From There to Here: Getting Your Data Into the SAS® System

Andrew T. Kuligowski, Nielsen Media Research
Nancy Roberts, Utah State University

ABSTRACT / INTRODUCTION

The SAS® System has numerous capabilities to store, analyze, report, and present data. However, those features are useless unless that data is stored in or can be accessed by the SAS System. This presentation will provide an overview into some of the methods that can be used to pass data into the SAS System. It will range from Base SAS's INFILE and INPUT statement to more advanced mechanisms such as SAS/ACCESS®. It is hoped that there will be some aspects of this presentation for the beginning, the intermediate, and perhaps even the advanced user of the SAS System.

It should be noted in advance that the details of this topic can vary from operating system to operating system. This presentation will be primarily aimed at providing an overview which is independent of operating system. The reader is strongly encouraged to use this paper in conjunction with the appropriate manuals and "Operating System Companions" to gain a full understanding of the options available to them.

SEQUENTIAL FILES

The first potential source of data that we shall discuss is the sequential file. There are two steps involved with making any external data known to the SAS System. The source of the data must be defined to SAS, and the data must be subsequently passed to SAS. There are two statements in the DATA step which perform these tasks. The **INFILE** statement will define the data source, while the **INPUT** statement will move the data into SAS.

The INFILE Statement

An external file is identified to a DATA step and subsequent INPUT statement(s) by the INFILE statement. There are three different ways to tie an external file to the INFILE statement. **See Figure A for an example of each method .**

The first mechanism is to actually specify the name of the external file, within quotes, in the INFILE statement. This method is useful under certain circumstances, such as one-time-only ad hocs. Its primary disadvantage is that it eliminates some flexibility -- in order to use the SAS routine to process different files, the user must either clone the routine and change the INFILE statement, or they must utilize the macro facility.

The second way is to provide a file reference to an external file. The file reference can be associated to the external file with a command to the host system, such as a JCL "DD" statement under IBM's MVS. An alternate method is to use the **FILENAME** statement under SAS. The FILENAME statement is a global statement which does not need to be included within a Data step. The syntax for this statement, in this context, is:

```
FILENAME fileref 'external file';
```

The third method is a variation of the second, which is only applicable to "aggregate storage locations", such as an IBM MVS partition dataset. Again, the INFILE statement is provided with a file reference to an external file, along with a specific file or member reference enclosed in parentheses.

Note that many of the options for the INFILE and FILENAME statements are dependent on the operating system. Please consult the appropriate "Companion" for your operating systems for details. Also note that it is possible to read from multiple external files from within the same DATA step. To accomplish this, there must be a separate INFILE statement for each external file. The appropriate INFILE statement should immediately proceed its corresponding INPUT statement.

Full file name in INFILE statement.
Example using Windows:

```
DATA _NULL_;
  INFILE 'c:\sugi22\sample.dat';
  <other statments...>
```

FILENAME and INFILE statements.
Example using CMS:

```
FILENAME  TDATA
         'SUGI22 TESTDATA A1';
DATA _NULL_;
  INFILE tdata;
  <other statments...>
```

Input file allocated by Operating System with INFILE statement.
Example using MVS:

```
//TDATA DD
DSN='userid.SUGI22.DATA',
//         DISP=SHR
DATA _NULL_;
  INFILE tdata(example1);
  <other statments...>
```

Figure "A" - INFILE Statement

The CARDS Statement

The most basic example of sequential input is the **CARDS** statement. When using this option, there *is* no separate sequential file. Instead, the last executable line of a DATA step will be the CARDS statement. The subsequent lines contain sequential input data. The end of the input data is signified by a single semicolon. (If your data might contain a semicolon, the CARDS4 statement can be substituted -- the end of input data is the signified by a string of 4 consecutive semicolons.) Even the use of the INFILE statement is optional when using CARDS; should the user choose to include it, there is a predefined CARDS file reference. **See Figure B for an example of the CARDS statement.**

```
DATA  EXAMPLE;
  INFILE CARDS;  /* Optional */
  INPUT  W  X  Y $  Z ;
  CARDS;
4  31.2 emu 141
6  5.18 fry 288
3  29.1 act 671
1  14.4 ion 93
;
```

