WHAT IS MISSING DATA?

When SAS® programmers discuss missing data, they are most often talking about numeric SAS missing values or blank character values. In particular, numeric missing values are valuable in SAS applications since they allow us to record data points that have no value at all — not zero and certainly not -1 or 99999999, commonly coded values that represent missing data in other environments.

However, the use of SAS missing values (including blank character values) does not cover all situations when data are missing or incomplete. For instance, empty cells in a table produced by PROC TABULATE or null macro variable values. In addition, there are subtle issues where information appears to be absent, such as leading blanks in character strings, which must be handled carefully.

The components of the SAS System provide options and techniques which permit us to: (1) detect and avoid undesirable missing data; (2) treat missing data as acceptable within the scope of an application; and (3) take advantage of missing data in special circumstances. This paper summarizes some considerations for dealing with a number of situations that involve missing data.

MISSING DATA IN EXTERNAL FILES (RAW DATA)

USE OF THE $CHAR. INFORMAT

Two features of the default informat for character data ($w.) are the conversion of a single period to a blank and the removal of leading blanks. To counteract these actions, the $CHAR. informat can be used.

Reading character values that are frequently missing can require the use of a placeholder character in the raw data. The default result with no informat is quite appropriate here:

```
DATA DOCUMENT:
  INPUT COL1 $CHAR. @;
  IF COL1='.' THEN INPUT 62 COMMAND $;
  ELSE INPUT 61 TEXT $CHAR40.:
DATALINES:
  figure

RUN;
```

The previous example also illustrates the use of the $CHAR. informat to retain leading blanks. Using the $40. informat would have left-justified the value of the variable TEXT.

This situation is best illustrated by the use of a coded field with values A, B, C, D, ..., X, Y, Z, AA, AB, AC, AD, etc., where the two-digit values must follow "Z" when the values are sorted. This requires the single-digit values to contain a leading blank; without a leading blank, AA would be positioned between A and B. If single-digit values were read using the $2. informat, the leading blank would be dropped. Using the $CHAR2. informat retains the leading blank.

```
DATA CODES:
  INPUT CODE $CHAR2.;
DATALINES;
  A
  C
  Z
RUN;
```

INVALID DATA CONVERTED TO MISSING VALUES

When reading numeric data, invalid numeric values cause the resulting variable to be assigned a missing value and trigger the "Invalid Data..." message in the SAS Log. This includes cases where alphabetic characters appear in the input stream or when numbers containing commas (or dollar signs) are read without using the COMMA. informat. Invalid raw data values can also generate missing values. For example, reading the string 11/31/93 with the MMDDYY8. informat results in a missing value since November 31 is an invalid date.

Unfortunately, these situations can be difficult to correct since the invalid string is not highlighted in the SAS Log. There is only a reference to the raw data line and the columns generating the problem. To write out a note displaying the actual invalid string, we can: (1) read the field as both character and numeric, detecting when the numeric value is missing but the character value is not missing [code shown below]; or (2) read the field as character and use the INPUT function to convert it to numeric, detecting missing value results from the INPUT function.

```
DATA DOCUMENT:
  INPUT COL1 $CHAR. @;
  IF COL1='.' THEN INPUT 62 COMMAND $;
  ELSE INPUT 61 TEXT $CHAR40.:
DATALINES:
  figure

RUN;
```

On the other hand, if each data line was distinguished by the presence or absence of a period in column 1, the $CHAR. informat would be necessary to maintain the "." value:
Efficient methods to perform checks on many different fields can be implemented using ARRAY processing in the DATA step.

If invalid data are anticipated, but the messages in the SAS Log are undesirable, adding a format modifier after the variable name in the INPUT statement may be helpful. The "?" modifier only suppresses the invalid data message. The "#" modifier also suppresses the printing of the input line and prevents the automatic _ERROR_ variable from being set to 1.

It is possible to override the default missing value assigned when invalid numeric data are encountered. The INVALIDDATA= system option allows the user to select the missing value used to represent invalid numeric data generated through the INPUT statement or an INPUT function. The default value for the option is INVALIDDATA='.' resulting in standard missing values. Assigning a different value generates the corresponding special missing value. For example, INVALIDDATA='B' produces .B values whenever invalid ("bad") data are encountered.

