Creating External Files Using SAS© Software

Clinton S. Rickards
Oxford Health Plans, Norwalk, CT

ABSTRACT

This paper will review the techniques for creating external files for use with other software. This process is the flip-side of reading external files. We will review the business need, statements for pointing to and creating the file, and common options. A number of specific applications will be used to illustrate the various techniques.

INTRODUCTION

The SAS system is a powerful application development, data analysis, and data manipulation tool. But its structures are proprietary, making usage by other software difficult. This may require you to reformat the data into a form that the other software can handle. For the purposes of this paper, we will limit ourselves to “traditional” flat files; that is, files that are supported by the native operating system. We will not concern ourselves with relational databases, xBase, or other proprietary data storage systems; SAS Institute and other vendors offer tools to work with those systems.

This paper provides an overview of the tools SAS makes available to you for creating external files in a DATA step. Although other areas of SAS (the Display Manager, SAS/AF, various procedures) can be used to create external files, you will eventually have a need to use the venerable DATA step to accomplish this task.

We will first discuss the upfront planning and analysis tasks you should take. Then we will examine the ways you tell SAS about the file to be created. We will describe the PUT statement, which actually writes the file, in some detail, and finally consider some complete examples.

All programs in this paper were created under SAS 6.11 running on Windows 95. The techniques presented here are applicable to versions 6.06 and higher.

DETERMINE YOUR NEEDS

The first step toward successfully creating an external file is to determine your needs. Spending the necessary time at this stage will greatly reduce the number of problems you encounter later on during coding and testing. Questions to ask: What is the purpose of the file? Can the file serve more than one purpose? How often will your program be run and how often might it be changed? A one-time, quickie job may allow you to cut some corners that you wouldn’t be able to with a more dynamic or long-lived program. Where will your data come from? Can the file be made to serve more than one purpose? What environment (e.g. PC, UNIX, MVS) will your program run in and where will the file be used?

Also, don’t neglect understanding the source of data. What does it look like, are there idiosyncrasies you need to be aware of? How often is the source created?

FILE DESIGN CONSIDERATIONS

Again, investing time at this stage will help you develop a program that fits your needs. Some questions to ask: Is the layout determined already? What is the layout? How much flexibility is there in changing the layout? What kind of translations and other manipulations are needed? Does the order of the records matter? How many records are involved? Are there different record formats in the file? Are there group fields (i.e. arrays)? How are number fields and dates formatted? Packed? Display? Are there variable areas? Will you use disk or tape?

POINTING TO EXTERNAL FILES

There are three ways to point SAS to external files: operating system control statements, FILENAME statements, and FILE statements.

1. Operating system commands

Some operating systems provide commands or statements that allow you to create a *fileref*. A fileref (file reference) is a nickname used in place of the full operating system filename. For example, MVS batch allows gives you the Job Control Language DD statement, and CMS has the FILEDEF command.

Generally, the fileref is created before a SAS session is started. Some platforms permit the use of the SAS X statement to assign a fileref after the session has started. Please check the SAS Companion for your operating system for details. A disadvantage of assigning filerefs with this method is that the filerefs will not be available in the Display Manager FILENAME window.

2. FILENAME statement

This statement creates a fileref that names a link between the physical file and the SAS system. For operating systems that do not support filerefs (e.g. PC-DOS and Windows), the FILENAME statement or the FILE statement (below) must be used. The filename statement is executed within your SAS session but before the DATA step begins.

The syntax of the FILENAME statement is:

FILENAME *fileref <device-type>*
'external-name' <host-options>;

where:
fileref is any valid SAS name

device-type indicates the type of device. It is optional and defaults to DISK.

'external-name' is the name of the file on the host system. The quotes are required.

host-options specify are options that vary from operating system to operating system. Carefully read the SAS companion for your operating system to understand the meaning and usage of these options.

3. FILE statement
FILE is used to tell a DATA step which file to write to when a PUT statement is executed. It either specifies a fileref or the actual external file name. It must be used in each data step writing an external file and must be executed before the put statement. You can have more than one FILE statement, which allows you to write multiple files in a single DATA step.

