Paper AD12-2011

What's in a (FILE)NAME: The important role of a simple statement

Chris Schacherer, Clinical Data Management Systems, LLC

ABSTRACT

The FILENAME statement has a very simple purpose—to specify the fileref (or, file reference) that serves as the link to an external file or device. The statement itself does not process any data, specify the format or shape of a dataset, or directly produce output of any type, yet this simple statement is an invaluable SAS® construct that allows SAS programs to interact with the world outside of SAS. Through specification of the appropriate device type, the FILENAME statement allows SAS to symbolically refer to external disk files for the purpose of reading and writing data, interact with FTP Servers to read and write remote files, send e-mail messages, and gather data from external programs—including the local operating system and remote web services. The current work explores these uses of the FILENAME statement and 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 to an improved toolset that ultimately makes data access more efficient, this increased efficiency has come at the cost of many new SAS users not being exposed to this powerful, flexible tool. Therefore, the current work attempts to close this gap in knowledge by introducing the reader to the FILENAME statement, providing some oft-encountered techniques for manipulating flat files with the INFILE and FILE statements, and demonstrating several popular (but less frequently utilized) techniques in which the FILENAME statement plays a central role.

To get started on 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.*

From this definition it is clear that the FILENAME statement can be used to generate a symbolic link to an external file (or device) that, in turn, can be used to refer to the file elsewhere in the SAS program. For example, in the following code, the file reference claims references the physical file \\finance\analytics\report\source\2011_07.txt. When the fileref is later encountered in the INFILE statement, SAS interprets the symbol "claims" to refer to the fully qualified path and file name indicated 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;
```

However, as you will see in the remainder of the paper, the FILENAME statement can be used to interact with the world outside of SAS in a number of interesting ways beyond reading and writing text files.

READING & WRITING EXTERNAL DISK FILES

Far and away the most frequent use of the FILENAME statement is still 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 an INFILE and INPUT statements[1]. Although an exhaustive discussion of the use of the INFILE and INPUT statements to read external flat files is beyond the scope of this discussion, a few

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

examples are provided in the following sections. For more in-depth explanations of using the INFILE and INPUT statements for reading files into SAS, the reader is referred to Aster and Seidman (1997), Kolbe (1997), and First (2008).]

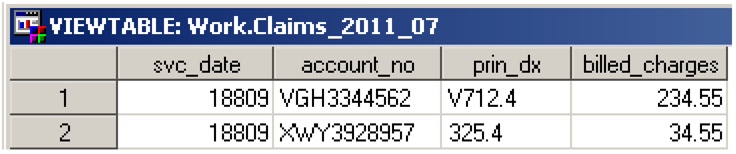
FIXED WIDTH, UNIFORM LAYOUT. The most fundamental characteristic for determining the appropriate approach for reading a flat file into SAS is whether the location of data elements on the record are defined in terms of "FIXED" positions or simply in terms of their order on the record as defined by a "DELIMITER" (e.g., commas, tabs, etc.). 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, is always given as the characters that lie between positions 1 and 10—"account_no" in positions 13 and 22, etc. When this file is read by the following DATA step, the INFILE statement fetches each row of data 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_charge" is identified as a numeric value that is being represented in the external file using the dollar12. format.[2]

```
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 $10. prin_dx $6. billed_charges dollar12.;
RUN;
```

Once the DATA step completes, the new dataset "work.claims_2011_07" contains the 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

The previous 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 that is recommended for fixed-width files is to use an explicit pointer (@<line position>) to indicate the location of each variable on the record followed by the variable name and the INFORMAT that should be used to read the data associated with that variable. This method of declaring variables in the INPUT statement is referred to as COLUMN ENTRY. Using column entry, the previous example would 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. @36 billed_charges dollar12.;
RUN;
```

In addition to forcing you to confirm your knowledge of the data structure before importing it, this method also makes validation of your program against any data mapping documentation much easier. 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 more easily be detected.

The preceding two examples also introduce an important OPTION to the INFILE statement—FIRSTOBS. FIRSTOBS identifies the row at which the DATA step should begin processing data from the file referenced in the FILENAME

[2] It is worth noting that if all of the values within this dataset were represented in strictly numeric or text form (e.g., 234.55 instead of $234.55) and fit the default field length of "8", the input statement could be simplified as:

```
INPUT svc_date account_no $ prin_dx $ billed_charges;
```

statement. If our "2011_07.txt" file contained additional header information, as in the following example, FIRSTOBS would be set to "5".

```
Claims System XYZ
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
```

DELIMITED, UNIFORM LAYOUT. Frequently, instead of fixed-width files, users are provided with delimited files (in which a special character is used to explicitly mark the separation between variables). In order to read these files using the INFILE statement, the DELIMITER (DLM) option is added to the INFILE statement. If our "2011-07.txt" file were delimited with commas, the data would look like this:

```
Claims System XYZ
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
```

With delimited files, the position of the values within the rows is no longer meaningful (see, for example, the values of "billed_charges"). However, the values are reliably separated by a comma, which allows us to use the DELIMITER option to accurately assign values to our SAS variables. 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=5 DLM =',' ;
   INPUT svc_date :mmddyy10. account_no :$10. prin_dx :$6.
         billed_charges :dollar12.;
RUN;
```

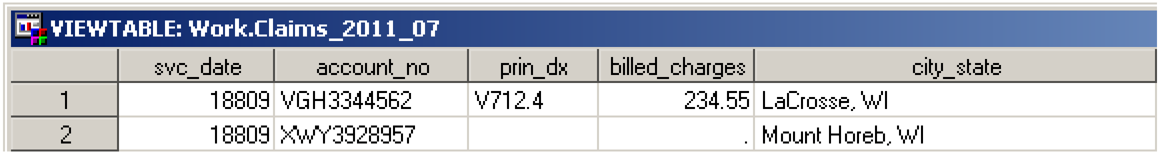
In addition to delimiters that can be typed in their human-readable form (e.g., comma [,], pipe [|], carrot [^], etc.), delimiters can also be specified in their hexadecimal form—for example "DLM='05'x specifies tab delimiters for ASCII systems.

When reading in 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 multiple, sequential delimiters in the second record will be interpreted as missing values for "principle_dx" and "billed_charges", and the commas in the values of "city_state" will be retained as part of the data 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

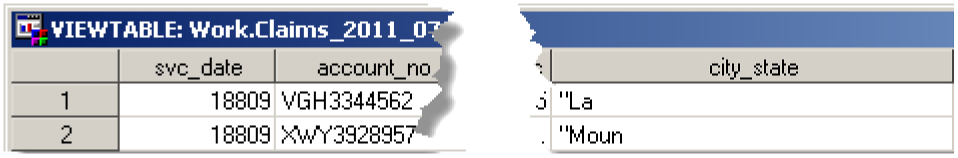
Two additional INFILE options that can significantly impact the reading of external files are LRECL (logical record length) and the options (TRUNCOVER, MISSOVER, STOPOVER, FLOWOVER) that determine what happens when the INPUT statement encounters the end of the external file record before all variables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

***LRECL** OPTION* - 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 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.

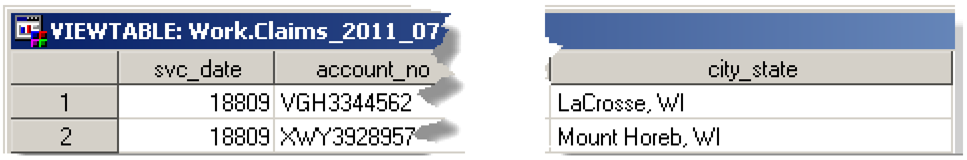
```
DATA work.claims_2011_07;
   INFILE claims FIRSTOBS=2 DLM =',' DSD MISSOVER;
   INPUT svc_date :mmddyy10. account_no :$10. <other variables> city_state :$25.;
RUN;
```


VIEWTABLE: Work.Claims_2011_07

	svc_date	account_no	city_state
1	18809	VGH3344562	"La
2	18809	XWY3928957	"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

LINE-END OPTIONS - If the INPUT statement reaches the end of a data record and not all variables have been assigned a value yet, the default behavior is for INPUT to continue reading data by proceeding to the next record 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".

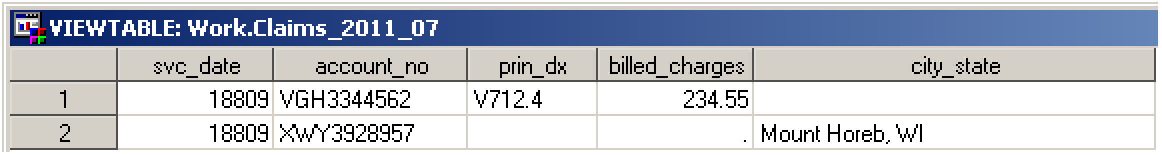
```
svc_date,account_no,principle_dx,billed_charges,...,city_state
07/01/2011,VGH3344562,V712.4,$234.55
07/01/2011,XWY3928957,,,"Mount Horeb,WI"
```

This is rarely the desired behavior. To correct this situation, you can use the MISSOVER option, which keeps INPUT from reading data from the next line and 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[3]. The STOPOVER option, on the other hand, raises an error and ends the DATA step if the end of a record is reached and not 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;
```