Figure "B" - CARDS Statement

The primary benefit of the CARDS methodology is simplicity. There is no need to locate and understand external data -- the external data is the user's to control. Conversely, this is also the main drawback. In a batch setting, the user must ensure that the data is included in the DATA step. In an interactive setting, it becomes the user's responsibility to actually *enter* the data at the keyboard. Further, the user is *supposedly* limited to working with one external file per DATA step. In actuality, by using the INFILE CARDS statement, making the CARDS the last input source used in the DATA step, and with some careful coding, it is possible to combine CARDS with other external files in the same DATA step. (Exactly why a user would *want* to do this is unclear, however.) **See Figure C for an example of multiple INFILE statements with a CARDS statement.**

```
FILENAME ADISK 'A:STATECNT.TXT';
DATA TEMP;
  DO UNTIL(LASTFILE);
    INFILE ADISK END=LASTFILE;
    INPUT  STATE $ COUNT ;
    OUTPUT;
  END ;
  DO UNTIL(LASTCARD);
    INFILE CARDS END=LASTCARD;
    INPUT STATE $ COUNT ;
    OUTPUT;
  END ;
  STOP ;
CARDS;
TX 2
TN 1
FL 1
IL 1
CA 1
CT 1
;
```

Figure "C" - Multiple INFILEs with CARDS

The INPUT Statement

The actual transfer of data from an external file to a SAS DATA step is performed by the INPUT statement. By default, the SAS System assumes all input data to be numeric. This can be overridden by pre-defining a variable to be character, or by associating either a dollar sign ($) or a character format with the variable on the INPUT statement.

A full discussion of the INPUT statement is not possible in this limited space - the Version 6 SAS Language Reference devotes 30 pages to the statement! However, there are some basic tenets that should be reviewed in this setting.

There are five basic methods to describe a record to an INPUT statement. These methods can be combined on a single INPUT statement. However, it is suggested that users restrict themselves to one method per INPUT statement; this will avoid introducing additional complications in a program which will assist in debugging and in future maintenance on the routine. **See Figure D for examples of each input method.**

The most basic method is *list input*, in which the INPUT statement simply contains the name of each variable. Each value on a line of input data must be delimited by one or more blank characters (or other delimiter, as defined by the DELIMITER= or DLM= option on the INFILE statement). For this reason, missing data must be explicitly listed in the input file. Further, the standard list input method cannot be used for character variables that may contain embedded blanks. This weakness can be overcome by the use of the Ampersand (&) format modifier. The Ampersand will permit single embedded blanks in character variables. Similarly, the Colon (:) format modifier will allow the INPUT statement to ignore its default 8-character maximum on character variables. Use of either the Ampersand or Colon format modifiers is referred to as *modified list input.*

In *Column input*, the start and end columns are listed after each variable name. Column input does not have the weaknesses described under list input -- character variables can contain embedded blanks and can exceed 8 bytes in length, and missing values do not have to be explicitly defined in the input file. However, it is only useful when input data is formatted consistently on each line; other methods must be employed for free-format data.

Formatted input is similar to the list and column input methods described above. However, each input variable must have an accompanying informat. This is useful for data in a "non-standard" form, such as packed decimal, dollar, or date values.

The least common method is *named input*. With this method, the INPUT statement has an Equal sign (=) following each variable. The input data must also contain the same series of "fieldname=value" pairs. Please note that this method is an exception to the "all methods are interchangeable" rule referenced earlier. Once an INPUT statement begins to reference named input, all remaining data on that line must be in named input format.