If a FILE statement has not executed, a PUT statement will default to the SAS log, a particularly useful feature for debugging or writing control totals. When SAS begins each iteration of a DATA step, it "forgets" which file you were writing to. This means that you must be careful to execute the FILE each time you process through the DATA step. Consider this program, in which the FILE statement is executed only once. The first observation is written to the external file but all remaining observations are written to the log.

```
data _null_;
  set somedata;
  if _n_ eq 1 then do;
    file onlyonce;
    cnt + 1;
    end;
  put var1 var2 var3;
run;
```

The general syntax of FILE is:

FILE *file-spec <options> <host options>*;

where *file-spec* identifies the source of the data. *file-spec* may have one of five forms:

fileref gives the fileref of an external file. The fileref must be created before the DATA step by using an operating system definition or a FILENAME statement.

'external-file' gives the name of an external file. This form is the functional equivalent of a separate FILENAME statement and of an operating system definition.

fileref(file) uses a previously defined fileref that points to an aggregate data store (i.e. a library). *file* is the name of a member in the aggregate store.

LOG indicates that the data should be written to the SAS log.

PRINT indicates that the data should be written to the SAS list.

options specify SAS options to control writing the file or to provide information about the file. Commonly used options will be described in detail later.

host-options are host specific options. See the SAS companion for your system for details.

Comparison of methods
As you can see, the three methods of pointing to an external file have considerable functional overlap. All other things being equal, I prefer to create a fileref - using an operating system command if possible else a FILENAME statement - and then using a FILE statement using that fileref. I rarely use a FILE directly pointing to the external file. This arrangement makes me think up front about what I am doing and tends to gather external file definitions together, which makes maintaining and porting the application to another operating system easier. Changing the file to be read by a production job is easier and faster, especially in MVS batch. Finally, certain maintenance tasks can be handled by support staff not familiar with SAS.

PUT STATEMENT BASICS
The PUT statement is used to actually write to the data file. When a PUT statement is executed, it writes to the file pointed to by the FILE statement most recently executed in that iteration of the DATA step or to the LOG if no FILE has executed. Remember, you can have more than one FILE in a DATA step, including one that references the LOG. There are four styles of PUT that may be mixed and matched as needed.

LIST PUT merely lists the desired variable names in the PUT statement. List put assumes that the data are separated by a space. It is the easiest style to code but offers the least control over where and how the data is actually placed in the file. Consider this example, which writes to the LOG since no FILE statement is used.

```
data _null_;
  a = '21may1954'd;
  b = a;
  format a mmddyy10.;
  put a b;
run;
```

```
05/21/1954 -2051
```

If a format has been defined for a variable, it is applied automatically before the data is written; otherwise the BEST format is used for numeric data or the $ format for character.

List out also allows you to write constant data surrounded by single or double quotes. Note, however, that there is no blank following the constant text.

```
data _null_;
  a = '21may1954'd;
  b = a;
  format a mmddyy10.;
  put a
      "THIS IS A CONSTANT"
      b ;
run;
```

```
05/21/1954 THIS IS A CONSTANT-2051
```

COLUMN PUT specifies the columns to be written for each variable. For example:

```
data _null_;
  a = '21may1954'd;
  b = a;
  format a mmddyy10.;
  put a 1-10 b 11-20;
run;
```

```
     -2051     -2051
```

Note that is a variable's defined format is ignored when using column put.

FORMATTED PUT includes a format for the variables. This form of put is the only form that will work when non-standard display data representations (e.g. packed, hexadecimal, date/time) are needed. For example:

```
data _null_;
  a = '21may1954'd;
  b = a;
  put a mmddyy10.
      b yymmdd8.;
run;
```

```
05/21/195454-05-21
```

NAMED PUT is of the form *variable*=.
This style of put places the variable name with the data, as in:

```
data _null_;
  a = '21may1954'd;
  b = a;
  format a mmddyy10.;
  put a= b=;
run;
```

```
A=05/21/1954 B=-2051
```

Note that defined formats are applied automatically. Should you need to apply a different format, specify the new format after the equal sign:

```
data _null_;
  a = '21may1954'd;
  b = a;
  format a mmddyy10.;
  put a=yymmdd8. b=;
run;
```

```
A=54-05-21 B=-2051
```

The special keyword *_all_* used in a put statement instructs SAS to write all variables in the DATA step using named put.