[3] But see Cates (2001) for in-depth explanation of situations in which these options differ.

```
   INPUT svc_date :mmddyy10. account_no :$10. city_state :$25.;
RUN;
```



VIEWTABLE: Work.Claims_2011_07

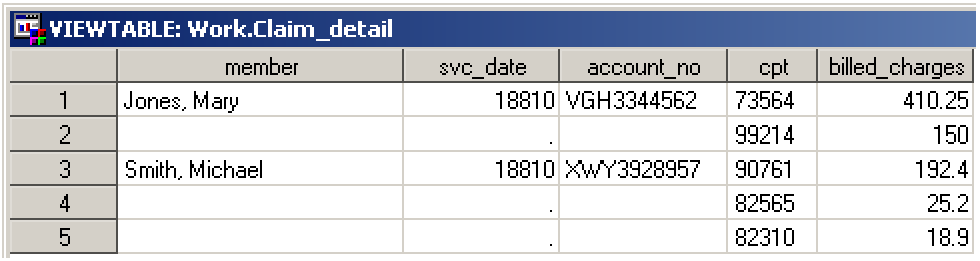
	svc_date	account_no	prin_dx	billed_charges	city_state
1	18809	VGH3344562	V712.4	234.55	
2	18809	XWY3928957		.	Mount Horeb, WI

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, but this is not always the case. Some older systems (and some newer ones for that matter) produce files in the structure of a hierarchical report similar to the following:

```
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 this file using only the methods demonstrated in the previous examples, only two of the five rows of data would contain the member name, service date, and account number—making summarizations of these data by account number or service date particularly difficult.

```
FILENAME  clm_dtl DISK 'C:\_CDMS\MWSUG 2011\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

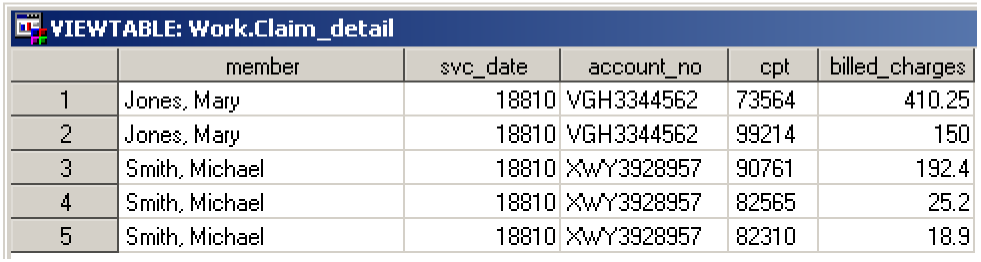
	member	svc_date	account_no	cpt	billed_charges
1	Jones, Mary	18810	VGH3344562	73564	410.25
2		.		99214	150
3	Smith, Michael	18810	XWY3928957	90761	192.4
4		.		82565	25.2
5		.		82310	18.9

Instead, a RETAIN statement is needed to copy these record headers to each of their subordinate rows. The RETAIN statement allows you to define a variable that will "retain" it's value from record to record throughout the DATA step until you assign that retained variable a new value. The following code takes advantage of this characteristic by assessing whether or not the "member" variable for the current record has a non-null value; if it does, the value of "member" from the current record is assigned to the retained variable "member_r", which carries this value from record to record until the next record with a non-null value for "member" is encountered. Therefore, when a subordinate record is encountered with a null value of "member", the retained value "member_r" can be used to assign that record the appropriate value that has been "borrowed" from the header record.

```
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

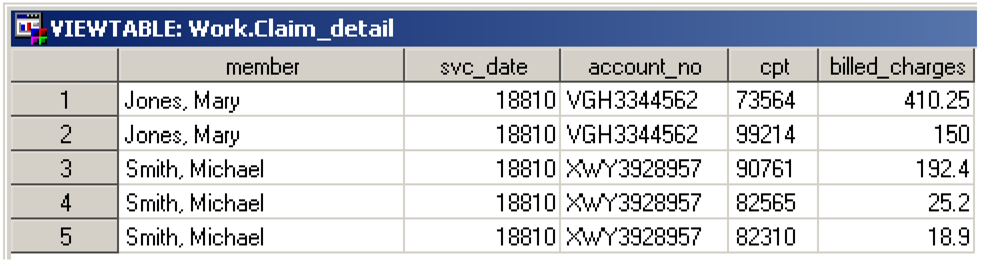
	member	svc_date	account_no	cpt	billed_charges
1	Jones, Mary	18810	VGH3344562	73564	410.25
2	Jones, Mary	18810	VGH3344562	99214	150
3	Smith, Michael	18810	XWY3928957	90761	192.4
4	Smith, Michael	18810	XWY3928957	82565	25.2
5	Smith, Michael	18810	XWY3928957	82310	18.9

MULTI-LINE FILES. Another file layout that can pose a problem for SAS programmers is that in which a data for a single record is written across multiple lines in a source file. In the following file structure, each record is written across two lines of the source data file. The first line contains the health plan member name, service date, and account number; the second line contains the current procedural terminology (CPT) code associated with a billing line-item as well as 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 statement.

```
   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;
```



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. Note here, that not only is it easy to control which columns are included/excluded, 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 '\\finance\analytics\report\source\2011_07_DETAIL.txt'

DATA _NULL_;
FILE clm_dtl DLM=',';
 SET work.claim_detail;
 PUT account_no svc_date mmddyy. ',' cpt billed_charges dollar12.2;
RUN;
```

Similarly, you could create a new fixed-width text file.

```
FILENAME  clm_dtl DISK '\\finance\analytics\report\source\2011_07_DETAIL.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 very specific file formats 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 previous examples work great for files written to disks on your local system or LAN directory, but it is still frequently the case that enterprise-level systems produce flat files that are distributed through a central FTP server. Although it is usually not a terribly difficult feat to obtain these files via your favorite FTP client, the FILENAME statement's FTP access method can be used to automate the task. This is particularly useful if you have large files that you want to process after hours, or if you have an analysis/report that is to be presented every Monday at your 8:00 a.m. status meeting.

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 later referenced, SAS will attempt to retrieve the file via an FTP connection. In this example, the file is located on the FTP server (HOST) "cdms-llc.com", and the credentials of user "chris" are used to establish a connection. Once the connection is specified, you can also specify options that will issue FTP commands once the connection to the server is established.[4]. For example, once connected to the FTP server, the CD option is used to issue a command to change the working directory on the FTP server to the directory where the physical file is located. The PROMPT and DEBUG options are included, respectively, to issue a prompt for entry of the user's password and to write informational messages received from the FTP server 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 as in the previous examples. 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, has a value, and can be used in other transformations performed in the DATA step.

[4] It should be noted that the FTP options require syntax specific to the OS and filesystem 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.

