DATA Step in Version 7: What's New?

William F. Heffner, SAS Institute Inc., Cary, NC

INTRODUCTION

This paper describes some enhancements in the DATA step and its associated environment in Version 7. These enhancements include new rules for names, longer character variables, new functionality for the FILE and INFILE statements, new SAS® file I/O features, new extensions to the DATA step language (including new functions), and some performance improvements.

NAME SPACE AND VARIABLE ATTRIBUTE ENHANCEMENTS

Names

The most visible (and most requested) change in Version 7 involves SAS names. By default, the maximum length for many SAS names has been increased to 32 (from 8 in previous versions). Specific to the DATA step, variable names (including array names), window names, and statement labels may now consist of up to 32 characters. The 'rules' for forming these names (except for the length) have not changed from Version 6. The first character must be a letter (A, B, C, …, Z) or an underscore (_), and subsequent characters can be letters, digits (0, 1, …, 9), or underscores.

Version 7 supports mixed (upper and lower) case names by default. The SAS System will store these names exactly as first given, without uppercasing. This differs from previous versions where names were always normalized (uppercased). Note, however, that even though Version 7 will store these names in mixed case, they will be processed in a case insensitive manner (i.e., as if they were normalized, or uppercased). Names whose spellings differ only in the case of the characters are considered to be the same name. The following names would all reference the same variable.

```
A_DOG_CHASES_A_CAT
A_Dog_Chases_A_Cat
a_dog_chases_a_cat
```

The SAS System will validate these SAS names based upon the value of the new option VALIDVARNAME=. There are four valid values (V7, UPCASE, V6, ANY) for this option to specify applicable name validation rules. The default setting is VALIDVARNAME=V7, and gives the behavior described above.

Setting the VALIDVARNAME option to UPCASE indicates that longer V7 names are allowed, but the names will be uppercased (as in Version 6) by the SAS System upon entry. To restrict these names even further, VALIDVARNAME=V6 may be used. This causes the Version 6 rules to be used (maximum of 8 characters and names are uppercased).

The last accepted value, ANY, is an experimental feature of VALIDVARNAME=, and will only work with the DATA step and the SQL procedure. This setting allows V7 names (32 characters, mixed case), but will additionally allow any character in any position in a name. This includes blanks, punctuation, arithmetic operators, and any other special characters. Use of these special characters in a name, however, requires a special delimiter, so that the SAS System can parse it correctly. That special delimiter consists of quotes (single or double) and the letter 'N' (or 'n'). The following example is valid when VALIDVARNAME=ANY and produces a data set with 3 variables ('A dog chases a cat', '3+3', and 'normvar_6').

```
data strange;
  'A dog chases a cat'n = 5.5;
  "3+3"n = 'Six';
  normvar_6 = '3+3'n;
run;
```

Other, more global, names in the SAS System also have new attributes. Member names of SAS data libraries (e.g., SAS data files, SAS data views, catalogs, and indexes) all have the new maximum limit of 32 characters. Whether these names can support mixed case is dependent upon the host platform or the underlying engine. Also dependent upon the host platform, catalog entry names may also be up to 32 characters and support mixed case. The SAS macro language also supports the new maximum limit. This includes macro names, macro variable and window names, and macro statement labels.

The maximum length for SAS file librefs, external I/O filerefs, passwords, formats, and informats all remain unchanged (8, except for informats, which is 7). Function and CALL routine name lengths remain at 16.

The maximum length for descriptive labels has also been increased in Version 7. The limit is now 256 characters, raised from the previous maximum of 40. DATA step variable labels (assigned with the LABEL or ATTRIB statement), as well as SAS data set labels (assigned via the LABEL= data set option), are affected.

Character Variables

In previous versions of the SAS System, no more than 200 characters could be contained in character variables. This limit has been increased in Version 7 to 32,767 (32K - 1) characters. All language constructs which explicitly specify lengths for character variables will allow this length. This includes statements (LENGTH, ATTRIB, and ARRAY), format and informat specifications, and any statements which bind character variables to formats or informats (FORMAT, INFORMAT, PUT, and INPUT).

