SAS TECHNIQUES FOR READING VARIABLE LENGTH RECORD FORMATS

Andrew V. Bowden, Jr., South Carolina Electric & Gas Company

INTRODUCTION

In order to compress as much information as possible into a small amount of storage space, many corporate computer departments are now using variable length records in their data bases. These files are typically built, modified and maintained with programs written in machine language or COBOL. In order to access this data with SAS, an analyst or statistician will find it necessary not only to understand the basic concept of variable length files, but to write a program with an enormous amount of flexibility.

This paper will first explain how variable length records are formatted, and then will offer some suggestions on how to write a SAS program to handle this type of data. The techniques that will be presented were first developed for use by analysts at South Carolina Electric and Gas Company, a utility whose customer master file has variable length records up to 5,000 bytes long and provides thirteen months of history on over 500,000 customers.

CONCEPT: VARIABLE LENGTH RECORD FORMATS

Computer departments use a variable length format to shorten the amount of space needed to store one record. For instance, consider a standard record of fixed length with the following layout:

Segment	Positions	Type of Data Stored
1	1 - 75	Customer Identification
2	76 - 350	Credit History
3	351 - 400	Special Information
4	401 - 1400	Electricity Usage History
5	1401 - 2400	Gas Usage History
6	2401 - 3000	Meter Information
7	3001 - 3140	Streetlighting Charges
8	3141 - 3300	Outstanding Transactions

Using this fixed format, a file containing information on 200,000 customers would occupy approximately 660,000 kilobytes of space. However, quite often each record in such a file has a few segments that do not contain any data. (Exactly which segments are blank may vary). In this case, computer programmers might decide to use a variable length record format to discard any empty segments on a record-by-record basis. For instance, if a record did not have any data for segments 5, 6, and 7 in the above layout, the variable length format might compress the information as follows:

Segment	Positions	Type of Data Stored
	1	Indicator Byte
1	2 - 77	Customer Identification
2	78 - 353	Credit History
3	354 - 404	Special Information
4	405 - 1405	Electricity Usage History
8	1406 - 1566	Outstanding Transactions

Then, this particular record would occupy less than half the space that it would have occupied in the fixed length format.

Retrieval programs that access variable length files need two pieces of information to calculate where each segment begins on any record. First, the record must be coded to tell the programs exactly which segments are present. Second, there must be a way to calculate the length of these segments. "Indicator bytes" are typically placed at the start of each record to accomplish the first function. An additional byte is then added to the beginning of each segment to give its length. Using this information and the logic presented in Figure I, a retrieval program can calculate the position of each segment and input the data it contains.

INDICATOR BYTES

In order to conserve space, computer programmers normally use bit-coding for the bytes placed at the beginning of variable length records to indicate which segments are present. In other words, each of the eight bits in one of these "indicator" bytes shows the status of a particular segment. If a segment is present, the corresponding bit is turned on (value of 1); otherwise, it is off (value of 0). Using this approach, three bytes can hold enough information to test for the presence of twenty-four (3 times 8) segments. The following macro performs the bit testing:

```
%MACRO TEST(BYTE);
    %LET BIT = %EVAL(&BYTE);
    IF TEST&BYTE = '1.......'B THEN S&BIT = 1; ELSE S&BIT = 0;
    %LET BIT = %EVAL(&BIT + 1);
    IF TEST&BYTE = '.1......'B THEN S&BIT = 1; ELSE S&BIT = 0;
    %LET BIT = %EVAL(&BIT + 1);
    IF TEST&BYTE = '..1.....'B THEN S&BIT = 1; ELSE S&BIT = 0;
    %LET BIT = %EVAL(&BIT + 1);
    IF TEST&BYTE = '...1....'B THEN S&BIT = 1; ELSE S&BIT = 0;
    %LET BIT = %EVAL(&BIT + 1);
    IF TEST&BYTE = '....1...'B THEN S&BIT = 1; ELSE S&BIT = 0;
    %LET BIT = %EVAL(&BIT + 1);
    IF TEST&BYTE = '.....1..'B THEN S&BIT = 1; ELSE S&BIT = 0;
    %LET BIT = %EVAL(&BIT + 1);
    IF TEST&BYTE = '......1.'B THEN S&BIT = 1; ELSE S&BIT = 0;
    %LET BIT = %EVAL(&BIT + 1);
    IF TEST&BYTE = '.......1'B THEN S&BIT = 1; ELSE S&BIT = 0;
%MEND TEST;
```