```
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, an FTP connection is attempted and the user will be prompted with a dialog box asking for the password associated with the userid specified as USER. After the password is provided by the user 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 lines of data are then read sequentially from the input buffer and parsed into the values that are assigned to the defined variables. The SAS log entries generated by the FTP server (because the DEBUG option was used) look something like the following:

```
NOTE: >>> USER chris
NOTE: <<< 331 Password required for chris.
NOTE: >>> PASS XXXXXXXXXXXX
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
```

In the previous 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 PASSWORD 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 operating procedures and policies that forbid the saving of passwords in a plain text form. 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;
```

Encoding the password (even with the addition of encryption using "METHOD=SAS002") 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. One additional security measure, then, is to save the encoded password in an external SAS dataset in a secure location—such as a directory to which only people 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 set up this extra security measure, first create a SAS dataset (again, in a secure location, but a location to which the program will have access when scheduled to run automatically by your system's job scheduler) and assign the encoded password to a variable in that dataset.

```
LIBNAME safe 'c:\';

DATA safe.passwords;
 ingredients = '''{sas002}75FF5D504546D3563DC8361D092F20F22FC8909B''';
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.

```
LIBNAME safe 'c:\';

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

Once the macro variable is assigned the value of the password, the macro variable can be assigned as the value to the PASS option, and the value of the password does not have to be shared in the .sas file.

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

So far, all of the FTP examples have involved reading data from a single file from a single server. Often, however, when creating analytic datasets, the dataset resulting from the transformations in your program are to be sent to yet 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). The FILENAME FTP access method can be used to accomplish this more complex set of tasks as well. In the following example, two filerefs based on the FTP access method are defined ("source" and "target"). Following assignment of the filerefs, a DATA _NULL_ step is executed to read the external file referenced by "source", perform a series of data transformations, and write a comma-delimited file out to the file referenced as "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 mmddyy. ',' 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 as in the following example to enable it to run on the first day of the month without alteration. 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". A macro variable is dynamically created to represent the name of the file to be read from the source site. When run in September, 2011 the fileref resolves 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;
```

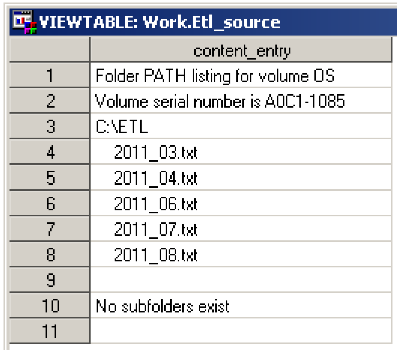
Whether you use macro variables to dynamically assign filenames, directory locations, or other parts of the fileref or hard-code values and managing changes to them manually, the FTP access method can be incredibly beneficial when trying to automate the processing of data with source or target files located on FTP servers. It should be noted, however, that because the FTP access method frequently interacts with hardware and software outside of the platform on which SAS is running, 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 hardware/software 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. When SAS encountered the fileref defined with the FTP access method, it used the information provided in the fileref to establish a connection to the FTP server and retrieve the specified file. 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 a one-way communication between two processes—wherein one process "pulls" data from another. A named pipe establishes a two-way communication wherein the two processes can interact—passing data in both directions. The examples provided here focus on unnamed pipes in which SAS initiates a process outside of SAS and then reads data from that "piped" data source. 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 "c:\etl" directory of the local machine on which SAS is running. The fileref "contents" is defined in terms of a call to the "tree" command in DOS. 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_03.txt
5	2011_04.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, scheduled 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 in the previous example, the following code queries the "etl_source" dataset to determine the number of files in the directory that have the same name as the source file that you are looking for. 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 what the program should do next.

```
PROC SQL NOPRINT;
 SELECT count(*) INTO :file_exists
   FROM etl_source
  WHERE compress(content_entry) = "&source_file1&source_file2";
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 the &terminator macro variable is encountered later in the program, it will resolve to a null, nonexistent value that is not interpreted as a command by SAS. 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.

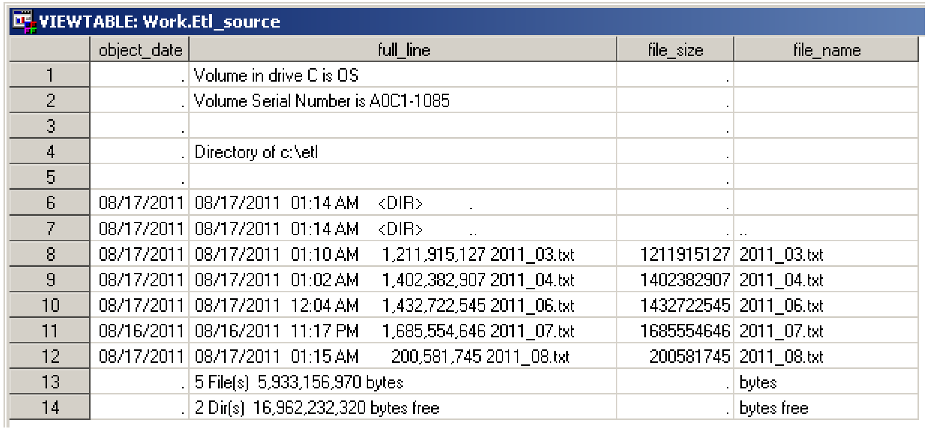
```
DATA _NULL_;
 IF &file_exists = 1 THEN
 CALL SYMPUT('terminator','');
 ELSE
 CALL SYMPUT('terminator','ENDSAS;');
RUN;

