Paper 119-2012

The FILENAME Statement: Interacting with the world outside of SAS®

Chris Schacherer, Clinical Data Management Systems, LLC

ABSTRACT

The FILENAME statement has a very simple purpose—creating a symbolic link to an external file or device. The statement itself does not process any data, specify the format or shape of a data set, or directly produce output of any type; yet, this simple statement is an invaluable construct that allows SAS programs to interact with the world outside of SAS. Through specification of the appropriate device type, the FILENAME statement allows you to symbolically reference external disk files, interact with FTP servers, send e-mail messages, and integrate data from external programs and processes—including the local operating system and remote web services. The current work provides examples of how you can use the different device types to perform a variety of data management tasks.

INTRODUCTION

The wide array of data access methods available to the SAS programmer today has resulted in a decrease in the number of new SAS programmers who explore the FILENAME statement. In fact, in one recent survey, only 14% of users reported using the FILENAME statement as a means of accessing their source data (Milum, 2011). Whereas this decline has come about due, largely, to an improved toolset that makes the majority of data access tasks more efficient, this increased efficiency has come at the cost of new SAS users not being exposed to this powerful tool. The current work attempts to close this knowledge gap by introducing the reader to the FILENAME statement, providing some examples of frequently used techniques for manipulating flat files with the INFILE and FILE statements, and demonstrating several techniques for leveraging the FILENAME statement against common data management tasks.

To begin this exploration, consider how the FILENAME statement was described in *The SAS Language Guide for Personal Computers (Release 6.03 Edition)* (SAS, 1988):

> *The FILENAME statement associates a SAS fileref (a file reference name) with an external file's complete name (directory plus file name). The fileref is then used as a shorthand reference to the file in the SAS programming statements that access external files (INFILE, FILE, and %INCLUDE). Associating a fileref with an external file is also called defining the file. Use the FILENAME statement to define a file before using a fileref in an INFILE, FILE, OR %INCLUDE statement.*

In short, the FILENAME statement creates a symbolic link to an external file (or device) that, in turn, can be used to refer to that file (or device) elsewhere in the SAS program. In the following code, the file reference "claims" is defined as the physical file "\\finance\analytics\report\source\2011_07.txt". When this fileref is later encountered in the INFILE statement, SAS interprets the symbol "claims" as the fully qualified path and file name defined in the FILENAME statement.

```
FILENAME claims DISK '\\finance\analytics\report\source\2011_07.txt';

DATA work.claims_2011_07;
   INFILE claims;  {interpreted as: INFILE '\\finance\analytics\report\source\2011_07.txt';}
   INPUT svc_date mmddyy10. account_no $10. principle_dx $8. billed_charges;
RUN;
```

READING & WRITING EXTERNAL DISK FILES

Far and away the most frequent use of the FILENAME statement is the reading and writing of external disk files. In the preceding example the file "2011_07.txt" was referenced in the FILENAME statement and subsequently read into the SAS data file "claims_2011_07" using the INFILE and INPUT statements[1]. Although an exhaustive discussion of using INFILE and INPUT statements to read external files is beyond the scope of this discussion, a few examples are provided here. For more in-depth explanation of the INFILE and INPUT statements, the reader is referred to Aster and Seidman (1997) and Kolbe (1997).

[1] The DISK device type was explicitly specified in this filename statement, but because DISK is the default access method, this keyword can be omitted when reading/writing external disk files.

FIXED WIDTH, UNIFORM LAYOUT

One of the fundamental characteristics that determine the appropriate approach for reading a flat file into SAS is the manner in which the variables are positioned on the data record. "Fixed-Width" data records contain data elements that are located in defined, absolute (i.e., "fixed") positions on the record; "Delimited" records are those in which values are written one after the other with a delimiter (such as a comma or a tab) defining their relative position on the record. In the case of the medical claims file "2011_07.txt" in the preceding example, the values of the variables are identified in terms of their fixed positions within the data record.

```
svc_date account_no principle_dx billed_charges
07/01/2011  VGH3344562  V712.4     $234.55
07/01/2011  XWY3928957  325.4       $34.55
|||||||||||||||||||||||||||||||||||||||||
1       10        20        30        40
```

The value of "svc_date", for example, lies between positions 1 and 10, "account_no" between positions 13 and 22, etc. When this file is read by the following DATA step, the INFILE statement fetches each data record from the file into the input buffer. The INPUT statement then parses the data into values associated with the listed variables using explicitly defined INFORMATS; "svc_date" is identified to SAS as representing a numeric date in the format of MM/DD/YYYY (or, mmddyy10.), "account_no" is indicated to be a 12-character text string, and "billed_charges" is identified as a numeric value that is being represented in the external file using the dollar12. format.

```
FILENAME claims DISK '\\finance\analytics\report\source\2011_07.txt';

DATA work.claims_2011_07;
   INFILE claims FIRSTOBS=2;
   INPUT svc_date mmddyy10. account_no $12. prin_dx $8. billed_charges dollar12.;
RUN;
```

Once the DATA step completes, the new dataset "work.claims_2011_07" contains "svc_date" as a SAS date, "billed_charges" as a numeric value, and "account_no" and "prin_dx" as character data.

VIEWTABLE: Work.Claims_2011_07

	svc_date	account_no	prin_dx	billed_charges
1	18809	VGH3344562	V712.4	234.55
2	18809	XWY3928957	325.4	34.55

This form of the INPUT statement is referred to as LIST ENTRY because the variables are simply listed in the order they are encountered on the data record. An alternative form of the input statement (one that is strongly recommended for fixed-width files) is to use an explicit column pointer (@) followed by the position at which the variable is located, the variable name, and the INFORMAT used to read the data into that variable. This method of declaring variables in the INPUT statement is referred to as COLUMN ENTRY. Using column entry, the previous example might be rewritten as follows:

```
FILENAME claims DISK '\\finance\analytics\report\source\2011_07.txt';

DATA work.claims_2011_07;
   INFILE claims FIRSTOBS=2;
   INPUT @1 svc_date mmddyy10. @13 account_no $10.
         @25 prin_dx $6. @31 billed_charges dollar12.;
RUN;
```

In addition to forcing you to confirm your knowledge of the data structure before reading in the data, this method also facilitates validation of your program against data mapping documentation supplied with the file. By explicitly identifying the location and INFORMAT of each variable being read, the intentions of your program are made explicit and errors in the INPUT statement can be detected more readily.

The preceding two examples also introduce an important OPTION to the INFILE statement—FIRSTOBS. FIRSTOBS identifies the row at which the INFILE statement begins reading data from the file. Because the first record on the file "2011_07.txt" contains variable names, FIRSTOBS is set to "2".

DELIMITED, UNIFORM LAYOUT.

Frequently, instead of fixed-width files, users are provided with delimited files (in which a special character is used to separate the variables). In order to read these files using the INFILE statement, the DELIMITER (DLM) option is added. A comma-delimited version of the file "2011_07.txt" might look like this: (Note also that in this example the file has three header records, so FIRSTOBS is set to "4")

```
Report:  ALL_MONTHLY_CLAIMS
Date Run:  August 1, 2011
svc_date, account_no, principle_dx, billed_charges
07/01/2011,VGH3344562,V712.4,$234.55
07/01/2011,XWY3928957,325.4,$34.55
```

In delimited files, the absolute position of the values within the row is no longer meaningful (see, for example, the values of "billed_charges"); instead, the relative position of the delimited values determines the variable to which each will be assigned. Note, however, that in specifying the INFORMATS in this example, you need to prefix the INFORMAT with a colon (:) to specify that the INPUT statement should read "from the first character after the current delimiter to the number of characters specified in the Informat or to the next delimiter, whichever comes first." (SAS, 2009).

```
DATA work.claims_2011_07;
   INFILE claims FIRSTOBS=4 DLM =',' ;
   INPUT svc_date :mmddyy10. account_no :$10. prin_dx :$6.
         billed_charges :dollar12.;
RUN;
```

In addition to delimiters that can be specified in their human-readable form (e.g., comma [,], pipe [|], caret [^], etc.), one can also specify delimiters using their hexadecimal form—for example, DLM='05'x specifies the TAB delimiter in the ASCII character set.

When reading an external file with a delimiter specified, the DSD (delimiter-sensitive data) option on the INFILE statement indicates that delimiters found within a data value enclosed in quotes should not be used as a reference point to separate values, but should instead be included in the value assigned to the variable. In addition, the DSD option indicates to SAS that repeated delimiters indicate value assignments of "missing".

```
svc_date, account_no, principle_dx, billed_charges,city_state
07/01/2011,VGH3344562,V712.4,$234.55,"LaCrosse, WI"
07/01/2011,XWY3928957,,,"Mount Horeb,WI"
```

By adding the DSD option to the INFILE statement, the successive delimiters in the second record will be interpreted as indicating missing values for "prin_dx" and "billed_charges", and the commas in the values of "city_state" will be retained as part of the value of that variable.

```
DATA work.claims_2011_07;
   INFILE claims FIRSTOBS=2 DLM =',' DSD;
   INPUT svc_date :mmddyy10. account_no :$10. prin_dx :$6.
         billed_charges :dollar12. city_state :$25.;
RUN;
```

VIEWTABLE: Work.Claims_2011_07

	svc_date	account_no	prin_dx	billed_charges	city_state
1	18809	VGH3344562	V712.4	234.55	LaCrosse, WI
2	18809	XWY3928957		.	Mount Horeb, WI

Additional options that can significantly impact the processing of external files are LRECL (logical record length) and the end-of-line options (TRUNCOVER, MISSOVER, STOPOVER, FLOWOVER) that determine what happens when the INPUT statement encounters the end of the external file record before all variables have been assigned a value.

LRECL - By default, the input buffer filled by the INFILE statement has a length of 256 bytes; therefore, if the length of a given row of data in your fileref is greater than 256 (single-byte) characters, the additional data (beyond 256) will be excluded because it is not read into the input buffer. Suppose, for example, that the "2011_07.txt" file had many more variables and some of the records extended beyond this 256 byte limit.

```
svc_date,account_no,principle_dx,billed_charges,...,city_state
07/01/2011,VGH3344562,V712.4,$234.55,…,"LaCrosse, WI"
07/01/2011,XWY3928957,325.4,$34.55,…,"Mount Horeb, WI"
|||||||||||||||||||||||||||||||||...|||
1        10        20        30       ...256
```

If no value is provided for LRECL, the values for "city_state" in the following two records would (by default) be truncated at the 256th character.

```
DATA work.claims_2011_07;
   INFILE claims FIRSTOBS=2 DLM =',' DSD MISSOVER;
   INPUT svc_date :mmddyy10. account_no :$10. <other variables> city_state :$25.;
RUN;
```

VIEWTABLE: Work.Claims_2011_07

	svc_date	account_no
1	18809	VGH3344562
2	18809	XWY3928957

city_state
"La
"Moun

With the LRECL option assigned to accommodate the longest record in the file, the entire line of data will be read into the (expanded) buffer and the full values of "city_state" will be assigned.

```
DATA work.claims_2011_07;
   INFILE claims FIRSTOBS=2 DLM =',' DSD LRECL=500 MISSOVER;
   INPUT svc_date :mmddyy10. account_no :$10. <other variables> city_state :$25.;
RUN;
```

VIEWTABLE: Work.Claims_2011_07

	svc_date	account_no	city_state
1	18809	VGH3344562	LaCrosse, WI
2	18809	XWY3928957	Mount Horeb, WI

END-OF-LINE OPTIONS - If the INPUT statement reaches the end of a record without having assigned all variables a value (even a "null" value), the default behavior is for the INPUT statement to continue reading data by proceeding to the next record and looking for values to assign to the remaining variables in the "current" record—the FLOWOVER option. In the following example, the first record of the dataset has no value for "city_state". If the default FLOWOVER option is allowed to control the reading of the data from this dataset, INPUT will continue to the second line and read "07/01/2011" as the value of "city_state".

```
svc_date,account_no,principle_dx,billed_charges,city_state
07/01/2011,VGH3344562,V712.4,$234.55
07/01/2011,XWY3928957,,,"Mount Horeb,WI"
```

This is rarely the desired behavior. To correct this situation, you can use the MISSOVER option, which keeps the INPUT statement from reading data from the next line and, instead, assigns missing values to all variables that do not yet have a value assigned at the time the end of the data line is reached. Similarly, when reading delimited data (or list-entry fixed-width data), the TRUNCOVER option generates the same result[2]. The STOPOVER option, on the other hand, raises an error and ends the DATA step if the end of a record is reached before all variables have been assigned a value.

