

# Advanced Techniques for Reading Difficult and Unusual Flat Files

Kim L. Kolbe Ritzow of Systems Seminar Consultants, Kalamazoo, MI

## Abstract

This paper will discuss the various tricks and techniques that can be used to read more difficult types of flat files. Topics to be discussed will include: reading multiple layout files, strategies for reading hierarchical files, advanced input pointers, and the various uses of the SCAN, INDEX and INDEXC functions to assist in reading data.

## Selectively Reading Data

The single trailing @ used on the INPUT statement is not only useful for selectively reading data, but is also an important tool in reading multiple layout files.

One way to use the single trailing @ is to selectively read data. The following code is not bad, but it is also not very efficient because it reads in all the fields and then determines if it wants the record or not:

```
DATA CONSUMER;
INFILE 'C:\FLATFILE\VEAT.DAT';
INPUT
  @1 ID          $5.
  @6 GENDER      $1.
  @7 AGE         2.
  @9 MEAL1       $2.
  @11 MEAL2      $2.
  @13 MEAL3      $2.;
IF GENDER='1' AND MEAL1='30';
RUN;
```

However, the following code using the single trailing @ to selectively read the data is a lot more efficient than the previous code:

```
DATA CONSUMER;
INFILE 'C:\FLATFILE\VEAT.DAT';
INPUT
  @6 GENDER      $1.
  @9 MEAL1       $2. @;
IF GENDER='1' AND MEAL1='30';
INPUT
  @1 ID          $5.
  @7 AGE         2.
  @11 MEAL2      $2.
  @13 MEAL3      $2.;
RUN;
```

This code is more efficient because it first reads in the necessary fields and holds the record with the single trailing @. The record is then checked with the subsequent logic to determine if the record should be kept or not. If the record meets the criterion it will continue on to the next INPUT statement and will eventually be output to the data set via the implicit output at the bottom of the Data Step. If the record does not meet the criterion, it continues no further and control is sent back up to the top of the data step to grab a new record. Thus, minimizing the amount of initial information that is read in before the data is subset.

## Reading Multiple File Layouts

Another way in which the single trailing @ can be useful is to read flat files which contain multiple file layouts within a single flat file. This type is identified by the fact that each line requires a different type of input statement and that there is no predictability to the order in which the record will appear (the first record could be a type 15 record, and then a type 5 record and another time there could be two type 5 records first and then a type 21 record, and so on).

```
05 PRODPAYR J1981 05/19/91 10:31:02
15 PROD.MONTHLY.PAYROLL 3380
15 PROD.MONTHLY.BACKUP 3400
21 1B1 145664
05 TSTCOMPL J1982 05/19/91 10:32:02
15 SYS1.LINKLIB 3380
```

```
DATA JOBS(KEEP=JNAME JNUM JDATE
           JTIME)
```

```
  DSNS(KEEP=DSN UNIT)
  TAPE(KEEP=ADDR VOLUME);
INFILE 'C:\FLATFILE\SYSTEM.DAT';
INPUT @1 TYPE $2. @;
IF TYPE='05' THEN
  DO;
    INPUT @4 JNAME $8.
          @13 JNUM $5.
          @19 JDATE MMDDYY8.
          @28 JTIME TIME8.;
    OUTPUT JOBS;
  END;
ELSE IF TYPE='15' THEN
  DO;
    INPUT @4 DSN $22.
          @28 UNIT $4.;
```

```

OUTPUT DSNS;
END;
ELSE IF TYPE='21' THEN
DO;
INPUT      @4 ADDR  $3.
           @13 VOLUME $6.;
OUTPUT TAPE;
END;
RUN;

```

### Reading Multi-Line Per Observation Files

Multi-line per observation files are similar to multiple layout files (demonstrated above) in that both types of files have differing types of record layouts for each line of data. The main difference between the two is that a multi-line per observation file will have the same number of lines for each observation and they will always come in the same, predictable order. The multiple layout files on the other hand, do not always have the same number of lines, nor do they have any predictability in the order in which the information will appear. Because of the predictability in the multi-line per observation files, they are relatively easy to read.