**Special Missing Values**

Special numeric SAS missing values are indicated with a single letter (or underscore) immediately following the "." character. They can be utilized not only for invalid data, but within applications that require distinction between different classifications of missing data. For instance, a survey question based on a 1-to-5 scale may also have non-responses which fall into categories such as "Refused", "Not Applicable", "Don't Know", "Invalid Response" and "No Answer".

Instead of storing the responses in character variables, which would permit coding these non-responses as "R", "N", "D", "I", and blank but would make numeric summarization (e.g., averages) more difficult, we can store non-responses as special numeric missing values such as .R, .N, .D, and .I in addition to the standard missing value (.). This is done by introducing a MISSING statement prior to reading the data with an INPUT statement:

```
MISSING R N D I;
```

The MISSING statement acts as a global statement in the same manner as a TITLE statement. The assignment of valid letters as special missing values remains in effect for the remainder of the program.

Special missing values are treated in the same manner as standard SAS missing values when producing summary statistics — they are ignored in all statistics except NMISS. When special missing values are printed by SAS procedures, the leading "." is suppressed automatically.

**The MISSOVER and STOPOVER Options**

By default, when an INPUT statement reaches the end of a raw data line and additional variables are to be read, the DATA step attempts to read the remaining variables from the next data line. This default action is called FLOWOVER mode.

When a pre-determined number of data values is expected on each raw data line, we can detect situations where a data line is incomplete by adding a MISSOVER or STOPOVER option to the INFILE statement. When fewer values are present on a data line than anticipated by the INPUT statement, these options trigger special DATA step routines.

The MISSOVER option permits processing to continue by setting all "leftover" fields on the INPUT statement to missing values. The STOPOVER option halts the DATA step immediately and generates an error message. For example, the statement

```
DATA SURVEY;
    INPUT (QUEST1-QUEST10) (L:);
    DATALINES:
    11122023 11/31/93 234.10 /* should be 234.10 */
    2/29/94 none
    RUN;
```

The MISSING statement acts as a global statement in the same manner as a TITLE statement. The assignment of valid letters as special missing values remains in effect for the remainder of the program.

**Calculations Involving Missing Values**

**Arithmetic Operations**

One of the most problematic situations involving missing data is the arithmetic calculation in which one or more of the operands contains a missing value or in which non-missing operands produce a missing value.

An example of the latter case is division by zero. The result cannot be expressed as a number, so a missing value is assigned. In previous versions of the SAS System, division by zero was a fairly painless situation. The missing value result is just as desirable now, but reports show that this default action can cause excessive CPU usage. It is recommended that any division operation that has the possibility of a "division by zero" condition be checked for a zero denominator first. If the denominator is
zero, explicitly set the result to missing; if not, proceed with the division.

Lack of understanding how missing values are handled in simple arithmetic expressions has introduced many bugs into SAS applications. Many of these bugs could have been avoided if the programmer had paid attention to "Missing Values Generated..." messages in the SAS Log during development. Occasionally, however, even thorough testing that we may consider "thorough" does not uncover the problem, particularly if all test data are non-missing.

The rule concerning missing values in arithmetic expressions is simple: if any values in the expression are missing, the result of the entire expression is missing. This is known as propagation of missing values. So, if A is 1 and B is missing, the following expressions are all missing: A+B, B+A, A-B, B-A, AxB, BxA, A+B, B+A.

DATA STEP FUNCTIONS

Some applications are set up such that a missing value should be treated as a zero within arithmetic expressions. To help resolve this category of problems, we can use DATA step functions such as SUM and MEAN. These functions ignore missing values when calculating numerical results. Therefore, while A+B is missing in our earlier example, SUM(A,B) is equal to 1.

Suppose that an observation in our database contains credit card account data for a year with monthly charges stored in 12 variables and payments stored in 12 other variables. Lack of activity is stored as a missing value. To obtain the total charges for January ($100) and February (missing, i.e., no charges), we could try a simple arithmetic operation, CHARGJAN+CHARGFEB, which gives us a missing value. Using a DATA step function, in this case, the SUM function, solves the problem:

\[ \text{SUM}(\text{CHARGJAN, CHARGFEB}) = 100 \]

Similarly, we can use the SUM function to perform subtraction, as in calculating a monthly change in outstanding balance, that is, charges minus payments. Suppose there was no payment made in January. The arithmetic expression CHARGJAN-PAIDJAN results in a missing value, while SUM(CHARGJAN-PAIDJAN) yields 100. [Note: We still get the "Missing values..." message due to the -PAIDJAN operation that gives us a missing value as the second argument to the SUM function.]

