "refused to answer"} & \ \text{then age} = \text{A - z}; \\
\text{else if} \ \text{comment = "don't remember"} & \ \text{then age} = \text{B};
\end{align*}
\]

Once a missing value has occurred 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:

\[
y = \text{NMISS(age, height, weight, name)}; \\
y = \text{N}(a,b,c,d); \\
z = \text{NMISS} \text{list}; \\
z \ \text{contains the number of missing values in the list};
\]

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 MEANS functions ignore missing values in calculations: SUM will add all the non-missing arguments and MEANS will add the non-missings and divide by the number of non-missings. If all the arguments to SUM or MEANS are missing, the result of the calculations will be missing. This works for MEANS, but not for SUM, particularly if the intention is to use the result in a later calculation:

\[
x = a + b + c; * \text{if any argument is missing, } x \text{ is } .; \\
x = \text{SUM(a,b,c,d); * if all arguments are missing, result will be } .;
\]

Since there are 90+ DATA step functions, the moral of the function story is to read 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:

\[
\begin{align*}
\text{retain total 0;} \\
\text{total} & = \text{total + add_it;} \\
\text{* this will work as long as add_it is never missing;}
\end{align*}
\]

The SUM statement combines all the best features of the RETAIN statement and the SUM function:

\[
\begin{align*}
\text{total + add_it;} \\
\text{* total is automatically RETAINed;} \\
\text{* add_it is added if using the SUM function;} \\
\text{* missings will not wipe out the accumulating total};
\end{align*}
\]

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 bandied around the question of re-ordering variables as they appear in the 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. This becomes their default position order in the PDV and data set descriptor. The best way to force a specific order is with a LENGTH statement, with attention to placement. Make sure it is the first reference to the variable and the attributes are correct. If using RETAIN for this purpose, beware of the other potential effects it has on the code:

\[
\text{data new;} \\
\text{set indata;} \\
x = a + b + c; \\
\text{length x } $35 \text{ a } $10 \text{ b } $7 \text{ c } $12; \\
*\text{LENGTH statement does not affect PDV;} \\
\text{run;} \\
\text{data new;} \\
\text{length x } $35 \text{ a } $10 \text{ b } $7 \text{ c } $12;
\]

83
Handling Character Data

Character-handling DATA step functions can simplify string manipulation. Keep in mind the defaults and how each function handles missing data.

- **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_nuD_t set inputdata;
x = "abcdefgijklmnopqrstuvwxyz";
y = scan(x,1,'k');
put y; run;
y = abedefgij; *y has length of 200;
```

The results from SUBSTR are different:

```
data odd;
x = "abcdefgijklmnopqrstuvwxyz";
b=3;
c=5;
run;
data even;
set odd; x = substr(a,23,4); y = substr(b,5,3); z = substr(c,0,9); put a = b = c = y = z = ; *a is length $26; *x y z have length $26; run;
data odd;
length idnum $10 name $25 age 8;
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 = "abcdefgijklmnopqrstuvwxyz";
substr(source, 12, 10) = "____________";
run;
```

- **numeric substring**

A similar function to SUBSTR if often desired for numerics. One possible solution involves performing numeric to character conversion, SUBSTR, the conversion back to numeric. Or, try the numeric functions MOD and INT to accomplish the same purposes:

```
data rew;
a=123456;
x = int(a/1000);
y = mod(a,1000);
z = mod(a,int(a/1000));
p = x = y = z = ;
run;
a=123456
x=123
y=456
z=34
```

- **handling imbedded blanks**

The TRIM and TRIMN functions are used to removed embedded blanks. Notice the different results:

```
data null;
string1="trimmed ";
string2= " ";
string3=" ";
string4=" ";
w=trim(string4)||string3;
x=trim(string4)||string3;
y=trim(string4)||string3;
z=trim(string4)||string3;
p = 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

The input data for this example consists of DEPTINFO, a list of department names (DEPTNAME) for each sort code (SORTCD):
Data set EXPENSES contains the expense data with only SORTCD as an identifier. It is required that all reports must display the long department.

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

**EXAMPLE 1: Table Lookup with IF statements:**

```sas
data lup lferr;
  set expenses;
  length deptname $25;
  if l001 le 1010 then do;
    if sorted = 1001 then deptname = 'Operations';
    else if sorted = 1002 then deptname = 'Hardware';
    else if sorted = 1003 then deptname = 'Software (IBM)';
    else if sorted = 1004 then deptname = 'Software (MAC)';
    output lufxp;
  end;
  else output lhferr;
run;
```

This method uses the IF/ELSE statements efficiently and accomplishes the objective. But having a separate dataset for users to track the sort codes they can still assign would be more useful and easily maintainable.