&terminator
```

A slight twist on this use of PIPE could be extended to do slightly more sophisticated tests of the source file "2011_07.txt". In the following example, suppose that you have been performing the processing of this monthly file for a while and you know that it is always larger than 500 MB. You could evaluate the size of the file before reading it in and processing it by using the following PIPE command that issues 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 command results in the following dataset:

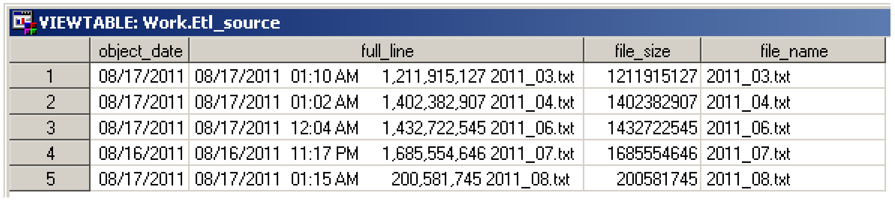


VIEWTABLE: Work.Etl_source

	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	08/17/2011	08/17/2011 01:14 AM <DIR> .	.	
7	08/17/2011	08/17/2011 01:14 AM <DIR> ..	.	..
8	08/17/2011	08/17/2011 01:10 AM 1,211,915,127 2011_03.txt	1211915127	2011_03.txt
9	08/17/2011	08/17/2011 01:02 AM 1,402,382,907 2011_04.txt	1402382907	2011_04.txt
10	08/17/2011	08/17/2011 12:04 AM 1,432,722,545 2011_06.txt	1432722545	2011_06.txt
11	08/16/2011	08/16/2011 11:17 PM 1,685,554,646 2011_07.txt	1685554646	2011_07.txt
12	08/17/2011	08/17/2011 01:15 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) 16,962,232,320 bytes free	.	bytes free

By adding a conditional statement like the following, 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	08/17/2011	08/17/2011 01:10 AM 1,211,915,127 2011_03.txt	1211915127	2011_03.txt
2	08/17/2011	08/17/2011 01:02 AM 1,402,382,907 2011_04.txt	1402382907	2011_04.txt
3	08/17/2011	08/17/2011 12:04 AM 1,432,722,545 2011_06.txt	1432722545	2011_06.txt
4	08/16/2011	08/16/2011 11:17 PM 1,685,554,646 2011_07.txt	1685554646	2011_07.txt
5	08/17/2011	08/17/2011 01:15 AM 200,581,745 2011_08.txt	200581745	2011_08.txt

You could then control whether processing should continue by evaluating the size of the current month's file. In this case, the macro variable "file_size" is assigned a value corresponding to the size of the source file you are about to process.

```
PROC SQL NOPRINT;
 SELECT put(file_size/1000000,best12.) INTO :file_size
   FROM etl_source
  WHERE compress(file_name) = "&source_file1&source_file2";
QUIT;
```

Following the assignment of the value of &file_size, the file size is evaluated, and in the case of processing the file "2011_08.txt", the program will write an error message to the SAS log and stop processing by assigning a %PUT statement and ENDSAS command to the value of the "&terminator" macro variable.

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

&terminator
```

Finally, in addition to using PIPE to execute commands that generate information consumed by SAS, the PIPE device type can also 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 on the local system (or associated LAN directories) in either a static or dynamic manner. In the following example, a new directory "c:\transofmred 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_07;
 SET work.claims_2011_07;
RUN;
```

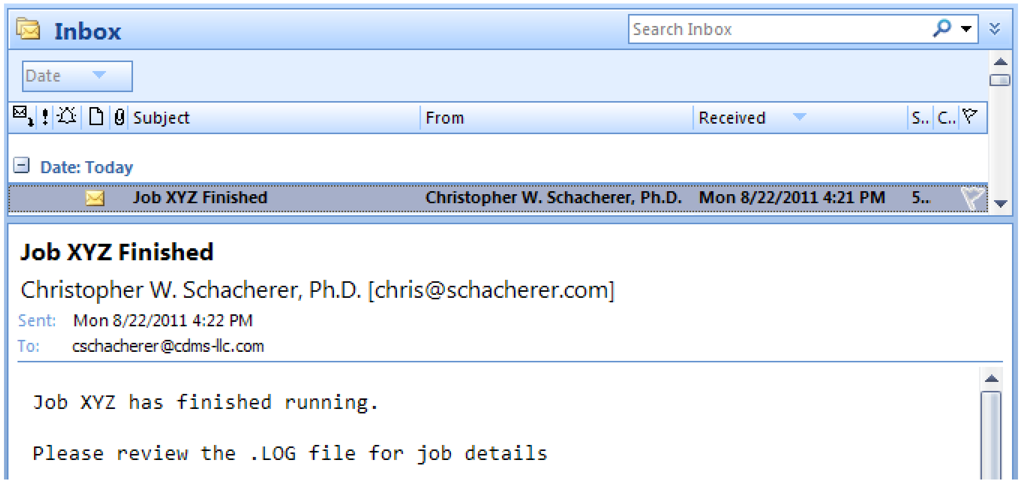
These examples of the PIPE device type only scratch the surface. The ability to interact with other programs and processes opens up a number of possibilities for developing creative SAS solutions.

E-MAILING FROM SAS

One of the most popular access methods available via the FILENAME statement is EMAIL. Using the EMAIL access method, 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 e-mail 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 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.



Subject	From	Received	S..
Job XYZ Finished	Christopher W. Schacherer, Ph.D.	Mon 8/22/2011 4:21 PM	5..

Job XYZ Finished

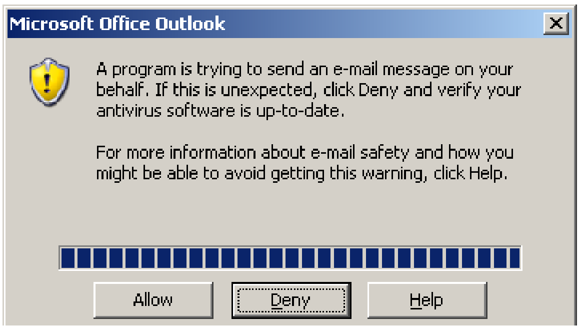
Christopher W. Schacherer, Ph.D. [chris@schacherer.com]

Sent: Mon 8/22/2011 4:22 PM

To: cschacherer@cdms-llc.com

```
Job XYZ has finished running.

Please review the .LOG file for job details
```

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


Microsoft Office Outlook

A program is trying to send an e-mail message on your behalf. If this is unexpected, click Deny and verify your antivirus software is up-to-date.

For more information about e-mail safety and how you might be able to avoid getting this warning, click Help.

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 Outlook, Eudora, Thunderbird, etc. 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 this method is going to be utilized regularly, you can change these options in your SAS configuration file (SAS, 2007) instead of specifying them in each program from which you send e-mail. 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 default mail 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 for generating and sending e-mail messages, there are a number of methods that can be used to customize the content and route the delivery of e-mail to meet a variety of needs. In the following example, a MACRO was used to read an externally generated SAS .LOG file of 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 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 by this code does 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, the program does generate errors during execution, the subject line and contents will be altered to alert the recipient to this fact, and the "To" and "Cc" lists will be altered to communicate the failure to a broader list of recipients.

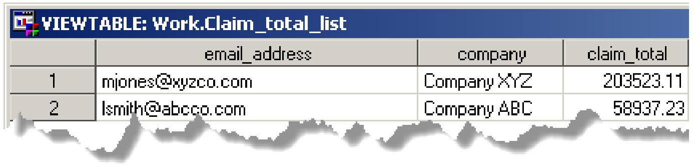
In addition to the conditional processing, this example also introduces the !EM_<directive>! notation to override e-mail attributes. As seen in the previous examples, EMAIL options can be set in the FILENAME statement.

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

EMAIL options can also be assigned in the FILE statement, and options indicated in the FILE statement override those defined in the FILENAME statement.

```
FILENAME job_done EMAIL;

DATA _NULL_;
 FILE job_done to="cschacherer@cdms-llc.com" subject="Job XYZ 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. In the following example (adapted from Tilanus, 2008 & SAS, 2011c), a dataset of monthly healthcare claim totals exists for a set of client companies. Each record contains the company name, the e-mail address of a contact in the benefits organization, and the total of healthcare claims for the month.