To process the preceding dataset without error, the MISSOVER option is used—resulting in the first record having a missing value for "city_state".

```
DATA work.claims_2011_07;
   INFILE claims FIRSTOBS=2 DLM =',' DSD LRECL=500 MISSOVER;
   INPUT svc_date :mmddyy10. account_no :$10. prin_dx :$6.
         billed_charges :dollar12. city_state :$25.;
RUN;
```

[2] See Cates (2001) for in-depth explanation of situations in which these options differ.

VIEWTABLE: Work.Claims_2011_07

	svc_date	account_no	prin_dx	billed_charges	city_state
1	18809	VGH3344562	V712.4	234.55	
2	18809	XWY3928957		.	Mount Horeb, WI

HIERARCHICAL FILES

The previous examples of reading data from a physical file assume that the file has a consistent structure across all lines of data. However, source system files can take on many forms that violate this assumption. One common variant is the hierarchical file—in which header records containing a full complement of identifying information are followed by subordinate records in which data elements from the header are implied by the order of the records on the file.

```
member            svc_date     account_no     cpt         charge
Jones, Mary       07/02/2011   VGH3344562     73564        410.25
                                              99214        150.00
Smith, Michael    07/02/2011   XWY3928957     90761        192.40
                                              82565         25.20
                                              82310         18.90
|||||||10|||||||20|||||||30|||||||40|||||||50|||||||60|63
```

If you were to read the preceding file using the methods demonstrated in the previous examples, only two of the five rows of data (the header records) would contain the member name, service date, and account number—making programmatic summarization of these data by member, service date, or account number impossible.

```
FILENAME  clm_dtl DISK 'C:\_CDMS\SGF 2012\filename\_Data\hierarchy.txt';

DATA work.claim_detail;
   INFILE clm_dtl FIRSTOBS=2  ;
   INPUT @1 member $18. @20 svc_date mmddyy10. @32 account_no $10. @46 cpt $5.
         @52 billed_charges best12.;
RUN;
```

VIEWTABLE: Work.Claim_detail

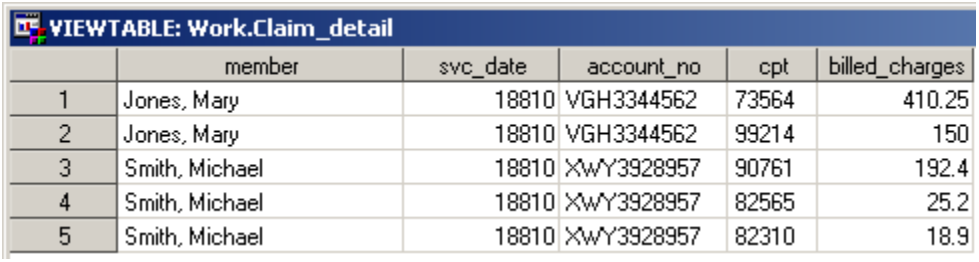
	member	svc_date	account_no	cpt	billed_charges
1	Jones, Mary	18810	VGH3344562	73564	410.25
2		.		99214	150
3	Smith, Michael	18810	XWY3928957	90761	192.4
4		.		82565	25.2
5		.		82310	18.9

The missing data elements in the subordinate rows need to be populated with data from the corresponding header row; this can be achieved using the RETAIN statement. The RETAIN statement allows you to define a variable such that it will "retain" its value across records—until you assign the retained variable a new value. The following code takes advantage of this characteristic of "retained" variables to populate the "member", "svc_date", and "account_no" variables in the "claim_detail" dataset. First, the RETAIN statement specifies the variables that will retain their values. Then, as each record is processed, the value of "member" is assessed to determine if it has a value. If the current record has a non-null value for “member”, the values of the retained variables “member_r”, "svc_date_r", and "account_no_r" are assigned the values of "member", "svc_date", and "account_no", respectively, from that record. Once assigned a value, these retained variables carry that value forward from record to record until a new value is assigned—which occurs, in this example, when the next header record is encountered. When records with null values of "member" (i.e., the subordinate rows) are encountered, the retained values are used to assign values to "member", "svc_date", and "account_no".

```
FILENAME  clm_dtl DISK '\\finance\analytics\report\source\2011_07_DETAIL.txt';

DATA work.claim_detail;
     RETAIN member_r svc_date_r account_no_r;
     INFILE clm_dtl FIRSTOBS=2  ;
```

```
      INPUT @1 member $18. @20 svc_date mmddyy10.
            @32 account_no $10. @46 cpt $5.
            @52 billed_charges best12.;
IF member ne '' THEN DO;
   member_r = member;
   svc_date_r = svc_date;
   account_no_r = account_no;
   END;
ELSE DO;
   member = member_r;
   svc_date = svc_date_r;
   account_no = account_no_r;
 END;
DROP member_r svc_date_r account_no_r;
RUN;
```

The result is a SAS dataset that has the member, service date, and account number assigned to each record. Now the data can be rolled up, sorted, or transposed by any of the variables in the dataset.



VIEWTABLE: Work.Claim_detail

	member	svc_date	account_no	cpt	billed_charges
1	Jones, Mary	18810	VGH3344562	73564	410.25
2	Jones, Mary	18810	VGH3344562	99214	150
3	Smith, Michael	18810	XWY3928957	90761	192.4
4	Smith, Michael	18810	XWY3928957	82565	25.2
5	Smith, Michael	18810	XWY3928957	82310	18.9

MULTI-LINE FILES

Another file layout that requires a specialized approach is that in which data for a single record is written across multiple lines in the source file. In the following example, each record is written across two lines of data. The first line contains the member name, service date, and account number; the second line contains the current procedural terminology (CPT) code associated with a billing line-item and the billed charge.

```
Jones, Mary       07/02/2011   VGH3344562
73564       410.25
Jones, Mary       07/02/2011   VGH3344562
99214       150.00
Smith, Michael    07/02/2011   XWY3928957
90761       192.40
Smith, Michael    07/02/2011   XWY3928957
82565        25.20
Smith, Michael    07/02/2011   XWY3928957
82310        18.90
|||||||10|||||||20|||||||30|||||||40||
```

To read in such multi-line records, you need to use the line pointer control (#) in the INPUT statement to indicate the line being defined by the INPUT syntax that follows.

```
FILENAME  clm_dtl DISK '\\finance\analytics\report\source\2011_07_DETAIL.txt';
DATA work.claim_detail;
   INFILE clm_dtl FIRSTOBS=2  ;
   INPUT #1 @1 member $18. @20 svc_date mmddyy10. @32 account_no $10.
         #2 @1 cpt $5. @7 billed_charges best12.;
RUN;
```

For each record represented in the file, after the variables from the first line of data are read, input continues from the second line of data—resulting in a single record comprised of data elements from each of the two lines.

VIEWTABLE: Work.Claim_detail

	member	svc_date	account_no	cpt	billed_charges
1	Jones, Mary	18810	VGH3344562	73564	410.25
2	Jones, Mary	18810	VGH3344562	99214	150
3	Smith, Michael	18810	XWY3928957	90761	192.4
4	Smith, Michael	18810	XWY3928957	82565	25.2
5	Smith, Michael	18810	XWY3928957	82310	18.9

WRITING TO EXTERNAL FILES

Finally, in addition to reading from external files, one can also write to a fileref using FILE and PUT statements. In the following example, the "claim_detail" dataset is written to "c:\test.txt" as a comma-delimited file with the "member" field excluded. It should be noted here that not only is it easy to control which columns are included in the output file (by including the variable name in the PUT statement), but you can also rearrange the order of the variables in the new file by placing them in the desired output order in the PUT statement.

```
FILENAME  clm_dtl DISK 'c:\test.txt';

DATA _NULL_;
FILE clm_dtl DLM=',';
 SET work.claim_detail;
 PUT account_no svc_date :MMDDYY10. cpt billed_charges :DOLLAR12.2;
RUN;
```

Similarly, using column pointers to place variables at specific locations on the file record, you could create a new fixed-width text file.

```
FILENAME  clm_dtl DISK 'c:\test.txt';

DATA _NULL_;
FILE clm_dtl;
 SET work.claim_detail;
 PUT @10 account_no @30 svc_date mmddyy10. @50 cpt @70 billed_charges dollar10.2;
RUN;
```

This type of manipulation is often required for the submission of flat file datasets to regulatory agencies or vendors that require specific file layouts for input to their data processing systems.

```
          VGH3344562          07/02/2011          73564                $410.25
          VGH3344562          07/02/2011          99214                $150.00
          XWY3928957          07/02/2011          90761                $192.40
          XWY3928957          07/02/2011          82565                 $25.20
          XWY3928957          07/02/2011          82310                 $18.90
|||||||10|||||||20|||||||30|||||||40|||||||50|||||||60|||||||70|||||||79
```

INTERACTING WITH FTP SERVERS

The examples in the previous section will help you get started reading and writing files to/from your local computer or LAN directory, but enterprise-level files are sometimes made available only through a central FTP server. Although obtaining these files via your favorite FTP client is usually a relatively simple task, the FILENAME statement's FTP access method can be used to fully automate the retrieval and processing of these files.

In the following example, the "claims" fileref is specified as a pointer to the file "2011_07.txt" via the FTP access method. When the fileref is referenced later in the program, SAS will attempt to establish an FTP connection to the server (HOST) "cdms-llc.com", using the credentials of user "chris". Once connected to the FTP server, the CD option is used to change the working directory on the FTP server to the directory containing the source file

"2011_07.txt"[3]. Finally, the PROMPT and DEBUG options are included, respectively, to issue a prompt for entry of the user's password and to write informational messages about the FTP transaction to the SAS log.

```
FILENAME claims FTP '2011_07.txt'
  USER = 'chris'
  HOST = 'cdms-llc.com'
    CD = "ftproot\public\" PROMPT DEBUG;
```

Once the fileref is defined, you are ready to use it in a DATA step. Note in the following example that the DATA step also creates a new variable "discount_rate" based on the conditional processing of a variable being read by the INPUT statement. This demonstrates that as soon as a variable is defined in the INPUT statement, it exists as a variable on the current record and can be used in other transformations performed in the DATA step.

```
DATA monthly_claims;
   INFILE claims FIRSTOBS=2 MISSOVER LRECL=300;
   INPUT @1 svc_date mmddyy10. @13 account_no $10. @25 prin_dx $6.
         @273 billed_charges dollar12.;

   IF billed_charges ne . THEN discount_rate = .8*billed_charges;
      ELSE discount_rate = 0.;
RUN;
```

Once the fileref is accessed by the INFILE statement, the specified FTP connection is attempted and the user will be prompted with a dialog box asking for the password associated with the USER account. After the password is provided and authenticated by the HOST, the connection is completed. At this point the CD command is executed—navigating to the directory containing the source file—and retrieval of the file begins. The data lines are then read sequentially from the input buffer and are parsed into values that are assigned to the defined variables. Because the DEBUG option was specified, the LOG includes the following messages describing the connection and data transfer:

```
NOTE: >>> USER chris
NOTE: <<< 331 Password required for chris.
NOTE: >>> PASS XXXXXXXXXXX
NOTE: <<< 230-Welcome to FTP server
NOTE: <<< 230 User chris logged in.
NOTE: >>> PORT 192,168,1,105,213,54
NOTE: <<< 200 PORT command successful.
NOTE: >>> TYPE A
NOTE: <<< 200 Type set to A.
NOTE: >>> CWD ftproot\public\
NOTE: <<< 250 CWD command successful.
NOTE: >>> PWD
NOTE: <<< 257 "/chris/ftproot/public" is current directory.
NOTE: >>> RETR 2011_07.txt
NOTE: <<< 150 Opening ASCII mode data connection for 2011_07.txt(372 bytes).
NOTE: User chris has connected to FTP server on Host cdms-llc.com .
NOTE: The infile CLAIMS is:
      Filename=2011_07.txt,
      Pathname= "/chris/ftproot/public" is current directory,
      Local Host Name=M2400,
      Local Host IP addr=192.168.1.100,
      Service Hostname Name=cmds-llc.com,
      Service IP addr=64.78.48.48,Service Name=FTP,
      Service Portno=21,Lrecl=300,Recfm=Variable
NOTE: <<< 226 Transfer complete.
NOTE: >>> QUIT
```

[3] It should be noted that the FTP options require syntax specific to the OS and file system hosting the FTP server. As such, you should work with the administrator of the FTP server if you encounter problems with the FTP options sent by your fileref.