Before invoking this macro, the SAS retrieval program must read the indicator bytes. For example, an input might look like:

```
INPUT
/*INDICATOR BYTE FOR SEGMENTS  1-8*/  @1 TEST1  $CHAR1.
/*INDICATOR BYTE FOR SEGMENTS  9-16*/ @2 TEST9  $CHAR1.
/*INDICATOR BYTE FOR SEGMENTS 17-24*/ @3 TEST17 $CHAR1.
@;
```

The variables TEST1, TEST9, and TEST17 can be bit-tested using the above macro. When the macro (also named TEST for convenience) is invoked, the segment to which the first bit corresponds is also specified. The macro then creates eight variables in an array named "S" with values equal to the status of the eight bits. For example:

```
%TEST(1)
%TEST(9)
%TEST(17)
```

are the instructions necessary to test the three indicator bytes that were read in the input statement above. Twenty-four new variables are created (S1-S8, S9-S16, and S17-S24) with values of 1 or 0, depending on whether or not the matching segments are present.

SEQUENTIAL PROCESSING

The easiest way to read a variable length file is to examine each of the segments in order. Beginning with the first segment, the program tests to see whether or not it is present (e.g. - does S1=1 in the above example). If it is not on the record (S1=0), the program goes to the second segment and tests for its existence. However, if the first segment is indeed present, then the program must either skip the positions it occupies or read the data it contains. In both cases, the beginning position of the next segment will need to be calculated. The only way to do this is to know the current segment's length.

Since the length of each segment is often subject to modification as additional information is added, and may sometimes even change from record to record, variable length files typically allocate one packed field at the beginning of the segment to designate its length. When the information contained by a segment is not important, the following macro can be used to skip the appropriate number of spaces:

```
%MACRO SEGMENT(NUM);
 IF (S&NUM   0) THEN DO;
  INPUT @POSITION SEG&NUM PIB1. @;
  POSITION = POSITION + SEG&NUM + (SEG&NUM   0)*1;
 END;
%MEND SEGMENT;
```

Thus, segment 11 can be skipped entirely by invoking the macro and passing the segment number to the instruction:

```
%SEGMENT(11)
```

On the other hand, an INPUT statement can be used to read relevant data from the segment. For example, the following logic uses the length to calculate the starting location of the next

segment after reading the necessary data from Segment 11:

```
IF S11=1 THEN DO;
   INPUT /* LENGTH OF SEGMENT 11 */ @POSITION SEG11 PIB1. @;
   CPOS=POSITION+1; INPUT
   /*   MAIL ADDRESS   */   @CPOS      MAILADD   $CHAR22.
   /*   MAIL TOWN      */              MAILTOWN  $CHAR22.
   /*   MAIL TOWN ZIP*/                MAILZIP   $CHAR9. @;
   POSITION = POSITION + SEG11 + (SEG11  0)*1;
END;
```

Figure II presents a sample program listing which illustrates the sequential processing of variable length files and includes all of the topics discussed in this paper.

CONCLUSIONS

Using SAS to read variable length record formats requires an understanding of: (a) the concept behind record segmentation and compression, (b) the intricacies of such technical tools as indicator bytes, bit testing and informatting, and (c) the need for flexibility. The sample macros listed in this paper provide a simple modular way of constructing SAS programs to handle these files.

The rewards can be tremendous. Analysts and statisticians can use the full power of SAS to manipulate data base information that was previously the almost exclusive domain of machine language and COBOL.

ACKNOWLEDGEMENTS

This paper would not have been possible without the assistance of Mrs. Bettie S. Stinnette, a fellow employee who co-wrote our first practical applications program to employ these concepts and participated in every phase of the project. The excellent Computer Services Department of South Carolina Electric and Gas Company was also very helpful.

FOR FURTHER INFORMATION

Please address questions, suggestions, or comments to:

Mr. Andrew V. Bowden, Jr. (045)
South Carolina Electric & Gas Co.
P O Box 764
Columbia, S.C. 29218