There is also a *null input*. An INPUT statement, followed by a semicolon, can be used to move to the next input record without any further processing of the current line.

```
DATA  EXAMPLE;
  /* list input      */
  INPUT X
        Y $ ;
  /* column input    */
  INPUT X   1-4
        Y $ 6-8;
  /* formatted input */
  INPUT X   4. +1
        Y $ 3.;
  /* named input     */
  INPUT X=
        Y= $ ;
  /* null input      */
  INPUT ;
  CARDS;
14.4 Bob
1492 Sue
1776 Ann
X=28.8 Y=Jay
dummy - ignored by null INPUT
;
```

Figure "D" - INPUT Statement

There are a number of mechanisms to control the column pointer when reading an input file. Defining the start column after each variable name when using the column input method will

move the pointer to that column. Similarly, the use of a format after a variable name will move the column pointer.

There are two symbols that move the column pointer within the current record. The Plus sign (+) will move the column pointer relative to its current position. For example, +6 will move the pointer six characters from its current position. It should be noted that the value passed to the relative column pointer (Plus sign) need not be positive. A negative number will move the pointer *backwards* from its current position, up to the first position of the file. Negative numbers must be enclosed in parentheses when using the Plus sign to reposition the column pointer. Similar to the Plus sign, the At sign (@) provides absolute column position. For example @6 will move the pointer to the 6th position within the current record. It is not necessary to hard-code a fixed number for the Plus and At signs; they can be followed by a SAS variable to provide additional flexibility.

It is also possible to control the line pointer. The Pound sign (#) will move the line pointer to a specified line number within the input buffer. For example, #4 will move to the 4th line of the input buffer. (By default, the input buffer will contain the maximum number of lines specified by using the Pound sign in an INPUT statement. This can be overridden by the N= option of the INFILE statement.) The At sign (@) can also be used to control the line pointer. The default action is to move to a new input line with each INPUT statement; however, by ending an INPUT statement with "@", the line pointer will remain on the *same* input line. (This is referred to as a "Trailing 'At' sign".) The line pointer will not be moved until it encounters an INPUT statement without a Trailing At sign -- this is the most common reason for a null input statement -- or the next iteration of the DATA step. The latter control can be overcome if necessary by using a Double Trailing At (@@).

There are some special considerations which should be undertaken when reading records from variable length files. The LENGTH= option on the INFILE statement will assign the length of the current record to a SAS variable. A null input statement with a Trailing @ will permit that variable to be assigned, while keeping the line pointer on the current input line. Finally, the $VARYING. format will allow for flexibility in length of character variables. **See Figure E for an examples of this approach.**

```
DATA  EXAMPLE;
  INFILE  varyfil
          LENGTH=ln_len ;
  INPUT @ ;
  INPUT NAME $VARYING40. ln_len;
RUN;
```

Figure "E" - Variable Length File

DDE

Dynamic Data Exchange, or DDE, allows a client application to request information from a server application in a Windows® or OS/2® environment. Effective with Release 6.08, the SAS System acts as a client application in this relationship. It can request data from a server application, with the requirement that the server application must be running. (It can also send commands and data to a server application, but that is a topic for another presentation.)

In order to use DDE, a connection must be established between the client application and the server application. This is accomplished by issuing a **FILENAME** statement with the additional keyword "DDE". The syntax for this statement, in this context, is:

```
FILENAME fileref DDE
         'DDE-triplet' ;
```

The DDE-triplet is a specialized argument, and is made up of three components:

```
application|topic!item
```

Application is the name of the server application, such as Excel. *Topic* is defined as the "topic of conversation"; basically, this is the file to be processed. *Item* is the "item of conversation"; in a spreadsheet, this is the range of cells that is to be included. For example, the DDE-triplet an Excel worksheet would be:

```
Excel|[Book1]Sheet1!R1C1:R250C4
```

Note that the application and topic are separated by a vertical bar (|), while the topic and item are separated by an explanation point (!).