In the preceding example, the invocation of the fileref caused SAS to prompt the user for the host system password associated with the userid specified in the FILENAME statement. This is fine if you are working with SAS interactively, but if you want to schedule the program to run unattended, you will need to use the PASS option on the filename statement to provide the associated host-system password. In this case, even if the PROMPT option remains on the statement, the user will not be prompted for a password, because one has already been provided.

```
FILENAME claims FTP  '2011_07.txt'
  USER = 'chris'
  PASS = 'mysecretpassword'
  HOST = 'cdms-llc.com'
    CD = "ftproot\public\" PROMPT DEBUG;
```

Of course, most organizations have policies that forbid the saving of passwords in a plain text form. In order to keep passwords safe from "casual, non-malicious viewing…" (SAS, 2011a) you can encode the password using PROC PWENCODE. In the following example, PROC PWENCODE is used to generate the encoded version of the password.

```
PROC PWENCODE IN='mysecretpassword' METHOD=SAS002;
RUN;
```

Once the PROC step is executed the encoded version of the password is output to the LOG—with the encoding method "{sas002}" indicated as part of the encoded password.

```
99   PROC PWENCODE IN=XXXXXXXXXXXXXXXXXX METHOD=SAS002;
100  RUN;

{sas002}5FAE9D593AF70EB951C670803B44F19538CDCDB301E8A357563480B0

NOTE: PROCEDURE PWENCODE used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds
```

The encoded password can be provided as the value of PASS in the filename statement, and SAS will decode it before sending it to the FTP server.

```
FILENAME claims FTP  '2011_07.txt'
  USER = 'chris'
  PASS = '{sas002}5FAE9D593AF70EB951C670803B44F19538CDCDB301E8A357563480B0'
  HOST = 'schacherer.com'
    CD = "ftproot\public\" DEBUG;
```

It should be noted that encoding the password does not keep the password safe from use by others who could simply copy both your userid and the encrypted password from your program and save it in their own SAS program. Therefore, one additional security measure is to save the encoded password in an external SAS dataset in a secure location—such as a directory to which only individuals managing the FTP program have access [but see Sherman & Carpenter (2009) for an important discussion of issues related to the safe-keeping of passwords]. To add this extra layer of security, first create a SAS dataset (again, in a secure location, but one to which the program will have access when scheduled to run from your system's job scheduler) and assign the encoded password to a variable in that dataset.

```
LIBNAME safe 'c:\';

DATA safe.passwords;
  pw = '''{SAS002}5FAE9D593AF70EB951C670803B44F19538CDCDB301E8A357563480B0''';
RUN;
```

Then alter the FTP program to query the value from this dataset into a macro variable (in this case, "password") before executing the filename statement.

```
PROC SQL NOPRINT;
 SELECT pw INTO :PASSWORD
 FROM safe.passwords;
QUIT;
```

Once the macro variable is assigned the value of the password, it can be assigned as the value of "PASS", and the password is no longer shared in the .sas file.

```
FILENAME claims FTP  '2011_07.txt'
  USER='chris'
  PASS=&PASSWORD
  HOST = 'schacherer.com'
  CD ="ftproot\public\" DEBUG;
```

The previous FTP example involved reading data from a single file on a single server. However, when creating analytic datasets, the product of your data processing sometimes needs to be sent to another location for further processing (e.g., loaded into a business intelligence system, sent to a third-party data aggregator, or reported to a regulatory agency). In the following example, two filerefs ("source" and "target") based on the FTP access method are defined. Following assignment of the filerefs, a DATA _NULL_ step is executed to read the external file "source", perform a series of data transformations, and write a comma-delimited file out to the file "target".

```
FILENAME source FTP  '2011_07.txt' USER='chris' PASS=&PASSWORD
         HOST = 'cdms-llc.com' CD ="ftproot\public\" DEBUG;
FILENAME target FTP  'HOSPITALXYZ_QUALRPT_2011_07.txt' USER='chris' PASS=&PASSWORD
         HOST = '<state hospital authority>' CD ="reports\hospital\XYZ\" DEBUG;

DATA _NULL_;
   INFILE source FIRSTOBS=2 MISSOVER LRECL=300;
   INPUT @1 svc_date mmddyy10. @13 account_no $10. @25 prin_dx $6.
         @273 billed_charges dollar12.;
   IF billed_charges ne . THEN discount_rate = .8*billed_charges;
      ELSE discount_rate = 0.;
   …
   <additional transformations>
   …
  FILE target DLM=',';
  PUT account_no svc_date :mmddyy10. cpt billed_charges :dollar12.2;
RUN;
```

If you wanted to automate this program to run each month as a scheduled batch job, you could change the program in the following manner to enable it to run on the first day of each month without having to edit the program to change the source file name. In this example, it is assumed that the source file is written on the last day of each month with the filename in the format of "YYYY_MM". Therefore, using the appropriate nested functions, you could dynamically specify the name of the source file to be read. When run in September, 2011 the concatenated macro variables in the FILENAME statement resolve to the name of the source file generated at the end of the previous month—"2011_08.txt".

```
%LET SOURCE_FILE1 = %SYSFUNC(YEAR(%SYSFUNC(INTNX(MONTH,%SYSFUNC(TODAY()),-1))))_;
%LET SOURCE_FILE2 = %SYSFUNC(MONTH(%SYSFUNC(INTNX(MONTH,%SYSFUNC(TODAY()),-1))),Z2.).txt;

FILENAME source FTP  "&source_file1&source_file2" USER='chris' PASS=&PASSWORD
        HOST = 'schacherer.com' CD ="ftproot\public\" DEBUG;
```

Whether you use macro variables to dynamically assign filenames, directory locations, or other parts of the fileref or hard-code values and manage changes to them manually, the FTP access method can significantly improve the efficiency with which you process data files stored on FTP servers. It should be noted, however, that because the FTP servers with which you will be interacting are often on different hardware/software platforms than the one on which your SAS software is operating, you may encounter unfamiliar errors referring to system constructs you do not understand. When that happens, work with the administrator of the FTP server to troubleshoot the issue; he or she has expertise on the particular platform to which you are attaching and will likely be able to help you debug your code.

In the previous examples the FTP device type was used to connect to remote FTP servers, issue FTP commands, and retrieve files for processing in SAS. Similarly, the PIPE command can be used to initiate a wide array of processes and capture the data returned by those processes.

PIPES

"A pipe is a channel of communication between two processes." (SAS, 2011b) An unnamed pipe establishes one-way communication between two processes—wherein one process writes to the pipe and the other reads from it. A named pipe enables two-way communication wherein data can pass in both directions. The examples provided here focus on unnamed pipes in which SAS invokes a process outside of SAS and then reads the data written to the pipe by the invoked process. The following example (adapted from Varney, 2008) shows how to use the PIPE device type to create a dataset containing the hierarchical directory structure of the directory "C:\ETL". The fileref "contents" is defined in terms of a call to the DOS command "TREE". When the INFILE statement initiates retrieval of the fileref "contents" the DOS command "TREE C:\ETL" is executed with the switches "/F" and "/A". Instead of returning these data to a DOS command window, as would happen if you were executing this command from the command line, the data are returned through the communication channel (the pipe) defined in the filename statement.

```
FILENAME contents PIPE 'TREE "C:\ETL" /F /A' LRECL=2000;

DATA etl_source;
 INFILE contents TRUNCOVER;
 INPUT content_entry $char2000.;
RUN;
```

VIEWTABLE: Work.Etl_source

	content_entry
1	Folder PATH listing for volume OS
2	Volume serial number is A0C1-1085
3	C:\ETL
4	2011_04.txt
5	2011_05.txt
6	2011_06.txt
7	2011_07.txt
8	2011_08.txt
9	
10	No subfolders exist
11	

You could use this information (for example) in an automated program to check for the existence of a given file and conditionally stop the program if the file is not found (Flavin & Carpenter, 2001; Schacherer, 2010). Having read the directory contents with the preceding DATA step, the following code queries the "etl_source" dataset to determine whether the "C:\ETL" directory contains the source file your program will later attempt to process. The result of this query is assigned to the macro variable "file_exists", and the value of "file_exists" can then be evaluated to determine the appropriate course of action.

```
PROC SQL NOPRINT;
 SELECT count(*) INTO :file_exists
   FROM etl_source
  WHERE compress(content_entry) = "2011_08.txt";
QUIT;
```

If the source file was found in the "etl_source" dataset, the macro variable "file_exists" will have a value of "1" and the outcome of the following DATA _NULL_ step will be the assignment of a null value to the macro variable "terminator". When "&terminator" is encountered later in the program, it will resolve to a null, nonexistent value that is not interpreted as executable SAS code. If, on the other hand, the file was not found in the query of "etl_source" (i.e., "file_exists" = 0), "terminator" is assigned the value "ENDSAS;", and when "&terminator" resolves following the DATA step, the SAS session is terminated.

```
DATA _NULL_;
 IF &file_exists = 1 THEN
 CALL SYMPUT('terminator','');
 ELSE
 CALL SYMPUT('terminator','ENDSAS;');
RUN;

&terminator
```

A slight twist on this use of PIPE could be used to perform more sophisticated tests of the source file being processed by this program. In the following example, suppose that you have been performing the processing of this monthly file for several years and you know that it is always larger than 500 MB. You could evaluate the size of the file before processing it by using the following PIPE command to issue a "DIR" command of the "ETL" directory.

```
FILENAME contents PIPE 'DIR "C:\ETL" /A' LRECL=2000;
DATA etl_source;
 FORMAT object_date mmddyy10.;
 INFILE contents TRUNCOVER;
 INPUT @1 full_line $200. @1 object_date mmddyy10.
       @22 file_size comma17. @40 file_name $15.;
RUN;
```

This DATA step results in the following dataset:

VIEWTABLE: Work.Etl_source

	object_date	full_line	file_size	file_name
1	.	Volume in drive C is OS	.	
2	.	Volume Serial Number is A0C1-1085	.	
3	.		.	
4	.	Directory of c:\etl	.	
5	.		.	
6	03/17/2011	03/17/2011 01:14 PM <DIR> .	.	
7	03/17/2011	03/17/2011 01:14 PM <DIR> ..	.	..
8	04/01/2011	04/01/2011 02:03 AM 1,211,915,127 2011_04.txt	1211915127	2011_04.txt
9	05/01/2011	05/01/2011 02:22 AM 1,402,382,907 2011_05.txt	1402382907	2011_05.txt
10	06/01/2011	06/01/2011 02:30 AM 1,432,722,545 2011_06.txt	1432722545	2011_06.txt
11	07/01/2011	07/01/2011 03:15 AM 1,685,554,646 2011_07.txt	1685554646	2011_07.txt
12	08/01/2011	08/01/2011 12:55 AM 200,581,745 2011_08.txt	200581745	2011_08.txt
13	.	5 File(s) 5,933,156,970 bytes	.	bytes
14	.	2 Dir(s) 15,533,596,672 bytes free	.	bytes free