Other DATA step functions actually generate missing values for certain argument ranges. Examples include the square root of a negative value and the logarithm of zero. Character functions can also generate missing values; for instance, the SCAN function produces a missing (blank) value when we search for the \( n \)th word in a string when only \( n-1 \) words exist.

RETAILING COMPUTED VALUES

When a new variable is created in a DATA step, the default action is to initialize the variable to a missing value at the start of each DATA step iteration. This automatic re-initialization can be disabled by declaring a variable in a RETAIN statement. The variable will maintain its value until explicitly re-assigned by a DATA step statement. The variable's value at the start of DATA step execution can be assigned in the RETAIN statement; the default starting value is missing.

One common application using the RETAIN statement is the accumulation of a total. The code

\[
\text{RETAIN TOTAL 0;}
\]
\[
\text{TOTAL=TOTAL+DETAIL;}
\]

would accumulate a total of the values of DETAIL as long as DETAIL never had a missing value.

Instead of using a standard assignment statement, the SUM statement performs the equivalent function. However, missing values are treated as zeros by the SUM statement. In fact, when a SUM statement is present, an implied RETAIN statement with an initial value of 0 is automatically generated for the corresponding variable.

\[
\text{TOTAL+DETAIL;}
\]

MISSING VALUES IN LOGICAL EXPRESSIONS

DETECTING MISSING VALUES

How can you tell if a set of one or more variables contains a missing value? The straightforward method would resemble

\[
\text{IF A=. OR B=. OR C=. OR D=. OR ...}
\]

Not so fast! What if we had used special missing values in the variables being checked? The standard missing value . won't match any of the special missing values. Since .Z is the "largest" special missing value, the new expression is

\[
\text{IF A <= .Z OR B <= .Z OR C <= .Z OR ...}
\]

But there is a DATA step function which can detect any missing value — the NMISS function. The expression

\[
\text{NMISS(A, B, C, D, ...)}
\]

will generate the number of standard and special missing values contained in the variables listed. In fact, to detect
whether a single numeric variable has any type of missing value, you could just use the expression NMISS(variable)=1.

MISSING VALUES IN IF EXPRESSIONS

If missing values have troublesome side effects in the simplest of arithmetic expressions, imagine the havoc they can cause in logical expressions like those used in the IF statement. In this instance, another of the less-recognized properties of the SAS missing value becomes painfully apparent — missing values are treated as smaller than any numeric value. In a sense, a missing value is equivalent to , even smaller than any negative number imaginable.

This feature's impact on IF-expressions is now clearer. An IF-expression such as \((X < 0)\) is true not only when \(B\) is negative but also when \(X\) is missing. Therefore, the basic IF-THEN-ELSE IF logic that performs different functions based on the "sign" of an expression must be enhanced. Once again, we check for any missing value by checking for values less than the special missing value .Z:

\[
\text{IF } X > 0 \text{ THEN } \ldots ; \\
\text{ELSE IF } X = 0 \text{ THEN } \ldots ; \\
\text{ELSE } X <= .Z \text{ THEN } \ldots ; \\
\text{ELSE } \ldots ;
\]

VARIABLE TYPE CONVERSIONS

CHARACTER TO NUMERIC

As mentioned earlier, using the INPUT function to convert character strings to numeric values results in the assignment of a missing value when the character string does not represent a valid value based on the informat specified. This concept is also true for implicit character-to-numeric conversions of standard numeric-like strings.

NUMERIC TO CHARACTER

Sometimes our "laziness" in letting the SAS System do so much work behind the scenes gets us in trouble. Implicit numeric-to-character conversions, those performed without an explicit PUT function, are a major problem in this area. The rule to remember — a rule that many SAS programmers, even very experienced ones, are totally unaware of — is that implicit numeric-to-character conversions always use the BESTn. format.