An example of what the data looks like coming in:

MARY	F	29
Y	N	132
109	3	
JOHN	M	38
N	N	280
003	9	

Multi-line per observation files, can be read one of three ways:

1. Separate input statements for each line:

```

DATA CONSUMER;
INFILE 'C:\FLATFILE\EAT2.DAT';
INPUT
  @1 NAME      $5.
  @7 GENDER    $1.
  @9 AGE       2.;

INPUT
  @1 MED_DIET  $1.
  @3 HEL_DIET  $1.
  @5 WEIGHT    3.
  @9 DIABETIC  $1.;

INPUT
  @1 MEAL1     $3.
  @5 BEV1      $1.;
RUN;

```

2. One input statement, one semicolon, and forward slash pointers:

```

DATA CONSUMER;
INFILE 'C:\FLATFILE\EAT2.DAT';
INPUT
  @1 NAME      $5.
  @7 GENDER    $1.
  @9 AGE       2. /
  @1 MED_DIET  $1.
  @3 HEL_DIET  $1.
  @5 WEIGHT    3.
  @9 DIABETIC  $1. /
  @1 MEAL1     $3.
  @5 BEV1      $1.;
RUN;

```

3. One input statement, one semicolon, and pound pointers (this method is the most flexible because you can selectively read cards of data in, but it also is the least efficient because the input buffer contains all three lines of data at once whereas the other two methods only ever have one line at a time in the buffer):

```

DATA CONSUMER;
INFILE 'C:\FLATFILE\EAT2.DAT';
INPUT #1
  @1 NAME      $5.
  @7 GENDER    $1.
  @9 AGE       2.

#2
  @1 MED_DIET  $1.
  @3 HEL_DIET  $1.
  @5 WEIGHT    3.
  @9 DIABETIC  $1.

#3
  @1 MEAL1     $3.
  @5 BEV1      $1.;
RUN;

```

Regardless of which style of input statement you use, the resulting output SAS data set will look the same.

### Hierarchical Files

Hierarchical files are simply a variation on the multi-line per observation files (as demonstrated above). Multi-line per observation files always have a fixed number of line per observation (in my example three lines per observation, but it could have just as easily been five lines or six lines of information for each observation. The key here is that it is always the same number of lines- whatever the number may be). Hierarchical files on the other hand, have

a varying number of lines per observation. One observation may have two lines of the data, the next observation may have four lines, and so on. Because of this variability, they are much like the multiple layout files that we saw a while back, but are different in that we are dealing with A SINGLE OBSERVATION which has a varying number of lines associated with it, whereas in the multiple layout files, there were different types of observations residing on the same incoming data set that were split off into separate outgoing SAS data sets after having been read in.

Hierarchical files usually have a record type (or card reference) on each record to help distinguish what type of record, or card, you are dealing with. Hierarchical files typically consist of a header record and a varying number of detail records.

An example of what the data looks like coming in:

1345601 29 Y N 134 Y	<=== Header record
1345602 004 1	<=== Detail record
1345602 098 5	<=== Detail record
1345602 111 9	<=== Detail record
3989201 63 Y Y 290 N	<=== Header record
3989202 208 1	<=== Detail record
3989202 035 1	<=== Detail record
3838201 43 N N 132 N	<=== Header record
3838202 084 1	<=== Detail record

```
DATA CONSUMER;
INFILE 'C:\FLATFILE\EAT3.DAT';
INPUT
    @1 ID                $5.
    @6 TYPE              $2. @;
IF TYPE='01' THEN
DO;
    RETAIN AGE MED_DIET HEL_DIET
    WEIGHT DIABETIC;
    INPUT
        @9 AGE            2.
        @12 MED_DIET     $1.
        @14 HEL_DIET     $1.
        @16 WEIGHT       3.
        @20 DIABETIC     $1.;
    END;
ELSE IF TYPE='02' THEN
DO;
    INPUT
        @9 MEAL          $3.
        @13 BEV          $1.;
    OUTPUT;
    END;
DROP TYPE;
RUN;
```

The key to reading this type of file is to first identify what type of record you have (by reading in that field, holding it and checking its value), making sure you RETAIN the header record's information so it will appear on the same line as the detail record's information, and making sure you output the record each time after you have read in a detail record.

This solution will work for those types of hierarchical files which have a header record and a varying number of detail records each on a separate line. As a result, this code will "flatten out" the hierarchical structure into a file with one line for each detail record with the header record information being repeated on each of the lines of detail data.