Length defaulting for implicitly defined character variables remains as in Version 6. If no length information is available upon the first variable reference, the length is 8. Target variables of assignment statements will default their length based on the length of the source expression. In the following example, the variables *one*, *two*, *three*, and *four* have lengths of 3, 8, 753, and 32,000, respectively.

```
data _null_;
  length a $ 3  d $ 32000;
  format c $char750.;
  set work.indata (keep a c d);
  one = a;
  input two $;
  three = c || a;
  four = compress( d );
```

```
datalines;
…
;
```

The variable *one* is defaulted to length 3 because it is the target of an assignment where the source expression (the variable *a*) is length 3. The variable *two* takes the default of 8, because there is no length information when the variable is first seen. The variable *three* is given length 753, which matches the source expression (length of *c*, 750, plus length of *a*, 3). And, because the function COMPRESS is defined to return a result variable the same length as the first argument, the variable *four* takes the same length as *d*, 32,000.

Certain functions (REPEAT and RESOLVE, for example) are defined to return result values whose length is the largest allowed. Their results are, basically, unbounded. In Version 7, these functions could return values with lengths of 32,767. To avoid defaulting character variables in older programs to this maximum length, the DATA step will not default target user variables of these type of functions larger than 200. Even though REPEAT could create a very large string, the result target variable will be implicitly defined with a length of 200. You can override this default by defining the variable with a LENGTH statement earlier in the DATA step program. In the following example, the variable *default200* will default to a length of 200. This will truncate the result of the REPEAT function, assuming the result is greater than 200 characters. The length of *length500* will be 500, of course.

```
data _null_;
  length length500 $500;
  input n 6. string $80.;
  default200 = repeat( string, n );
  length500 = repeat( string, n );
datalines;
…
;
```

If OPTION MSGLEVEL = I has been set (N is the default), the DATA step will output an informational message regarding this character variable default.

```
INFO:  Character variables have defaulted to a length of 200 at
       places given by:  (Line) : (Column).  Truncation may
       result.
       4:3          default200
```

EXTERNAL I/O EXTENSIONS

Footnotes

The DATA step FILE statement has been extended to handle text lines specified by the FOOTNOTE global statement. The FOOTNOTE statement specifies lines of text to be added to the bottom of each page of printed output. Historically, the FILE statement has processed TITLE lines, but not FOOTNOTE lines.

When enabled via the FILE statement, any active TITLE and FOOTNOTE text lines will be processed by the DATA step. These lines will be written to any file with the 'print' attribute. This includes FILE PRINT, any FILE statement with the PRINT option, or any file with print attributes (as defined by the host operating system). The writing of these lines are controlled by the TITLES and FOOTNOTES options on the FILE statement:

FILE fileref **PRINT** < ***TITLES*** > < **FOOTNOTES** > … ;
< **NOTITLES** >< ***NOFOOTNOTES*** >

The default for TITLE lines in previous versions is **TITLES**. It will remain so for Version 7. The default for FOOTNOTE lines will be **NOFOOTNOTES**.

Just as with TITLE lines, FOOTNOTE lines will reduce the number of text lines available for use within the body of each page of output. The number of lines available when the FILE option FOOTNOTES is specified should be exactly the number of lines available when NOFOOTNOTES is specified less the number of active FOOTNOTES.

DELIMITER= and DSD

In Version 6, the INFILE statement was enhanced with the DELIMITER= option. This option allows the specification of a character string containing alternate delimiters. A delimiter is a character which separates data values when data lines are scanned. When using list input, this character (or characters) is used in place of a blank for the scan.

This option has now been added to the FILE statement. Data lines that are output when this option is in effect will contain the specified character delimiter (instead of a blank) following the data item when list output is used. Even though a character string or variable with a length greater than one is accepted, only the first character of the string or variable is used as the output delimiter. This differs from INFILE DELIMITER= processing.

Note that the delimiter will appear in the column immediately following the data item, before any pointer controls ('+' or '@') are executed. No delimiter is output following the last data item on a line, though, since the end of the line is an implicit delimiter. To output data which contains the delimiter character, the DSD option (see below) should probably be used.

The DELIMITER= character is honored only if list output is being used. Formatted, column, and named output will ignore this option (since the output data is not 'delimited'). Modified list output (which combines formatted and list output) may be used, however.

If a data item contains the delimiter, it will be treated as any other data item. However, an analogous INPUT statement will not read the same data values because of the embedded delimiter. The DSD option overcomes this limitation.