To illustrate this point, let's create a new character variable to contain the numeric value 1 using the statements:

```
LENGTH CHARVAR $ 8;
CHARVAR=1;
```

The value of CHARVAR is not "1", it is " 1". The result has seven leading blanks because the implicit conversion uses the BEST8. format (since the length of the character variable is 8). And the BEST8. format right-justifies its result.

How does this relate to missing values? Consider the treatment received by numeric values when used with functions that expect character operands (such as SUBSTR). These situations also perform implicit conversion using the BEST. format, but the width is 12 (i.e., the BEST12. format). Thus, the resulting text string will be 12 digits long and right-justified with leading blanks.

Suppose we wish to check the magnitude of the numeric variable NUMBER that ranges from 100 to 999. You attempt to create the variable HUNDREDS by obtaining the first digit of NUMBER:

```
HUNDREDS=SUBSTR(NUMBER, 1, 1);
```

The result is always a blank, i.e., a character missing value. The SUBSTR function converts NUMBER to a 12-digit string with nine leading blanks. You obtain the first digit — which is always a blank. No error message ... no warnings ... no unexpected notes in the SAS Log. But the results — all missing values — are incorrect.

Another common occurrence of this problem deals with the storing of values into macro variables using the SYMPUT routine. The simple statement

```
CALL SYMPUT('VAR',[_N_, 'XYZ']);
```

will not generate macro variables named VAR1, VAR2, etc., because the leading blanks from the implicit conversion of _N_ to a character string are inserted after the literal string 'VAR'. One way to solve this type of problem is to left-justify the converted string before performing the concatenation:

```
CALL SYMPUT('VAR',LEFTCH(PUT(_N_,S.),'XY'));
```

Even an explicit conversion would require left-justification:

```
CALL SYMPUT('VAR',LEFTCH(PUT(_N_,5),,'XY'));
```

Yet another problem case is the merging of data sets by a "numeric" BY-variable when the variable is stored as a character field in one of the data sets. Converting the character version to numeric and then merging typically clears up the discrepancy. Converting the numeric version to character, however, requires more care since the original character version may or may not have leading blanks. In addition, it is important to specify the proper length for the converted character value, particularly when each of the matching values will be right-aligned.
USING UPDATE TO LOAD MISSING VALUES

The power of the UPDATE statement to modify data values in a SAS data set can occasionally be compromised when missing values must be loaded to replace non-missing values. The only way to accomplish this using the UPDATE statement is to convert standard numeric missing values in the transaction data set to special "_" missing values and to convert blank character values to a single underscore character ("_"). The "_" and "_" values will "erase" the corresponding non-missing numeric and character data.

```sas
DATA NEWDATA2;
  SET NEWDATA;
  IF NEWCHAR=' ' THEN NEWCHAR='_';
  IF NEWNUM=. THEN NEWNUM='_';
  DATA UPDATED;
  UPDATE MASTER NEWDATA2;
  BY UPDVAR;
```

DISPLAYING MISSING VALUES IN PROCEDURES

There are many ways to display missing values in SAS procedure output. Many procedures have special options which permit the user to select the mode in which missing values are to be handled. In other cases, SAS system options must be set or reset.

**PROC PRINT**

As indicated earlier, the PRINT procedure will display special missing values as the alphabetic portion only, i.e., the value .X is displayed as simply X with the leading "." suppressed.

On the other hand, standard missing values are displayed based on the value of the MISSING= system option. The default value for this option is MISSING=":" To display each standard missing value as a question mark (?) instead of a period, the statement

```
OPTIONS MISSING='?';
```

would be inserted prior to the appropriate PROC PRINT step. The MISSING='?' option would stay in effect until another MISSING= option was specified within the program. [Do not confuse the MISSING= system option and the MISSING statement discussed earlier.]