An example of what the data looks like by the time it makes it to the SAS data set:

13456 29 Y N 134 Y 004 1
13456 29 Y N 134 Y 098 5
13456 29 Y N 134 Y 111 9
39892 63 Y Y 290 N 208 1
39892 63 Y Y 290 N 035 1
38382 43 N N 132 N 084 1

### Other Variations on a Hierarchical File

Hierarchical files can come in different varieties. For example, rather than having one header record with a varying number of detail records on separate lines as we saw in the previous example, one could also have a hierarchical file that has a header record with a varying number of detail records on the SAME line as the header record. Typically these files are not fixed block in their length, but rather, are variable length files.

There are many different ways in which these types of hierarchical files can be read. Which way you read them, largely depends on how the file was created and what information you can use to key off of to determine if you are on a header or detail record, or when there are no more detail records left to be read. The following are some of the possible scenarios for these types of files:

1. In the header record there is information indicating how many detail records are to be expected:

An example of what this type of data might look like:

```
13456 29 Y N 134 Y 4 082 3 004 1 098 5 111 9
39892 63 Y Y 112 N 3 011 5 104 8 051 3
38382 43 N N 221 N 2 131 2 021 8
```

```
DATA CONSUMER;
INFILE 'C:\FLATFILE\EAT4.DAT';
INPUT
  @1 ID          $5.
  @7 AGE        2.
  @10 MED_DIET $1.
  @12 HEL_DIET  $1.
  @14 WEIGHT    3.
  @18 DIABETIC $1.
  @20 HOWMANY  2. @;
DO I=1 TO HOWMANY;
INPUT
  MEAL          $CHAR4.
  BEV           $CHAR2. @;
OUTPUT;
END;
DROP I HOWMANY;
RUN;
```

In this solution, it is important to hold the record at the end of reading in the header record because the detail information will be located on the same line and a subsequent INPUT statement has to be coded with a DO loop surrounding it. The DO loop will simply control how many times the detail information will be read on a given line.

On the second INPUT statement (the one within the DO loop), it is EXTREMELY important that the INPUT statement not be coded with any pointers or column references. If it is, you will hang the pointer up in a given spot and you will never read in anything but the first detail record's information. It is the width of the informat that advance the pointer. It is also extremely important that within this second INPUT statement a single trailing @ be coded at the end of the INPUT statement so that the current line will be held in order for additional detail records to be read. If the single trailing @ is forgotten, the record will be released and information from the subsequent observations will appear on the same line as the first observations data on the output SAS data set. The current line will be released at the end of the DO loop.

Finally, it is important that the record be OUTPUT within the DO loop so that each detail record will be output, one at a time. A RETAIN is not required in this solution, as in the previous solution, because the header record information is on the same line as the detail record's information, whereas in the previous example it was physically located on a

different line and therefore its value needed to be retained to get the information on the same line as the detail record prior to outputting the data.

An example of what this data looks like by the time it makes it to the SAS data set:

```
13456 29 Y N 134 Y 082 3
13456 29 Y N 134 Y 004 1
13456 29 Y N 134 Y 098 5
13456 29 Y N 134 Y 111 9
39892 63 Y Y 112 N 011 5
39892 63 Y Y 112 N 104 8
39892 63 Y Y 112 N 051 3
38382 43 N N 221 N 131 2
38382 43 N N 221 N 021 8
```

- Another variation on this type of file is that it may tell you when there ARE NO MORE records rather than how many records to expect:

An example of what this type of data might look like:

```
13456 29 Y N 134 Y 082 3 004 1 098 5 111 9 $
39892 63 Y Y 112 N 011 5 104 8 051 3 $
38382 43 N N 221 N 131 2 021 8 $
```

```
DATA CONSUMER;
INFILE 'C:\FLATFILE\EAT5.DAT';
INPUT
  @1 ID          $1.
  @7 AGE        2.
  @10 MED_DIET  $1.
  @12 HEL_DIET  $1.
  @14 WEIGHT    3.
  @18 DIABETIC $1. @;
DO UNTIL(EOFLAG='$');
INPUT MEAL      $
      BEV       $
      EOFLAG    $2. +(-1) @;
OUTPUT;
END;
DROP EOFLAG;
RUN;
```

Here too, it's important that the second INPUT statement does not contain any pointer or column references and that the OUTPUT statement is coded within the DO loop. The use of a DO UNTIL

is important, because it does not check the condition until the BOTTOM of the DO loop, at which time the indicator being checked has already been read in.