The DDE triplet for an application should be defined in the documentation for that application. However, most people find it easier to let SAS determine the proper DDE triplet. The following is a step-by-step method to obtain the proper DDE triplet for an application, assuming both SAS and that application are active:

- Toggle to your application, and use the standard PC "cut" techniques to store the portion of the client application to be processed in the Windows Clipboard. (For example, use the mouse to highlight the area to be "cut", then select CUT or COPY on the EDIT pop-up menu of most Windows applications.)
- Toggle to your SAS session, and click on the "Options" menu on the Menu Bar in SAS.
- The Options menu will contain an option called "DDE Triplet". Click on it.
- This will display an Information Box, which will contain the DDE-triplet.
- Enter this DDE-triplet into the FILENAME statement of your SAS routine.

If the user is willing to perform a little manual intervention, it is even possible to use DDE without ever knowing the name of the DDE triplet!

- As above, toggle to your application, and use the standard PC "cut" techniques to store the portion of the client application to be processed in the Windows Clipboard.
- Toggle to your SAS session, and replace the FILENAME statement with the following:

```
FILENAME fileref DDE CLIPBOARD;
```

The SAS routine is now ready to be executed. The weakness in this approach is that the data to be processed must be stored in the Clipboard prior to each invocation of your SAS routine. The benefit is that there is no need to ever know the DDE-triplet for your application to use DDE. (Please note that this approach will only work if an application is DDE compliant.)

In order to use DDE with the SAS System, the server application must be running while SAS is running. If the server application is not active, then it can be invoked from within the SAS session with the "X" command. However, the SAS options XSYNC and XWAIT must be turned off before issuing this command, or control will not be returned to the SAS session until that external application is closed -- this defeats the purpose of a DDE link! (Of course, the user could also simply toggle over to the Windows Program Manager and manually invoke the application.)

The actual transfer of data from the external application to the SAS System is done via the combination of an INFILE and INPUT statement. The actual code to accomplish this task looks exactly like the code to read a sequential file into SAS. **See Figure F for a DDE example.**

```
OPTIONS  NOXSYNC NOXWAIT;
X 'C:\EXCEL SUGI22.XLS' ;
FILENAME  SUGI22  DDE
'Excel|[Book1]Sheet1!R1C1:R25C4'
;

DATA SUGISCHD;
   INFILE SUGI22;
   INPUT DAY TIME TITLE AUTHOR;
RUN;
```

Figure "F" - INPUT with DDE

OLE / ODBC

Object Linking and Embedding, or *OLE*, is another client/server methodology which will allow the transfer of data between SAS and other products. In the SAS System, OLE functionality is attained through SAS/AF® using FRAME catalog entries.

OLE is more graphically oriented than DDE; OLE will permit the SAS System to share a number of different objects, such as graphs and charts, with other products. As the name implies, OLE consists of an *object linking* process and an *object embedding* process. The object link permits an object to be updated on either the client or server side of the link, with the change reflected on the other side of the link. An embedded object, on the other hand, can only be changed from the client application.

Open DataBase Connectivity, or *ODBC*, started off as a standard for the exchange of data between DataBase Management Systems (DBMS) under Microsoft's Windows environment. Since those early days, interfaces

to other operating systems and machines, such as the Apple MacIntosh, have been developed. It is necessary to use the SAS/ACCESS Interface to ODBC in order to use ODBC to bring data into the SAS System.

Even a brief discussion of OLE and ODBC requires more detail than can be covered in a single section of a Beginning Tutorial. As such, for more information on these topics, the reader is encouraged to refer to the appropriate SAS Institute publications, and to other presentations in the Proceedings from SUGI and the various regional SAS User Group conferences.

SAS/ACCESS ENGINES

SAS/ACCESS software is available for a variety of host systems, covering traditional mainframe, personal computer, and UNIX environments. It provides a method to view and transfer data from several common database management systems (DBMS) and a number of common PC file formats, into the SAS System.