By adding a conditional statement, you can reduce the dataset to contain only records associated with individual files.

```
DATA etl_source;
 FORMAT object_date mmddyy10.;
 INFILE contents TRUNCOVER;
 INPUT @1 full_line $200. @1 object_date mmddyy10.
       @22 file_size comma17. @40 file_name $15.;
 IF object_date = . or file_size = . THEN DELETE;
RUN;
```

VIEWTABLE: Work.Etl_source

	object_date	full_line	file_size	file_name
1	04/01/2011	04/01/2011 02:03 AM 1,211,915,127 2011_04.txt	1211915127	2011_04.txt
2	05/01/2011	05/01/2011 02:22 AM 1,402,382,907 2011_05.txt	1402382907	2011_05.txt
3	06/01/2011	06/01/2011 02:30 AM 1,432,722,545 2011_06.txt	1432722545	2011_06.txt
4	07/01/2011	07/01/2011 03:15 AM 1,685,554,646 2011_07.txt	1685554646	2011_07.txt
5	08/01/2011	08/01/2011 12:55 AM 200,581,745 2011_08.txt	200581745	2011_08.txt

You can then programmatically determine whether processing should continue by evaluating the size of the current source file (2011_08.txt). First, the macro variable "file_size" is assigned a value corresponding to the size of the file in megabytes—approximately 200.

```
PROC SQL NOPRINT;
 SELECT PUT(file_size/1000000,best12.) INTO :file_size
   FROM etl_source
  WHERE compress(file_name) = "2011_08.txt";
QUIT;
```

Then, a DATA _NULL_ step is used to conditionally assign a value to the macro variable "terminator" based on the value of "file_size". If "file_size" is less than 500, the "terminator" macro variable is assigned a value containing a %PUT statement followed by the ENDSAS command. When "terminator" resolves following the DATA _NULL_ step, an error message will be written to the log file, and the SAS session and will be terminated.

```
DATA _NULL_;
IF &file_size < 500 THEN
CALL SYMPUT('terminator','%PUT Error-Source file too small to continue; ENDSAS;');
ELSE
CALL SYMPUT('terminator','');
RUN;

&terminator
```

Finally, in addition to using PIPE to execute commands that generate information consumed by SAS, the PIPE device type can be used to execute commands that impact the system on which SAS is running. As demonstrated by Varney (2008) and Wei (2009), filerefs indicating the PIPE device type can be used to issue MKDIR commands to create new directories in either a static or dynamic manner. In the following example, a new directory "c:\transformed claims" is created so that it can be used to save a copy of a SAS dataset from the WORK library to a persistent storage area.

```
FILENAME new_dir PIPE 'MKDIR "c:\transformed claims"' ;

DATA _NULL_;
 INFILE new_dir;
RUN;

LIBNAME persist 'c:\transformed claims';

DATA persist.claims_2011_08;
 SET work.claims_2011_08;
RUN;
```

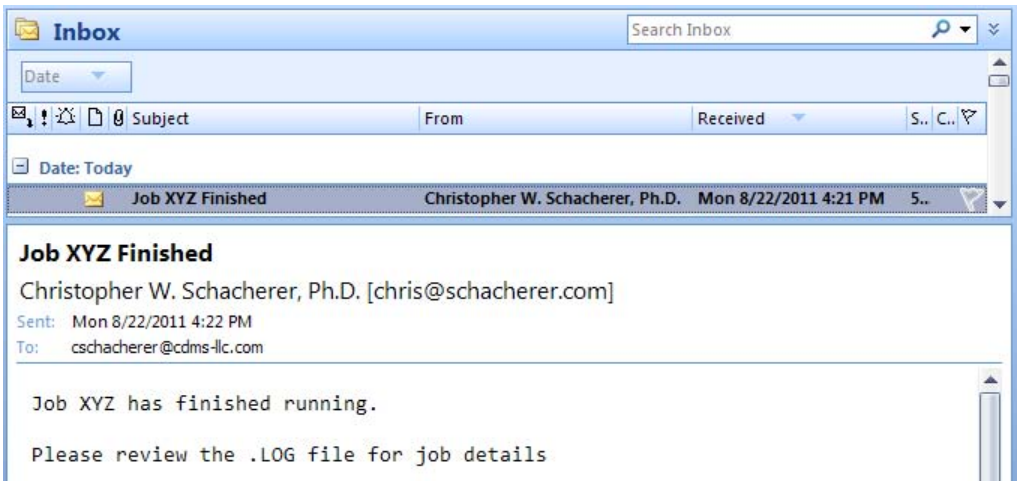
These examples merely scratch the surface of what can be achieved with the PIPE device type. The ability to interact with other programs and processes opens up a number of possibilities for developing creative SAS solutions. Another device type that allows you to do some very creative things with SAS is the EMAIL device type.

E-MAILING FROM SAS

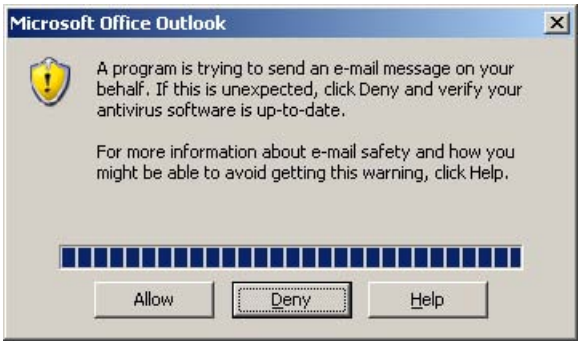
Using the EMAIL device type, you can send "success/failure" messages at the end of long-running processes, keep track of multi-stage processes by sending status messages following milestone events in your programs, and automate the delivery of reports and output. For example, you could put the following code at the end of your SAS program to notify you when the program has finished. In this example, the fileref "job_done" is defined with the EMAIL access method, and the options "to" and "subject" are added to define the recipient of the e-mail and the subject line. When referenced in the DATA _NULL_ step, the fileref, by default, invokes the installed MAPI client to send the e-mail using the default user account. The PUT statements that follow the FILE statement define the content of the e-mail message, and the message is queued for delivery when the RUN statement is encountered.

```
FILENAME job_done EMAIL to="cschacherer@cdms-llc.com"
                        subject="Job XYZ Finished";
DATA _NULL_;
 FILE job_done;
 PUT "Job XYZ has finished running.";
 PUT " ";
 PUT "Please review the .LOG file for job details";
RUN;
```

The delivered message is depicted below in the recipient's e-mail inbox.



As previously mentioned, the EMAIL access method defaults to using your system's MAPI client. Unfortunately, for many users, programmatically invoking your e-mail client will result in a dialog box similar to the following:



The user must then explicitly "Allow" the e-mail message to be sent; which is fine if you are working with your SAS session interactively, but causes significant problems if your intention is to schedule the SAS job to run unattended. In the case of unattended SAS programs, you will want to change the option specifying the e-mail system to use when sending e-mail from SAS. By specifying SMTP as your default e-mail system, you bypass the MAPI client and access you SMTP server directly—essentially, generating the e-mail directly on the server instead of sending it through your MAPI client. An example of this approach is provided below.

```
OPTIONS EMAILSYS="SMTP"
        EMAILHOST="mail.<mycompany>.com"
        EMAILPORT=25
        EMAILID="chris@<mycompany>.com"
        EMAILPW=mysecretpassword;
```

Now when the "job_done" fileref is defined with the EMAIL access method, it is referencing SMTP as the e-mail system, instead of MAPI. It will use the e-mail credentials specified in EMAILID and EMAILPW to connect to the SMTP server through port 25 and send the e-mail without going through the e-mail client and requiring manual intervention.

```
FILENAME job_done EMAIL to="cschacherer@cdms-llc.com"
                        subject="Job XYZ Finished";
DATA _NULL_;
 FILE job_done;
 PUT "Job XYZ has finished running.";
 PUT " ";
 PUT "Please review the .LOG file for job details";
RUN;
```

Although the EMAILxxx options can be specified in your SAS program, if you are going to use this method regularly, you may want to change these options in your SAS configuration file (SAS, 2007) instead of specifying them in each program. For SAS 9.2, this file is located by default at:

C:\Program Files\SAS\SASFoundation\9.2\nls\en\sasv9.cfg

The entries are entered at the top of the configuration file as follows:

```
-EMAILSYS="SMTP"
-EMAILHOST="mail.<mycompany>.com"
-EMAILPORT=25
-EMAILID="chris@<mycompany>.com"
-EMAILPW="mysecretpassword";
```

Following these changes to your configuration file, subsequent filerefs specifying the EMAIL access method will be routed directly to the SMTP server instead of going through the MAPI client.

In addition to the SMTP and MAPI mail systems, SAS also provides an interface (VIM – Vendor Independent Messaging) that can be used to connect to other e-mail clients—for example, Lotus Notes—in a fashion similar to the MAPI client (Pagé, 2004).

Regardless of the interface used to generate and send e-mail messages from SAS, there are a number of methods that can be used to customize their content and route their delivery. In the following example, a MACRO was used to read an externally generated SAS .LOG file from the SAS session, count the number of occurrences of the text "ERROR", and assign that value to the macro variable "num_errors". The DATA _NULL_ step in the following example is then used to conditionally send one of two e-mails based on the number of errors found in the .LOG file (Schacherer & Steines, 2010).

```
FILENAME job_done EMAIL;

DATA _NULL_;

 FILE job_done;
 IF &num_errors = 0 THEN
    DO;
      PUT '!EM_TO! (BI_Analyst@companyxyz.com)';
      PUT '!EM_CC! (etl_admin@cdms-llc.com)';
      PUT 'The XYZ ETL process has completed successfully';
    END;
 ELSE
    DO;
      PUT '!EM_SUBJECT! JOB FAILURE - Company XYZ ETL ';
      PUT '!EM_TO! (etl_admin@cdms-llc.com etl_backup@cdms-llc.com)';
      PUT '!EM_CC! (BI_Analyst@companyxyz.com BI_Manager@companyxyz.com
                    Reporting_Manager@companyxyz.com)';
      PUT 'The XYZ ETL process has failed';
      PUT ' ';
      PUT 'Please check the .LOG file for the source of errors.';
    END;
RUN;
```

If the program being assessed did not generate any errors during execution, the value of "num_errors" will be "0" and a message stating that the process completed successfully will be sent to the BI Analyst at Company XYZ and will be Cc'd to the ETL Administrator at CDMS, LLC. If, on the other hand, errors are encountered during execution (i.e., "num_errors" is not "0"), a different subject line and message will be generated, and the "To" and "Cc" lists will be altered to communicate the failure to a broader list of recipients.