EXAMPLES

Assume that you need to create a file containing the name, birth date, and gender of the students in a class and that the data are stored in a SAS data set. A complete program (under Windows 6.11) would be:

```
126  filename rstrfile 'a:\file1.dat';
127  data _null_;
128    set roster;
129    file rstrfile;
130    put name birthdte gender;
131  run;

NOTE: The file RSTRFILE is:
      FILENAME=a:\file1.dat,
      RECFM=V,LRECL=256

NOTE: 3 records were written to the file
RSTRFILE.
      The minimum record length was 9.
      The maximum record length was 11.
NOTE: The DATA statement used 0.22 seconds.
```

As you can see, SAS places a great deal of useful information in the log for you. The resulting file looks like this:

```
Dan 157 M
Mary 74 F
Mary Anne 3168 M
```

This file presents some difficulties to the program that will be actually reading it. First, the placement of BIRTHDTE and GENDER varies. This could potentially be overcome by having the reading program use blanks as delimiters but the record for Mary Anne has an imbedded blank. Second, the value for BIRTHDTE is meaningless unless the program knows how SAS stores date values internally. All of these problems could have been avoided had the source data set ROSTER carried the proper formats. Since it doesn't, this is a better program:

```
170  filename rstrfile 'a:/file2.dat';
171  data _null_;
```

```
172    set roster;
173    file rstrfile;
174    put name           $10.
175        birthdte   mmddyy10.
176        gender          $1.
177      ;
178  run;

NOTE: The file RSTRFILE is:
      FILENAME=a:\file2.dat,
      RECFM=V,LRECL=256

NOTE: 3 records were written to the file
RSTRFILE.
      The minimum record length was 21.
      The maximum record length was 21.
NOTE: The DATA statement used 0.11 seconds.


Dan       06/06/1960M
Mary      03/15/1960F
Mary Anne 09/03/1968M
```

MISSING NUMERIC DATA

When SAS encounters missing numeric data, what it writes depends on the format being used and on the MISSING= system option. If the format has an “other” or “low” defined, that value is used. Most often, MISSING=” “ or MISSING=”.” is used, telling SAS to write that character in place of missing values. Reading programs may or may not have difficulties with blanks but most will have difficulties with periods, so consider this carefully.

ADVANCED PUT STATEMENT FEATURES

The last program manifests a couple of additional problems with the put techniques we have examined so far. First, the location of GENDER is not immediately obvious - you must add the formatted lengths of NAME and BIRTHDTE to determine where GENDER begins. Second, if BIRTHDTE is not needed but GENDER must remain in its current location, you must consume precious resources writing 10 spaces. Although acceptable in that small example, these problems could be unmanageable when writing large files. This section provides the tools to overcome these and other problems.

POINTER CONTROLS

SAS maintains two pointers, the **column pointer** and the **line pointer**, to keep track of where data will be written during the execution of a PUT statement. You can change these pointers, for example, to rewrite data, change the order in which data is written, or to create logical records that are defined by multiple physical records. Use of these pointer controls also makes the PUT statement more self-documenting.

COLUMN pointer controls:
@expression
+expression

@ moves the pointer to the column number resulting from *expression. expression* can be a numeric constant, a numeric variable, or a formula enclosed in parenthesis. For example, @44, @START, @(START+COUNT*5) are valid expressions.

+ moves the pointer left or right the number of columns resulting from *expression. expression* can be a numeric constant, a numeric variable, or a formula enclosed in parenthesis. Expressions resulting in positive numbers move the pointer to the right, negative results move the pointer to the left. Negative numeric constants must be enclosed in (). For example, +5, +START, +(-5), and +(START+COUNT*5) are valid expressions.

These examples accomplish the same task. Note in example 3 that PEARS is written before APPLES by using the +5 and +(-10) column pointer controls.

```
1. PUT @1   VENDOR    $CHAR20.
       @20  APPLES          5.
       @25  PEARS           5.;

2. START = 1;
   PUT @START  VENDOR  $CHAR20.
               APPLES        5.
               PEARS         5.;

3. PUT           VENDOR  $20.
        +5       PEARS     5.
        +(-10)   APPLES    5.;
```

LINE pointer controls:
#expression
/

moves the pointer to a specific line number based on the result of *expression*. *expression* can be a numeric constant, a numeric variable, or a formula enclosed in parenthesis.

/ moves the pointer to the next line. You can use multiple / if needed.

