The INPUT Statement: Where It's @

Ronald Cody, Ed.D.
Robert Wood Johnson Medical School

Introduction

One of the most powerful features of SAS software is the ability to read data in almost any form. For example, you can have data values separated by blanks or other delimiters or you can arrange your data in columns, using one or more lines of data for each subject. You can also read selected data fields and then decide how to read the remaining data values.

This tutorial will give you an overview of the immense power and flexibility of the SAS INPUT statement.

Reading Space Delimited Data

A common form of data entry is to separate each data value by one or more spaces. This is handy for small data sets that are entered by hand, especially for test purposes. This arrangement of data is often called "list directed" data. The rule here is that you must specify all the variables in the data lines and all the data values must be separated by one or more spaces. You must also indicate which variables are to be read as character data. Look at the following example:

```
***LIST DIRECTED INPUT;
DATA LIST;
   INPUT X Y A $ Z;
DATALINES;
1 2 HELLO 3
4     5     GOODBYE   6
;
PROC PRINT DATA=LIST;
   TITLE 'LIST DIRECTED INPUT';
RUN;
```

Notice that you need to list the variable names on the INPUT statement and to place a dollar sign ($) after any variable that is to hold character values. Also notice that the second line of data which has multiple spaces between each data value causes no problems at all.

Delimiters Other Than Spaces

With just a small change to the program, you can indicate any delimiter you like, in place of the default blank. To understand how this works, we need to jump ahead a bit to see how a SAS program reads data from an external data file (as opposed to data following a DATALINES statement). You do this by including an INFILE statement which tells the program where to find the data. The INFILE statement has several options that provide additional information on how to read these external data lines, one of them being an option to define a delimiter other than a blank. The option has the form: DLM= '*your_delimiter*'. Since we want to demonstrate this option with "in-stream" data, we use the reserved fileref (file reference) DATALINES. This allows you to supply INFILE options with "in-stream" data. Here is the program:

```
***OTHER DELIMITERS;
DATA DELIM;
   INFILE DATALINES DLM='#';
   INPUT X Y A $ Z;
DATALINES;
1#2#HELLO#3
4   #   5   #  GOODBYE  # 6
;
PROC PRINT DATA=DELIM;
   TITLE 'OTHER DELIMITERS';
RUN;
```

In this example, we use a number sign (#) as the data delimiter. The INFILE statement uses the reserved fileref DATALINES followed by the DLM= option.

Special Case of Comma Delimited Files

There is a common data arrangement used by many personal computer applications. That is, to separate data values by commas and to place string values in double quotes. Furthermore, two commas together indicate that there is a missing value. The INFILE option DSD takes care of all of these features. Besides allowing commas as the data delimiter, this

option reads character strings enclosed in double quotes and strips off the quotes before assigning the value to the character variable. It also allows you to include commas within a character string. Finally, two commas together are interpreted as a missing value. The program that follows demonstrates all these features:

```
***SPECIAL COMMA DELIMITED FORMAT;
DATA SPECIAL;
   INFILE DATALINES DSD;
   INPUT X Y A $ Z;
DATALINES;
1,2,HELLO,3
4  ,   5  ,   GOODBYE ,  6
7,,"HI THERE",8
9,10,"HI,THERE",11
;
PROC PRINT DATA=SPECIAL;
   TITLE 'SPECIAL COMMA DELIMITED
FORMAT';
RUN;
```

To see that this program is working as expected, here is the output from PROC PRINT:

```
Special Comma Delimited Format

OBS    X     Y    A           Z

 1     1     2    HELLO       3
 2     4     5    GOODBYE     6
 3     7     .    HI THERE    8
 4     9    10    HI,THERE   11
```

When you use the DSD INFILE option, the default delimiter is a comma. You may use the DSD and DLM= options together to allow all the features just discussed but with a delimiter other than a comma.