In addition to the use of conditional processing to control e-mail messages, this example also introduces the use of the !EM_<directive>! notation to override e-mail attributes defined in either the FILENAME or FILE statements. As demonstrated in the first EMAIL example, EMAIL options can be set in the FILENAME statement:

```
FILENAME job_done EMAIL to="cschacherer@cdms-llc.com"
                        subject="Job XYZ Finished";
```

When established as part of the fileref definition, these options serve as the default values for emails generated by invoking that fileref. However, EMAIL options can also be assigned in the FILE statement, and options indicated in the FILE statement override those defined in the FILENAME statement. In the following example, even though the "job_done" fileref specifies "cschacherer@cdms-llc.com" as the recipient, the e-mail will be sent to "admin@cdms-llc.com" as specified in the FILE statement.

```
DATA _NULL_;
 FILE job_done to="admin@cdms-llc.com" subject="Job 123 Finished";
 PUT "Job XYZ has finished running.";
 PUT " ";
 PUT "Please review the .LOG file for job details";
RUN;
```

Similarly, the !EM_<directive>! notation embedded in a PUT statement can be used to override options specified in the FILENAME and FILE statements. This notation can also be used to control the e-mail activity being generated by the DATA step. The following example (adapted from Tilanus, 2008 & SAS, 2011c), depicts a dataset of monthly healthcare claim totals for a set of client companies. Each record in the dataset contains the company name, the e-mail address of a contact person at that company, and the healthcare claims total for the month. Using the PUT statement and the !EM_<directive>! notation, an e-mail containing the total claims for a given company will be sent to that company's contact.

VIEWTABLE: Work.Claim_total_list

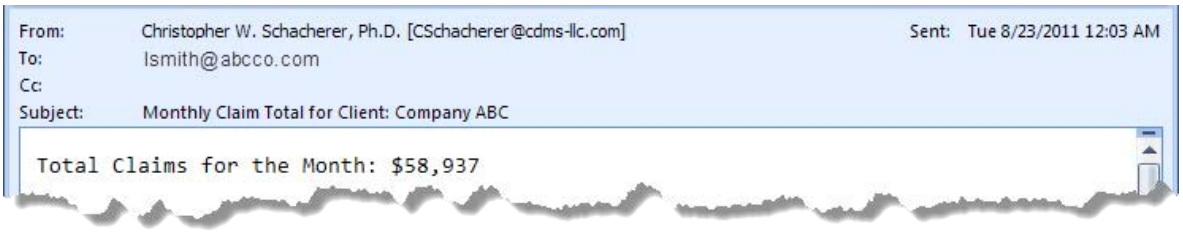
	email_address	company	claim_total
1	mjones@xyzco.com	Company XYZ	203523.11
2	lsmith@abcco.com	Company ABC	58937.23

As the dataset "claim_total_list" is processed in the following DATA _NULL_ step, each invocation of the "claims" file reference will generate an e-mail for which the "subject" and "to" attributes are dynamically assigned using the values of "company" and "email_address" in the current record, and the body of each e-mail will dynamically change to include the "claim_total".

```
FILENAME claims EMAIL;

DATA _NULL_ ;
  SET claim_total_list END=last_record;
  FORMAT claim_total dollar12.;
 FILE claims ;
      PUT '!EM_SUBJECT! Monthly Claim Total for Client:' company;
      PUT '!EM_TO!' email_address;
      PUT 'Total Claims for the Month: ' claim_total;
      PUT '!EM_SEND!';
      PUT '!EM_NEWMSG!';
    IF last_record THEN PUT '!EM_ABORT!';
RUN;
```

!EM_SEND! and !EM_NEWMSG! play an important role in generating multiple e-mail messages from the "claim_total_list" dataset. Without these two directives, only one e-mail would be sent; after all of the records have been processed, a single e-mail would be sent to the e-mail address listed on the last record and the subject line of the e-mail would include the name of the company on that last record. As each record is encountered in the DATA step, the !EM_TO! and !EM_SUBJECT! directives reassign the value of the "subject" and "to" e-mail attributes, but by default no e-mail message is sent until the DATA step completes. Interestingly, however, the PUT statements are still being executed for each record in the dataset; so, in addition to the "to" and "subject" attributes being reassigned for each record, a line of text stating "Total Claims for the Month: …" is being generated to the body of the e-mail.

The !EM_SEND! directive makes sure that a "send" is executed for each record in the dataset. After the e-mail message is completed and sent, the !EM_NEWMSG! directive purges the content of the message associated with the current record, so that the next record processed by the DATA step creates a new message associated with that record. Finally, after the last record is processed, the !EM_ABORT! directive is issued to stop processing directed at the fileref at the top of the DATA step. The result is one e-mail sent for each record in the dataset.

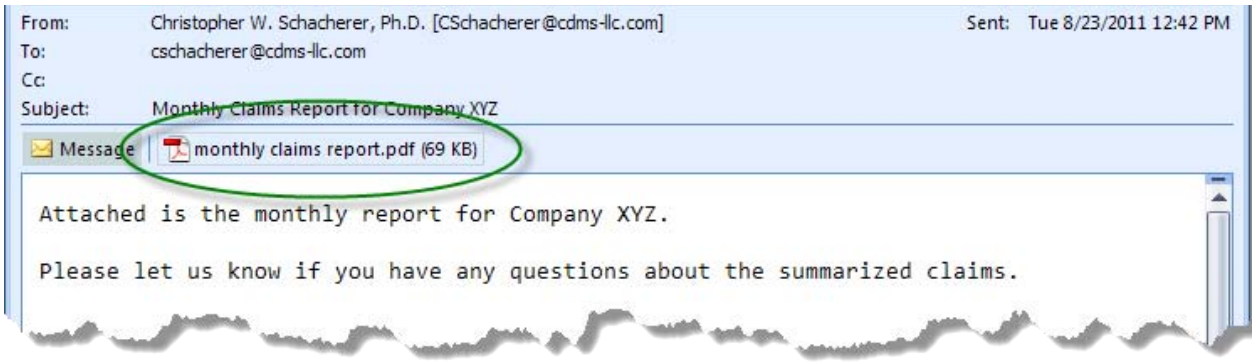


In addition to generating data-driven, dynamic e-mails that incorporate SAS data into the body of the e-mail, the EMAIL access method can also be used to deliver output generated by the Output Delivery System® (ODS) as an attachment to the e-mail message. In the following example, a monthly claims report is generated for client company "xyz" by running a PROC TABULATE within an ODS statement.

```
ODS PDF FILE='C:\monthly claims report.pdf' STYLE = Journal
TITLE 'Claim Payments by Paid Date';
PROC TABULATE DATA=xyz;
  CLASS datepaid;
  VAR   payment ;
  TABLE datepaid='',sum=''*(payment='Claim Total')*FORMAT=DOLLAR20.
        /BOX='Paid Date';
RUN;
ODS PDF CLOSE;
```

Once the file is generated, it is attached to an e-mail by specifying the name and location of the file in the ATTACH atttribute of the e-mail.

```
FILENAME report EMAIL;

DATA _NULL_;
 FILE report to="cschacherer@cdms-llc.com"
             subject="Monthly Claims Report for Company XYZ "
             attach="c:\monthly claims report.pdf";
 PUT "Attached is the monthly report for Company XYZ.";
 PUT " ";
 PUT "Please let us know if you have any questions about the summarized claims.";
RUN;
```



Like the other e-mail attributes, "attach" can be specified either on the FILENAME statement or the FILE statement, and there is an !EM_<directive>! form of "attach" that can be used to add greater flexibility to the e-mail by attaching alternative reports based on specific conditions. In the following example, different e-mails (and attachments) are sent depending on the value assigned to a macro variable "range_deviations". If "range_deviations" exceeds "0", a detailed report is sent; otherwise, the recipient receives the normal monthly report.

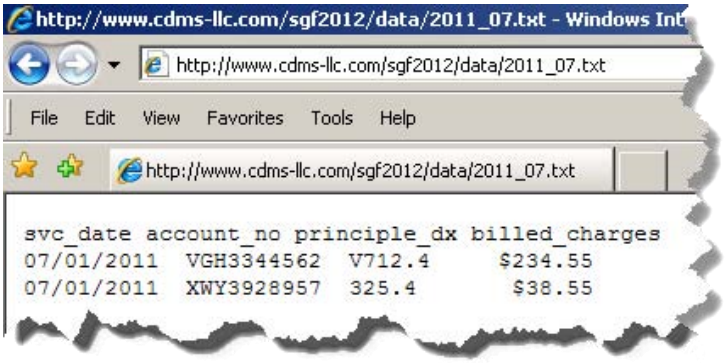
```
FILENAME report EMAIL;

DATA _NULL_;
 FILE report to="cschacherer@cdms-llc.com"
             subject="Monthly Claims Report for Company XYZ ";

 IF &range_deviations > 0 THEN DO;
   PUT '!EM_ATTACH! "c:\monthly claims report - detail.pdf" ';
   PUT "The monthly report revealed deviations from the expected range.";
   PUT " ";
   PUT "Please review the attached detailed report.";
 END;
 ELSE DO;
   PUT '!EM_ATTACH! "c:\monthly claims report.pdf" ';
   PUT "Attached is the monthly report for Company XYZ.";
   PUT " ";
   PUT "Please let us know if you have any questions about the summarized claims.";
 END;
RUN;
```

With so many ways to customize the delivery of information from your SAS programs, it is easy to see why the EMAIL access method appeals to so many users[4]. But whereas the EMAIL access method can facilitate your ability to disseminate data and information from your SAS programs, the URL access method has the potential to profoundly impact how data are integrated into your analytic solutions from a variety of different data sources.

URL

The URL access method has been utilized in a number of creative solutions (see, for example, DeGuire, 2007; Zdeb, 2010; Fuller, 2010; Langston, 2009; Bartlett, Bieringer, & Cox, 2010). In the basic example described below, a text file is posted on a website, and the URL access method is used to access the file in a manner similar to the FTP method. The "claims" fileref is defined as the URL specifying the location of the file, and then the file is accessed using an INFILE statement in the context of a DATA step.

[4] For additional examples of the EMAIL access method, the reader is referred to Schacherer (2008), DeGuire (2007), Pagé (2004), Hunley (2010), and Tilanus (2008).




```
FILENAME claims URL 'http://www.cdms-llc.com/sgf2012/data/2011_07.txt';

DATA work.claims_2011_07;
   INFILE claims FIRSTOBS=2;
   INPUT svc_date mmddyy10. account_no $12. prin_dx $8. billed_charges dollar12.;
RUN;
```

In order to facilitate the resolution of errors encountered using the URL access method, it is a good idea to add the DEBUG option to the FILENAME statement. The DEBUG option will generate detailed information to the LOG regarding the HTTP communications between SAS and the web server. Without the DEBUG option, the following error (caused by misspelling the source file name) can be a bit confusing.

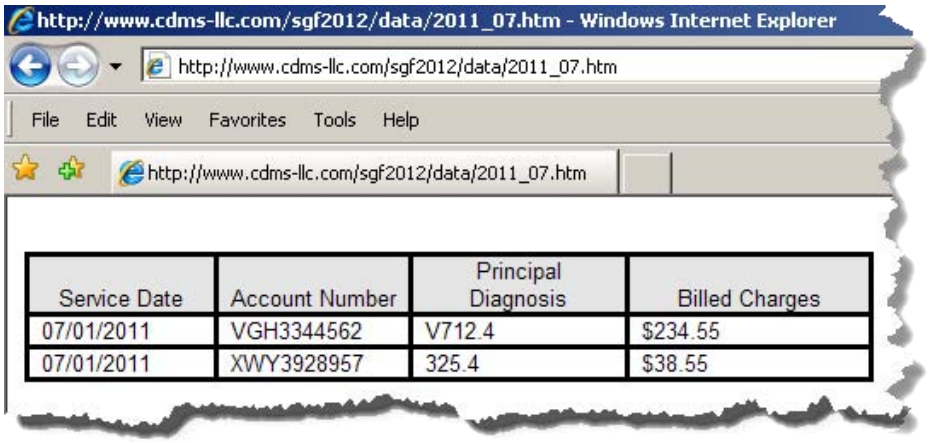
```
38   FILENAME claims URL 'http://www.cdms-llc.com/sgf2012/data/201107.txt';
39
40   DATA work.claims_2011_07;
41      INFILE claims FIRSTOBS=2;
42      INPUT svc_date mmddyy10. account_no $12. prin_dx $8. billed_charges dollar12.;
43   RUN;

ERROR: Invalid reply received from the HTTP server. Use the debug option for more info.
```

Re-running the same code with the DEBUG option specified, you can see the GET request being sent from SAS, and the response informs you that the specified web page was not found ("HTTP/1.1 404 Not Found").

```
FILENAME claims URL 'http://www.cdms-llc.com/sgf2012/data/201107.txt' DEBUG;

DATA work.claims_2011_07;
   INFILE claims FIRSTOBS=2;
   INPUT svc_date mmddyy10. account_no $12. prin_dx $8. billed_charges dollar12.;
RUN;

NOTE: >>> GET /sgf2012/data/201107.txt HTTP/1.0
NOTE: >>> Host: www.cdms-llc.com
NOTE: >>> Accept: */*.
NOTE: >>> Accept-Language: en
NOTE: >>> Accept-Charset: iso-8859-1,*,utf-8
NOTE: >>> User-Agent: SAS/URL
NOTE: >>>
NOTE: <<< HTTP/1.1 404 Not Found
NOTE: <<< Content-Length: 1635
NOTE: <<< Content-Type: text/html
NOTE: <<< Server: Microsoft-IIS/6.0
NOTE: <<< X-Powered-By: ASP.NET
NOTE: <<< Date: Thu, 25 Aug 2011 20:20:32 GMT
NOTE: <<< Connection: close
NOTE: <<<
ERROR: Invalid reply received from the HTTP server. Use the debug option for more info.
```

Correcting the file name addresses this issue, and you are able to access the data file.