FIGURE I

RETRIEVAL LOGIC FOR VARIABLE LENGTH RECORDS

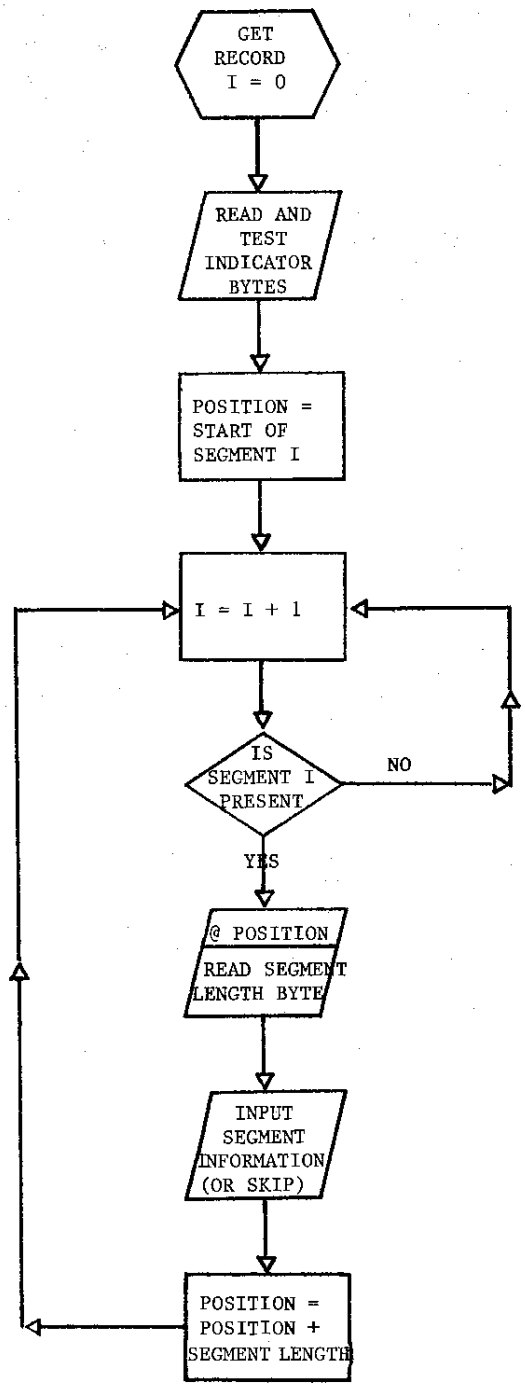

FIGURE II

EXAMPLE OF SAS PROGRAM TO READ VARIABLE LENGTH RECORDS

```
%MACRO TEST(BYTE);
  %LET BIT = %EVAL(&BYTE);
    IF TEST&BYTE='1.......'B THEN S&BIT = 1; ELSE S&BIT=0;
  %LET BIT = %EVAL(&BIT + 1);
    IF TEST&BYTE='.1......'B THEN S&BIT = 1; ELSE S&BIT=0;
  %LET BIT = %EVAL(&BIT + 1);
    IF TEST&BYTE='..1.....'B THEN S&BIT = 1; ELSE S&BIT=0;
  %LET BIT = %EVAL(&BIT + 1);
    IF TEST&BYTE='...1....'B THEN S&BIT = 1; ELSE S&BIT=0;
  %LET BIT = %EVAL(&BIT + 1);
    IF TEST&BYTE='....1...'B THEN S&BIT = 1; ELSE S&BIT=0;
  %LET BIT = %EVAL(&BIT + 1);
    IF TEST&BYTE='.....1..'B THEN S&BIT = 1; ELSE S&BIT=0;
  %LET BIT = %EVAL(&BIT + 1);
    IF TEST&BYTE='......1.'8 THEN S&BIT = 1; ELSE S&BIT=0;
  %LET BIT = %EVAL(&BIT + 1);
    IF TEST&BYTE='.......1'B THEN S&BIT = 1; ELSE S&BIT=0;
  %MEND TEST;


%MACRO SEGMENT(NUM);
 IF (S&NUM > 0) THEN DO;
 INPUT  @POSITION SEG&NUM PIB1. @;
  POSITION = POSITION +  SEG&NUM + (SEG&NUM>0)*1;
  END;
  %MEND SEGMENT;

DATA DDOUT.EXAMPLE;
  INFILE DDIN LENGTH=LENVAR;

INPUT
  /*    ACCOUNT NUMBER          */   @1         ACCNO       $CHAR8.
  /*    NAME                    */   @9         NAME       $CHAR22.
  @ ;

INPUT
  /*    TEST SEGMENT 1-8        */   @31        TEST1       $CHAR1.
  /*    TEST SEGMENT 9-16       */   @32        TEST9       $CHAR1.
  @ ;

 * DO ACTUAL BIT TESTING;
  %TEST(1)
  %TEST(9)

POSITION=33;
```

```
* This is the macro to perform bit testing.
* Macro variable BYTE=1st segment to be tested
* Macro variable BIT set equal to value of BYTE
* SAS variable S&BIT (eg S9) equal to 1st bit
* Increment value of BIT by one
* S&BIT (eg S10) set equal to value of 2nd bit
* Increment value of BIT by one
* Test 3rd bit and set S&BIT (e.g. S11=1 or 0)
* Increment value of BIT by one
* Test 4th bit
* Increment value of BIT by one
* Test 5th bit
* Increment value of BIT by one
* Test 6th bit
* Increment value of BIT by one
* Test 7th bit
* Increment value of BIT by one
* Test 8th bit
* end of macro TEST

* This is the macro to skip unwanted segments
* Macro variable NUM = # of segment skipped
* Checks to see if segment is present
* Reads segment length at start of segment
* Calculates start of next segment
* END statement for IF ... DO statement above
* end of macro SEGMENT

* Output example data set to disk or tape
* Infile statement. LENVAR = length of record
*   (LENVAR is used here only for debugging)
* Input statement
*  Since every record has this information, it
*  is placed before the variable length portion
* Trailing @ holds record for next input

* Input statement to read indicator bytes that
* are bit coded.  The number of the first
* segment indicated by the bytes is used in the
* variable name. Trailing @ holds the record.

* Call TEST macro to perform bit testing of
* the indicator bytes from above.  Passes number
* of segment represented by 1st bit of the byte.

* Starting field of variable length portion
```

```
  POSITION=33;                                                          * Starting field of variable length portion

  IF S1=1 THEN DO;                                                      * Is the first segment present? If so, then
    INPUT /* LENGTH OF SEGMENT 1  */ @POSITION SEG1 PIB1. @;            * read its length from packed field.
    CPOS=POSITION+1;                                                    * Set position of 1st variable in the segment
    INPUT                                                               * Input the relevant information from the
     /*    ACCOUNT STATUS          */    @CPOS      STATUS      $CHAR1.  * segment.  Note that not everything in the
     /*    ADDRESS                 */    +10        ADDRESS    $CHAR22.  * segment must be read.
     /*    TOWN                    */               TOWN       $CHAR22.  *
     /*    ZIP CODE                */               ZIPCODE     $CHAR9.  *