VIEWTABLE: Work.Claim_total_list

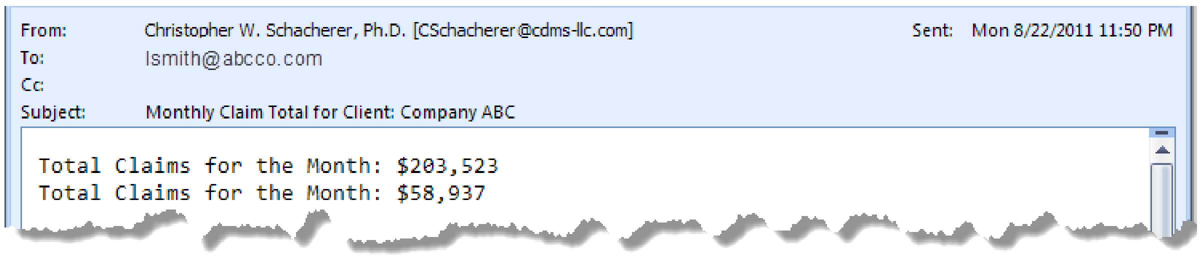
	email_address	company	claim_total
1	mjones@xyzco.com	Company XYZ	203523.11
2	lsmith@abcco.com	Company ABC	58937.23

Using the PUT statements and the !EM_<directive>! notation, an e-mail containing the total claims for the month will be sent to the benefits contact in each of the client organizations. As the dataset "claim_total_list" is processed in the DATA _NULL_ step, the SUBJECT of the e-mail generated for each record in the dataset is dynamically assigned using the current value of "company" and the e-mail address is dynamically assigned using !EM_TO! and the "email_address" variable.

```
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 ' 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;
```

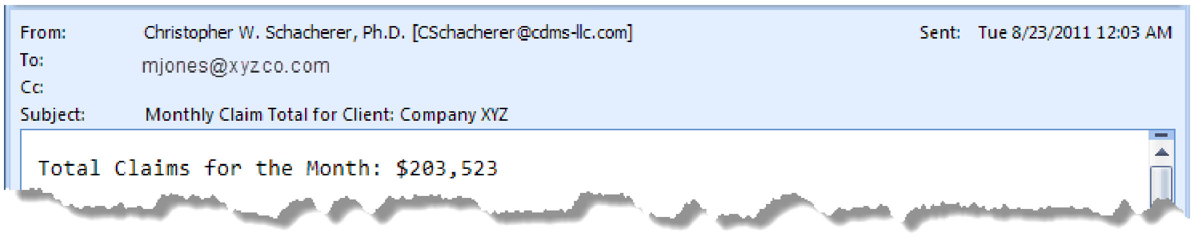
!EM_SEND! and !EM_NEWMSG! play an important role in the processing of multiple e-mail messages being generated from the "claim_total_list". Without these two directives, only one e-mail would be sent; when all of the records have been processed, a single e-mail would be sent to the e-mail address listed 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! notation would 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, all of the PUT statements are 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 for the body of the e-mail:

```
Total Claims for the Month: $203,523
Total Claims for the Month: $58,937
...
```

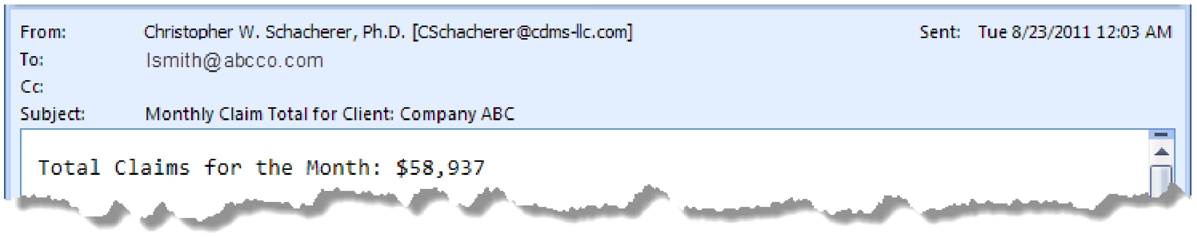

From: Christopher W. Schacherer, Ph.D. [CSchacherer@cdms-llc.com] Sent: Mon 8/22/2011 11:50 PM
To: lsmith@abcco.com
Cc:
Subject: Monthly Claim Total for Client: Company ABC

```
Total Claims for the Month: $203,523
Total Claims for the Month: $58,937
```

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 only 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.


From: Christopher W. Schacherer, Ph.D. [CSchacherer@cdms-llc.com] Sent: Tue 8/23/2011 12:03 AM
To: mjones@xyzco.com
Cc:
Subject: Monthly Claim Total for Client: Company XYZ

```
Total Claims for the Month: $203,523
```


From: Christopher W. Schacherer, Ph.D. [CSchacherer@cdms-llc.com] Sent: Tue 8/23/2011 12:03 AM
To: lsmith@abcco.com
Cc:
Subject: Monthly Claim Total for Client: Company ABC

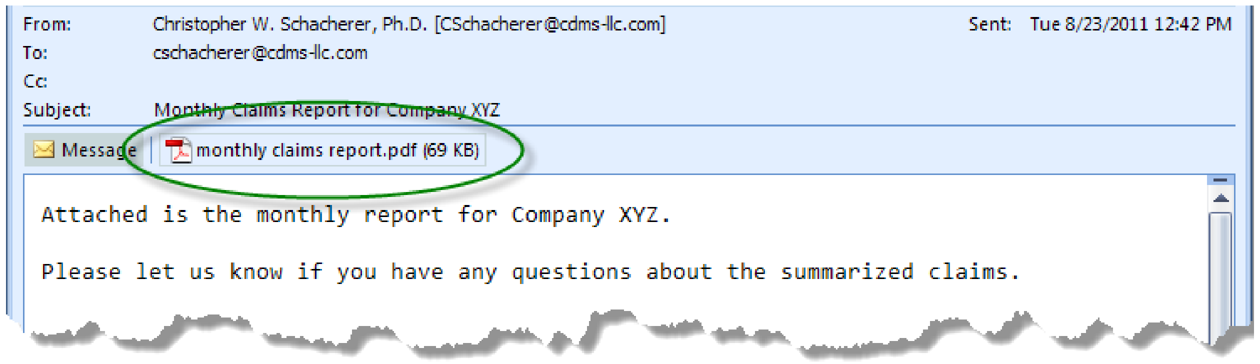
```
Total Claims for the Month: $58,937
```

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 attach output generated by the Output Delivery System® (ODS) to the generated 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 resulting in a .PDF file that will be e-mailed at the end of the program.

```
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 will be attached to an e-mail by specifying the name and location of the file in the "attach" option 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;
```


From: Christopher W. Schacherer, Ph.D. [CSchacherer@cdms-llc.com] Sent: Tue 8/23/2011 12:42 PM
To: cschacherer@cdms-llc.com
Cc:
Subject: Monthly Claims Report for Company XYZ

Message | monthly claims report.pdf (69 KB)

```
Attached is the monthly report for Company XYZ.

Please let us know if you have any questions about the summarized claims.
```

Like the other e-mail options, "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 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 normal monthly report is sent.

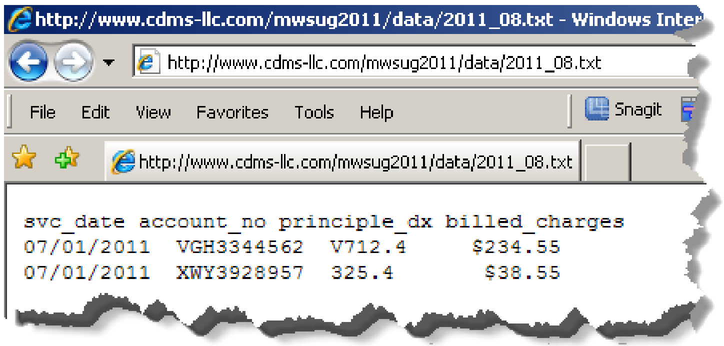
```
FILENAME report EMAIL attach="c:\monthly claims report.pdf";

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;
```

As with the FTP and PIPE device types, the EMAIL examples provided here only reveal a fraction of the types of creative solutions available for automating deliver of SAS output[5]. The integration and delivery of data has also been profoundly impacted by the many methods of data sharing made available on the internet. The most basic of these is the URL access method.

