Paper 5103-2020

Some _FILE_ Magic

Mike Zdeb, FSL, University at Albany School of Public Health, Rensselaer, NY

ABSTRACT

The use of the SAS® automatic variable _INFILE_ has been the subject of several published papers. However, discussion of possible uses of the automatic variable _FILE_ has been limited to postings on the SAS-L listserv and on the SAS Support Communities web site. This paper shows several uses of the variable _FILE_, including creating a new variable in a data set by concatenating the formatted values of other variables and recoding variable values.

INTRODUCTION

When you use a data step with an INPUT statement, SAS creates an input buffer where it holds your data prior to moving the values of variables into the program data vector. If your data step reads a SAS data set with a SET statement and no INPUT statement is present, no input buffer is created. If there is an input buffer, you can access the contents of that buffer using the variable name _INFILE_. _INFILE_ is an *automatic* variable whose value is accessible within a data step but is not output to any data set being created in the data step. The use of _INFILE_ is shown in several papers cited in the references (an explanation of how data step works including creation of the input buffer is found on page 335 of the SAS documentation also cited in the references). The following examples show some aspects of _INFILE_ and the input buffer.

In this first example, you want to read some data and have all the character variables in uppercase. You can use a $UPCASE informat, or you could move each record into the input buffer and hold the record in the buffer with a trailing @-sign ①. The contents of the input buffer is named _INFILE_ and you can use an UPCASE function to convert the contents to uppercase ②. The next INPUT statement moves variable values from the input buffer into the program data vector ③.

Obs	name	age	city	state
1	MIKE	25	ALBANY	NY
2	SARA	15	WASHINGTON	DC

data set NAMES ... uppercase using _INFILE_

```
data names;
input @; ①
_infile_ = upcase(_infile_); ②
input name :$10. age city :$10. state :$2.; ③
datalines;
mike 25 albany ny
Sara 15 Washington DC
;
```

Someone gives you a data file in which two consecutive have been used as a delimiter. Missing data are indicated by four consecutive slashes. Admittedly it is a somewhat contrived example, but it does show how you can modify the input buffer in those instances where it seems as if it would be difficult to read variable values properly.

Obs	name	age	city	state
1	MIKE	25		NY
2	SARA	15	WASHINGTON	DC
3	JESSICA	9		CA

data set NAMES ... alter delimiter using _INFILE_

```
data names;
infile datalines dsd dlm='/'; ①
input @; ②
_infile_ = tranwrd(_infile_,'//','/'); ③
input name :$10. age city :$10. state :$2.; ④
datalines;
MIKE//25////NY
SARA//15//WASHINGTON//DC
JESSICA/9////CA
;
```

The DLM option is used on the INFILE statement to specify that a slash separates variable values ①. One consequence of a DSD option in an INFILE statement is that two consecutive delimiters are treated as indicating a missing value. A record is placed in the _INPUT_ buffer and held there using a trailing @-sign ②. The TRANWRD function is used to convert tow slashes to one ③, then data are read into the program data vector using list input ④.

While it is easy to find papers and documentation about _INFILE_ and the input buffer in a data step, it is difficult to find much similar information about _FILE_ and the output buffer. Again, several examples are used, but now to show aspects of _FILE_ and the output buffer.

```
proc format; ①
value ag low-<13.5  = '0'   other = '1';
value ht low-<64.15 = '0'   other = '1';
value wt low-<107.25 = '0'  other = '1';
run;

filename nosee dummy; ②

data males (drop=sex);
file nosee; ③
set sashelp.class (where=(sex eq 'M'));
put age ag. height ht. weight wt. @; ④
htwt = _file_; ⑤
put; ⑥
run;
```