While the initial data coming in to this data step (EAT5.DAT) looks slightly different than the data in the previous example (EAT4.DAT), the resulting output SAS data set would look exactly the same because the data step's syntax was adjusted accordingly to accommodate the different structure of the data coming in.

3. Finally, you know neither how many detail records to expect, nor when there are no more detail records. Rather, you must figure this out yourself:

An example of what this type of data might look like:

```
13456 29 Y N 134 Y 082 3 004 1 098 5 111 9
39892 63 Y Y 112 N 011 5 104 8 051 3
38382 43 N N 221 N 131 2 021 8
```

```
DATA CONSUMER;
INFILE 'C:\FLATFILE\EAT6.DAT'
LENGTH=LRECL;
INPUT
@1 ID $5.
@7 AGE 2.
@10 MED_DIET $1.
@12 HEL_DIET $1.
@14 WEIGHT 3.
@18 DIABETIC $1. @;
HOWMANY=INT((LRECL-18)/5);
DO I=1 TO HOWMANY;
INPUT MEAL $
BEV $ @;
OUTPUT;
END;
DROP I HOWMANY;RUN;
```

The LENGTH= option on the INFILE statement will return to the pseudo-variable you defined as LRECL which will contain the length of each record at compile time. Knowing that each header record has a fixed length of 18 bytes, or characters, it can be subtracted from the total length of the record and then divide by the length of the detail records to determine how many detail records are on the file (notice one the number of detail records is determined, the solution is much like what was done when that information is actually provided on the file, as we saw in an earlier example).

While the initial data coming in to this data step (EAT6.DAT) looks slightly different than the data in the previous example (EAT5.DAT), the resulting output SAS data set would look exactly the same because the data step's syntax was adjusted accordingly to accommodate the different structure of the data coming in.

### Variable Length Files

Sometimes hierarchical files are the source of variable length files, but not all variable length files have to be hierarchical. A variable length file has differing lengths for each record on the file. The records physically end in different positions. Without some special tricks and techniques these files can be difficult to read. There are several ways in which a variable length file can be read, but one of the easiest ways is to use the LRECL= option in addition to the PAD option on the INFILE statement. The purpose of the LRECL= option is to specify the length of the longest record being read in (this information can usually be found with the documentation on the file or through the use of some system utility). The PAD option will then pad out the remaining bytes on the file out to the length specified on the LRECL= with blanks. These two options essentially turn an

otherwise variable length file into a fixed length file which we can then read like we would any other fixed length file.

An example of what this type of data might look like:

```
13456 29 Y N 134 Y CHANGED MEDS
39892 63 Y Y 112 N FOLLOW UP IN 2 MOS
38382 43 N N 221 N CHECK HDLS
```

```
DATA CONSUMER;
INFILE 'C:\FLATFILE\EAT7.DAT'
LRECL=45 PAD;
INPUT
@1 ID $5.
@7 AGE 2.
@9 MED_DIET $1.
@10 HEL_DIET $1.
@12 WEIGHT 3.
@18 DIABETIC $1.
@20 DESCRIB1 $24.;
RUN;
```

### Free Form Files

A lot of times people confuse variable length files with free form files, each of which require different

types of solutions in order to read them. Variable length files typically have columns of information that start in the same place but the last variable's value physically ends in a different place for each record. A free form file, on the other hand, has information that does not begin and end in nice, neat columns but does have the same length for each record. This type of data requires the use of list input and oftentimes, a bunch of other special tricks and techniques to get the data read in correctly.

An example of what this type of data might look like:

```
SUE H 12 3239.12 123.83
JENNIFER S 14 5692.10 388.38
MARK H 24 1928.23 28.12
```

```
DATA SOFTSALE;
  INFILE 'C:\FLATFILE\SOFTSALE.DAT';
  INPUT NAME $
         DIVISION $
         YEARS
         SALES
         EXPENSE;
RUN;
```

This style of input does assume that the fields are blank delimited. If they are delimited by something other than a blank, such as commas and double quotes (as if often the case when reading in an export file from a spreadsheet package like Lotus, Excel, or Symphony) then the DLM= option needs to be used on the INFILE statement:

```
"SUE", "H", 12, 3239.12, 123.83
"JENNIFER", "S", 14, 5692.10, 388.38
"MARK", "H", 24, 1928.23, 28.12
```

```
DATA SOFTSALE;
  INFILE 'C:\FLATFILE\SOFTSAL2.DAT'
    DLM=",";
  INPUT NAME $
         DIVISION $
         YEARS
         SALES
         EXPENSE;
RUN;
```

#### Tab Delimited Files

Files that are tab delimited can be read one of two ways by either using the DLM= or the EXPANDTABS options on the INFILE statement.