The **ACCESS** procedure can create *descriptor* files that will provide information about the data stored in the DBMS table or PC file format, and use that information to create a SAS data file. In addition, use of an *interface view engine* will allow SAS to read data from the file formats directly into SAS routines. The interface view engine is used by the SAS SQL procedure to directly access external data bases without leaving the SAS session. However, the SQL statements used in the procedure are beyond the scope of this Beginning Tutorial.

There are two types of descriptor files created by PROC ACCESS: an *access descriptor* and a *view descriptor*. Access descriptors provide information regarding the structure of the file to be accessed. This includes data types, table names, and column names, as well as the related SAS dataset information such as variable names and formats. This access descriptor can then be used to create the view descriptor which will contain criteria to be used to select columns and rows from the selected DBMS table or PC file. The data can be used directly from the view descriptor in the SAS routine, or it can be extracted from the DBMS or PC file into a SAS data file.

The type of DBMS or PC file to be used is specified in the PROC ACCESS statement in the form:

PROC ACCESS DBMS=*filetype*

Filetype can take on many different values. To cite just a few examples, the user can obtain data from DB2®, SYBASE®, and ORACLE® by selecting filetypes *DB2*, *SYBASE*, and *ORACLE*, respectively. In a Windows environment, *XLS*, and *WKn (*where *n* is a valid version number) will allow the transfer of data from Excel® or Lotus® spreadsheets, while *DIF* is obviously the filetype for interfacing with a DIF format file.

The actual access or view descriptor is then created using the following syntax:

CREATE libref.member-name.ACCESS

or

CREATE libref.member-name.VIEW

The PROC ACCESS and CREATE statements are followed by a statement that identifies the name of the DBMS, DBF, XLS or other file that will be accessed. In addition, there are other editing statements; these provide information about the structure of the DBMS or PC file being accessed, and select columns to be viewed. It is suggested that the reader refer to the appropriate SAS/ACCESS manual for their DBMS for details on these statements. **See Figure G for an example of using PROC ACCESS create a view, with a subsequent use of its output.**

```
LIBNAME VWLIB 'c:\sugidat\';
PROC ACCESS DBMS=xls;
  CREATE vwlib.states.access;
  PATH 'c:\sugidat\state.xls';
  <editing statements omitted>
  CREATE vwlib.states2.view;
  <select, format, and
   subset statements omitted>
RUN;

DATA _NULL_;
  SET vwlib.states2;
  <statements omitted>
RUN;
```

Figure "G" - PROC ACCESS : Creation and use of a View

The SAS view descriptor can be used in any PROC or DATA step just like a SAS data set. It is also possible to use PROC ACCESS to create a SAS dataset from the view descriptor. This is accomplished by issuing PROC ACCESS with the VIEWDESC=*libref.view-descriptor* and OUT=*libref.sas-data-filename* options. **See Figure H for an example of using PROC ACCESS create a view, with a subsequent use of its output.**

```
PROC ACCESS
     VIEWDESC=vwlib.states2
     OUT=vwlib.stdata ;
RUN;

PROC PRINT DATA=vwlib.stdata;
RUN;
```

Figure "H" - PROC ACCESS : Creation and use of a SAS Dataset

It is not possible to fully cover PROC ACCESS in the limited space of this paper -- there are a number of manuals dedicated to the topic! For further information, including details of the DBLOAD procedure that will transfer data from SAS to the assorted DBMS and PC products, the reader is directed to the assorted SAS/ACCESS manuals.

THIRD PARTY PRODUCTS

In addition to the methods built into the Base SAS product and additional modules from SAS Institute, there are products produced by third-party providers that facilitate the transfer of data into SAS. In some cases, the user base for a given product does not warrant a separate SAS/ACCESS engine. The vendor for that product may provide their own interface to the SAS System. In other cases, an entrepreneur may believe that they have found a niche market which they believe SAS Institute has overlooked, or that they can provide an interface which is faster / quicker / cheaper / etc.; the marketplace will determine whether or not they have succeeded.