Obs	Name	Age	Height	Weight	htwt
1	Thomas	11	57.5	85.0	000
2	James	12	57.3	83.0	000
3	John	12	59.0	99.5	000
4	Robert	12	64.8	128.0	011
5	Jeffrey	13	62.5	84.0	000
6	Alfred	14	69.0	112.5	111
7	Henry	14	63.5	102.5	100
8	Ronald	15	67.0	133.0	111
9	William	15	66.5	112.0	111
10	Philip	16	72.0	150.0	111

data set MALES ... variable HTWT created using _FILE_

Formats are created that will be used in the subsequent data step to check whether males in the data set SASHELP.CLASS have age, height, and weight that are below (0) or at/above (1) the median for each variable. The FILENAME statement sets up a FILEREF named NOSEE ②. The special physical file DUMMY specifies that no actual file will be produced, but you can still write to that file using the FILE statement ③ and PUT statements in the data step. The PUT statement writes formatted values (0s and 1s) of the three variables to the output buffer and the results are held there using a trailing @-sign. Just as contents of the input buffer are available using _INFILE_, contents of the output buffer can be accessed using _FILE_. A new variable named HTWT is createdf from the contents of the output buffer (formatted values of age, height, and weight) ⑤. The output buffer is cleared with a PUT statement ⑥. That second PUT statement clears the output buffer by writing its contents to the file DUMMY.

Obs	htwt
1	000
2	000000
3	000000000
4	000000000011
5	000000000011000
6	000000000011000111
7	000000000011000111100
8	000000000011000111100111
9	000000000011000111100111111
10	000000000011000111100111111111

variable HTWT if second PUT statement not used

In the data set on the above right you can see the values of age/height/weight that are below (0) or at/above the median (1) by looking the variable HTWT. Without the second PUT statement, the presence of the trailing @-sign on the first PUT statement would result in the formatted values of the age, height, and weight for each observation to be added onto the end of a string of values from all the previous observation(s), as shown on the right.

A single trailing @-sign on a PUT statement persists during multiple passes through the data step, not like an INPUT statement where the INPUT buffer is cleared at each pass through the data step if there is only one trailing @-sign (two, or @@, are needed to ensure that the contents of the input buffer are not cleared).

Data set MALES could also be produced with the following data step ...

```
data males (drop=sex);
set sashelp.class (where=(sex eq 'M'));
put @1 age ag. height ht. weight wt. @; ①
htwt = _file_;
run;
```

Notice that there is no FILE statement within the data step. That would normally cause the results of any PUT statement to be written to the LOG. But the trailing @-sign on the PUT statement causes the results of the PUT statement to be held in the input buffer and not written to the LOG ①. Also notice the @1 in the PUT statement that causes the formatted values of age, height, and weight to overwrite each other on every pass through the data step (rather than accumulating in one string as shown on the previous page). Upon completion of the data step, the output buffer is emptied and one line is written to the LOG.

What follows are a set of examples that show how a combination of PUT and _FILE in a data step can be used for a number of different tasks, as alternatives to other methods you are possibly already using.

COMPLEX SEARCHING FOR VARIABLE VALUES

There are a number of ways to determine if a collection of variables contain a specified value. Given the data set ANSWERS, you want to create a new data set containing all observations where a subject had at least one answer with a value of 'Y' for any of the ten questions (variables q1-q10). There are many ways to create that data and among them are the following: the FIND and CATT functions ①; the WHICHC function ②; an ARRAY and the IN operator ③.

```
data atleast1y;
set answers;
if find(catt(of q:),'Y'); ①
run;

data atleatst1y;
set answers;
if whichc('Y', of q:); ②
run;

data atleats1y;
set answers;
array q(10);
if 'Y' in q; ③
run;
```

id	q1	q2	q3	q4	q5	q6	q7	q8	q9	q10
A1234	Y	Y	Y	Y	Y	Y	Y	Y	N	N
A2345	N	N	N	N	N	N	N	N	N	N
A3456	N	N	N	N	N	N	N	N	N	Y
A4567	N	N	N	N	Y	N	N	N	N	N
A5678	Y	N	N	Y	N	N	Y	N	N	Y