These PUT statements are functionally equivalent:

```
1.  PUT  #1 @1   LASTNAME  $CHAR25.
         #2 @16  PHONENUM        9.
         #4 @30  CITY          $25.;

2.  PUT     @ 1  LASTNAME  $CHAR25.
         /  @16  PHONENUM        9.
         // @30  CITY          $25.;

3.  CITYLOC  =  4;
    PUT          LASTNAME  $CHAR25.
         #2 @16  PHONENUM        9.
       #CITYLOC @30  CITY      $25.;
```

LINE HOLD SPECIFIER

A line hold specifier is used to "hold" the current line in the external file through multiple PUT statements. Placed at the end of the PUT statement, it instructs SAS to <u>not</u> write a new record when the next PUT statement is executed. This capability is the key element of techniques used to writing more complex files or to improving efficiency.

@ **(trailing at-sign)** tells SAS to keep this record current until another PUT is executed, even if that PUT is in a different iteration of the DATA step. The record is written when a PUT statement without a trailing @ is processed, the column pointer moves past the end of the line, or the DATA step ends. For example, this program writes different data depending on the results of a merge operation.

```
data _null_;
  file trash;
  merge policy (in=a)
        agents (in=b);
  by agtcode;
  if a;
  put polnum @;
  if b then
    put agtcode agtstats;
  else
    put agtcode "NOT FOUND";
run;
```

The @@ (double trailing at-sign) is equivalent to @.

WARNING: When using @, care must be taken to ensure that the record is released when you want it released. It is easy to attempt to write past the end of the record or to only write one record. For example, this program will write only 1 record to the file STREETS, when data set ADDRESS is exhausted. It will have the street of the last observation in ADRESS.

```
DATA _null_;
  set address;
  file streets;
  put street $ 1-20 @;
run;
```

Remember that using line hold specifiers may require you to either 1) explicitly end execution of the DATA step with a STOP statement; or 2) release the line with a PUT statement without line hold a specifier.

GROUPING VARIABLES AND FORMATS
You may group variables and formats into lists to reduce the size of the PUT statement. SAS recycles the format list until the variable list is exhausted. This technique, shown in Program 3, is particularly useful when you are writing arrays or numbered variables.

COMPARISON OF METHODS
In general, I tend to use formatted put in conjunction with column and line pointer controls. Some types of data (e.g. packed decimal, signed numeric, hexadecimal, or dates) can be written only with formatted put and this technique gives me maximum control over the writing of the data without excessive worrying about coding errors in the data. In addition, the formatted put will ensure that the data is written exactly as I want it to be written without any surprises caused by the default processing SAS could take. Finally, the code is self-documenting to some extent by fully defining where the data is and its structure. However, you must choose the techniques appropriate to your applications.

FILE STATEMENT OPTIONS
FILE has many options, some specific to the host operating system and some generic to any SAS application. In this section, we will explore the more commonly used SAS options.

DROPOVER, FLOWOVER, STOPOVER are mutually exclusive options that define what action SAS is to take when the program attempts to write past the end of a record.

DROPOVER tells SAS to drop remaining variables from the record and to continue processing the DATA step.

FLOWOVER tells SAS to continue writing variables on succeeding lines. This is the default action. Note that a new record will be written when SAS attempts to write past the end of the record even if a trailing @ on the PUT statement is used.

STOPOVER tells SAS to stop processing the DATA step after writing the current record. SAS sets _ERROR_ equal to 1.

N=*n* tells SAS how many lines to make available to the PUT statement. Set this value if there is more than 1 record to be written at a time. The default is 1. The line pointer controls / and # would be used. See Example 5.

LRECL=, BLKSIZE=, RECFM= are used to describe the logical record length, block size, and record format, respectively, of the external file. These options control the overall size and structure of your file.

MOD, OLD are mutually exclusive options that tell SAS to append records to the file (MOD) or to replace the existing file (OLD). OLD is the default.

The following options are generally used when creating reports in a DATA step. Although this paper doesn’t deal with this aspect of DATA step programming, they are presented for completeness.

LINESIZE= sets the number of columns available to SAS. LRECL= sets the overall record length but LINESIZE= tells SAS how much of that line to use.

PAGESIZE= sets the number of lines per page in your report. After reaching this limit, SAS automatically moves to line 1 of the next page.

NOTITLES, PRINT|NOPRINT controls whether title lines are printed and whether printer carriage control characters are used, respectively. Usually, NOTITLES and NOPRINT are used when creating files and PRINT is used for reports.
HEADER= defines a label that is executed whenever SAS begins a new output page.