The FILE statement DSD option will cause data items to be quoted with double quotes, if necessary. The necessity is determined by scanning the value for the delimiter. If the delimiter is found, the value is quoted. Any double quotes embedded in the data value will be expanded to two double quotes. For example, assume the DSD option is specified, the delimiter is a comma (','), and the data value to be output is the character string:

```
Jim said, "Hello" to the startled employee.
```

The value actually written to the file is quoted, with the embedded quotes being expanded:

```
"Jim said, ""Hello"" to the startled employee."
```

If no delimiter is found in the data item, no quotes are added. This quoting of data items can be forced, however, by specifying

the ~ (tilde) format modifier for a specific data item. This causes the data item to always be quoted.

If the DSD option is specified for the FILE statement, a default delimiter of comma (',') is assumed. This can be overridden using the DELIMITER= option.

INFILE Pseudo-Variable

The ability to directly access input data buffers has been enhanced. In previous releases, the _INFILE_ construct gave the ability to write out the contents of the current input buffer to a different output file. The only access, though, to this _INFILE_ buffer was via the PUT statement. You could modify the contents of the buffer with the PUT statement before it was output, but nothing else.

In Version 7, this _INFILE_ construct is allowed almost anywhere a variable reference may occur. Most notably, it may be used as source or target in an assignment statement, or as a function call argument. When specified, it references the entire contents of the current input buffer. 'Current input buffer' is defined as the last buffer read by the most recently executed INPUT statement. This INPUT statement will refer to the file denoted by the most recently executed INFILE statement.

The following example uses the _INFILE_ variable as an argument to the SCAN function, counting and outputting 'words' from the data lines. In previous releases, assumptions would have to be made about either the maximum number of words on a line, or the maximum length of any data record.

```
data _null_;
  input;
  wordn = 1;
  word = scan( _infile_, 1 );
  do while( word ne ' ');
    put 'Word' wordn '= ' word;
    wordn = wordn + 1;
    word = scan( _infile_, wordn );
  end;
datalines;
This is Version Seven.
Not in Version Six.
;

Word1 = This
Word2 = is
Word3 = Version
Word4 = Seven
Word1 = Not
Word2 = in
Word3 = Version
Word4 = Six
```

Although it can now be referenced like a variable, note that _INFILE_ is **not** an actual variable. There is no additional memory associated with _INFILE_. It is not a copy of the input data – it is a reference directly into the current input buffer. As such, some care should be taken when modifying the contents of this variable (i.e., as the target of an assignment statement). After modifying this variable, the next PUT _INFILE_ will reflect those modifications.

The _INFILE_ character pseudo-variable is automatically RETAIN'd (initialized to blanks), and is not written to any output SAS data set. Its length cannot be overridden in LENGTH or ATTRIB statements, although informats can be specified for it. The length varies at execution – it depends on the last record read into the current buffer. The maximum length during execution is the LRECL (logical record length) of the file to which it refers. Since this length is not known during the compile phase, this variable's length is considered to be the maximum allowed, 32,767. Some care must be taken when using _INFILE_ since this maximum length is used for length defaulting purposes. The following code would default the length of variable *infile_copy* to 32,767.

```
data one;
  infile filein;
  input x y z;
  infile_copy = _infile_;
run;
```

The DATA step will only update the contents of the _INFILE_ variable when it would normally read a new buffer of data from the INFILE. A new buffer is read only when an INPUT statement executes and was preceded by an INPUT statement which released its record (or block of records for N=). If N=1, a record is released at the end of an INPUT statement if there is no trailing @ (single or double). If N>1, a block of records is released at the end of an INPUT statement if the line pointer is on the last record of the block (and there is no trailing @).

You can also create an _INFILE_ pseudo-variable which is specific to a particular INFILE with the _INFILE_= option. A variable created via this INFILE option will always refer to the buffer of the INFILE for which it was specified, no matter what INFILE is considered current. The 'global' _INFILE_ variable will always refer to the current buffer from the current INFILE.

```
data _null_;
  infile this _infile_=thisbuf; /* Current INFILE is THIS         */
  input;
  x = _infile_;                 /* References THIS                */
  y = thisbuf;                  /* References THIS                */
  infile another;               /* Current INFILE is ANOTHER      */
  input;
  x = _infile_;                 /* References ANOTHER             */
  y = thisbuf;                  /* References THIS                */
run;
```

The same characteristics and restrictions apply to the _INFILE_= pseudo-variable as to _INFILE_ (see above). Additionally, the _INFILE_= reference must be the defining (i.e., first) reference in the DATA step program for the named variable.