Data Arranged in Columns

One of the most common forms of data entry is to arrange the data values in specified columns. This has several advantages over space or comma delimited data. First, you can pack more data values together without wasting space. Second, you can read only those columns of data that you want, and third, you can read the data values in any order you choose. Let's look at a SAS program that reads data arranged in columns:

```
***COLUMN INPUT;
DATA COL1;
   INPUT X    1-2
         Y      3
         A $ 4-10
         Z     11;
DATALINES;
 12HELLO  3
4 5GOODBYE6
;
PROC PRINT DATA=COL1;
   TITLE 'COLUMN INPUT';
RUN;
```

The rules are very simple. You list each of the variable names followed by the starting and ending columns (or just the starting column if there is only one column). You also place a dollar sign after any variable name that will hold character data. Notice the value of X in the two lines of data in this example. It doesn't matter whether this value is right adjusted (placed in the right-most columns of the field) or not. Good programming practice dictates that numbers should be right adjusted but SAS will read numbers correctly regardless.

Reading Only Selected Variables

As we mentioned earlier, once you have arranged your data values in columns, you can read only those variables that you need. So, using the same data lines as above, here is a program that only reads values for variables X and Z:

```
***COLUMN INPUT (SELECTED VARIABLES);
DATA COL2;
   INPUT X 1-2
         Z 11;
DATALINES;
 12HELLO  3
4 5GOODBYE6
;
PROC PRINT DATA=COL2;
   TITLE 'COLUMN INPUT';
RUN;
```

It's just as easy as that. You can also read these variables in any order you choose as demonstrated in the program below:

```
***COLUMN INPUT (DIFFERENT ORDER);
DATA COL3;
   INPUT Y      3
         A $ 4-10
         Z    11
         X    1-2;
DATALINES;
 12HELLO  3
4 5GOODBYE6
;
```

```
PROC PRINT DATA=COL3;
   TITLE 'COLUMN INPUT';
RUN;
```

Using Pointers and INFORMATS to Read Data

As an alternative to using column specifications, you can use column pointers and INFORMATS. This method is actually more flexible than using column specifications since you can use a SAS INFORMAT or a user-defined INFORMAT to specify how a data value is to be read. The example that follows uses almost the same data arrangement as the program that used column specifications except for the addition of a date value. We added that to demonstrate the advantage of this method.

```
***POINTERS AND INFORMATS;
DATA INFORM1;
   INPUT @1  X    2.
         @3  Y    1.
         @4  A    $7.
         @11 Z    1.
         @12 DATE MMDDYY10.;
   FORMAT DATE DATE9.;
DATALINES;
 12HELLO  310/21/1946
4 5GOODBYE611/12/1997
;
PROC PRINT DATA=INFORM1;
   TITLE 'POINTERS AND INFORMATS';
RUN;
```

The @*n* symbols are columns pointers. For example, @3 says to move to column 3. The INFORMAT following the variable name tells the program how many columns to read and how to read the data value. The INFORMAT n. indicates a numeric variable occupying n columns. The $n. INFORMAT is used for character variables and the MMDDYY10. INFORMAT converts dates in the form MM/DD/YYYY to a SAS date value (that's a topic for another talk). We chose MMDDYY10. instead of the more traditional MMDDYY8. because the year 2000 is fast approaching and, even with the YEARCUTOFF option, it is a good idea to use 4-digit years.

Using INFORMATS with List-Directed Input

You may want to read list-directed or some form of delimited data and still provide an INFORMAT. For example, you may want to read a character variable more than 8 bytes in length or one of the data values might be a date which needs an INFORMAT to be read correctly. Here comes the colon modifer to the rescue! By placing a colon (:) after the variable name, you can then supply an INFORMAT to be used. The program will search for a delimiter and begin reading the first non-blank data value according to the INFORMAT you supply. Look at the following example:

```
***USING INFORMATS: LIST DIRECTED
   DATA (COLON MODIFIER);
DATA COLON;
   INPUT X :    2.
         Y :    1.
         A :    $11.
         Z :    1.
         DATE : MMDDYY10.;
   FORMAT DATE DATE9.;
DATALINES;
1 2 HELLO 3 10/21/1946
4   5   ARRIVEDERCI  6  11/12/1997
;
PROC PRINT DATA=COLON;
   TITLE 'INFORMATS: COLON MODIFIER';
RUN;
```

Notice that variable A is now 11 bytes and the variable DATE will be a true SAS date.