URL

The URL access method has been used 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 web page, 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 by an INFILE statement in the context of a data step. The result of this DATA step is the creation of the SAS dataset "claims_2011_08".


http://www.cdms-llc.com/mwsug2011/data/2011_08.txt - Windows Inter
http://www.cdms-llc.com/mwsug2011/data/2011_08.txt
File
Edit
View
Favorites
Tools
Help
Snagit
http://www.cdms-llc.com/mwsug2011/data/2011_08.txt

```
svc_date account_no principle_dx billed_charges
07/01/2011  VGH3344562  V712.4     $234.55
07/01/2011  XWY3928957  325.4       $38.55
```

```
FILENAME claims URL 'http://www.cdms-llc.com/mwsug2011/data/2011_08.txt';

DATA work.claims_2011_08;
   INFILE claims FIRSTOBS=2;
   INPUT svc_date mmddyy10. account_no $12. prin_dx $8. billed_charges dollar12.;
RUN;
```

If you encounter errors when accessing data via the URL access method (as in the example below, where the filename was misspelled as "201108.txt"), you can add the "DEBUG" option to obtain more detail about the HTTP requests being sent from SAS and the response received from the web server.

```
38   FILENAME claims URL 'http://www.cdms-llc.com/mwsug2011/data/201108.txt';
39
40   DATA work.claims_2011_08;
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.
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.CLAIMS_2011_08 may be incomplete.  When this step was stopped there
         were 0 observations and 4 variables.
```

```
FILENAME claims URL 'http://www.cdms-llc.com/mwsug2011/data/201108.txt' DEBUG;

DATA work.claims_2011_08;
   INFILE claims FIRSTOBS=2;
   INPUT svc_date mmddyy10. account_no $12. prin_dx $8. billed_charges dollar12.;
RUN;
```

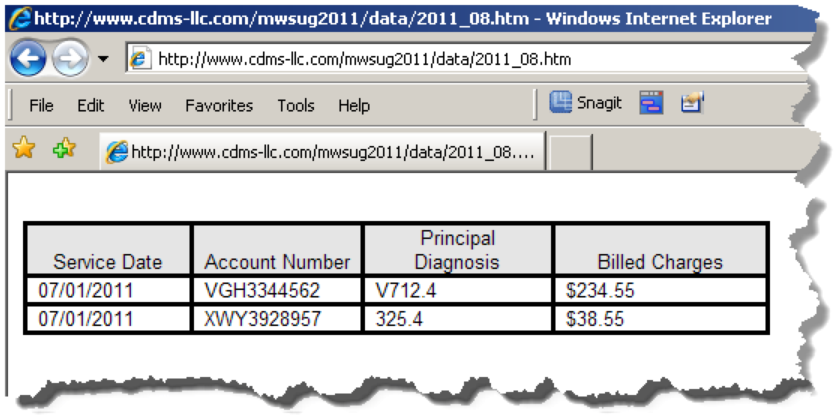
[5] For additional examples of the EMAIL access method, the reader is referred to Schacherer (2008), DeGuire (2007), Pagé (2004), Hunley (2010), and Tilanus (2008).

Re-running the erroneous code with the DEBUG option, you can see the details of the GET request being sent, and the response informs you that the web server is returning a "404" error—indicating that the web page of interest is not found.

```
NOTE: >>> GET /mwsug2011/data/201108.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 name of the data file (or, webpage) addresses this issue, and you are able to access the data file via the URL method.

However, serving up text files as web pages is not very common; it is more likely that the data you will access with the URL method is served up via the web in a more "human readable" form that makes it attractive and more user-friendly to someone viewing the data rather than reading it programmatically—what Richardson & Ruby (2007) refer to as the "human web".


http://www.cdms-llc.com/mwsug2011/data/2011_08.htm - Windows Internet Explorer
http://www.cdms-llc.com/mwsug2011/data/2011_08.htm
File
Edit
View
Favorites
Tools
Help
Snagit
http://www.cdms-llc.com/mwsug2011/data/2011_08....

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

This means, of course, that the program-readable data is more likely in the form of HTML or some other web-centric formatting language focused on making data consumable by humans—sometimes to the detriment of SAS programmers. An example of the data received from calling the previous URL is presented below. In order to create a corresponding SAS dataset for analysis of the August, 2011 claims, these data would need to be parsed using conditional logic and SAS functions in order to produce the target dataset. This often requires a significant bit of trial and error.

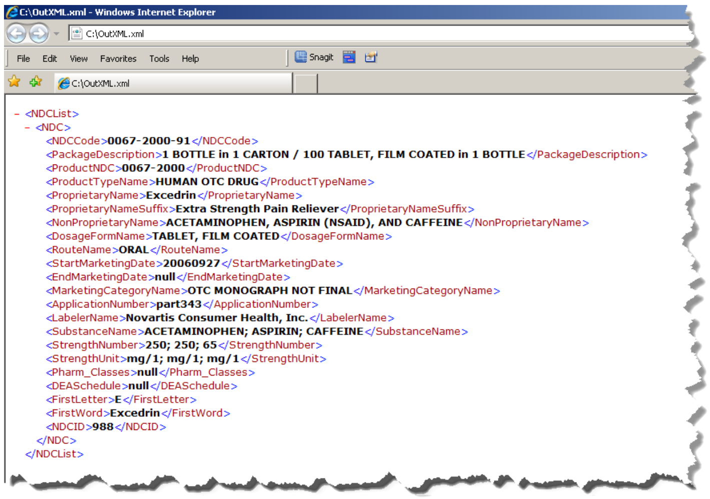
```
<tr style='mso-yfti-irow:1'>
  <td width=94 valign=top style='width:70.85pt;border:solid black 2.25pt;
  border-top:none;padding:0in 5.4pt 0in 5.4pt'>
  <p class=MsoNormal><span style='font-size:10.0pt'>07/01/2011</span></p>
  </td>
[additional html]
  <p class=MsoNormal><span style='font-size:10.0pt'>VGH3344562</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 creating a "network that you can use whether you're serving data to human beings or computer programs." Whereas the 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 "getcode" service associated with National Drug Code (NDC) data is being queried to return descriptive data associated with a specific NDC drug code[6]. 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 "token"—representing, respectively, the NDC code for which information is being requested, the form of the data to be returned in response to our request, and a security token that identifies you as an authorized user[7]. In the following macro variable assignments, %QSYSFUNC executes the URLENCODE function in a manner that masks special characters, and the URLENCODE function encodes special characters that might otherwise impact resolution of a URL. The %NRSTR and %SUPERQ functions are then used to assign the value of macro variable "url". %NRSTR is used to mask the special characters in components of the URL that will represent the parameter inputs and %SUPERQ to put the values of the parameters in quotes.