Admittedly, serving up raw text files as web pages is not very common; it is more likely that the data you will access with the URL method are served up via the web in a more "human readable" form that makes it attractive and more user-friendly to someone viewing the data—what Richardson & Ruby (2007) refer to as the "human web". The data you are trying to access programmatically is likely in the form of HTML or some other web-centric format focused on making data consumable by humans.



Service Date	Account Number	Principal Diagnosis	Billed Charges
07/01/2011	VGH3344562	V712.4	$234.55
07/01/2011	XWY3928957	325.4	$38.55

As a result, the preceding table of July 2011 claims data is very easy for a human to understand, however, the following example of the data contained in the file "2011_07.htm" suggests that you will need to parse each line of the file using conditional logic and SAS functions in order to produce the corresponding SAS dataset. Although it is absolutely achievable, it often requires a significant bit of trial and error.

```
<tr style='mso-yfti-irow:1'>
  <td width=94 valign=top style='width:70.85pt;border:solid black 2.25pt;
  border-top:none;padding:0in 5.4pt 0in 5.4pt'>
  <p class=MsoNormal><span style='font-size:10.0pt'>07/01/2011</span></p>
  </td>
[additional html]
  <p class=MsoNormal><span style='font-size:10.0pt'>VGH3344562</span></p>
[additional html]
  <p class=MsoNormal><span style='font-size:10.0pt'>V712.4</span></p>
  </td>
[additional html]
  <p class=MsoNormal><span style='font-size:10.0pt'>$234.55</span></p>
  </td>
```

The flip-side of the "human web" is the "programmable web." A term Richardson & Ruby (2007) use to describe "programmer-friendly technologies for exposing a web site's functionality in officially sanctioned ways—RSS, XML-RPC, and SOAP." They argue that these two webs should be reunited with the goal of creating a "network that you can use whether you're serving data to human beings or computer programs." Whereas the current author does not disagree with this goal, he is admittedly biased toward the programmable web—with its drive toward providing data to eager analysts.

In the following example (adapted from Mack, 2010), the URL method is used to query a RESTful web service available at "hipaaspace.com". In this example, the proprietary "getcode" service associated with National Drug Code (NDC) data is being queried to return descriptive data associated with a specific NDC[5]. In order to send a request to the "getcode" service using the URL access method, the parameters expected by the service are assigned to the macro variables "ndc", "return_type", and "pass_code"—representing, respectively, the NDC code for which information is being requested (ndc), the form of the data to be returned in response to the request (return_type), and a security token that identifies you as an authorized user (pass_code)[6]. In assigning the value of macro variable "ndc", %QSYSFUNC (which masks special characters in the arguments of the executed functions) executes the URLENCODE function (which encodes arguments with the URL escape syntax).

[5] Please note that in this example, values for the "pass_code" and "target" macro variables have been wrapped to maintain the readability of the code in the paper; in order to run this example, the whitespace inserted in these values should be removed.

[6] As of the time the paper was written, the security token in this example was available from hipaaspace.com to allow prospective users to test their web services

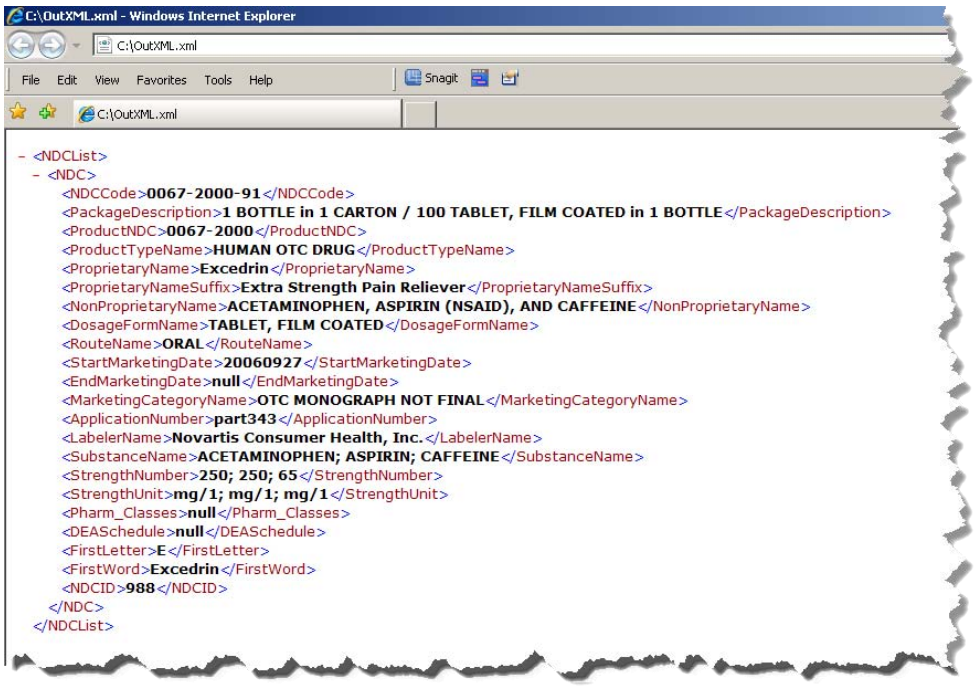
```
%LET ndc          =%QSYSFUNC(URLENCODE(0067-2000-91));
%LET return_type =xml;
%LET pass_code    =3932f3b0-cfab-11dc-95ff-0800200c9a663932f3b0-cfab-11dc-95ff-
                   0800200c9a66;
```

In assignment of the macro variable "target", the %NRSTR function is used to mask special characters in the static text contributing to the URL, and the URL is completed by the inclusion of the resolution of the previously assigned macro variables.

```
%LET target = %NRSTR(http://www.HIPAASpace.com/api/ndc/getcode?q=)&ndc
              %NRSTR(&rt=)&return_type%NRSTR(&token=)&pass_code;
```

Following assignment of the complete URL as the value of "target", the fileref "inurl" is defined in terms of the URL access method using the %SUPERQ function to assign the value of "target" as the URL reference without resolving the macro variable at the time the fileref is assigned. An XML file on the local hard drive is defined as the fileref "outxml"—to which the data read from the web service will be written. The subsequent DATA _NULL_ step then connects to the specified URL, reads the output returned from the web service, and writes the XML data out to the "outxml" file.

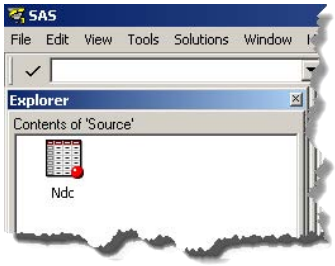
```
FILENAME inurl URL "%SUPERQ(target)" LRECL=4000 DEBUG;
FILENAME outxml "c:\OutXML.xml";

DATA _NULL_;
INFILE inurl LENGTH=len;
 INPUT record $varying4000. len;
  FILE outxml NOPRINT NOTITLES RECFM=n;
   PUT record $varying4000. len;
RUN;
```



If the goal of your program was simply to generate the XML file for input into another system, you could simply write the "outxml" file to the specified location (e.g., LAN, FTP server, etc.) and your task would be complete. If, on the other hand, you want to create a SAS dataset based on the XML file, you can use the XML LIBNAME engine to create a SAS library through which you can access the data contained in the structured XML file.

In the following example, the fileref "outxml" is assigned as the value of XMLFILEREF to identify the XML file to which the "source" library is referring. The result is a library that provides access to the data stored in "c:\OutXML.xml".

```
LIBNAME source XML XMLFILEREF=outxml;
```



Alternatively, the LIBNAME can be defined as follows:

```
LIBNAME source XML 'C:\outxml.xml';
```

Once the "source" library is defined, you can navigate to that library and see that the NDC dataset exists there. You can operate against these data using DATA and PROC steps, but (as is noted in the following error) the SAS XML engine does not support double-clicking on the dataset to open it in browse mode.

```
ERROR: The PROC has requested that the XML Engine perform an unsupported function. As a work-
       around you might consider copying the data into the WORK library and having the PROC
       process from the copy.
```

You can, however, save a copy of the dataset to another library—creating a native SAS dataset—and then open that SAS dataset in browse mode.

```
DATA work.ndc_from_xml;
 SET source.ndc;
RUN;
```

VIEWTABLE: Work.Ndc_from_xml

	NDCID	FIRSTWORD	FIRSTLETTER	DEASCHEDULE	PHARM_CLASSES	STRENGTHUNIT	STRENGTHNUMBER	SUBSTANCENAME	LABELERNAME
1	988	Excedrin	E	null	null	mg/1; mg/1; mg/1	250; 250; 65	ACETAMINOPHEN; ASPIRIN; CAFFEINE	Novartis Consumer H

It should also be noted that beginning in SAS 9.2, access to RESTful web services can be simplified using PROC HTTP. Utilizing this PROC, you can simplify the previous example of making a call to the "getcode" web service by submitting your request in a more concise manner as follows:

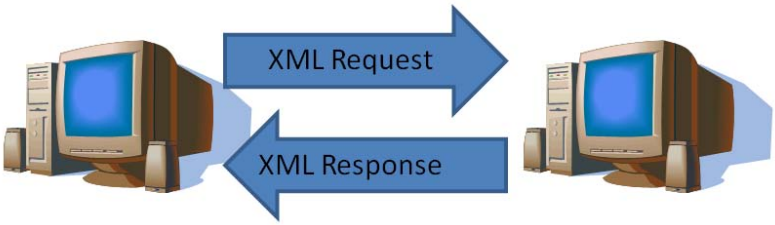
```
FILENAME outfile "c:\OutXML.xml";
PROC HTTP
     OUT = outfile
     URL = 'http://www.HIPAASpace.com/api/ndc/getcode?q=0067-2000-91&rt=xml
           &token=3932f3b0-cfab-11dc-95ff-0800200c9a663932f3b0-cfab-...'
  METHOD = 'GET';
RUN;
```

The most obvious advantage provided by PROC HTTP is the ability to specify the URL without the need for the complex assignment of macro variable values needed to implement the FILENAME approach.