An Alternate Method for Supplying INFORMATS

An alternative to the program above is preceed the INPUT statement with an INFORMAT statement, associating each of the variables with an INFORMAT. The program below will produce exactly the same data set as the one above. Which method you choose is up to you.

```
***USING INFORMATS: LIST DIRECTED
   DATA (INFORMAT STATEMENT);
DATA INFORM2;
   INFORMAT X 2. Y Z 1. A $11.
            DATE MMDDYY10.;
   INPUT X Y A Z DATE;
   FORMAT DATE DATE9.;
DATALINES;
1 2 HELLO 3 10/21/1946
4   5    ARRIVEDERCI   6   11/12/1997
;
```

```
PROC PRINT DATA=INFORM2;
   TITLE 'INFORMAT STATEMENT';
RUN;
```

Space Delimited Data Values Containing Blanks

What happens if you are using blanks as data delimiters and you have a character value that contains a blank, such as a first and last name? By replacing the colon modifer with an ampersand (&), the system will continue reading a character value, even if it contains single blanks. The program will know that a data value is finished when it encounters two or more blanks. Notice variable A in the program below and the values of "HELLO THERE" and "A BIENTOT" in the data lines. When you use the ampersand modifier, be sure to remember to follow the variable with two or more blanks.

```
***USING INFORMATS: LIST DIRECTED
   DATA (AMPERSAND MODIFIER);
DATA AMPER;
   INPUT X : 2.
         Y : 1.
         A & $11.
         Z : 1.;
DATALINES;
1 2 HELLO THERE  3
4     5     A BIENTOT   6
;

PROC PRINT DATA=AMPER;
   TITLE 'AMPERSAND MODIFIER';
RUN;
```

A Shortcut Way of Specifying Variable Names and INFORMATS

First, look at the rather long and inelegant program below:

```
***WITHOUT VARIABLE AND INFORMAT
   LISTS;
DATA NOINLIST;
   INPUT @1 Q1 1.
         @2 Q2 1.
         @3 Q3 1.
         @4 Q4 1.
         @5 Q5 1.
         @6 A  $1.
         @7 B  $1.
         @8 C  $1.;
DATALINES;
12345XYZ
;
PROC PRINT DATA=NOINLIST;
   TITLE 'WITHOUT VARIABLE AND
INFORMAT LISTS';
RUN;
```

Even beginning SAS programmers know that whenever a program gets tedious to write, there is usually a better and shorter way to accomplish the same thing. In this case, you can use a variable list and an INFORMAT list to shorten the program. Take a look at this program:

```
***VARIABLE AND INFORMAT LISTS (1);
DATA INLIST1;
   INPUT @1 (Q1-Q5 A B C)
            (5*1. 3*$1.);
DATALINES;
12345XYZ
;
PROC PRINT DATA=INLIST1;
   TITLE 'VARIABLE LISTS (1)';
RUN;
```

You can place a list of variables, including the base*n*-base*m* notation (Q1-Q5 for example), in parentheses followed by a list of INFORMATS, also placed in parentheses. The notation 5*1. or 3*$1. means to repeat the INFORMAT 5 or 3 times respectively. Each variable in the variable list is read with the corresponding INFORMAT in the INFORMAT list. If the INFORMAT list is shorter than the variable list, the program will "recycle" the INFORMAT list, that is, go back to the first INFORMAT in the list and go through the list as many times as necessary. You can take advantage of this feature to simplify the program like this:

```
***VARIABLE AND INFORMAT LISTS (2);
DATA INLIST2;
   INPUT @1 (Q1-Q5)(1.)
         @6 (A B C)($1.);
DATALINES;
12345XYZ
;
PROC PRINT DATA=INLIST2;
   TITLE 'VARIABLE LISTS (2)';
RUN;
```

By grouping variables together that use the same INFORMAT, you can list the INFORMAT just once and it will be used for each of the variables in the list.