FILE Pseudo-Variable

Analogous to the _INFILE_ variable, which refers to the current input buffer, the _FILE_ pseudo-variable refers to the current output buffer. 'Current output buffer' is defined as the last buffer formatted by the most recent PUT statement. This PUT statement will refer to the file denoted by the most recently executed FILE statement. The FILE statement also supports a _FILE_= option, which creates a pseudo-variable that always references the buffer of a specific FILE.

These pseudo-variables have the same characteristics and restrictions as _INFILE_ pseudo-variables: automatically RETAIN'd character variable that is not written to any output SAS data set. The length (which cannot be overridden) varies at runtime, depending on the contents of the buffer. The maximum length is governed by the LRECL of the file to which it refers, with an absolute maximum of 32,767 (which would be used for length

defaulting purposes). See previous _INFILE_ discussion and example.

The DATA step will update the contents of this variable only after it releases the current data buffer(s) for output to the FILE. A buffer is released only when the PUT statement column and line pointers move past the end of the data buffer (or block of buffers for N=). At this point, the data buffer is cleared (initialized to blanks). If N=1, a buffer is cleared at the end of a PUT statement if there is no trailing @ (single or double). If N>1, a block of buffers is cleared at the end of a PUT statement if the line pointer is on the last record of the block (and there is no trailing @).

You may access a _FILE_ variable before or after the PUT statement executes. Before the PUT statement executes, the output buffer is empty. In this instance, you would probably want to initialize the buffer with some data. This data would be reflected in the next PUT statement. Note that modifying the _FILE_ variable via programming statements only affects the length of the current output buffer; the column pointer associated with the output buffer is unchanged. In the first example, the line output to FILE PRINT is 'This is PUT!'. In the second example, the line is 'Where it be!'.

```
data _null_;
  file print;
  _file_ = 'Where is it?';
  put 'This is PUT!';
run;

data _null_;
  file print;
  _file_ = 'Where is it?';
  put @ 7 'it be!';
run;
```

You may also access the _FILE_ variable after a PUT statement executes, provided that the buffers are held in memory, either by using a trailing @ or N= and # line pointer controls. This may be useful if you need to save the output line, and process it further. This also allows you to use the formatting capabilities of the PUT statement to build complex strings for other uses.

```
data projects;
  infile projfile;
  input project phase $ targetdate;
  put project userfmt. '(' phase +(-1) ') ' targetdate monyy7. @;
  proj_info = _file_;
  put ;                          /* Release line to FILE LOG   */
                                 /* Further use of proj_info   */
run;
```

SCANOVER

In Version 6, the @ column pointer control was enhanced to allow a character specification. This would cause the DATA step to scan the input data for that character string in order to position the column pointer. If the end of a data line was reached while scanning, then the behavior is governed by any specified end-of-record option. If FLOWOVER (the default) is specified, the DATA step will go to the next data line and continue searching for the character string. For MISSOVER, STOPOVER, and TRUNCOVER, however, the DATA step will stop the scan, and take the appropriate end-of-record action.

The new SCANOVER option allows you to separate the @'char' scanning end-of-record action from the variable processing end-of-record action. It only affects the end-of-record action during the @'char' scan. If you desire the @'char' operator to scan over multiple lines to find its target, but for some other behavior (MISSOVER, STOPOVER, or TRUNCOVER) to take place while processing variables on the data line, then use SCANOVER in conjunction with the other end-of-record option. Specification of SCANOVER with FLOWOVER is allowed, but has no effect.

The following example will scan as many lines as required to find the string 'HERE'. But, once the string is found, we will set any remaining variables to missing when we encounter the end of the record. Note the output from the step.

```
data one;
  infile cards missover scanover;
  input @'HERE' var1 var2 var3;
  put var1= var2= var3=;
cards;
111 122 133 NOPE 144 155
211 222 233 HERE 244 255
HERE 311 322 333 344 355
;

var1=244 var2=255 var3=.
var1=311 var2=322 var3=333
```

Note: SAS went to a new line when INPUT @'CHARACTER_STRING' scanned past the end of a line.