**Table Lookup using MERGE**

**EXAMPLE 2: Table Lookup with MERGE**

```sas
proc sort data=expenses;
  by sorted;
run;
proc sort data=deptinfo;
  by sorted;
run;
data expenses errdept missmnth;
merge expenses(in=lufxp) deptinfo(in=indpt);
by sorted;
```

The MERGE provides the users with a "bonus" file by coding multiple dataset 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:**

```sas
proc format;
  value regfmt 190-200 = 'NE';
    201-300 = 'NW';
    301-400 = 'SE';
    401-500 = 'SW';
run;
```

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 dataset to generate the PROC FORMAT:**

```sas
data fdn{lteep.start end label Cntname);
  set deptinfo end=eofdept;
  length label $32;
  start = sorted;
  end = sorted;
  label = deptname;
  fmtname = 'convdept';
  output fdn;
```
I

```sas
if fdept then do;
  start .... label = 'ERROR';
  output ;
  end;
end;'!
```

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:

```sas
proc format cntlin=work.FDNM;
```

Once the format (or informal) has been created, it can be used to read the expense data with an informat statement, print the expense data using the format in a PROC PRINT, or apply the format to the SORTCD variable in the expense program giving the users the monthly expense and error reports required:

```sas
data intexp interr;
set espense5;
long deptname $25;
 deptname = put(sortcdpmvdept.);
if deptname = 'ERROR' then output ;
else output ;
run;
```

Using format from CNTLIN with PROC PRINT:

```sas
data intexp fnterr; set expenses;
length deptname $25;
 deptname = put(sortcd,convdept.);
if deptname = 'ERROR' or index(contains(deptname),"UNASSIGNED") gt 0) then output ;
else output ;
run;
```

### 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 which 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 and always unique situations 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 the "real world":

- 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

### Referencing a Data Set at Compile Time

It is often necessary to reference or capture a variable value from a data set at compile time:

```sas
data real; call SYMPUT('n_obs', put(n_obs, 5.)); stop;
set intdata nobs = 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 1: 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):

```
BOSTON bill 107000
BOSTON dave 40
BOSTON dave 40
BOSTON dave 40
BOSTON jean 58
BOSTON jean 58
BOSTON jean 58
BOSTON jean 58
```

A simple way to solve this problem uses the features of the SAS 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.

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

```data_droll keepfirst;
set citysale;
by city hour;
if first.hour then
do;
output droll;
delete;
end;
output keepfirst;
run;
```

Using first.hour would drop these:

```
BOSTON bill 107000
BOSTON dave 40
BOSTON jean 58
BOSTON jean 58
```

Using first.hour would keep these:

```
BOSTON dave 40
BOSTON dave 40
BOSTON dave 40
BOSTON jean 58
BOSTON jean 58
BOSTON jean 58
```

Real World 2: 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:

```
%name newlable 'c:\saspaper';
%name outlable '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=newlable;by note1 lastname name1;
run;
```

```proc print data=newlable;
  title 'SAS data .. "newlabel" barb';
  title2 'Envelope label';
  label;
  var
    name: dept10r notel note2;
run;
```

The resulting ASCII file is ready to bring into WORD (tab characters do not display):

```
NAME LOCATION
LORD-CRUFFY VANDEMEER 111511
ROBERT ALFIELE 504772
LEITFOLD BLOOM 1050099
HERMAN MELVILLE 902133
GEORGE OSWIND 496177
JANET KALE 1057097
MAX ANDERSON 465125
BILBO BAGGINS 902133
WRANGLE BARDON 476179
R.T. FRAZUILLA 1095037
SEANUS KYOS 805090
DONNA REED 705092
THOMAS A. BOTTLE 1057077
PAULINE TAME 953312
KIMBER APPOG 955356
```

Real World 3: 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.
Advanced Tutorials

If this is a 40 page story late for a publishing deadline: can:

- beg secretary to make the changes in WP package;
- make changes in the word processor yourself; or
- write using character manipulation functions.

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