**PROC FORMAT**

When specifying ranges in user-defined formats it is important to remember that missing values, standard or special, are treated no differently than non-missing values, particularly with respect to the "OTHER" range. It is possible, and frequently desirable or necessary, to explicitly list missing values in ranges.

```sas
PROC FORMAT;
  VALUE NUMFMT
     .='MISSING'
     OTHER='NON-MISSING'
  VALUE $CHARFMT
     .='MISSING'
     OTHER='NON-MISSING'
;
```

**PROC MEANS/SUMMARY/TABULATE**

In procedures that utilize classification variables (MEANS, SUMMARY, TABULATE), observations with a missing value for a CLASS variable are ignored. To permit missing values to be treated as valid classes, the MISSING option must be added to the PROC statement.

```sas
PROC MEANS DATA=DETAILS MISSING;
  CLASS GROUP; /* has missing values */
  VAR NUMBER;
```

**PROC FREQ**

When generating frequency tables with the FREQ procedure, missing values are, by default, summarized with a message indicating "Frequency Missing = " at the bottom of the table. Percentages are based solely on non-missing values.

To include the frequency count for missing values within the body of the table, add the MISSPRINT option to the TABLES statement. To include missing values in the calculation of percentages as well, use the MISSING option instead. [The MISSING option implies MISSPRINT.]

An effective application of the MISSING option is the summarization of a data set with many variables. If the only information desired is the percentage of missing values in each variable, we can combine the MISSING option with the NUMFMT and $CHARFMT formats defined earlier to produce a concise output report. [Note the use of the keywords _ALL_, _NUMERIC_ and _CHARACTER_ to represent the corresponding set of variables.]

```sas
PROC FREQ;
  TABLES ALL / MISSING;
  FORMAT _NUMERIC NUMFMT;
  _CHARACTER $CHARFMT;
```

DISPLAYING MISSING DATA IN PROCEDURES

The concept of missing data is best distinguished from missing values when discussing empty cells in cross-tabulations. An empty cell indicates that there are no records that correspond to that combination of classification values. It is frequently important to differentiate the existence of missing values from the lack of any data at all.

**PROC TABULATE**

In tables produced by PROC TABULATE, empty cells are filled with a period (.), the standard SAS missing
value character. A different text string can be displayed in all empty cells by using the MISSTEXT option on the TABLE statement. When a table consists of exclusively frequency counts, the appropriate text to display in empty cells is the digit "zero" — MISSTEXT='0'. In other cases, an explicit indicator of lack of data is more appropriate — MISSTEXT='None'.

PROC TABULATE DATA=TABDATA;
CLASS GROUP1 GROUP2;
TABLE GROUP1, GROUP2 / MISSTEXT='0';

PROC FREQ

When the LIST option is used in PROC FREQ or when an output data set of frequency counts is produced, cross-tabulation combinations that do not contain data are suppressed. If the existence of empty cells is important in these circumstances, adding the SPARSE option to the TABLES statement will generate lines/observations representing all possible combinations. Empty cells can be identified by a frequency count of zero.

PROC FREQ DATA=TABDATA;
TABLES GROUP1 * GROUP2 / OUT=FREQS SPARSE;

SELECTING OR OMITTING MISSING VALUES

WHERE STATEMENT

The WHERE statement is extremely useful in subsetting a SAS data set, within either a DATA step or a PROC step. One restriction to the WHERE statement is the requirement that the type of subsetting variable must match the type of the subsetting value(s). If there is a type mismatch (numeric/character, character/numeric), an error occurs. A type mismatch most commonly occurs in generalized, user-driven systems in which the programmer cannot fully anticipate what variable(s) may be specified by the user for subsetting purposes.

To overcome this limitation in the DATA step, the WHERE statement can be replaced by an equivalent IF statement. This can be less efficient, but the code will typically run successfully, even considering the less-desirable implicit type conversions performed during DATA step execution. Unfortunately, there is no IF statement for use with PROC steps.

There is, however, a more robust alternative for the WHERE statement when the subsetting involves missing values but the subset variable's type is uncertain. Instead of requiring

WHERE NUMVAR = . ;

for numeric variables or

WHERE CHARVAR = ' ' ;

for character variables, we can use the general form

WHERE VARIABLE IS MISSING ;

instead. The equivalent phrasing for detecting non-missing values is "IS NOT MISSING".

PROC REPORT

In a similar situation, when output from the REPORT procedure is based on a subset of observations containing missing values specified "blindly" by an end-user, it is possible that one or more columns with nothing but missing values will appear on the report. To suppress a column whenever every value is missing (or zero), add the NOZEROS option to the associated DISPLAY statement.