Skipping Around Using Relative Column Pointers

Suppose you have several X,Y pairs (X1,Y1; X2,Y2; and X3,Y3 for example). You might think the only way to read these data would be an INPUT statement that looked like this:

```
INPUT X1 1 Y1 2 X2 3 Y2 4 X3 5 Y3 6;

OR

INPUT (X1 Y1 X2 Y2 X3 Y3)(1.);
```

Well, there is an easier way. By using a relative columns pointer (a + sign), you can move the column pointer right or left, relative to its last position. You can use this to read each of the X-values first, skipping over the columns occupied by the Y-values and then go back to column 2 (where the Y-values start) and read all the Y-values, skipping over the columns occupied by the X's. Here is the program:

```
***RELATIVE COLUMN POINTERS;
DATA RELATIVE;
   INPUT @1  (X1-X3)(1. + 1)
         @2  (Y1-Y3)(1. + 1);
DATALINES;
123456
;
PROC PRINT DATA=RELATIVE;
   TITLE 'RELATIVE COLUMN POINTERS';
RUN;
```

The INPUT statement above says to start reading in column 1 (@1) and then read a single digit value (1.) and then skip a column (+1). Do this for all the X-values. When you are finished, go back to column 2 (@2) and read the Y-values in the same way. If you had more than three X's and Y's, this technique could save you lots of typing.

Reading Data from an External File

So far, all our examples used data lines "in-stream" or part of the program. Most SAS programers, especially when large amounts of data are involved, keep the data separately from the program. It is quite simple to have the program read data from an external raw data file instead of "in-stream" data. First, you use an INFILE statement to point to the data file and, second, you remove the DATALINES statement. It's as easy as that.

There are two ways that an INFILE statement can point to a file. The first is to name the file directly in the INFILE statement as follows:

```
***READING FROM AN EXTERNAL FILE
   (METHOD 1);
DATA EXTERN1;
   INFILE 'C:\SASTALKS\DATA1';
   INPUT X Y A $ Z;
RUN;

PROC PRINT DATA=EXTERN1;
   TITLE 'DATA IN AN EXTERNAL FILE';
RUN;
```

Notice that the file name is place in quotes (single or double). If the file is in the same subdirectory as the program, you can omit the full path description and just supply the file name. We prefer that you include the entire path and file name as above so that the program is transportable and can be run from another subdirectory without changes. A second method of supplying the file name is to first define a fileref (file reference) with a FILENAME statement. This fileref is then named on the INFILE statement. The very important difference between this method and the previous method is that the fileref is not placed in quotes on the INFILE statement. Here is an example of this method of specifying an external file:

```
***READING FROM AN EXTERNAL FILE
   (METHOD 2);
DATA EXTERN2;
   FILENAME PAT 'C:\SASTALKS\DATA1';
   INFILE PAT;
   INPUT X Y A $ Z;
RUN;

PROC PRINT DATA=EXTERN2;
   TITLE 'DATA IN AN EXTERNAL FILE';
RUN;
```

INFILE Options

Whether you are reading data following a DATALINES statement or from an external file, you may need to exercise some control on how the data values are read. There are a number of INFILE options that are useful. We will only discuss a few here. If you read complicated data structures or read data from multiple files,

you will want to investigate all the possible INFILE options available to you.

The first option we will show you concerns data lines, either "in-stream" or from external files, that may be missing data values at the end of a line of data. In the example below, line 2 only contains 3 values (for X, Y, and A). When the program attempts to read this list-directed data, it goes to the third line to find a value for Z. Then, when the data step iterates again, the pointer tries to move to the next record but meets an end-of-file instead and the data step stops. To see this clearly, look at the output from PROC PRINT shown following the program.

```
***INFILE OPTIONS;
DATA INOPT1;
   ***PROGRAM WITHOUT MISSOVER;
   INPUT X Y A $ Z;
DATALINES;
1 2 HELLO 3
4 5 GOODBYE
7 8 LAST 9
;
PROC PRINT DATA=INOPT1;
   TITLE 'INFILE OPTIONS';
RUN;
```

```
Infile Options

OBS    X    Y       A       Z

 1     1    2    HELLO      3
 2     4    5    GOODBYE    7
```