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

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

Following assignment of the &url macro variable, the fileref "inurl" is defined using the URL access method pointing to the URL that corresponds to the value of the &url macro variable, and an XML file on the local hard drive is defined as the fileref "outxml", to which the data read from the URL will be written. The subsequent DATA _NULL_ step connects to the specified URL, and writes the XML "found" at that URL to the "outxml" file.

```
FILENAME inurl URL "%SUPERQ(url)" 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;
```


C:\OutXML.xml - Windows Internet Explorer
C:\OutXML.xml
File Edit View Favorites Tools Help
Snagit
C:\OutXML.xml

```
- <NDCList>
  - <NDC>
      <NDCCode>0067-2000-91</NDCCode>
      <PackageDescription>1 BOTTLE in 1 CARTON / 100 TABLET, FILM COATED in 1 BOTTLE</PackageDescription>
      <ProductNDC>0067-2000</ProductNDC>
      <ProductTypeName>HUMAN OTC DRUG</ProductTypeName>
      <ProprietaryName>Excedrin</ProprietaryName>
      <ProprietaryNameSuffix>Extra Strength Pain Reliever</ProprietaryNameSuffix>
      <NonProprietaryName>ACETAMINOPHEN, ASPIRIN (NSAID), AND CAFFEINE</NonProprietaryName>
      <DosageFormName>TABLET, FILM COATED</DosageFormName>
      <RouteName>ORAL</RouteName>
      <StartMarketingDate>20060927</StartMarketingDate>
      <EndMarketingDate>null</EndMarketingDate>
      <MarketingCategoryName>OTC MONOGRAPH NOT FINAL</MarketingCategoryName>
      <ApplicationNumber>part343</ApplicationNumber>
      <LabelerName>Novartis Consumer Health, Inc.</LabelerName>
      <SubstanceName>ACETAMINOPHEN; ASPIRIN; CAFFEINE</SubstanceName>
      <StrengthNumber>250; 250; 65</StrengthNumber>
      <StrengthUnit>mg/1; mg/1; mg/1</StrengthUnit>
      <Pharm_Classes>null</Pharm_Classes>
      <DEASchedule>null</DEASchedule>
      <FirstLetter>E</FirstLetter>
      <FirstWord>Excedrin</FirstWord>
      <NDCID>988</NDCID>
    </NDC>
  </NDCList>
```

XML is a common form of output generated by web services, and it is frequently used as a means of data

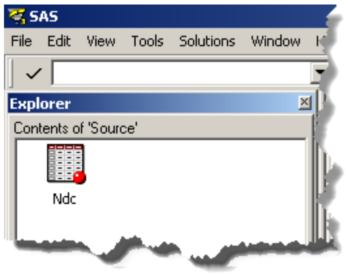
[6] Please note that the value for the "token" and "url" macro variables have been wrapped for the purpose of maintaining the readability of the code in the paper; in order to run this example, the whitespace inserted in these values should be removed.
[7] As of the time the paper was written, the security token in this example was available from hipaaspace.com to allow prospective users to test their web services

interchange for web-based applications. 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 your XML file, you can use the XML libname engine to create a SAS library through which you can access the data contained in the file. In the following example, the path to the .xml file is assigned to the fileref "source", and this fileref is used as the name of the LIBRARY. When accessed, the library "source" resolves to the XML file specified in the fileref.

```
FILENAME source 'C:\outxml.xml';
LIBNAME  source XML ACCESS=READONLY;
```

Alternatively, the LIBNAME to the XML file can be defined as follows:

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

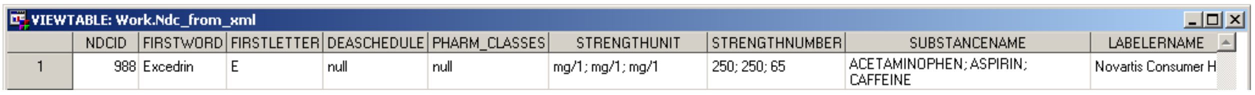


Once defined, you can see the NDC dataset in the "Source" library. However, the SAS XML engine does not support double-clicking on the datasets in this library to open them in browse mode—as is noted in the following error code—even though the data are available for analysis and reporting via DATA and PROC steps.

```
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 persistent SAS dataset.

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



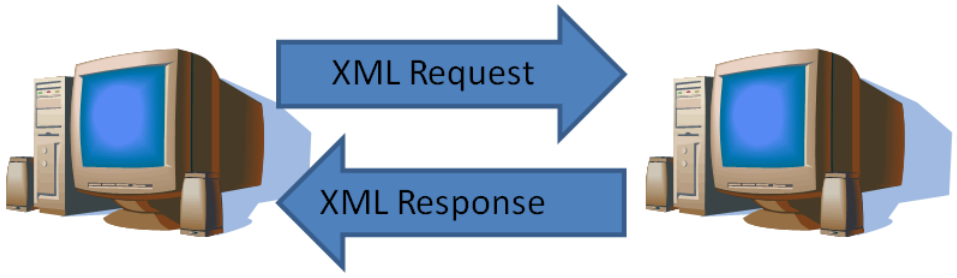
VIEWTABLE: Work.Ndc_from_xml

	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

TEMP

In the preceding example, the filename 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) and Jahn (2008) for discussions of interacting with web services using SAS. For the present purposes, it will suffice to differentiate the previous call to a RESTful web service from a call to a SOAP web service by describing the RESTful web service call as the specification of a particular URL (with the appropriate parameter-value pairs) whereas a SOAP-based web service is one in which a structured request message is sent to the web service and a structured response is returned.

As depicted below, making a call to a SOAP-based web service involves sending an XML request file to the web service provider and receiving an XML response file that will be parsed into a SAS dataset.



In the example presented below, the request being sent to the SOAP web service "QueryItem" at hipaaspace.com 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 syntax is saved in a file (NDC Request.txt) on the local system running SAS. This request specifies that the type of query you are performing is for NDC codes. The code (or codes) being requested will be identified by resolution of the SAS macro variable "NDCCode", and the token "3932…" is used as a security identifier to confirm your authorization to use the web service.

In the following example, 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 request file 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 code is run with a different value of the macro variable NDCcode, a different version of the temporary request file "tempreq" is generated, containing the current value of "&NDCcode". The corresponding XML response file received from the web service contains the web service response generate by that request's invocation of the web service. In the case where these responses are to be extracted into a SAS dataset, there is no reason to maintain a persistent file. In fact, identifying a name and location for such files add unnecessary complexity to the program. Using the TEMP access method, SAS manages the creation and naming of the temporary files, and they are deleted as soon as the fileref is unassigned.

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

   FILENAME request   'C:\_CDMS\MWSUG 2011\filename\NDC Request.txt';
   FILENAME tempreq  TEMP;
   FILENAME response TEMP;
```

Following assignment of the filerefs, a NULL DATA step identifies fileref "tempreq" as a target file to which data will be written and the fileref "request" as a source dataset from which data will be read. The INPUT statement then fetches each line of data from the "request" file, but unlike the previous examples, the INPUT statement is not used to parse the line of data into constituent variables. Instead the INPUT statement simply fetches the line of data into the automatic variable "_INFILE_". The value of _INFILE_ (which contains the entire line of data read from the "request" file) is then assigned to the local variable "local_infile" and is written to the temporary file "tempreq" using the PUT statement.

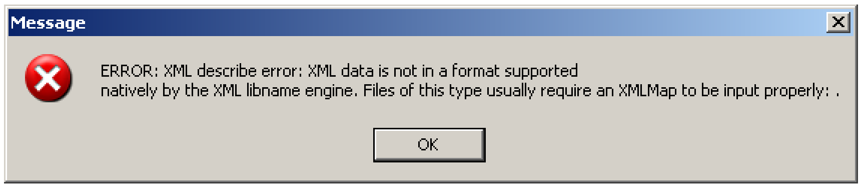
```
   DATA _NULL_;
      FILE tempreq;
    INFILE request;
     INPUT;

    local_infile = RESOLVE(_INFILE_);

    PUT local_infile;
```

Next, PROC SOAP is used to send the contents of the "tempreq" temporary file 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 RESTful service example, however, defining an XML library to read the "response" data from this web service is not as straight-forward. Attempting to access the "response" data through the following LIBNAME statement results in an error indicating that the file structure is not natively understood by the SAS XML engine.

```
LIBNAME  source XML ACCESS=READONLY;
DATA work.ndc_from_xml;
 SET xmlout.ndc;
RUN;
```

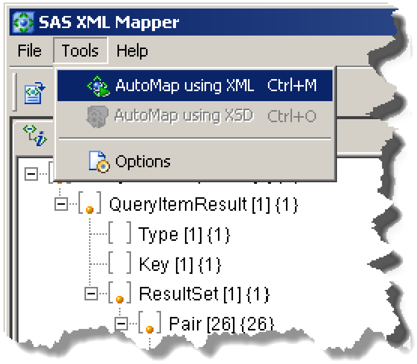


Message

ERROR: XML describe error: XML data is not in a format supported natively by the XML libname engine. Files of this type usually require an XMLMap to be input properly: .