SPECIFIC APPLICATIONS

PROGRAM 1: CREATING A BASIC FIXED FORMAT FILE

This example presents the author's favorite technique: using pointer controls and formatted puts to achieve maximum control over the placement and format of the data.

```
file outtools 'a:\outfile1.dat';
data tools;
  set nesug.tools;
  file outtools;
  put @1   item      $char10.
      @11  quantity  5.
      @16  saledate  mmddyy8.
      @22  clerk     $char5.
      @22  clrkgrp   2.
      @24  initials  $char3. ;
run;
```

PROGRAM 2: CREATING A MULTIPLE FORMAT FILE

This program illustrates the use of the line hold specifier @. The output data has a date record, then several detail records, and a final control record, each with a different layout. The first character of each record identifies the record type.

```
filename outtools
  'c:\data\clint\writing-files\out2.dat';
data tools;
  set writing.tools end=eof;
  file outtools noprint notitles;
  if _n_ eq 1 then
    link daterec;

  link detail;

  if eof then
    link control;
return;

daterec: /* write rest of date record */
  created  = compress(
          put(today(),mmddyy10.),'/');
  asofdate = compress(
          put(today()-1,mmddyy10.),'/');
  put @1  '0'       /* date record type */
      @2  created $8.
      @10 asofdate $8.;
  totrecs + 1;         /* count records */
return;

detail: /* write rest of detail record */
  wrksaldt = compress(
          put(saledate,mmddyy10.),'/');
  put @1  '1'     /* detail record type */
      @2   item      $char10.
      @12  quantity  z5.
      @17  wrksaldt  $8. @;
  if clerk ne ' ' then
    put  @25  clerk     $char5.;
  else
    put  @25  deptcode  $5.;
  totrecs + 1;  /* count records */
return;

control: /* write rest of control record */
  totrecs + 1;  /* count records */
  put @1  '9'    /* control record type */
      @2  totrecs   z9.;
return;
run;
```

```
NOTE: The file OUTTOOLS is:
      FILENAME=a:\out2.dat,
      RECFM=V,LRECL=256

NOTE: 3 records were written to the file
OUTOOLS.
      The minimum record length was 10.
      The maximum record length was 29.
NOTE: The data set WORK.TOOLS has 1 observations
and 9 variables.
NOTE: The DATA statement used 0.17 seconds.
```

A date record (sometimes called a header record) is created the first time through the DATA step. Every record is written to the file. Note the line-hold specifier after WRKSALDT. This allows the substitution of DEPTCODE in place of CLERK when needed. Also note the slightly convoluted means to achieve a date in ccyymmdd format; SAS does not supply a format matching this need. When the end of WRITING.TOOLS is reached then a control record is written.

PROGRAM 3: REPEATING FIELD PATTERNS

Program 3 demonstrates one way to write a file with repeated fields. Each record has an ACCOUNT followed by 4 occurrences of a set of repeated fields pertaining to cash flow into the policy. All fields are separated by blanks.

```
filename out3
  'a:\out3.dat';
data _null_;
 file out3 noprint notitles;
 set writing.sales;
 put @1  policy         $char8.
    @10 (fund1-fund3) ($char3. +14)
    @14 (date1-date3) (yymmdd6. +11)
    @21 (amount1-amount3) (5.2  12);
run;
```

```
NOTE: The file OUT3 is:
      FILENAME=a:\out3.dat,
      RECFM=V,LRECL=256

NOTE: 1 record was written to the file OUT3.
      The minimum record length was 59.
      The maximum record length was 59.
NOTE: At least one W.D format was too small for
the number to be printed. The decimal may be
shifted by the "BEST" format.
NOTE: The DATA statement used 0.33 seconds.
```

The PUT statement writes the data from the variables by using **grouped format lists**. Grouped format lists are a very compact way to write repeating fields because the format lists are recycled until all of the variables are exhausted. Grouped format lists consist of 2 lists, each enclosed in parentheses: the first is the list of variables and the second is the format list to be recycled. The format lists can also include column pointer controls, vital in this case due to the intermixing of the data.

To illustrate, consider the fund information. FUND1 is written using the $CHAR3. format and then the +14 moves the pointer past DATE1, AMOUNT1, and the intervening blanks. The pointer is now positioned at the data for FUND2. Looking at the variable list, SAS then writes FUND2 using the recycled format list, which leaves the pointer at the data for FUND3. This recycling continues until the variable list is exhausted. Pointer controls are not used after the last variable in the list is written. SAS then processes the next part of the put, which writes the date and amount fields in similar fashion.