To solve the problem of missing data at the end of a line, when using list-directed data entry, use an INFILE option called MISSOVER. The program below uses the reserved fileref DATALINES on the INFILE statement. If you are reading data from an external file, the MISSOVER option is used in the same way. This option instructs the program to set any variables to a missing value if the end of a line (or record) is reached and values have not been read for all the variables in the INPUT list. Look at the program below and the resulting output.

```
***INFILE OPTIONS: MISSOVER;
DATA INOPT2;
   ***PROGRAM WITH MISSOVER;
   INFILE DATALINES MISSOVER;
   INPUT X Y A $ Z;
DATALINES;
1 2 HELLO 3
4 5 GOODBYE
7 8 LAST 9
;
PROC PRINT DATA=INOPT2;
   TITLE 'INFILE OPTIONS';
RUN;
```

```
Infile Options

OBS    X    Y    A          Z

 1     1    2    HELLO      3
 2     4    5    GOODBYE    .
 3     7    8    LAST       9
```

With column oriented data or when you use pointers and INFORMATS, the same problem can occur. The PAD option is the best choice for resolving this problem. This option says to PAD out short data lines with blanks to the length of the logical record (set by default or by the LRECL option).

```
***INFILE OPTIONS: PAD;
DATA INFORM;
   INFILE DATALINES PAD;
   INPUT  X    1-2
          Y      3
          A $ 4-10
          Z     11;
DATALINES;
 12HELLO  3
4 5GOODBYE
 78LAST   9
;
PROC PRINT DATA=INFORM;
   TITLE 'USING THE PAD OPTION';
RUN;
```

Reading Data "Blindly"

Suppose you have a data file and are not given information on the data layout. Are you dead in the water, finished, kaput? Not at all. Use the strange looking program below to first take a look at the file. Notice that there are no variables listed on the INPUT statement. This form of the INPUT statement will read an entire logical record (or line of data) but will not assign any values to variables. This does not seem very useful except for the fact that you can use a PUT statement with the keyword _INFILE_ to write out the contents of the data record to the log (the default output location) or to your

output window (by using a FILE PRINT statement). Look at the program below and the resulting log file. (Note: this form of INPUT statement is obviously useful when you are reading data from and external file. The in-stream data in this example is for illustrative purposes only.)

```
***NO VARIABLES LISTED;
DATA READIT;
   INFILE DATALINES;
   INPUT;
   PUT _INFILE_;
DATALINES;
12345ABCDEXYZ
1122334455667
 12HELLO  310/21/1946
4 5GOODBYE611/12/1997
;
```

Here is a listing of the SAS log after running the program above:

```
44   DATA READIT;
45      INFILE DATALINES;
46      INPUT;
47      PUT _INFILE_;
48   DATALINES;

12345ABCDEXYZ

1122334455667

 12HELLO  310/21/1946

4 5GOODBYE611/12/1997

NOTE: THE DATA SET WORK.READIT HAS 4
OBSERVATIONS AND 0 VARIABLES.
NOTE: THE DATA STATEMENT USED 0.17 SECONDS.
```

Using More than One INPUT Statement: The Single Trailing @

There are times when you want to be able to read one or more values from a data line and then, depending on the value, decide how to read the remaining values. In the example that follows, there are two data layouts in a single file. Type 1 records have age in columns 1 and 2; type 2 records have age in columns 3 and 4. The record type is stored in column 6. You want to be able to read the record type first and then decide where to read the age value. If you use two INPUT statements like this:

```
INPUT @6 TYPE $1.;
IF TYPE = '1' THEN INPUT AGE 1-2;
ELSE IF TYPE = '2' THEN INPUT AGE
3-4;
```

it will not work. After the first INPUT statement is executed, the pointer moves automatically to the next line. Age values are then read from the wrong line. You want to tell the program to "hold the line" after reading the value for TYPE. You do this with a single trailing @ as shown next:

```
***WHERE IT'S @;
DATA TRAILING;
   INPUT @6 TYPE $1. @;
   IF TYPE = '1' THEN INPUT AGE 1-2;
   ELSE IF TYPE = '2' THEN
      INPUT AGE 3-4;
   DROP TYPE;
DATALINES;
23   1
  44 2
;
PROC PRINT DATA=TRAILING;
   TITLE 'SINGLE TRAILING @';
RUN;
```

After a value is read for TYPE, the trailing single @ tells the program not to go to the next data line for the next INPUT statement in the data step. When the data step finishes, the pointer will then move to the next line of data for another iteration of the data step.