One example of a proprietary third-party product is PROC SAGE™. The Computer Corporation of America (CCA) supplies and supports a mainframe database called Model 204®. They also supply an interface called PROC SAGE, which passes data from a Model 204 database into a SAS dataset.

Conceptual Software, Inc. markets two products which can be used to transfer external data into the SAS System. The first package, DBMS/COPY™, provides connectivity between over 80 different spreadsheets, databases, statistical packages, and other assorted files -- including the SAS System. The newer product, DBMS/Engines™, provides engines which allow the SAS System to interact with all of the products that can interface with DBMS/COPY. Both products are available for assorted Windows and UNIX environments.

TRANSPORT FILES

There is one potential source of data that might not be thought of as an "external source" at first glance -- the SAS System itself, licensed on another computer. However, it is not possible to simply move SAS datasets from one operating system to another. Instead, the data must be converted to a format that is consistent on both machines. The "brute force" method of accomplishing this task would be to use PROC PRINT or the PUT statement to output the contents of the SAS dataset to a sequential file on the first machine, then to input that file into the SAS System on the second machine using the techniques described previously in this paper. However, an easier method exists; the SAS System permits the transfer of SAS datasets across operating systems as *SAS Transport Files.*

Transport files are used to move one or more SAS data sets from one host system to another. The transport file is a sequential file which is independent of the host operating system. This file can be readily transferred electronically or on permanent media to the destination host system. There are three basic steps involved:

- *Export* - creating the transport file on the original host system,
- *Transport* - moving the file via network protocols, tapes, or floppy media, and
- *Import* - reading the file back into SAS with the format of the destination host system.

Due to space limitations, only the procedure for moving a single data file will be discussed here. Further documentation on moving entire data libraries or catalogs can be found in the assorted Operating System Companion manuals and in Technical Report P-195.

Export : Creating the Transport File

The first step in the transport process is to *export,* or create the SAS transfer data set on the host system. Typically, this process starts by the allocation of the transport file on the host system. The SAS System has strict requirements on the allocation of a transport file. A SAS transport file ***must*** have a fixed record length of 80. In addition, it is highly recommended that it have a block size of 8000. (Note that block size is a meaningless concept under OS/2.) **See Figure I for examples of tape allocations.**

Under VMS:

```
DEFINE TRANFILE REEL
ALLOCATE TRANFILE
MOUNT/FOREIGN/BLOCKSIZE=8000
TRANFILE
```

Under MVS:

```
//TRANFILE DD DSN=mvs.dset.name,
//   DISP=(NEW,CATLG,DELETE),
//   UNIT=TAPE,VOL=SER=volser,
//   LABEL=(1,NL),
//   DCB=(RECFM=FB,LRECL=80,
//        BLKSIZE=8000,
//        DEN=density)
```

Figure "I" - Allocating a tape

The SAS System has strict requirements on the allocation of a transport file. A SAS transport file **must** have a fixed record length of 80. In addition, it is highly recommended that it have a block size of 8000. (Note that block size is a meaningless concept under OS/2.)

Two **LIBNAME** statements are required. One LIBNAME statement will identify the location of the file to be transported. The other LIBNAME statement specifies the name that will be given to the transport file, and defines the XPORT engine to identify the destination file as a transport file. **See Figure J for examples of LIBNAME statements.**

Under VMS:

```
LIBNAME outxp XPORT
   ' [directory]filename.dat';
LIBNAME outxp  XPORT;
LIBNAME libref b '[directory]';
```

Under MVS:

```
LIBNAME ddname  XPORT;
LIBNAME alibref
       'mvs.dataset.name.';
```