DATA ENTRY IN SAS/FSP® OR SAS/AF® SOFTWARE

PAD ATTRIBUTE

For each variable on a SAS/FSP screen or SAS/AF display, there is a PAD attribute that determines what character will indicate a valid space for entry of a value for that variable. The default PAD character is the underscore (_). Therefore, a data entry screen in SAS/FSP or SAS/AF resembles a fill-in-the-blank application.

Problems arise when the value to be entered contains an underscore. Obvious examples in the SAS environment are data set names or variable names. When the underscore character is initially typed in, it cannot be distinguished from the PAD character so the system "erases" the embedded underscore, converting it to a blank just as it converts any trailing PAD character underscores to trailing blanks. If the underscore was in the first position, the remaining value is left-aligned.

This inconvenience can be avoided by re-defining the PAD character for the variable(s) in question. Typically, the PAD character should be changed to a blank or another character which cannot be part of the keyed string. Then, when an underscore is typed in, it remains in place and does not "disappear".

PREDICTED VALUES IN SAS/STAT® SOFTWARE

Frequently we wish to extrapolate or estimate unknown information from known information, particularly in statistical applications. SAS/STAT software has several tools which allow us to model a problem based on known information, then use the model to prepare an estimate for other cases. Some of these techniques require several passes of data or the invocation of multiple procedures. Other methods can perform the same or similar tasks within one execution of a single procedure.
PROC REG

Since a regression model can be defined based on observations with known dependent variable values and known independent variable values, fitting the model to observations with unknown dependent values can still be accomplished as long as all required independent variables have values. This is exactly what happens when a data set with missing dependent values is processed by the REG procedure.

There are several occasions when this feature is helpful. If we must obtain predicted values for observations that have a dependent value and those that do not have a dependent value, we do not have to estimate the model parameters using PROC REG, then use those parameters in another procedure such as PROC SCORE. We can also remove observations from the parameter estimation step by setting the dependent variable to missing without deleting those records from the data set, even enabling us to produce predicted values for those records based on all other records. This can be especially valuable when dealing with potential influence points.

```sql
DATA PREDICTING 1993 AFTER THE FACT;
SET ALLYEARS;
IF YEAR=1993 THEN DEPVAR=.;
PROC REG DATA=ERASE93;
MODEL DEPVAR=DpBPVAR;
OUTPUT OUT=POST_BOC P=PREDICT;
PROC DISCRIM

Like regression models, discriminant models are generated based on known values (independent variables) in an attempt to divide observations into known classes (dependent variable). While it is possible to create a separate TESTDATA= data set with unknown classifications, all observations may remain in a single input data set, some with known classes, some with unknown classes.

NULL MACRO VARIABLES

Unlike DATA step variables which can be either numeric or character, macro variables are handled exclusively as character strings. One important difference between DATA step character variables and macro variables is the definition of length. The length of a DATA step variable is identical on each observation. A macro variable, on the other hand, has a dynamic length, that is, its length changes based on its value.

Since a DATA step character variable has a minimum length of 1, a missing character value is represented by a blank. If a macro variable contains a blank character as its value, it would also have a length of 1. But a macro variable can have a length of zero, so a new concept is required — the null macro variable. A null macro variable exists, but has no value — not blank, not zero, just no value.

INITIALIZATION AND ASSIGNMENT

When a macro variable is added to the macro symbol table, through a %GLOBAL statement, a %LOCAL statement, or as a positional macro parameter, a "slot" is assigned in the table for the macro variable's name, but no value is assigned at that time. In the case of a positional parameter, if no value is specified in the macro call, the parameter retains its null value.

It is also possible to assign a null value to a macro variable directly. The simplest way is to use a %LET statement:

```sql
%LET MACVAR = ;
```

The %LET statement automatically trims all leading and trailing blanks from the assigned value, so the number of blanks between the equal sign and the semi-colon does not matter; ten blanks is the same as one blank, and one blank is the same as no blanks.

MACRO FUNCTION RESULTS

Another way to create a null value is as the result of a macro function. Just as the DATA step SCAN function produces a missing value when searching for the n-th word when only n-1 words exist, the macro function %SCAN generates a null value in the same situation.

CONDITIONAL EXECUTION

One common use of macro variables, including parameters, in macro programs is testing for conditional processing. When one is trying to detect a null value, the resulting %IF condition looks unbalanced.