TEMP

In the preceding section, the URL access method was used to call a web service based on REST methodologies—a so-called RESTful web service. Another common type of web service—that based on simple object access protocol (SOAP) is used to demonstrate the TEMP access method. An in-depth discussion of web service architectures (or even the SOAP protocol) is beyond the scope of this paper, but the reader is referred to Mack (2010), Jahn (2008), and Pratter (2011) for further discussion of interacting with web services using SAS. For the purpose of the present discussion, it will suffice to differentiate requests sent to RESTful and SOAP web services by describing the RESTful web service request as the specification of a particular URL (with the appropriate parameter-value pairs) and the SOAP web service request as one in which a structured request message is sent to the web service.

As depicted below, making a call to a SOAP-based web service involves sending a request file to the web service provider and receiving a response file—which, in this case, will be parsed to create a SAS dataset.



In the example presented below, the request being sent to the SOAP web service "QueryItem" takes the following form:

```
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:ns="http://HIPAASpace.com/webservice/2008">
   <soapenv:Header/>
   <soapenv:Body>
      <ns:QueryItem>
         <!--Optional:-->
         <ns:Type>NDC</ns:Type>
         <!--Optional:-->
         <ns:SearchRequest>&NDCCode</ns:SearchRequest>
         <!--Optional:-->
         <ns:Token>3932f3b0-cfab-11dc-95ff-0800200c9a663932f3b0-cfab-11dc-95ff-
0800200c9a66</ns:Token>
      </ns:QueryItem>
   </soapenv:Body>
</soapenv:Envelope>
```

This request specifies that you are searching for information on one or more NDC codes; the code (or codes) being requested will be identified by resolution of the SAS macro variable "NDCCode", and the token "3932f3b0-…" is the security identifier that confirms your authorization to use the web service. This XML request syntax is saved in the text file "C:\NDC Request.txt" that will be used to make a SOAP request from SAS.

In the following SAS code, the value of the macro variable "NDCCode" is assigned as "055045-1807-*9"; this will ultimately be the NDC code that is sent in the XML request. The fileref "request" references the persistent request template "NDC Request.txt" that will be used to create the request sent to the web service. Next, the filerefs "tempreq" and "response" are defined with the TEMP access method. The TEMP access method "creates a temporary file that exists only as long as the filename is assigned. The temporary file can be accessed only through the logical name and is available only while the logical name exists" (SAS, 2011b). The TEMP access method is used here to avoid retaining persistent files for the requests and responses being sent to/from the web service. Each time the SAS code is executed, a new temporary request file containing the current value of "&NDCcode" is generated. After the request is sent, the corresponding XML file received from the web service is written to a new temporary "response" file. By using the TEMP access method to define the request and response files, SAS manages the creation and naming of the temporary files and manages there clean-up (deletion) from the temporary storage location when the SAS session ends.

```
%LET NDCcode=055045-1807-*9;

FILENAME request   'C:\_CDMS\SGF2012\filename\NDC Request.txt';
FILENAME tempreq  TEMP;
FILENAME response TEMP;
```

Following assignment of the filerefs, a DATA _NULL_ step is used to create the temporary file "tempreq" that will eventually be sent as the request to the web service. The INFILE statement indicates that the request template "NDC Request.txt" will be used as the source data for creating "tempreq", and the INPUT statement then fetches each line of data from the "request" file. Unlike many of the previous examples, however, the INPUT statement is not used to parse the line of data into individual variables. Instead, the INPUT statement simply fetches the line of data into the automatic variable _INFILE_, and the value of _INFILE_ is assigned to the local variable "local_infile". This local variable is then written to the temporary file "tempreq" using the PUT statement. It is important to note that because the RESOLVE function is used in the assignment of the "local_infile" value, when the line containing the "NDCCode" macro variable reference is read from "NDC Request.txt", the macro variable will be resolved to its current value "055045-1807-*9"—resulting in a web service request for data related to that particular NDC code.

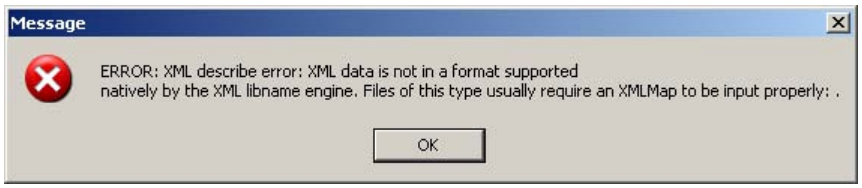
```
DATA _NULL_;
   FILE tempreq;
 INFILE request;
  INPUT;
 local_infile = RESOLVE(_INFILE_);
 PUT local_infile;
RUN;
```

Next, PROC SOAP sends the request written in "tempreq" to the web service identified by the location of the web service definition (URL) and the name of the requested service (SOAPACTION).

```
PROC SOAP IN=tempreq
          OUT=response
          URL="http://www.hipaaspace.com/wspHVLookup.asmx"
   SOAPACTION='http://HIPAASpace.com/webservice/2008/QueryItem';
RUN;
```

Unlike the previous RESTful service example, defining an XML library to read the response received from this web service is not as straight-forward as specifying the filename and the XML engine. Attempting to access the "response" data through the library "source" results in an error that indicates the file structure is not natively understood by the SAS XML engine.

```
LIBNAME source XML XMLFILEREF=response;
DATA work.NDC_Code_Data;
 SET source.pair;
RUN;
```

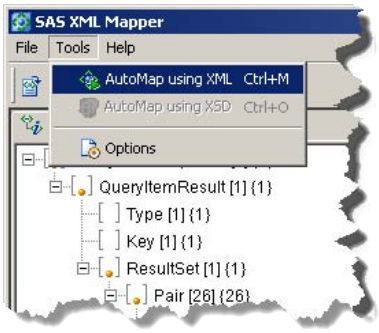


The "XMLMap" to which this error refers is a file that instructs SAS how to interpret the structure of the XML file being accessed. A detailed discussion of the XML engine and XMLMaps is beyond the scope of this paper, but a simple example of creating an XMLMap is provided below[7]. [For more in-depth descriptions of how to use the XMLMapper tool see Hoyle (2010) and Mack (2010).] The first step is to create a persistent response file. In the following example, the fileref "response" is defined as a reference to the physical file "Response.xml" at the root of the "C:" drive so that after the PROC SOAP step is executed the physical response file can be easily located.

```
%LET NDCcode=055045-1807-*9;

FILENAME request   'C:\_CDMS\SGF2012\filename\NDC Request.txt';
FILENAME tempreq  TEMP;
FILENAME response 'C:\Response.xml';

DATA _NULL_;
   FILE tempreq;
 INFILE request;
  INPUT;
 _infile_ = RESOLVE(_INFILE_);
 PUT _infile_;
RUN;

PROC SOAP IN=tempreq
          OUT=response
   URL="http://www.hipaaspace.com/wspHVLookup.asmx"
   SOAPACTION='http://HIPAASpace.com/webservice/2008/QueryItem';
RUN;
```

[7] At the time the paper was written, the SAS XML Mapper is available for download at: http://support.sas.com/demosdownloads/setupcat.jsp?cat=Base+SAS+Software (SAS, 2011d) Usage Note *33584:* Download location for the latest version of the XML Mapper

Once you have the response file, launch XMLMapper (SAS, 2010d), choose "Open XML" from the "File" menu, and open the response file "C:\Resonse.xml". With the file open, choose "AutoMap using XML" from the "Tools" menu, and the XML Mapper tool will generate the XMLMap file necessary to enable the XML LIBNAME engine to interpret the response file. Save the .map file by selecting "Save XMLMap as" from the "File" menu, and save the .map file as "C:\NDCMap.map".



With the .map file saved, you can now define the library "source" in terms of the XMLFILEREF "response" and the XMLMAP "respmap".

```
FILENAME respmap 'C:\NDCMap.map';
LIBNAME source XML XMLFILEREF=response XMLMAP=respmap;
```

As in the REST example, you can then read these data as you would the datasets in any other library.

```
DATA work.NDC_Code_Data;
 SET source.pair;
RUN;
```

Beginning in SAS 9.3 users have yet another tool for interacting with web services—six new SOAP functions. As a brief example of one of these (SOAPWEB), the following code shows that after the SOAP request is generated by the the previous DATA _NULL_ step you could send the SOAP request and write the response within a DATA step instead of using PROC SOAP:

```
DATA _NULL_;
  url = "http://www.hipaaspace.com/wspHVLookup.asmx";
  soapcall = "http://HIPAASpace.com/webservice/2008/QueryItem";
  rc = SOAPWEB("tempreq", url, "response", soapcall, , , , , ,);
RUN;
```

SOCKET & DDE

The last two access methods that reach outside of SAS are the SOCKET and DDE access methods. To use the SOCKET method you define filerefs in terms of a TCP/IP port on a host, and SAS reads and writes data to that TCP/IP socket. Ward (2000) provides a number of examples of using this access method to: (1) read web pages by sending HTTP GET requests through a socket to the web server and receiving the responses returned from that socket, (2) send and receive e-mail by sending SMTP messages to a mail server and attaching to a POP3 server to retrieve messages, and (3) execute FTP file transfers. Helf (2005) describes the use of the SOCKET access method to read web pages—expanding on Ward's methodology and providing additional examples of sending/receiving HTTP messages via sockets. Helf also points out that, interestingly, the SOCKET access method is unique in that the same fileref is often used to both read and write data within the same DATA step.

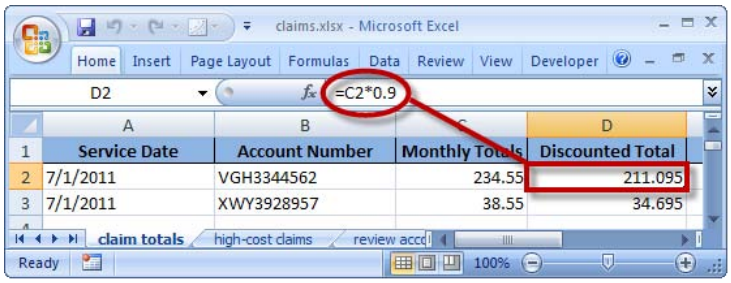
However, the SOCKET access method is not used broadly as an all-purpose means of accessing data with BASE SAS—mostly due to the number of alternative methodologies for serving up data via the web and (more recently) due to the growth of service oriented architectures that make data available via relatively simple calls to RESTful and SOAP-based web services.

The Dynamic Data Exchange (DDE) access method, on the other hand, still enjoys a devoted following of users that utilize the access method largely for reading/writing data to/from Microsoft Excel. There are a number of excellent papers that describe the DDE access method—including Vyverman (2001, 2002), Watts (2005), and Derby (2008), but the short explanation of this access method is that it involves creating a client-server relationship between SAS (the client) and a DDE server application (e.g., Microsoft Word, Excel, PowerPoint).

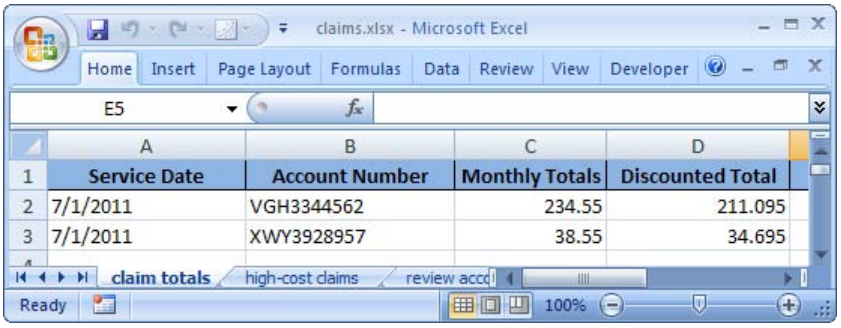
With the DDE application file open, the DDE fileref is used to establish a connection between SAS and the server application by referring to the DDE triplet format (Server/Topic/Item) that is used to identify the external file. In the following example, the fileref "clmout" references the server application "excel", the topic specified is the "claim totals" worksheet within the "c:\claim reports\claims.xlsx" file, and the item or data range is "r2c1:r4c3" (row 2, column 1 – row 4, column 3). The DATA _NULL_ step is then used to read data from the "2011_07.txt" file and write the data to file, worksheet, and cells identified as "clmout".

```
FILENAME clmin  '\\finance\analytics\report\source\2011_07.txt';
FILENAME clmout DDE "excel|c:\claim reports\[claims.xlsx]claim totals!r2c1:r4c3";

DATA _NULL_;
FORMAT svc_date mmddyy10.;
   INFILE clmin FIRSTOBS=2;
    INPUT svc_date mmddyy10. account_no $12. prin_dx $8. billed_charges dollar12.;

     FILE clmout;
      PUT svc_date account_no  billed_charges ;
RUN;
```

Starting from the first column in the second row (r2c1) the data are written across to the third column (c3) of each row. By writing data to specific locations within an Excel file, you can take advantage of native Excel formatting and functionality to create elegant output. In the example depicted here, the column headers are filled with a predefined background color, the font is bold face and centered, and a formula is assigned to the cells in column "D" such that the "Discounted Total" is automatically recalculated as data are written from the SAS dataset.