A very useful application of the single trailing @ is to decide whether to read a line of data or not. For example, suppose you only want to read data on females from a raw data file. One way to do this is to read a line of data in the usual way and to delete all observations where the value of GENDER is not equal to 'F'. A much more efficient way is to first read the value of GENDER and only read the remaining values if GENDER is female. Here is how it's done.

```
***ANOTHER @ EXAMPLE;
DATA TRAIL2;
   INPUT @1 GENDER $1. @;
   IF GENDER NE 'F' THEN DELETE;
      INPUT @3 AGE 2.
            @5 HEIGHT 2.;
DATALINES;
M 2368
F 4462
;
PROC PRINT DATA=TRAIL2;
   TITLE 'ANOTHER @ EXAMPLE';
RUN;
```

If there are a lot of variables in each observation this method can save considerable computer time. Remember that when the DELETE statement is executed, control returns to the top of the DATA step.

Creating Several Observations from One Line of Data: The Double Trailing @

What if you want to create several observations from a single line of data? In the example below, several X,Y pairs are placed on a single line to save space. A single trailing @ would not help here. After a value was read for X and Y, the data step would iterate again and the pointer would move to a new line. Try it. You will have only two observations, the first with X=1, Y=2 and the second with X=7, Y=8. To hold the line for multiple iterations of the data step, use the double trailing @ as shown in the examle below:

```
***WHERE IT'S REALLY @@;
DATA DOUBLE;
   INPUT X Y @@;
DATALINES;
1 2 3 4 5 6
7 8
;
PROC PRINT DATA=DOUBLE;
   TITLE 'DOUBLE TRAILING @';
RUN;
```

Reading Multiple Lines of Data for a Single Observation

What do you do when you have more than one line of data for each observation you want to create? You use a line pointer #. In the example below, there are two lines of data for each observation. The line pointer, #, is used to tell the program that ID and DOB are on the first line and HEIGHT and WEIGHT are on the second line.

```
***READING MULTIPLE LINES FOR ONE
OBSERVATION;
DATA MULT1;
   INPUT #1  @1 ID $11.
             @13 DOB MMDDYY8.
         #2  @5 HEIGHT 2.
             @8 WEIGHT 3.;
   FORMAT DOB MMDDYY10.;
DATALINES;
123-45-6789 10211946
    68 158
253-65-5455 11111960
    62 102
;
PROC PRINT DATA=MULT1;
   TITLE 'READING MULTIPLE LINES';
RUN;
```

If you only want to read the first two lines of data but you have more than two lines of data for each subject, make sure you include a #n at the end of your INPUT statement, where n is the number of data lines for each subject (it **must** be the same for each subject). The program below demonstrates this with 4 lines of data for each subject with values being read from only the first 2.

```
***READING MULTIPLE LINES FOR ONE
OBSERVATION;
DATA MULT2;
   INPUT #1  @1 ID $11.
             @13 DOB MMDDYY8.
         #2  @5 HEIGHT 2.
             @8 WEIGHT 3.
         #4;
   FORMAT DOB MMDDYY10.;
DATALINES;
123-45-6789 10211946
    68 158
9879876987698769876987
0987098709870987098709
253-65-5455 11111960
    62 102
9876987698769876987698
0987098709870987098709
;
PROC PRINT DATA=MULT2;
   TITLE 'READING MULTIPLE LINES';
RUN;
```