data set ANSWERS ... find Y

What if the task was a bit more complex? Given a data set with multiple diagnoses, find all the observations where at least one diagnosis starts with the characters '250' (if you are familiar with ICD-9-CM codes, any diagnosis that starts with those characters indicates diabetes). One way to find the diabetes cases is with an ARRAY and a LOOP.

```
data diabetes (drop=j);
set diagnoses;
array dx(5);
do j=1 to 5 until (dx(j) eq: '250'); ①
end;
if j lt 6; ②
run;

data diabetes;
set diagnoses;
array dx(5); ①
if '250' in : dx; ② ③
run;
```

id	dx1	dx2	dx3	dx4	dx5
01	486	5849	5990	04104	45119
02	5589	27651	5990	78079	
03	51881	49121	V1582	V1251	78650
04	5781	V1042	V1041	25060	25050
05	496	4280	25040	58281	5859
06	486	496	340	311	

data set DIAGNOSES ... find DIABETES

If no diabetes diagnoses are found, the value of the INDEX variable J ① will be 6 when the loop is completed, so any value less than 6 ② indicates that a diabetes case was found. The combination of a ARRAY ①, IN operator ②, and COLON modifier ③ give the same result with much less SAS code.

An even more complex task is finding if many variables contain many values. Referring back to the previous example with an ARRAY and the IN operator, it is equivalent to asking if there is an operator that allows the equivalent of ...

```
IF <many values> IN <many variables represented by an ARRAY>;
```

There is no such operator, but there is an alternative method that uses a PUT statement, the variable _FILE_, and the FIND function. The new problem to solve is given a data set with multiple diagnoses, find all the observations where a patient has traumatic brain injury (TBI). The task is more difficult than the previous search for diabetes since the indication for TBI includes the following individual diagnosis codes and code ranges: 800-80199, 803-80499 , 850-85419, 9501-95039, 95901, 99555. Thinking in terms of the IN operator and an ARRAY, the task for the first observation (ID=01) is as follows.

```
IF <95901, 78039, 4280, 87342, 81612> IN <800-80199,  803-80499 , 850-85419,
9501-95039, 95901, 99555>;
```

Since there is no such search mechanism in SAS, here is a solution.

```
proc format; ①
value $tbi
'800'-'80199', '803'-'80499' , '850'-'85419',  '9501'-'95039',
'95901' , '99555' = '1'  other = '0';
run;

data tbi;
set diagnoses;
put @1 (dx1-dx5) ($tbi.) @; ②
if find(_file_,'1'); ③
run;
```

id	dx1	dx2	dx3	dx4	dx5
01	95901	78039	4280	87342	81612
02	78039	41400	4019	85301	
03	82009	30501	496	2875	41400
04	9949	2765	4280	4240	78039
05	8730	9120	80001		

data set DIAGNOSES ... find TBI diagnoses

The first step is to create a format that will convert all TBI diagnosis codes to 1 and all non-TBI codes to 0, and ①. The diagnosis codes are read in a data step and a PUT statement ② is used to write the formatted values of the codes (1s and 0s) to the output buffer where the results of the PUT statement are held using the trailing @-sign (otherwise the results would be written to the LOG). The contents of the OUTPUT buffer are available using the variable _FILE_ and a FIND ③ function checks for the presence of a 1 in that variable (indicating the presence of TBI).

id	dx1	dx2	dx3	dx4	dx5
01	95901	78039	4280	87342	81612
02	78039	41400	4019	85301	
05	8730	9120	80001		

data set TBI ... created using PUT+ _FILE_ + FIND

If you also wanted to create a variable that pointed to the TBI diagnosis(es), you could use the following and the TBI data set is shown on the right. Each 1 in the variable TBI indicates a traumatic brain injury diagnosis code was found.

```
proc format; ①
value $tbi
'800'-'80199', '803'-'80499' ,
'850'-'85419', '9501'-'95039',
'95901' , '99555' = '1'
other = '0'
' ' = 'x';
run;

data tbi;
set diagnoses;
length tbi $5; ②
put @1 (dx1-dx5) ($tbi.) @;
if find(_file_,'1');
tbi = _file_; ③
run;
```

id	dx1	dx2	dx3	dx4	dx5	tbi
01	95901	78039	4280	87342	81612	10000
02	78039	41400	4019	85301		0001x
05	8730	9120	80001			001xx

data set TBI ... with new variable TBI showing the location of traumatic brain injury diagnosis codes (95901 85301 80001)