Interface to Output Delivery System (ODS)

The Output Delivery System (ODS) in Version 7 implements and controls the formatting of all SAS procedure output. In previous versions of the SAS System, all procedures wrote exclusively to the SAS listing file and to output data sets. In Version 7, all procedures produce ODS *output objects* – binary objects that are rendered to various output destinations by the ODS sub-system.

An output object consists of two component parts: *a data object* containing the raw data values for the piece of output, and a *template* describing how the piece of output should look. The output object is added to the system and ODS decides, based on the output destinations set by the user, how to render the output.

The data object is the vehicle that a procedure uses to move the data for a piece of output to the ODS system. The data object is primarily an internal object over which the SAS user has no control. However, the SAS user has the flexibility to convert all ODS data objects into SAS data sets.

A template is a description of how you would like a piece of output to look when it is rendered. Templates contain formatting information like data column order, text for data column headings, format specifications for columns, and stylistic references. Every template in the SAS System is fully editable via the TEMPLATE procedure in batch, and from the SAS Explorer in DMS mode.

Each output object that is produced by a run of the SAS System is rendered to whatever output destinations the user has selected. For the initial release of ODS, the SAS listing file, output data sets, and HTML will be supported. For subsequent releases, support for PS (PostScript) and PCL (Printer Control Language) destinations, an RTF (rich text format) destination, and a persistent output document destination are being investigated. Each output destination can be controlled by the ODS global statement. Multiple output destinations can be active at the same time,

i.e., one step can create one output object which is rendered to multiple destinations.

The **ODS** global statement is a new statement in Version 7 that gives users some control over ODS. The **ODS** statement contains sub-statements which control each of the various output destinations, select or exclude individual pieces of output, and manipulate the SAS template concatenation path.

Although primarily created for procedure output, this new functionality is also available for DATA step programs, through the FILE and PUT statements. The FILE statement is used to define an ODS output object via the new ODS= option. With this new FILE statement option, the template is specified, and an ODS data object is defined and created. Data columns for the data object are specified, and mapped to variables in the DATA step program data vector. The PUT statement can then be used to move data from these DATA step variables into the ODS output object.

A detailed discussion of ODS is beyond the scope of this paper. For a complete discussion of the DATA step interface to ODS, see the SUGI 23 paper entitled ODS: The DATA Step Knows.

SAS FILE I/O EXTENSIONS

SET Statement option OPEN=DEFER

In DATA step SAS data set input processing, all data sets specified in any SET, MERGE, UPDATE, or MODIFY statement are opened during the compile phase. This is required to build the list of variables for the DATA step's program data vector. Also, the runtime behavior for most forms of these statements requires processing multiple data sets simultaneously.

For example, a SET statement with multiple data sets and an associated BY statement requires reading an observation from each data set before proceeding. The same is true for MERGE (with or without a BY), UPDATE (BY required), and MODIFY (with BY). A SET statement with a POINT= or KEY= would require random positioning over multiple data sets. Thus, all the data sets must be open simultaneously.

If all data sets do not need to be opened simultaneously, though, more memory is used than is needed. More importantly, for streaming input devices (e.g., tape drives), one device is required for each open data set, when one input device may be sufficient to handle all data sets.

This is true for a common scenario: a SET statement with no associated BY statement. Neither simultaneous processing nor random access is required. All data sets specified are processed sequentially, one after another. As long as each data set in the SET statement has the same structure (same variable names and types), they need not all be opened during the compile phase. If the data set opens could be deferred until required, the amount of memory necessary for the step would be reduced. If the specified data sets are tape data sets, the hardware requirements for the step would be reduced, because only one tape drive is needed.

The new OPEN= option for the SET statement will enable this. The default setting is OPEN=IMMEDIATE, which mirrors the current behavior. All data sets specified are opened during the compile phase, and kept open until the end of the step. Specifying OPEN=DEFER will cause only the first data set specified on the SET statement to be opened during the compile phase. During the execution phase, each data set is processed, then closed, and the next data set in the SET statement specification is opened. This continues until all data sets have been processed. The last data set opened remains open until the end of the step.

Since subsequent data sets are not opened during the compile phase for OPEN=DEFER, all variables that need to be processed by the DATA step program should be present on the first data set listed in the SET statement. Any variables found in subsequent data sets which were not found in the first data set will be ignored, and a NOTE is written to the SAS LOG. Processing will continue. The order of variables in subsequent data sets is not significant.