The "XMLMap" to which this error refers is a file that instructs SAS how to interpret the file structure of the XML file being accessed. Though details about the specification of the XML engine and XMLMaps is beyond the scope of this paper, SAS does provide an easy-to-use tool for creating these mapping files[8]. [For more in-depth descriptions of how to use the XMLMapper tool see Hoyle (2010) and Mack (2010).] For the preceding PROC SOAP call, the first step in creating an XMLMap file can be accomplished by generating a disk file containing a response from the web service instead of using the TEMP access method that will be used in the production solution.

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

FILENAME request   'C:\_CDMS\MWSUG 2011\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;
```



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 specify the library "response" using the XML engine and the XMLMAP= option.

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

As in the previous REST example, you can then operate on these data as you would the datasets in any other read-only LIBRARY.

```
DATA work.NDC_Code_Data;
```

[8] 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

```
 SET response.pair;
RUN;
```

SOCKET & DDE

The last two access methods that reach outside of SAS are the SOCKET and DDE access methods. When using 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 via SOCKET connections. Helf (2005) expands on 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 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.

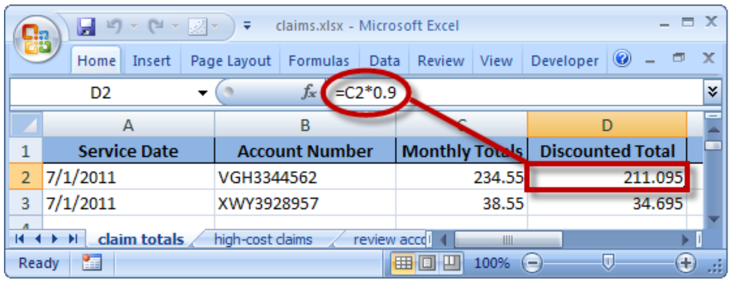
However, the SOCKET access method is not used broadly as an all-purpose means of accessing data from BASE SAS—mostly due to the myriad methodologies developed for presenting data via the web and (more recently) due to the growth of service oriented architectures that make data available via relatively simple calls to RESTful and SOAP-based web services and return data in formats that are readily interpreted without complex text parsing.

The Dynamic Data Exchange 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 on using the DDE access method—including Vyverman (2001, 2002), Watts (2005), and Derby (2008), but the short description of this access method is that it involves creating 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 make the connection between SAS and that 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 described as the "claim totals" worksheet within the "c:\claim reports\claims.xlsx" file, and the item or data range "r2c1:r4c3" (row 2, column 1 – row 4, column 3). The DATA _NULL_ step is then used to read the data from the "2011_07.txt" file and write the data to the defined excel fields in the claims excel file referenced as "clmout".

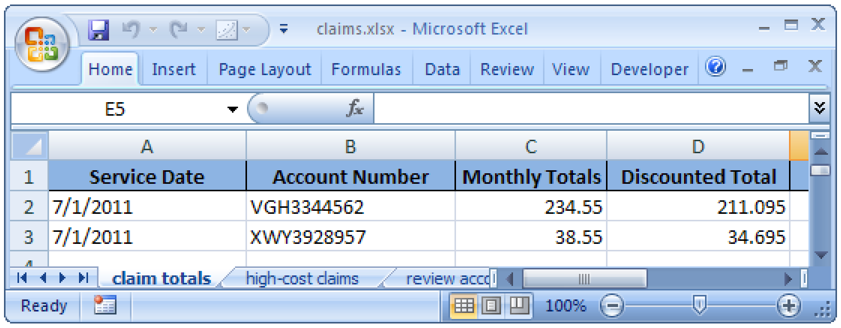
```
FILENAME clmin  '\\finance\analytics\report\source\2011_07.txt';
FILENAME clmout DDE "excel|c:\claim reports\[claims.xlsx]claim totals!r2c1:r4c3";

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 of each row (c3). This example begins to demonstrate the power of DDE to place data in worksheets that are formatted in a manner that takes advantage of Excel's available functionality. The column headers are filled with a predefined background color, the font is bold face and centered, and formulas can be assigned to cells that are not being overwritten by the data communication between the SAS client and the server application—as demonstrated by the calculation of the 90% "Discounted Total" that is dynamically computed as data are written to the worksheet.



	A	B	C	D
1	Service Date	Account Number	Monthly Totals	Discounted Total
2	7/1/2011	VGH3344562	234.55	211.095
3	7/1/2011	XWY3928957	38.55	34.695

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 the server application, 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].



	A	B	C	D
1	Service Date	Account Number	Monthly Totals	Discounted Total
2	7/1/2011	VGH3344562	234.55	211.095
3	7/1/2011	XWY3928957	38.55	34.695

CATALOG, DUMMY, CLIPBOARD, & OTHER ACCESS METHODS

The remainder of the access methods are grouped together in this last section, not because they are any less important (o.k., some might be less important if judged by the breadth of their use across the SAS user community), but because the author has not commonly used them in practice, nor has he seen them widely used. They likely play a crucial role in creative solutions of which the author remains naïve. The CATALOG access method, for example, can be used to organize and store SAS program code, macros, formats, and data elements used as parameters in production programs. In the following example (adapted from SAS, 2011e), the CATALOG access method is used to store source code for a series of programs in the "cdmsllc" catalog. In this case, the PUT statement that writes the "PROC OPTIONS; RUN;" source code to the "cdmsllc3" entry in the CDMSLLC catalog in the SRC_CODE library.

```
LIBNAME SRC_CODE 'C:\';
FILENAME prgrms CATALOG 'SRC_CODE.CDMSLLC.cdmsllc3.source';

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

Once these programs are written to the catalog, they can later be %INCLUDEd by calling them through the "cdms" fileref pointing to the catalog. In this example, the program source code "cdmsllc2" is called followed by "cdmsllc3".

```
FILENAME cdms CATALOG 'src_code.cdmsllc';
%INCLUDE cdms (cdmsllc2);
%INCLUDE cdms (cdmsllc3);
```

Writing to the DUMMY access method results in the data going 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 data. The other use DeGuire gives for the DUMMY access method is to define a file for the purpose of passing a file name to SAS Macros that require a filename as an input.

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

Finally, the author's favorite use 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 then 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. Through relatively simple command syntax, users can utilize SAS as an FTP client, an SMTP e-mail system, and a flexible application for consuming web services. Additionally, through the DDE access method, powerful solutions can be created using Excel (and Word and PowerPoint). All of these methods expand the capabilities of the SAS system for users who develop the necessary skills, and will help you achieve more efficient, more fully-automated solutions to the programming challenges you face day to day. These skills increase your productivity and 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.

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
Phone: 608.478.0029
E-mail: CSchacherer@cdms-llc.com
Web: www.cdms-llc.com