Supressing Error Messages in the SAS LOG

For our last two examples, we will show you how to eliminate error messages from being written to the SAS log when bad data values, such as character data in a numeric field, are encountered. In the example that follows, values of 'NA' and '?' indicate that a value was missing. A normal INPUT statement generates NOTES and listings of the offending data lines in the SAS log. The following program generates such error messages:

```
***SUPRESSING ERROR MESSAGES;
DATA ERROR;
   INPUT X Y Z;
DATALINES;
```

```
1 NA 3
4 5 ?
;
PROC PRINT DATA=ERROR;
   TITLE 'SUPRESSING ERROR MESSAGES';
RUN;
```

Here is a listing of the SAS log from the program above:

```
56   ***Supressing Error Messages;
57   DATA ERROR;
58      INPUT X Y Z;
59   DATALINES;

NOTE: Invalid data for Y in line 60 3-4.
RULE:----+----1----+----2----+----3----+----4--
60   1 NA 3
X=1 Y=. Z=3 _ERROR_=1 _N_=1
NOTE: Invalid data for Z in line 61 5-5.
61   4 5 ?
X=4 Y=5 Z=. _ERROR_=1 _N_=2
NOTE: The data set WORK.ERROR has 2 observations
and 3 variables.
NOTE: The DATA statement used 0.22 seconds.
```

If you know the there are invalid data values, such as NA (not applicable) in a numeric field and you want to avoid all the NOTES in the SAS log, use a ? modifier in your INPUT statement. A single ? following the variable name tells the program to omit the NOTES from the log. Look at the program below and the resulting log.

```
***SUPRESSING ERROR MESSAGES;
DATA NOERROR1;
   INPUT X ? Y ? Z ?;
DATALINES;
1 NA 3
4 5 ?
;
PROC PRINT DATA=NOERROR1;
   TITLE 'SUPRESSING ERROR MESSAGES';
RUN;
```

```
67   ***Supressing Error Messages;
68   DATA NOERROR1;
69      INPUT X ? Y ? Z ?;
70   DATALINES;

RULE:----+----1----+----2----+----3----+----4---
71   1 NA 3
X=1 Y=. Z=3 _ERROR_=1 _N_=1
72   4 5 ?
X=4 Y=5 Z=. _ERROR_=1 _N_=2
NOTE: The data set WORK.NOERROR1 has 2 observa-
tions and 3 variables.
NOTE: The DATA statement used 0.16 seconds.
```

Finally, to completely eliminate both the NOTES and the data listing, use a double ?? following the variable name like this:

```
***SUPRESSING ERROR MESSAGES;
DATA NOERROR2;
   INPUT X ?? Y ?? Z ??;
DATALINES;
1 NA 3
4 5 ?
;
PROC PRINT DATA=NOERROR2;
   TITLE 'SUPRESSING ERROR MESSAGES';
RUN;
```

Notice that the SAS log (following) indicates no errors. Use this with caution! Make sure you understand your data before overriding error messages.

```
78   ***Supressing Error Messages;
79   DATA NOERROR2;
80      INPUT X ?? Y ?? Z ??;
81   DATALINES;

NOTE: The data set WORK.NOERROR2 has 2 observa-
tions and 3 variables.
NOTE: The DATA statement used 0.11 seconds.
```

One final example shows how a double ?? can allow you to read a date field with invalid dates such as 99/99/99, or MISSING, in place of a valid date.

```
***SUPRESSING ERROR MESSAGES;
DATA NOERROR3;
   INPUT  @1 DATE ?? MMDDYY8. @10 X;
   FORMAT DATE MMDDYY8.;
DATALINES;
10/21/46 3
MISSING  4
99/99/99 5
;
PROC PRINT DATA=NOERROR3;
   TITLE 'SUPRESSING ERROR MESSAGES';
RUN;
```

Conclusion

We have demonstrated some of the power and versitility of the seemingly simple INPUT statement. Remember that the INPUT statement is one of the strengths of the SAS language that

allows you to read such diverse types of data easily.

Ronald P. Cody, Ed.D.
Robert Wood Johnson Medical School
Department of Environmental and Community Medicine
675 Hoes Lane
Piscataway, NJ 08822
(732)235-4490
cody@umdnj.edu