If you fail to accommodate for the tab delimiters in the file, the tabs will prematurely release the current line's information even before its done reading in all of its data.

```
SUE H 12 3239.12 123.83
JENNIFER S 14 5692.10 388.38
MARK H 24 1928.23 28.12
```

These types of files are often misleading because they **look like** they are blank delimited when they really are not. If you use the DLM= option on the INFILE statement to handle this problem, what will be type in quotes is the hex equivalent of a tab which is a non-displayable character in order to read the data:

```
DATA SOFTSALE;
  INFILE 'C:\FLATFILE\SOFTSAL3.DAT'
    DLM=' ';
  INPUT NAME $
         DIVISION $
         YEARS
         SALES
         EXPENSE;RUN;
```

Another possibility is to use the EXPANDTABS option on the INFILE statement. This option is only useful in reading files that contain tabs which were created on the same host system as it's trying to be read in on:

```
DATA SOFTSALE;
  INFILE 'C:\FLATFILE\SOFTSAL3.DAT'
    EXPANDTABS;
  INPUT NAME $
         DIVISION $
         YEARS
         SALES
         EXPENSE;
RUN;
```

#### Advanced input pointers

There are a variety of advanced features available on the input statement which can help read certain types of data:

```
N=5;
INPUT @N NAME $CHAR10.;
```

```
N=2;
INPUT @(N+3) NAME $CHAR10.;
```

```
INPUT @'Mr.' NAME $CHAR10.;
```

```
SEARCH='Mr.';
INPUT @SERACH NAME $CHAR10.;
```

```
SEARCH='Mr';
INPUT @(SEARCH!!'.') NAME $CHAR10.;
```

```
N=4;
INPUT +N NAME $CHAR10.;
```

```
INPUT +(-5) NAME $CHAR10.;
```

We also have the ability to use formatted lists with pointers, which are useful when you need to read in a bunch of data that has an identifiable and repetitive pattern associated with it:

```
AK065 AZ101 AL092 AR078.....WY054
```

```
DATA TEMPS;
  INFILE 'C:\FLATFILE\TEMPDATA.DAT';
  INPUT @1 (STATE1-STATE50) ($2. +4)
        @3 (TEMP1-TEMP50) ( 3. +3);
RUN;
```

### Using the \$VARYING Informat

One client I currently work for reads in a lot of data that isn't truly a flat file, rather it's a report stored as a text file. These files come either from the user creating them by using PROC PRINTTO (a SAS procedure used to capture procedural output see the bibliographic reference for "Customized Report Writing Using PROC PRINTTO" for an example of why they might be doing this); or, these files can come from some other software that the company owns which allows them to track competitor's sales. This software product cannot build a flat file of the information, rather it builds a report which the analyst then saves to disk. Once its been saved to disk it is then reads in, a SAS data set is built, and it is merged with other internal information for reporting purposes.

Reading reports as an incoming flat file can sometimes be tricky, but using the \$VARYING informat can make the process a lot easier.

Lets first take the case attempting to read in procedural output created by using PROC PRINTTO (see **Tables 1 and 2** for examples of what the data looks like coming in and the code used to read it in, respectively).

In this example, each line of the report's text is read in as a varying length record. The \$VARYING200. with the LRECL option says, "read to a length of

200 or to the length of the record, whichever comes sooner". Then, through the use of SCAN function, the line of text is parsed apart and macro variables are created as a result.

Obviously, whenever scanning the text of a report, as is being done in this example, one runs the risk of the report's format changing and the code suddenly no longer working. However, sometimes when you need access to a certain statistic's value that you cannot otherwise obtain through a SAS data set, this may be the next best alternative.

***A couple more examples of the \$VARYING informat using the INDEX and INDEXC functions will be shown during the presentation, but will not appear within the paper due to limitations on the number of pages.***

### In Summary

This paper has demonstrated a number of advanced techniques that can be used to read more difficult types of flat files.