However, if a variable in a subsequent data set differs in its type from the same-named variable in the first data set, that is a runtime error for OPEN=DEFER. (It is a compile-time error when OPEN=IMMEDIATE is specified.) An error message is output, and processing stops.

```
data ytd;
  set qtr1 qtr2 qtr3 qtr4 open=defer;
run;

ERROR: Variable stock has been defined as both character (in
       WORK.QTR3) and numeric (in WORK.QTR1).
NOTE: The SAS System stopped processing this step because
      of errors.
```

DATASTMTCHK= Option

The DATASTMTCHK= option slightly restricts the one-level output SAS data set names which may be specified on the DATA statement. It attempts to protect you from inadvertently destroying production data sets because of a specific typographical error: a missing semicolon at the end of a DATA statement.

Consider the following DATA step program.

```
Data temp
  set prod.big_important_data_set;
  /* DATA step statements */
  …
run;
```

Because the semicolon is left off the DATA statement, this program would create three output data sets (temp, set, and prod.big_important_data_set), which probably contain very little. The production data set (prod.big_important_data_set) has been wiped out, and there is no recourse for data recovery.

The DATASTMTCHK= option will catch these dangerous errors by disallowing certain DATA step statement names as one-level output data set names. There are three possible values for this new option: NONE, COREKEYWORDS, and ALLKEYWORDS. NONE gives the previous behavior – no checking is done. COREKEYWORDS (the default) disallows the names SET, MERGE, UPDATE, and RETAIN as one-level output names. Two-level names (e.g., WORK.SET, PROD.MERGE, etc.) are allowed. The ALLKEYWORDS value will disallow any DATA step keyword that can begin a statement (e.g., ABORT, ARRAY, INFILE, etc.). Note that this list of words does not include global statements (e.g., TITLE, OPTION, etc.).

IORCMSG() Built-In Function

A new built-in DATA step function has been added to aid in deciphering codes contained in the _IORC_ variable. This variable contains return codes set by all forms of the MODIFY statement and the SET statement with the KEY= option. The IORCMSG function will return the formatted error message associated with the most recently posted _IORC_ code.

This function will be most useful in handling unexpected or unknown errors encountered with MODIFY and SET with KEY=, especially when using SAS/ACCESS® data sets. IORCMSG() will help by providing a verbose message that may indicate the cause of the error. The following example uses _IORC_ to handle error conditions and uses IORCMSG() to help diagnose the problem when an unknown condition occurs..

```
data acclib.master_ds;
  set otherlib.transaction;              /* Obtain id value */
  modify acclib.master_ds key=id;
  if (_iorc_ eq %sysrc(_SOK)) then
  do;                                    /* Data okay; typical case */
    …                                    /* DATA step statements */
    replace;
  end;
  else
  if (_iorc_ eq %sysrc(_DSENOM)) then
  do;                                    /* Handle 'no match found' error */
    …                                    /* DATA step statements */
    output;
    _error_ = 0;                         /* Avoid variable dump */
  end;
  else
  do;                                    /* Handle unknown error */
    acclib_error_msg = iorcmsg();
    put 'Unknown error occurred…' / acclib_error_msg;
    _error_ = 0;                         /* Avoid variable dump */
  end;
run;
```

Other SAS FILE I/O Extensions

There are other SAS data set I/O enhancements which are global in nature to the SAS System. These enhancements are beyond the scope of this paper, but are referenced here for completeness.

- SAS Library concatenation
- SAS Catalog concatenation
- SAS Data Set Versioning (Generations)
- Integrity constraints
- Cross-environment data access (CEDA)
- Access by observation number of compressed data sets

DATA STEP LANGUAGE EXTENSIONS

RENAME

Variable lists are now supported in the RENAME statement and the RENAME= data set option. A variable list is a short-hand notation for a longer list of variables with similar characteristics. The most common form of the variable list is Xn-Xm, where X denotes a common variable name root, and n and m denote ascending (n < m) or descending (n > m) numeric suffixes for the root variable name. This is the only form of variable list supported by RENAME.