The program also illustrates that care must be taken when deciding on the formats to use. In this case, the variable AMOUNT3 has a value 100, which when formatted with 2 decimal places (100.00) cannot fit into 5 characters. SAS substitutes the BEST format in order to give you as much precision as possible. Unfortunately, the note does not tell you the name of the offending variable; you have to figure that out on your own.

PROGRAM 4: REPEATING FIELD PATTERNS

This program shows how to create a file with repeating fields from a SAS data set with one observation for each group of fields. The data elements are separated by blanks. Note the use of trailing-@ to hold the record until an account break is reached.

```
filename sales  'a:\outfile4.dat';
data _null_;
  file sales noprint notitles lrecl=80;
  set writing.fundinfo;
  by account;
  if first.account then
    put @1  account $char8. @;
  put +1  fund    $char3.
      +1  date    yymmdd6.
      +1  amount  5.2   @   ;
  if last.account then
    put;  /* release the record */
run;
```

```
NOTE: The file SALES is:
      FILENAME=a:\outfile4.dat,
      RECFM=V,LRECL=80

NOTE: 2 records were written to the file SALES.
      The minimum record length was 76.
      The maximum record length was 76.
NOTE: The DATA statement used 1.32 seconds.
```

PROGRAM 5: MULTIPLE RECORDS PER OBSERVATION

This program segment writes a policy information file that spans multiple physical records.

```
data _null_;
  file 'a:\out5.dat'
        noprint notitles n=4
        lrecl=72 blksize=7200;
  set writing.policy;
  put #1 @7  policy   $8.
         @28 issuedte yymmdd8.
         #2 @33 agent    $40.
         #4 @65 state    $3. ;
run;
```

```
NOTE: The file 'a:\out5.dat' is:
      FILENAME=A:\out5.dat,
      RECFM=V,LRECL=72

NOTE: 4 records were written to the file
'a:\out5.dat'.
      The minimum record length was 0.
      The maximum record length was 72.
NOTE: The DATA statement used 3.18 seconds.
```

The #1, #2, and #4 line pointer controls moves the pointer to those lines before the next part of the PUT statement is executed. Line 3 is completely blank

PROGRAM 6: COMMA DELIMITED, QUOTED TEXT FILE

This type of file is frequently referred to as a CSV file, or comma separated values, file. You must include the quotes that surrounds text fields and the commas that separate the variables as constant text.

```
data _null_;
  file 'a:\out6.dat' noprint notitles;
  length policy $15 agent $4 state $2;
  informat issuedte yymmdd8.;
  input policy  issuedte agent state;
  workdate =
put(compress(put(issuedte,mmddyy10.),'/'),$8.);
  put '"' policy   $  +(-1) '",'
          workdate          ','
       '"' agent    $ +(-1)  '",'
       '"' state    $ +(-1)  '"';
cards;
JOHN 19950228 A123 CT
ALFRED 19950515 A459 MA
run;
```

```
"JOHN",02281995 ,"A123","CT"
"ALFRED",05151995 ,"A459","MA"
```

The use of the +(-1) column pointer controls is necessary since list put is being used. In Observation 1, for example, the length of variable POLICY is 4 characters. SAS writes the 4 characters then 1 blank, leaving the column pointer at 6. The +(-1) moves the column pointer back to 5 before the double quote and comma are written.

```
"JOHN",02281995 ,"A123","CT"
"ALFRED",05151995 ,"A459","MA"
```

CONCLUSION

This paper has only touched upon the basics of creating external files using SAS software. Obtain and use the SAS appropriate documentation for your system. There are also numerous NESUG and SUGI papers that will be helpful. The author highly recommends Marge Scerbo's paper "DATA Step Reporting" from NESUG 95 as an excellent introduction to that topic.

Any given file layout can probably have several solutions that will work, each with their own advantages and disadvantages. Take the time to play and experiment with different techniques.

Electronic mail may be sent to the author at crickard@oxhp.com.

References:

SAS Language: Reference, Version 6.06, First Edition, SAS Institute, Inc., Cary, NC

SAS Technical Report P-222, Changes and Enhancements to Base SAS Software. Release 6.07, SAS Institute, Inc., Cary, NC

SAS Technical Report P-242, SAS Software: Changes and Enhancements. Release 6.08, SAS Institute, Inc., Cary, NC

Scerbo, Marge, “DATA Step Reporting”, NESUG Proceedings, 1995