```sql
%IF expression% THEN ....
```

In a DATA step, an IF condition must be balanced, with a value on each side of the operator, since even a missing value has an explicit representation, either a blank or a period. In the macro facility, the only way to test for a null value is to ask "Is the expression equal to nothing?" So, although it may look to the inexperienced macro programmer as if there is something missing in the %IF statement, we are really taking full advantage of the null value feature.

RECOVERING SYMBOL TABLE STORAGE SPACE

One way to maximize the utility of null values is to recover valuable storage space in macro symbol tables after a macro variable is no longer needed. Unlike DATA step variables, macro variables cannot be "dropped"; once added to a symbol table, a macro variable cannot be deleted from the table. Therefore, an effective way to free
up space in a symbol table is to reset all unneeded macro variables to null values. This is especially true for macro variables that may have contained very long values. A simple %LET statement will do the trick. If the SYMPUT routine must be used, refer to the section below dealing with blank macro variables and CALL SYMPUT.

BLANK MACRO VARIABLES

%STR Function

As noted earlier, the %LET statement automatically trims all leading and trailing blanks when assigning a value to a macro variable. To store one or more blanks or to retain leading and/or trailing blanks in the stored value, a macro quoting function is required, namely, the %STR function.

%LET BLANK = %STR( );
%LET LEADING = %STR( Before );
%LET TRAILING = %STR( After );

To test a macro variable created by the SYMPUT routine to see if it contains a blank value, the %IF condition may require multiple quoting functions to distinguish between blanks and null values during macro execution:

%IF %QUOTE( 'TESTVAR' ) = %STR( ) THEN ... 

When using a blank as an argument to a macro function, it is also necessary to use a quoting function. For example, to search for a blank in a macro expression:

%INDEX( MACVALUE, %STR( ) )

or to use the blank character as the only delimiter in the %SCAN function:

%SCAN( STRING, 2, %STR( ) )

CALL SYMPUT

Instead of using macro quoting functions within a %LET statement, blank values and values with leading and/or trailing blanks can be assigned to macro variables using the DATA step SYMPUT routine. Unlike %LET, CALL SYMPUT does not automatically trim blanks.

CALL SYMPUT(' BLANK ', ' ');
CALL SYMPUT(' LEADING ', ' Before ');
CALL SYMPUT(' TRAILING ', ' After ');

Although this feature of the SYMPUT routine is frequently useful, it can also be inconvenient, particularly when leading and/or trailing blanks are not desired. The DATA step functions LEFT and TRIM are commonly used in conjunction with CALL SYMPUT, especially in the second argument, to remove unwanted leading and trailing blanks. Watch out for leading blanks when storing numeric values in macro variables; the earlier section on implicit numeric-to-character conversions using the BESTn. format illustrates the potential problems.

CONCLUSION

Many of the difficulties we encounter as SAS programming professionals are directly related to the completeness of our underlying data, or lack thereof. There are times when we must try to detect the potential for missing data and take action to avoid or circumvent it. There are other times when missing data are inevitable or expected given the nature of an application. In some cases, we can even take advantage of the special characteristics of missing data to make our applications complete, even sophisticated.

It is impossible in a paper of limited length to discuss all of the aspects of missing data within the SAS software environment. It is even difficult to explain in sufficient detail the varied topics covered by this paper. The extensive documentation published by SAS Institute, including the list of references at the end of this paper, describes these and other related topics in much greater detail, showing additional examples and uses of the described techniques. When the need arises for a better understanding of how our programs work given the limitations or special properties of our data, these publications will provide much guidance; they are resources that should be fully utilized.

ABOUT THE AUTHOR

Warren Repole is a Senior Technical Consultant for CSC Professional Services Group in Rockville, MD. He has used SAS software since 1979. He may be contacted at:

CSC Professional Services Group
1301 Piccard Drive, 2nd floor
Rockville, MD 20850
voice: 301/258-5300
fax: 301/258-6897

REFERENCES


ACKNOWLEDGEMENTS

Special thanks to Dr. Doug Cho for his assistance with the preparation of the interactive presentation of this material.

SAS, SAS/AF, SAS/FSP, and SAS/STAT are registered trademarks of SAS Institute Inc., Cary, NC, USA.