The syntax of the RENAME statement and the RENAME= data set option:

RENAME old1< **-** oldn > **=** new1 < **-** newn >

RENAME = (old1 < **-** oldn > **=** new1 < **-** newn > **)**

In the example below, the first RENAME statement will rename the variables *var1*, *var2*, *var3*, *var4*, and *var5* to *myxyz1*, *myxyz2*, *myxyz3*, *myxyz4*, and *myxyz5*. The second statement will rename *var10* and *var11* to *bar522* and *bar523*.

```
rename var1 - var3 = myxyz1 - myxyz5;
rename var10 - var11 = bar522 - bar523;
```

Note that regardless of the case of the original variables, the renamed variables will match the case of the first variable in the 'new name' list (new1). For example, the resulting variable names from the following RENAME statement will all contain the root *STATUS*.

```
input x1 - x5;
rename x1 - x5 = STATUS1 - status5;
```

Initialization List Iteration Factor

The ARRAY and RETAIN statements support an initialization list. This list contains a series of constants which will initialize the given variables in the ARRAY or RETAIN statement before execution begins. This initialization list now supports iteration factors and nested sub-lists.

ARRAY / RETAIN … (*constant_sublist*) ;

…*where constant_sublist* can be one or more of the following…

< *iteration_factor* ***** > < **(** > constant | *constant_sublist* < **)** >

… where *iteration_factor* is an integral constant number

The iteration factor indicates how many times the given sub-list should be used in building the initialization list. The sub-list can itself contain other iteration factors and associated constant sub-lists. The following examples all denote the same initialization list (10 constant values of 5.2).

```
( 5.2, 5.2, 5.2, 5.2, 5.2, 5.2, 5.2, 5.2, 5.2, 5.2 )
( 10 * 5.2 )
( 5 * ( 5.2, 5.2 ) )
( 5.2, 5.2,  3 * ( 5.2, 5.2 ),  5.2, 5.2 )
( 2 * ( 5.2,  2 * ( 5.2, 5.2 ) ) )
```

Note that either a comma (',') or a blank (' ') may be used to separate the constant items in the list. The iteration factor and sub-lists are expanded during the compile phase. Therefore, the iteration factor must be a constant. The runtime processing of the initialized variables is not affected.

DATA Step Functions

New functions have been added. Most of these deal with variable attribute retrieval, and have names beginning with the letter 'V'. These V* functions differ from the SAS/AF® SCL-like VAR* functions added to the DATA step late in Version 6 development. The earlier VAR* functions return variable attributes associated with variables in SAS data sets. The new V* functions are DATA step

built-in functions, and return attributes associated with the variables in the DATA step program data vector. The VNAME and VLABEL functions are equivalent to the existing VNAME and LABEL CALL routines, and were added for consistency.

- VNAME(var) - returns the name of the given variable.
- VLABEL(var) - returns the label associated with the given variable. If there is no label, the variable name is returned.
- VTYPE(var) - returns the type of the given variable. 'N' is returned for numeric variables, and 'C' is returned for character variables.
- VLENGTH(var) - returns the defined compile-time size of the given variable. This is different from LENGTH(), in that LENGTH() examines the variable at runtime, trimming trailing blanks to determine the length. VLENGTH() returns a compile-time constant value, which reflects the maximum length. For example, LENGTH() returns 3 and VLENGTH() returns 8 in the following:

```
length x $8;  x = 'abc';  y = length(x);  z = vlength(x);
```

- VFORMAT(var) - returns the format associated with the given variable. This is the complete format FMTw.d name, including lengths and the dot ('.'), i.e., '$CHAR20.'.
- VFORMATN(var) - returns the format name, excluding any lengths, associated with the given variable, i.e., '$CHAR'.
- VFORMATW(var) - returns the format width value (w) associated with the given variable.
- VFORMATD(var) - returns the format decimal value (d) associated with the given variable.
- VINFORMAT(var) - returns the informat associated with the given variable. This is the complete informat INFMTw.d name, including lengths and the dot ('.'), i.e., '$CHAR20.'.
- VINFORMATN(var) - returns the informat name, excluding any lengths, associated with the given variable, i.e., '$CHAR'.
- VINFORMATW(var) - returns the informat width value (w) associated with the given variable.
- VINFORMATD(var) - returns the informat decimal value (d) associated with the given variable.
- VARRAY(var) - returns 1.0 if the given variable denotes an array name, and 0.0 if it does not.
- VINARRAY(var) - returns 1.0 if the given variable is an element of any array, and 0.0 if it is not.