Despite the powerful solutions that can be developed with DDE (and like any other methodology) DDE has its shortcomings. Gebhardt (2007) points out that the server application (i.e., the external file) must be running in order to send data to it, the access method works only on the Windows platform, and the DATA step manipulations needed to create the output can be onerous; although he concedes that "almost anything can be done with [DDE]". Partly because DDE can be a bit difficult to work with, and partly because of the increasing functionality available via the Output Delivery System (ODS), many users are turning to the compromise of using the excelxp tagset in ODS to generate their Excel output—but see Miralles (2011) for a concise, focused comparison of options for moving data between SAS and Excel.



CATALOG, DUMMY, CLIPBOARD, & OTHER ACCESS METHODS

The remaining access methods are grouped together in this last section, not because they are seen as less powerful, but because the author has not commonly used them in practice, nor has he seen them used widely. They likely play a crucial role in numerous creative solutions. The CATALOG access method, for example, can be used to centrally organize SAS program code, macros, formats, and data elements used in multiple applications. In the following example (adapted from SAS, 2011e), the fileref "src_code" is defined as source code entry "cdms3" in the "CODE" catalog in library "CDMSLLC". Next, the PUT statement in the DATA _NULL_ step writes the syntax "PROC OPTIONS; RUN;" to the catalog entry.

```
LIBNAME CDMSLLC 'C:\';
FILENAME src_code CATALOG 'CDMSLLC.CODE.cdms3.source';
```

```
DATA _NULL_;
   FILE src_code;
   PUT 'proc options; run;';
RUN;
```

Once such programs are written to the catalog, you can later issue an %INCLUDE statement to call them through a fileref pointed to the catalog where they are stored. In this example, the source code "cdms2" is called from the "code" catalog, followed by "cdms3".

```
FILENAME cdms CATALOG 'cdmsllc.code';
%INCLUDE cdms (cdms2);
%INCLUDE cdms (cdms3);
```

Writing to the DUMMY access method results in the data going (well) nowhere. DeGuire (2007) provides a couple of potentially useful examples of this method. The first is to use DUMMY to discard unwanted log file information during development. In the following example, the DUMMY access method is assigned to the fileref "nowhere", and PROC PRINTTO points the generation of log entries to "nowhere"—resulting in no log being generated. The other use DeGuire gives for the DUMMY access method is to define a file for the purpose of being able to pass a valid filename to a SAS macro that requires a filename as an input (but for which you cannot, or do not want to, specify an actual file).

```
FILENAME nowhere DUMMY;
PROC PRINTTO nowhere;
RUN;
```

Finally, the author's favorite demonstration of an access method in this last group is Tabachnik et al.'s (2010) ingenious demonstration of the CLIPBOARD access method to enable users to right-click SAS datasets in the SAS Explorer and select a menu option to copy variable names to the clipboard. Once in the clipboard, the contents of the clipboard can be pasted into the SAS Editor for use in KEEP and DROP statements in a DATA step, to form the basis of a SELECT statement within PROC SQL, etc.

CONCLUSION

The FILENAME statement plays an important role in a wide array of creative solutions that involve the interaction of SAS with other systems. With relatively simple command syntax, users can utilize SAS as an FTP client, an SMTP e-mail system, or a means of consuming web services. Additionally, through the DDE access method, powerful solutions can be created using Excel (and Word and PowerPoint). Developing a facility with one or all of these implementations of the FILENAME statement will help you create more efficient, more fully-automated solutions to the programming challenges you face daily and will help you achieve The Power to Know®.

ACKNOWLEDGMENTS

The author wishes to thank Antoine Lavoisier and the folks at Hipaaspace.com for allowing him to use their REST and SOAP based web services in examples of the URL and TEMP access methods.

REFERENCES

Aster, R. & Seidman, R. (1997). Professional SAS Programming Secrets. McGraw-Hill.

Bartlett, J., Bieringer, A., & Cox, J. (2010). Your Friendly Neighborhood Web crawler: A Guide to Crawling the Web with SAS®. Proceedings of the SAS Global Forum 2010. Cary, NC: SAS Institute, Inc.

Cates, R. (2001). MISSOVER, TRUNCOVER, and PAD, OH MY!! or Making Sense of the INFILE and INPUT Statements. Proceedings of the 26th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

DeGuire, Y. (2007). The FILENAME Statement Revisited. Proceedings of the SAS Global Forum 2010. Cary, NC: SAS Institute, Inc.

Derby, N. (2008). Revisiting DDE: An Updated Macro for Exporting SAS® Data into Custom-Formatted Excel® Spreadsheets: Part I - Usage and Examples. Proceedings of the SAS Global Forum 2008. Cary, NC: SAS Institute, Inc.

First, S. (2008). The SAS INFILE and FILE Statements. Proceedings of the SAS Global Forum 2008. Cary, NC: SAS Institute, Inc.

Flavin, J. M. & Carpenter, A. L. (2001). Taking Control and Keeping It: Creating and using conditionally executable SAS® Code. Proceedings of the 26th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Fuller, J. (2010). Mashups Can Be Gravy: Techniques for Bringing the Web to SAS®. Proceedings of the SAS Global Forum 2010. Cary, NC: SAS Institute, Inc.

Gebhart, E. (2007). ODS and Office Integration. Proceedings of the SAS Global Forum 2007. Cary, NC: SAS Institute, Inc.

Helf, G. (2005). Extreme Web Access: What to Do When FILENAME URL Is Not Enough. Proceedings of the 30th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Hoyle, L. (2010). Using XML Mapper and Enterprise Guide to Read Data and Metadata from an XML File. Proceedings of the SAS Global Forum 2010. Cary, NC: SAS Institute, Inc.

Hunley, C. (2010). SMTP E-Mail Access Method: Hints, Tips, and Tricks. Proceedings of the SAS Global Forum 2010. Cary, NC: SAS Institute, Inc.

Jahn, D. (2008). Using SAS BI Web Services and PROC SOAP in a Service-Oriented Architecture. Proceedings of the SAS Global Forum 2008. Cary, NC: SAS Institute, Inc.

Kolbe, K.L. (1997) Advanced Techniques for Reading Difficult and Unusual Flat Files. Proceedings of the 22nd Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Langston, R. (2009). Creating SAS® Data Sets from HTML Table Definitions. Proceedings of the SAS Global Forum 2009. Cary, NC: SAS Institute, Inc.

Mack, C. E. (2010). Using Base SAS® to Talk to the Outside World: Consuming SOAP and REST Web Services Using SAS® 9.1 and the New Features of SAS® 9.2®. Proceedings of the SAS Global Forum 2010. Cary, NC: SAS Institute, Inc.

Milum, J. (2011). Let's Get Connected: How We Link to our Data. Proceedings of the SAS Global Forum 2011. Cary, NC: SAS Institute, Inc.

Miralles, R. Creating an Excel report: A comparison of the different techniques. Proceedings of the SAS Global Forum 2011. Cary, NC: SAS Institute, Inc.

Pagé, Jacues. (2004). Automated distribution of SAS results. Proceedings of the 29th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Pratter, F.E. (2011). Web Development with SAS by Example. Cary, NC: SAS Institute, Inc.

Richardson, L. & Ruby, S. (2007). RESTful Web Services. Sebastopol, CA: O'Reilly.

SAS Institute Inc. (1988). The SAS Language Guide for Personal Computers (Release 6.03 Edition). Cary, NC: SAS Institute, Inc.

SAS Institute Inc. (2007). Usage Note 19767: Using the SAS® System to send SMTP e-mail. Cary, NC: SAS Institute, Inc.

SAS Institute Inc. (2009). TS-673: Reading Delimited Text Files into SAS®9. Cary, NC: SAS Institute, Inc.

SAS Institute Inc. (2011a). Base SAS® 9.3 Procedures Guide. Cary, NC: SAS Institute, Inc.

SAS Institute Inc. (2011b). SAS® 9.2 Companion for Windows, Second Edition. Cary, NC: SAS Institute, Inc.

SAS Institute Inc. (2011c). TS-605: Sending Electronic Mail within the SAS System under OS/390. Cary, NC: SAS Institute, Inc.

SAS Institute Inc. (2011d). Usage Note 33584: Download location for the latest version of the XML Mapper. Cary, NC: SAS Institute, Inc.

SAS Institute Inc. (2011e). SAS® 9.2 Language Reference: Dictionary, Fourth Edition. Cary, NC: SAS Institute, Inc.

Schacherer, C.W. & Steines, T.J. (2010). Building an Extract, Transform, and Load (ETL) Server Using Base SAS, SAS/SHARE, SAS/CONNECT, and SAS/ACCESS. Proceedings of the Midwest SAS Users Group.

Schacherer, C.W. (2008). Utilizing SAS as an Integrated Component of the Clinical Research Information System. Proceedings of the SAS Global Forum 2008. Cary, NC: SAS Institute, Inc.

Sherman, P.D. & Carpenter, A.L. (2009). Secret Sequel: Keeping your password away from the LOG. Proceedings of the SAS Global Forum 2009. Cary, NC: SAS Institute, Inc.

Tabachneck, A.S., Herbison, R., Clapson, A., King, J., DeAngelis, R., Abernathy, T. (2010). Automagically Copying and Pasting Variable Names. Proceedings of the SAS Global Forum 2010. Cary, NC: SAS Institute, Inc.

Tilanus, E.W. (2008). Sending E-mail from the DATA step. Proceedings of the SAS Global Forum 2008. Cary, NC: SAS Institute, Inc.

Varney, B. (2008). Check out These Pipes: Using Microsoft Windows Commands from SAS®. Proceedings of the SAS Global Forum 2010. Cary, NC: SAS Institute, Inc.

Vyverman , K.(2001). Using Dynamic Data Exchange to Export Your SAS® Data to MS Excel — Against All ODS, Part I. Proceedings of the 26th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Vyverman , K.(2002). Creating Custom Excel Workbooks from Base SAS with Dynamic Data Exchange: a Complete Walkthrough. Proceedings of the 27th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Ward, D.L. (2010). You can do THAT with SAS Software? Using the socket access method to unite SAS with the Internet. Proceedings of the SouthEast SAS Users Group.

Watts, P. (2005). Using Single-Purpose SAS® Macros to Format EXCEL Spreadsheets with DDE. Proceedings of the 30th Annual SAS Users Group International Meeting. Cary, NC: SAS Institute, Inc.

Zdeb, M. (2010). Driving Distances and Times Using SAS® and Google Maps. Proceedings of the SAS Global Forum 2010. Cary, NC: SAS Institute, Inc.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Chris Schacherer, Ph.D.
Clinical Data Management Systems, LLC
E-mail: CSchacherer@cdms-llc.com
Web: www.cdms-llc.com