I have never, in my many years of SAS programming, found a flat file that I cannot read in SAS. Oftentimes, it just requires rethinking the problem, approaching it in a different and perhaps unfamiliar way, and introducing some new and unusual techniques to get the data read in correctly. When in doubt, ask for some help and chances are someone can come up with alternate idea as to how to approach the problem.

### Trademark Notice

SAS is a registered trademark of the SAS Institute Inc., Cary, NC, USA and other countries.

### Useful Publications

Kolbe Ritzow, Kim (1996), "Customized Report Writing Using PROC PRINTTO", Proceedings of the 7th Annual MidWest SAS Users Group Conference

Any questions or comments regarding the paper may be directed to the author:

Kim L. Kolbe Ritzow  
Systems Seminar Consultants  
Kalamazoo Office  
927 Lakeway Avenue  
Kalamazoo, MI 49001  
Phone: (616) 345-6636  
Fax: (616) 345-5793  
E-mail: KRITZOW@AOL.COM

Model: MODEL1  
Dependent Variable: RESULT

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Value	Prob>F
Model	1	4224476.4575	4224476.4575	1466.742	0.0001
Error	19	54723.35203	2880.17642		
C Total	20	4279199.8095			

  

Root MSE	53.66728	R-square	0.9872
Dep Mean	458.09524	Adj R-sq	0.9865
C.V.	11.71531		

Parameter Estimates

Variable	DF	Parameter Estimate	Standard Error	T for H0: Parameter=0	Prob >  T
INTERCEP	1	62.314059	15.61882273	3.990	0.0008
CONC	1	1.168974	0.03052305	38.298	0.0001

Table 1

```

FILENAME OUTFILE 'C:\OUTFILE';
OPTIONS NOCENTER NODATE;
TITLE;
PROC PRINTTO NEW PRINT=OUTFILE;
RUN;
PROC REG;
MODEL RESULT=CONC;
RUN;
PROC PRINTTO;
RUN;
DATA _NULL_;
INFILE OUTFILE LENGTH=LRECL;
INPUT LINE $VARYING200. LRECL;
IF SCAN(LINE,1)='Root' THEN
  DO;
    CALL SYMPUT('ROOTMSE',SCAN(LINE,3, ' '));
    CALL SYMPUT('RSQUARE',SCAN(LINE,5, ' '));
  END;
ELSE IF SCAN(LINE,1)='Dep' THEN
  CALL SYMPUT('DEP_MEAN',SCAN(LINE,3, ' '));
ELSE IF SCAN(LINE,1)='INTERCEP' THEN
  DO;
    CALL SYMPUT('INTERCPT',SCAN(LINE,3, ' '));
  END;

```

\*point to external flat file ;

\*whenever using PRINTTO its a good idea to turn off centering and suppress the date for ease of scanning later on ;

\*also a good idea to turn off title;

\*turn on capture facility ;

\*capture output of regression ;

\*turn off capture facility ;

\*no need to build SAS data set ;

\*find the length of each rec. read;

\*reads a varying length field ;

\*search string is case sensitive ;

\*assign macro var. values ;

\*third parameter on scan ;

\*function tells SAS only use blanks as valid delimiters ;

```

CALL SYMPUT('INTERLNE',LINE);
IF SCAN(LINE,6,' ') < .05 THEN
  CALL SYMPUT('INTERPRB','YES');
ELSE CALL SYMPUT('INTERPRB','NO');
END;
ELSE IF SCAN(LINE,1)='CONC' THEN
DO;
  CALL SYMPUT('SLOPE',SCAN(LINE,3,' '));
  CALL SYMPUT('SLOPELNE',LINE);
  IF SCAN(LINE,6,' ') < .05 THEN
    CALL SYMPUT('SLOPEPRB','YES');
  ELSE CALL SYMPUT('SLOPEPRB','NO');
END;
RUN;

```

\*write macro vars out to make sure I got  
the right values ;

```

%PUT ROOTMSE=&ROOTMSE RSQUARE=&RSQUARE DEP_MEAN=&DEP_MEAN
INTERCPT=&INTERCPT INTERLNE=&INTERLNE SLOPE=&SLOPE SLOPELNE=&SLOPELNE
INTERPRB=&INTERPRB SLOPEPRB=&SLOPEPRB;

```

Table 2