None of these functions accept expressions as arguments. Only scalar or array references are allowed. The actual name is the argument, not the contents of the variable. Therefore, most of these 'functions' are actually references to constant values at runtime (unless array references are involved).

However, there are versions of these functions which do resolve the argument at runtime, and therefore, allow expressions as arguments. Attributes are returned for the variable indicated by the contents of the argument. These functions' names end with 'X': VLABELX, VTYPEX, VFORMATX, etc. For example, the result of the VTYPE function below is 'C' (the variable *cvar* is character), but the result of the VTYPEX function is 'N'. VTYPEX() inspects the runtime value of the variable *cvar* ('nvar'), and returns attributes for the variable specified by that runtime value. The variable *nvar* is numeric, so 'N' is returned.

```
data _null_;
  length cvar $8 nvar 8;
  cvar = 'nvar';
  x = vtype(cvar);
  y = vtypex(cvar);
run;
```

Other new functions include the following:

- IORCMSG() - returns the formatted message associated with the most recently returned _IORC_ return code. See discussion above under SAS FILE I/O EXTENSIONS.
- MISSING(expr) - returns 1.0 if the expression is a missing value, and 0.0 if it is not. 'expr' can be of type character or numeric. 'Missing' is defined for a character type as all blanks. For a numeric type, the standard missing ('.') and all special missing values ('._', '.A' - '.Z') will return a result of 1.0.

Stored Program Facility and Views

The DATA step Stored Program Facility has been enhanced to support data set options which are specific to a particular host platform or SAS I/O engine. Previously, they could not be saved with the stored program and would be flagged with an error when the program was compiled. These option settings can now be restored when the stored program or DATA step view (which uses the Stored Program Facility) is executed.

PERFORMANCE ENHANCEMENTS

Code Generation

The DATA step is a 'compile and run' language implementation. The source is parsed and compiled into an intermediate representation, called quad codes. These quad codes are translated into host specific object code, using an internal code generation subsystem. This object code (called a code stream) is then executed (or 'run'), and later deleted.

The code generation subsystem now contains an optimization phase. This phase may modify the code stream in several ways, attempting to reduce the size or increase the speed of the code stream. It will attempt to place highly referenced items in registers, remove dead code, replace certain operation sequences with better or smaller ones, and remove redundant code (especially useful for array referencing). These modifications should result in smaller, faster code streams generated for DATA step programs.

PUT and INPUT Functions

A major performance improvement has been implemented for the PUT and INPUT functions. A portion of the processing for these functions is now done with inline code generation. This can significantly reduce the low-level call overhead for each PUT or INPUT function invocation.

DATA Step Views

DATA step views have always supported both sequential access and random access. For Version 7, this random access processing has been improved in the area of 'spill file' generation. A 'spill file' is a temporary data file used by random access view processing to hold data observations as they are created. Once an observation has been created, it can be retrieved from the spill file instead of the executing view.

In previous releases, when the first observation is requested from a DATA step view opened for random access, the view would execute to completion, creating the **entire** spill file. For views which generate large numbers of observations, this meant a noticeable delay before the first observation is returned. Of course, after the first observation, retrieval performance is optimal. But, if the intent is to inspect only a few observations, then the resources used for the spill file creation (disk space and CPU time) have been wasted.

In Version 7, a DATA step view opened for random access will create an observation only if it is needed to satisfy a request. (Actually, the view will create enough 32K-byte buffers of observations to satisfy the request. No partial buffers are created.) This will allow the first observation to be returned very quickly. If an observation previously created is requested, it is retrieved from the spill file. If an observation not previously created (i.e., beyond EOF in the spill file) is requested, then the view is asked to create more observations until the request can be satisfied.

For applications that process the view entirely, this change merely distributes the resource usage more evenly along the life of the application. But, when only a small subset of observations are accessed, this change should result in disk space and CPU time savings.

ACKNOWLEDGMENTS

The author greatly appreciates the contributions by Nancy Agnew, Deanna Tawiah, Mandy Womble, and Pauline Leveille for new feature testing, and Susan O'Connor and Jason Secosky for assistance in reviewing this paper.

The author can be contacted at

William Heffner
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Phone (919) 677-8000
FAX (919) 677-4444
Email saswfh@sas.com