     @ ;                                                                * Trailing @ again holds record.
    POSITION = POSITION +  SEG1 + (SEG1>0)*1;                           * Calculate starting position of next segment.
  END;                                                                  * End corresponding to IF ... DO above.

  IF S2=1 THEN DO;                                                      * Is the second segment present? If so, then
    INPUT /* LENGTH OF SEGMENT 2  */ @POSITION SEG2 PIB1. @;            * Read its length and hold the line with @.
    CPOS=POSITION+8;                                                    * Calculate position of 1st variable of interest
    INPUT                                                               * Input relevant variables
     /*    NAME OVERFLOW           */  @CPOS     NAMEOVER    $CHAR22. @; *
     @ ;                                                                *
    POSITION = POSITION +  SEG2 + (SEG2>0)*1;                           * Calculate starting position of next segment.
  END;                                                                  * End corresponding to IF ... DO above.

  %SEGMENT(3)                                                           * Call macro SEGMENT to skip 3rd segment.

IF S4=1 THEN DO;                                                        * Is the fourth segment present?  If so, then
    INPUT /* LENGTH OF SEGMENT 4 */ @POSITION SEG4 PIB1. @;            * input its length from a packed field.
    CPOS=POSITION+1;                                                    * CPOS is the position of the first variable
    INPUT                                                               * Input all relevant information.
     /*    ADDRESS OVERFLOW        */  @CPOS   ADDOVER     $CHAR22. @ ; *
     @ ;                                                                *Always use trailing @ to hold the record.
    POSITION = POSITION +  SEG4 + (SEG4>0)*1;                           * Calculate start of the next segment.
  END;                                                                  * End corresponding to IF ... DO statement.

  %SEGMENT(5)                                                           * Call macro SEGMENT to skip segments 5,6,7,& 8.
  %SEGMENT(6)                                                           * This macro calculates the beginning of the
  %SEGMENT(7)                                                           * next segment.
  %SEGMENT(8)                                                           *

  IF S9=1 THEN DO;                                                      * Is the ninth segment present.  If so, then
    INPUT /* LENGTH OF SEGMENT 9 */ @POSITION SEG9 PIB1. @;            * input the length of segment nine.
    CPOS=POSITION+25;                                                   * Calculate location of first variable to read.
    INPUT                                                               * Input the desired information.
     /*    ACCOUNT BALANCE           */  @CPDS     BALANCE     PD6.    *
     /*    ACCOUNT ARREARS           */  +2        ARREARS     PD6.    *
     /*    OVERDUE CHARGES           */            OVERDUE     PD4.    *
     @ ;                                                                * Always remember the trailing @ to hold record.
    POSITION = POSITION +  SEG9 + (SEG9>0)*1;                           * Calculate beginning of next segment (not
  END;                                                                  * really necessary since this is the last one).

    INPUT;                                                              * Final input disposes of record being held.

   DROP TEST1 TEST9 S1-S16;                                             * Drop variables specifically created to read
                                                                        * variable length record format.
```