Figure "J" - Allocating a tape

At this point, **PROC COPY** is used to actually create the SAS transport file. PROC COPY will read in the host system formatted file that is named in the SELECT statement, and create the transport file. **See Figure K for an example of PROC COPY.**

```
PROC COPY IN=sasdata
          OUT=xprtdata;
   SELECT sas-datasetname;
RUN;
```

Figure "K" - PROC COPY

Transport : Moving the Transport File

The middle step in the transport process is to actually *transport* the SAS transfer data set on from the original host system to the new operating system. In the early days of the SAS System, this usually required the allocation of a round tape which could be then mounted on another machine. This process may still involve the physical transfer of a tape or floppy disk between two machines. However, the technology of today permits easy electronic data transfer between different machines and operating systems. Note that SAS transport files should be treated as binary data when using network commands such as ftp.

Import : Reading the Transport File

The final step in the transport process is to *import* the SAS transfer data set into the SAS System on the destination machine. This also involves the use of LIBNAME statements and PROC COPY. This time, however, the process is reversed; the LIBNAME for the transport file is used for the IN= parameter of PROC COPY, while the LIBNAME of the SAS dataset is used for the OUT= parameter. Again, a SELECT statement appears in the PROC COPY procedure. Figure 2 shows the syntax for importing the transport file on the new host system. (Due to space limitations, there is no example of using PROC COPY to import a transport file. The process is almost identical to the transport process described earlier in this section.)

It is also possible to use **PROC CPORT** and **PROC CIMPORT** to transfer SAS datasets between different machines and operating systems. However, this topic will not be explored in this tutorial due to space limitations.

CONCLUSION

There are a number of methods to introduce external data into the SAS System. It would be impossible to provide in-depth information on all of them in the limited space of this presentation. It is hoped that the material contained in this paper will serve to stimulate the curiosity of the reader, and that they will continue their education by researching the appropriate manuals and technical papers devoted to the specific topics discussed within this paper. Ultimately, however, it will be through real-life trial and error that true comprehension and retention of this knowledge will be attained.

REFERENCES / FOR FURTHER INFORMATION

Conceptual Software, Inc. (1997), "Conceptual Software, Inc. Home Page". *http://www.conceptual.com.*

Dalberth, Paul. (1996), "A Relational Database Primer for SAS Programmers". *Proceedings of the Fourth Annual Conference of the SouthEast SAS Users Group*. USA.

Praxis International, Inc. (1993), *SAGE/204 User's Guide Release 2.1*. USA: Praxis International, Inc.

Riba, S. David (1996), *Course Notes: Connecting With Your Data*. Clearwater, FL: Jade Tech, Inc.

Riba, S. David, and Riba, Elisabeth A. (1996), "ODBC: Windows to the Outside World". *Proceedings of the Fourth Annual Conference of the SouthEast SAS Users Group*. USA.

Riba, S. David (1996), "Open DataBase Connectivity and SAS". *The NESUG Express,* October 1996 Edition.

SAS Institute, Inc. (1993), *SAS Companion for the Microsoft Windows Environment*. Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1994), *Getting Started with SAS/ACCESS Software, Version 6, First Edition.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1995), *SAS/ACCESS Software for PC File Formats: Reference, Version 6, First Edition.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1994), *SAS/ACCESS Software for Relational Databases: Reference, Version 6, First Edition.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1990), *SAS Language: Reference, Version 6, First Edition.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1990), *SAS Procedures, Version 6, Third Edition.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1994), *SAS Software: Abridged Reference, Version 6, First Edition.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1989), *SAS Technical Report P-195, Transporting SAS Files between Host Systems.* Cary, NC: SAS Institute, Inc.

The authors can be contacted via e-mail as follows:
Andrew T. Kuligowski
0005949476@mcimail.com
kuligoat@tvratings.com

Nancy Roberts
nancyr@chimaera.declab.usu.edu
nancyr@cc.usu.edu

ACKNOWLEDGMENTS

The authors would like to acknowledge the contribution and support of Brad Keller and S. David Riba during the creation of this presentation.