The $TBI format is modified to convert missing diagnoses to an x (notice that variable TBI in the data set TBI has an x for missing diagnoses). The LENGTH statement ② is important since the length of _FILE_ is 32,767 and TBI is created from _FILE_ in a subsequent statement ③ and would have the same length as _FILE_. The variable TBI shows the diagnosis that led to selection of the observation.

RECODING VARIABLE VALUES

We can use the last example in the previous section to show how the _FILE_ statement can be used to recode variables. The last example produced a new variable named TBI that showed with 1s and the 0s the presence or absence of a traumatic brain diagnosis code. Rather than have one variable (TBI) with all the 1s and 0s, what if you wanted to convert the five diagnosis codes (variables DX1-DX5) to a series of binary variables (TBI1-TBI5) with the value of either 1 or 0. One way to create the new binary variables would be with an ARRAY and a LOOP in a data step.

```
proc format; ①
value $tbi
'800'-'80199', '803'-'80499' , '850'-'85419',  '9501'-'95039',
'95901' , '99555' = '1'  other = '0'  ' ' = ' ';
run;

data tbi (drop=j);
set diagnoses;
array dx(5);
array tbi(5);
do j=1 to 5;
   tbi(j) = input(put(dx(j),$tbi.),1.); ②
end;
run;
```

The $TBI format is modified ① so missing diagnosis codes will produce a missing value for the variables TBI1 through TBI5. A loop is used within a data step to first apply the format to the diagnosis codes with a PUT statement ②. Since that produces a character value, the INPUT function is used to produce numeric variables TBI1-TBI5. The following shows how the same task can be done with PUT and _FILE combined with INPUT and _INFILE_ (using the same $TBI format as was used with the ARRAY and LOOP).

```
data tbi;
if _n_ eq 1 then input @; ①
set diagnoses;
put @1 (dx1-dx5) ($tbi.) @; ②
_infile_ = _file_; ③
input @1 (tbi1-tbi5) (1.) @@; ④
datalines;
*
;
```

id	dx1	dx2	dx3	dx4	dx5	tbi1	tbi2	tbi3	tbi4	tbi5
01	95901	78039	4280	87342	81612	1	0	0	0	0
02	78039	41400	4019	85301		0	0	0	1	.
03	82009	30501	496	2875	41400	0	0	0	0	0
04	9949	2765	4280	4240	78039	0	0	0	0	0
05	8730	9120	80001			0	0	1	.	.

data set TBI ... created using PUT + _FILE_ + _INFILE_ + INPUT

An INPUT statement ① is used to establish an INPUT buffer and it will be empty since no variable values are used (since an INPUT statement requires either a DATALINES or external file to read, a single line DATALINES file is used and it does not matter what that line contains). The task of establishing the input buffer need only be done once so the first INPUT statement only executes on the first pass through the data step. Once again, a PUT statement writes the formatted values of the diagnosis codes to the output buffer ②. The next statement moves the contents of the output buffer to the input buffer ③ and the data in the input buffer is read with an INPUT statement ④. Since we want to hold onto the INPUT buffer, a double @@ is used on the INPUT statement to preserve the buffer for the next pass through the data step.