```
I'll never see another lost like that. No more soldiers, no more blood.
She hadn't even 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 three tourists,
said to the tumbling sky. "What's the good of all these dead customs, anyway?"
```

(undamaged text deleted)

The simplest solution is a sort of the input data by ID and descending YR. This order allows the first.yr automatic variable to be the last year in the patient’s data. When first.yr is greater than or equal to 92, then a delete flag (DELFLAG) will be set. The code creates two data sets: KEEPIT and DELETEDT; however, in a production environment, it might only be necessary to use a subsetting IF (if delflag=0) to output only the desired observations:

```
proc sort data=yrinfo;
  by id descending yr;
run;
```

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

```
real world 5: 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>100</td>
<td>NET</td>
<td>TUNER</td>
<td>OPERATIONS</td>
<td>30</td>
</tr>
<tr>
<td>200</td>
<td>CFP</td>
<td>MILLER</td>
<td>SOFTWARE</td>
<td>25</td>
</tr>
<tr>
<td>300</td>
<td>CFP</td>
<td>MILLER</td>
<td>LAN/WAN</td>
<td>25</td>
</tr>
<tr>
<td>400</td>
<td>NET</td>
<td>MILLER</td>
<td>SOFTWARE</td>
<td>25</td>
</tr>
<tr>
<td>500</td>
<td>NET</td>
<td>MILLER</td>
<td>LAN/WAN</td>
<td>75</td>
</tr>
<tr>
<td>600</td>
<td>NET</td>
<td>MILLER</td>
<td>SOFTWARE</td>
<td>25</td>
</tr>
<tr>
<td>700</td>
<td>BUS</td>
<td>MILLER</td>
<td>SOFTWARE</td>
<td>75</td>
</tr>
<tr>
<td>800</td>
<td>CFP</td>
<td>MILLER</td>
<td>SOFTWARE</td>
<td>25</td>
</tr>
<tr>
<td>900</td>
<td>BUS</td>
<td>MILLER</td>
<td>SOFTWARE</td>
<td>75</td>
</tr>
<tr>
<td>1000</td>
<td>BUS</td>
<td>MILLER</td>
<td>SOFTWARE</td>
<td>25</td>
</tr>
</tbody>
</table>

To complete the tracking process, users want two reports: one, a summary by line of business and director showing the total number of minutes for each ticket and the number of areas affected; and a second 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;
run;
Subsequent processing uses the SUPRSORT variable to
produce the detail report and summary report file in one
data step:

proc sort data=sortexmp;
  by suprsort linebusn director area ticket;
run;
filename detail 'c:Waspaper\suprdetrapm';
data dirrotl(keep=linebusn director ticket numarea duration);
by suprsort;
retain dirrotl tnl8';
filename print;
if first.suprsort then do;
dirrotl=0;
umarea=0;
put @8 'detail listing by line of business';
put @ linebusn directotlarea ticket duration;
end;
dirrotl + duration;
umarea + 1;
put @ linebusn director area ticket duration;
if last.suprsort then output dirrotl;
run;
proc sort data=dirrotl;
  by linebusn director;
run;
proc print data=dirrotl;
  by linebusn;
sum dirrotl;
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 DIRTOTL is:

<table>
<thead>
<tr>
<th>OBS</th>
<th>DIRECTOR</th>
<th>TICKET</th>
<th>NUMAREA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JOHNS</td>
<td>436</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>MILLER</td>
<td>436</td>
<td>75</td>
</tr>
<tr>
<td>3</td>
<td>MILLER</td>
<td>565</td>
<td>60</td>
</tr>
<tr>
<td></td>
<td>LINEBUSN</td>
<td></td>
<td>160</td>
</tr>
</tbody>
</table>

And the detail report looks like:

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:

<table>
<thead>
<tr>
<th>Line of Business</th>
<th>APP</th>
<th>Call</th>
<th>Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUS</td>
<td>Call</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINEBUSN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BUS</td>
<td>Call</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MILLER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BUS</td>
<td>Call</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MILLER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINEBUSN</td>
<td></td>
<td></td>
<td></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:

Once the data is sorted by the coded variable and
appname, the NICENAME format can be applied to
substitute the long name for lines of business and
manipulate the order of presentation on the users' reports:

<table>
<thead>
<tr>
<th>Line of Business</th>
<th>APP</th>
<th>Call</th>
<th>Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUS</td>
<td>Call</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINEBUSN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BUS</td>
<td>Call</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MILLER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BUS</td>
<td>Call</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MILLER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINEBUSN</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(Line of Business=CORPORATE, etc., follow.)

