Now _INFILE_ is an Automatic Variable – So What?
Howard Schreier, U.S. Department of Commerce, Washington, DC

ABSTRACT
In earlier versions of SAS® software, _INFILE_ could only be used in the context of the PUT statement. Now it is an automatic variable. What are the practical benefits of this expanded functionality?

INTRODUCTION
The keyword ”_INFILE_” (not to be confused with the infile statement) has long been part of the SAS language, but in a very limited context: as a “specification” in the PUT statement. The statement

    put _infile_;

copies the contents of the most recently filled input record buffer to the SAS Log or whatever output destination is in effect.

In Version 7 and later versions, _INFILE_ is an automatic variable. This means that it can still be used in the original context, but it can also appear virtually any place where it is appropriate to refer to a character variable. It can be used in an expression or as the target of an assignment statement.

This enhancement is useful, but not tremendously so. That is because just about anything which can be accomplished by using the automatic variable _INFILE_ can also be done using an explicit (user-defined) variable. In Version 6 and earlier versions, _INFILE_ could be particularly useful in transcribing records exceeding (or potentially exceeding) 200 characters in length, which could not be contained in ordinary character variables. But that particular advantage is a thing of the past, now that 32K-long variables are possible.

Nevertheless, _INFILE_ provides convenient and efficient techniques for solving certain problems. We’ll look at a couple of examples. The first is a read-only problem (inspecting the buffer) and the second is a read/write problem (altering the buffer).

EXAMPLE 1
Here is a little data file:

```
# Bordick 28Sep2001 4 2
29Sep2001 5 1
# Mora 28Sep2001 4 0
29Sep2001 5 2
30Sep2001 2 1
# Ripken 28Sep2001 4 1
30Sep2001 3 3
```

The programming problem is in distinguishing between the lead records (with player names) and the trailing records (without). The classic DATA step technique is to read in and test a character variable while “holding” the input record for subsequent INPUT statements.

```
data then;
    infile 'batting';
drop firstcol;
    input firstcol $1 01 0;
    if firstcol='#' then
        input +2 player:$15. 0;
    retain player;
    input gamedate:date9. atbats hits;
    run;
```

Here’s a way to exploit _INFILE_ to simplify this a little.

```
data now;
    infile 'batting';
    input @;
    if substr(_infile_,1,1)=# then
        input +2 player:$15. 0;
    retain player;
    input gamedate:date9. atbats hits;
    run;
```

There is no longer any need to create (and then DROP) the additional variable FIRSTCOL. The first INPUT statement loads a new record into the buffer. Then the SUBSTR function is used to test for the record-type indicator.

EXAMPLE 2
Here is the data file:

```
Monday 14 3 1.1
Tuesday 10 -2 0
Wednesday 9 _4 .2
Thursday 11 0 0
Friday 11 1 0
```

The problem is that sloppy data preparation has substituted underscores for some of the minus signs. A read-in process which does not allow for this, such as:

```
data nogood;
    infile 'weather';
    input day:$9. hitemp lotemp precip;
    run;
```

encounters errors:

```
NOTE: Invalid data for lotemp in line 3 13-14.
RULE: =-------1--------2-------3

Wednesday 9 _4 .2
```

Here’s one traditional way to address this; it creates a temporary character variable for each problem field, repairs it, extracts the numeric value, and excludes the character variable from the output:

```
data hardfix;
    infile 'weather';
    drop day:9 hitemp lotemp;
    input day:$9. hitemp : lotemp $ precip;
    hitemp = input(translate(hitemp,'-','_'),8.);
    lotemp = input(translate(lotemp,'-','_'),8.);
    run;
```

It works fine, but here’s a way to make use of _INFILE_ to simplify the solution:

```
data now;
    infile 'weather';
    input day:$9. hitemp lotemp precip;
    run;
```

A read-in process which allows for this, such as:

```
data now;
    infile 'weather';
    input @;
    data now;
    if substr(_infile_,1,1)=1 then
        input +2 day:$9. hitemp lotemp precip;
    run;
```

allows for the underscores. This is slightly more convenient than the solution above, and is very similar to the original technique.
data easyfix;
infile 'weather';
input @;
_infile_ = translate(_infile_,'-','_');
input day:$9. htemp lotemp precip;
run;

The first INPUT makes the record available in the buffer. Then the entire record is repaired via an assignment statement before the second INPUT statement loads the variables.

DETAILS: VARIABLE LENGTHS; LEADING AND TRAILING BLANKS

_INFILE_ is created automatically, so the DATA step does not have to specify its length. Of course, the value of the LRECL option for the input file (256, by default) provides an upper bound. Even though _INFILE_ cannot possibly take on a value with a length exceeding the LRECL value, SAS does not know the LRECL value when it allocates memory, so it reserves the maximum possible space (32767 bytes), and a variable created via assignment from _INFILE_ will by default have that length. To illustrate, consider:

data out;
infile 'weather';
input;
keepit = _infile_;
run;

The variable KEEPIT will have a length of 32767. So one should code a LENGTH statement when creating such a variable.

However, within the DATA step, _INFILE_ itself (as distinguished from variables derived from _INFILE_) behaves as if its length were variable. In other words, the TRIM function is not needed to remove trailing blanks (unless such blanks are explicitly present in the buffer). To illustrate, start with this little data file:

| a |
| b |
| c |

Read in each record and bracket it:

data _null_;  
infile 'trailing'; 
input; 
sandwich = '|'||_infile_||'|'; 
put sandwich; 
second = substr(_infile_,2,1); 
run;

The log shows the "self-trimming" behavior:

```
| abc |
| ab  |
| a   |
```

There is another indicator of the same phenomenon: the SUBSTR function generates an error on the third record (and only on the third record) because the second argument is out of range relative to the length of the _INFILE_ variable at that point.

```
NOTE: Invalid second argument to function SUBSTR
sandwich=|a| second=a _ERROR_=1 _INFILE_=a
_N_=3
```

In these respects, _INFILE_ does not behave like an ordinary character variable. If such a variable were used in these two assignment statements, all of the SANDWICH values would be the same length, and SUBSTR would encounter an error on either all of the records or none of the records.

Leading blanks in _INFILE_ values are preserved automatically. That is not really surprising on input. On output, it means that there is no need to apply a $CHAR format. Thus, _INFILE_ makes it easy to transcribe data records without the attention to leading blanks which is needed when explicit variables are used.

REFERENCES

CONTACT INFORMATION
Your comments and questions are valued and encouraged.
Contact the author at:
Howard Schreier
U.S. Department of Commerce
H-2815
Washington DC 20230
(202) 482-4180
Fax: (202) 482-4614
Howard_Schreier@ita.doc.gov

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are registered trademarks or trademarks of their respective companies.