CONVERT MISSING NUMERIC DATA TO ZERO

Given a data set (on the right) with missing values for some or all the numeric variables, you want to replace the missing values with zeroes. However, you would also like to keep track of the values that have been converted so you can distinguish the new zeroes (converted from missing) from those originally present in the data set.

name	gender	age	height	weight
MIKE	M	21	60	.
MARY	F	0	.	.
MARK		45	72	0

data set WITH_MISSING

One method would be to use an ARRAY, a LOOP, plus the CATT function to create a variable that identifies variable values that were originally missing.

```
data no_missing;
length missing $3; ①
set with_missing;
array nm(*) _numeric_; ②
do _n_=1 to dim(nm); ③
   missing = catt(missing, nm(_n_) eq .); ④
   nm(_n_) + 0; ⑤
end;
run;
```

A new character variable named MISSING is to be created using the CATT function. Since the default length of such variables is 200, a LENGTH statement sets the length to 3 ① (the number of numeric variables in the data set). The LENGTH statement also establishes that MISSING is a character variable. Otherwise when it is first used later in the data step in the CATT function, it would assumed to be a numeric variable and result in a LOG message about numeric-to-character conversion. The ARRAY statement sets up an array named NM ②. You need not specify the variable names since the text _NUMERIC_ places all numeric variables in the array. A LOOP is used to cycle through the numeric variables within each observation ③. The N=new variable MISSING will have a 1 for each missing variable and a 0 for all others ④. Missing values are converted to 0 using a SUM statement ⑤.

You can also use PROC STDIZE convert missing numeric values to zero, however you still need a data step if you want to create the new variable MISSING to keep track of variables that were originally missing and are now zero.

```
proc format; ①
value fix low-high='0' other='1';
run;

data with_missing;
length missing $3; ②
set with_missing;
put @1 (_numeric_) (fix.) @; ③
missing = _file_; ④
run;

proc stdize data=with_missing out=no_missing reponly missing=0; ⑤
run;
```

missing	name	gender	age	height	weight
001	MIKE	M	21	60	0
011	MARY	F	0	0	0
000	MARK		45	72	0

data set NO_MISSING ... created using a data step + PROC STDIZE

The format FIX is created ① that will convert all non-missing values (LOW-HIGH) to 0 and missing values (OTHER) to 1. The LENGTH statement ② is important since the length of _FILE_ is 32,767 and MISSING is created from _FILE_ in a subsequent statement. Formatted values of the numeric variables are written to the output buffer ③ and the contents of that buffer (_FILE_) are placed into the variable MISSING ④. PROC STDIZE ⑤ is then used to replace (REPONLY) missing numeric values with zeroes (MISSING=0). The data set on the above right shows the variable MISSING that indicates with a 1 that WEIGHT was missing in the first observation and both HEIGHT and WEIGHT were missing in the second observation. The WEIGHT of zero in the third observation was in the original data. Note the "data step only" method shown first on this section produces the same NO_MISSING data set.

REFERENCES

SAS® 9.4 Language Reference: Concepts, Sixth Edition
https://documentation.sas.com/api/docsets/lrcon/9.4/content/lrcon.pdf

Now _INFILE_ is an Automatic Variable - So What?
https://www.lexjansen.com/nesug/nesug01/cc/cc4018bw.pdf

More _Infile_ Magic
https://support.sas.com/resources/papers/proceedings/proceedings/sugi28/086-28.pdf

Using _INFILE_, Substr, Translate and TranWrd to Manipulate Strings of Text Within the Input Buffer
https://www.lexjansen.com/nesug/nesug02/cc/cc014.pdf

ACKNOWLEDGMENTS

Thanks to Howard Schreier who first made me aware of possible uses of the output buffer (the variable _FILE_) in a SAS-L posting ...
http://listserv.uga.edu/cgi-bin/wa?A2=ind0811B&L=sas-l&P=R15431

Thanks also to Pierre Gagnon who gave me the idea of exchanging data between a variable and the input buffer (the variable _INFILE_) in a SAS Support Communities posting ...
https://communities.sas.com/message/134104

CONTACT INFORMATION

The author can be contacted using e-mail... Mike Zdeb *mike.zdeb@gmail.com*