Real World 7: MERGE vs. UPDATE
If a file only needs a few changes, why recreate the entire file just to make those changes? This scenario demonstrates the benefit of the UPDATE statement over the MERGE for some applications. The master file (MASTER) contains names, birthdays, gift ideas and other information:

<table>
<thead>
<tr>
<th>NAME</th>
<th>BDATE</th>
<th>SIZE</th>
<th>COLOR</th>
<th>INTEREST</th>
<th>WHAT</th>
<th>LIMIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>jody</td>
<td>03/22/84</td>
<td>4.5</td>
<td>pink</td>
<td>Goose Bumps</td>
<td>math</td>
<td>20</td>
</tr>
<tr>
<td>phil</td>
<td>10/24/83</td>
<td>5.0</td>
<td>green</td>
<td>Goose Bumps</td>
<td>math</td>
<td>20</td>
</tr>
<tr>
<td>mary</td>
<td>12/23/83</td>
<td>3.5</td>
<td>green</td>
<td>Goose Bumps</td>
<td>math</td>
<td>100</td>
</tr>
<tr>
<td>marie</td>
<td>02/14/84</td>
<td>4.5</td>
<td>green</td>
<td>Goose Bumps</td>
<td>math</td>
<td>90</td>
</tr>
<tr>
<td>susan</td>
<td>11/13/84</td>
<td>5.0</td>
<td>green</td>
<td>Goose Bumps</td>
<td>math</td>
<td>5</td>
</tr>
</tbody>
</table>

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 BDATE:

```
data newmstr2;
merge master(i=imast) updttrns(in=intrn);
by name bdate;
if (inmast and intrn) or (imast and not intrn) then output newmstr2;
if intrn 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:

<table>
<thead>
<tr>
<th>NAME</th>
<th>BDATE</th>
<th>SIZE</th>
<th>COLOR</th>
<th>INTEREST</th>
<th>WHAT</th>
<th>LIMIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>jody</td>
<td>03/22/84</td>
<td>4.5</td>
<td>pink</td>
<td>Goose Bumps</td>
<td>math</td>
<td>20</td>
</tr>
<tr>
<td>phil</td>
<td>10/24/83</td>
<td>5.0</td>
<td>green</td>
<td>Goose Bumps</td>
<td>math</td>
<td>20</td>
</tr>
<tr>
<td>may</td>
<td>10/23/83</td>
<td>3.5</td>
<td>green</td>
<td>Goose Bumps</td>
<td>math</td>
<td>100</td>
</tr>
<tr>
<td>marie</td>
<td>02/14/84</td>
<td>4.5</td>
<td>green</td>
<td>Goose Bumps</td>
<td>math</td>
<td>90</td>
</tr>
<tr>
<td>susan</td>
<td>11/13/84</td>
<td>5.0</td>
<td>green</td>
<td>Goose Bumps</td>
<td>math</td>
<td>5</td>
</tr>
</tbody>
</table>

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;
input name $ bdate mmddyy8.;
input interest $15 limit $8;
missing _;
cards;
run;
```

The UPDATE statement produces the desired result:

```
data newmstr;
update master updtttrns;
by name bdate;
run;
```

The updated data set after using the UPDATE statement:

<table>
<thead>
<tr>
<th>NAME</th>
<th>BDATE</th>
<th>SIZE</th>
<th>COLOR</th>
<th>INTEREST</th>
<th>WHAT</th>
<th>LIMIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>jody</td>
<td>03/22/84</td>
<td>4.5</td>
<td>pink</td>
<td>Goose Bumps</td>
<td>math</td>
<td>20</td>
</tr>
<tr>
<td>phil</td>
<td>10/24/83</td>
<td>5.0</td>
<td>green</td>
<td>Goose Bumps</td>
<td>math</td>
<td>20</td>
</tr>
<tr>
<td>may</td>
<td>10/23/83</td>
<td>3.5</td>
<td>green</td>
<td>Goose Bumps</td>
<td>math</td>
<td>100</td>
</tr>
<tr>
<td>marie</td>
<td>02/14/84</td>
<td>4.5</td>
<td>green</td>
<td>Goose Bumps</td>
<td>math</td>
<td>90</td>
</tr>
<tr>
<td>susan</td>
<td>11/13/84</td>
<td>5.0</td>
<td>green</td>
<td>Goose Bumps</td>
<td>math</td>
<td>5</td>
</tr>
</tbody>
</table>

References