Paper 142-2017

Create a Unique Datetime Stamp for Filenames or Many Other Purposes

Joe DeShon, Boehringer Ingelheim Animal Health

ABSTRACT

This paper shows how to use Base SAS to create unique date/time stamps which can be used for naming external files. These file names provide automatic versioning for systems and are intuitive and completely sortable, providing enhanced flexibility when compared to generation data sets, which could be created by SAS or by the operating system.

INTRODUCTION

When using SAS to create external files, it is often desirable to create unique file names for these files. These file names should be easy to generate, should be sortable in the order that they were created, and should be completely unique, with no chance of file name collision in the future. These files can be used to keep track of various versions of the external files that are created for either development or for historical and archival purposes.

This paper illustrates how to create a SAS macro that can be called at any time during a SAS program. The macro generates a series of SAS macro variables that can be used to create unique file names for such purposes.

COMMON USES FOR DATE-TIME STAMPS

Common applications that might benefit from date/time stamp macro variables:

- An application that creates a CSV (Comma-Separated Value) file to be delivered to a vendor once a week. Since the file is created on the same directory each week, each successive run of the process must generate a unique file name. Further, it would be beneficial if the name of the file indicated the date and time that the process ran. It is also necessary that, when sorted alphabetically, the file names are also in their creation order.
- An application generates Excel spreadsheets that are delivered to various people in the organization. Each spreadsheet is unique to that individual (i.e., Bob's spreadsheet doesn't look like Mary's spreadsheet). But there is a desire to indicate through the name of the file which spreadsheet was run in which batch (in other words, a method to determine that Bob's spreadsheet "A" was generated in the same batch as Mary's spreadsheet "A" and the same is true for Bob's spreadsheet "B" and Mary's spreadsheet "B").

EXISTING SOLUTIONS IN BASE SAS® (GENERATION DATA SETS)

Base SAS provides a similar feature, but it works only for SAS data files, not for external formats.

To create a generation dataset in base SAS, include the GENMAX option in a DATA statement:

```
data a (genmax = 4);
  set sashelp.class;
run;
```

That creates a dataset that is part of a set of datasets that contain up to four versions:

A = base (current) version
A#003 = most recent (youngest) historical version
A#002 = second most recent historical version
A#001 = oldest historical version

This feature has several advantages:

- It is easy to implement and automatically maintained by SAS
- It is easy to reference the generations without the need to explicitly know the physical name of the generation

But it lacks a few features that we were most interested in

- It works only for SAS datasets, not for non-SAS formats such as CSV or XLSX
- It is a serial stamp, not a date/time stamp, so there is no intuitive indication when the dataset was created
- There is no method to coordinate multiple files created as part of the same process; every dataset's generation numbering is specific only to that dataset

So we determined that SAS's generation dataset feature was not adequate for what we wanted to do.

REQUIREMENTS

Before we designed the system, we came up with some general guidelines. Although none of the features was a "show-stopper" -- it would be okay if not all the requirements were fulfilled (after all, we figured some of them are actually self-contradictory) -- we wanted to create a system that would as much as possible satisfy the following general guidelines:

AMBIGUITY IS A VERY BAD THING

Confusion between similar-looking characters such as the letter "O" and the numeral zero should be avoided. If the characters are to be used, it should be very obvious from their context which is intended.

ACCURATE TO WITHIN ONE SECOND

A filename suffix should be unique for one second of time, and then never be generated again.

COMPACT AS POSSIBLE

Since we are proposing a system that will create a suffix for a file name -- and file names are already pretty long, the shorter we can make the suffix, the better.

FIXED LENGTH

This attribute makes it predictable; however long the suffix is, that's how long it should be expected to be in all cases.

READING IT SHOULD BE INTUITIVE

With very little training, a person should be able to look at a file name suffix and know the time and date the file was created.

EMBARRASSING / OFFENSIVE COMBINATIONS SHOULD BE AVOIDED

If alpha characters are used to create serial representations of time stamps, it is possible that un-intended offensive words or double entendre could be created. Steps should be taken to prevent this when possible.

CASE-AGNOSTIC

The suffix could be upper or lower case, but it should be irrelevant which. In other words, a suffix of "ABC123" should be equivalent to "abc123".

STAMP SHOULD BE UN-AMBIGUOUSLY SORTABLE BY DATE/TIME

Not only should be date and time be intuitive, it should be sortable. For that reason, January can't be represented by "JAN" because it wouldn't sort correctly. For more on the unambiguity nature of this feature, see the next requirement.

ANY GIVEN CHARACTER SHOULD BE ALPHA OR NUMERIC, BUT NOT BOTH

This requirement helps satisfy the unambiguous nature of sortability. If a given position is always numeric, or always character, there is no problem wondering if numerics sort before or after alphabetics. This also helps to solve the ambiguity issue of similar-appearing characters. For example, if it is known that the first character is always alphabetic, then it should be obvious that "O" is the letter "O", not the numeral zero.

DATE SHOULD NOT RESET ITSELF TOO SOON

Admittedly, this requirement is somewhat subjective. Since time marches forward infinitely, but a character-based system is limited, eventually any system will reset itself. A four-digit year will reset after 10,000 years; a two-digit year will reset after only 100. Using characters instead of numbers gives a larger domain to work in. At what point is a reset acceptable?

IMPLEMENTATION

We decided to create a macro that would create global macro variables that will remain static until regenerated by another invocation of the macro. This macro would be run in open code at any time to create date/time suffixes that could be used to create file names. The global macro variables could be used multiple times to create as many file names as necessary that could all later be tied together with the common suffix.

If desired, the macro could be run multiple times within the same session to generate multiple and unique suffixes.

In addition, it was determined that since many of the above requirements are mutually-exclusive, the same macro would create multiple, redundant macro variables using various algorithms. The process calling the macro can pick the most appropriate one for that particular use.

To simplify the syntax, all calculations are done in a _NULL_ data step in base SAS. Variables are created in the data vector first, rather than directly as macro variables. At the end of the step, the SYMPUT function is used to convert the data vector variables into macro variables.

INITIALIZATION

All date/time variables are initialized at the beginning of the process so they will be consistent throughout the implementation of the step:

```
datetime = datetime();
datetime_decimal =              /** Strip off just the decimal        **/
  datetime - int(datetime);     /** portion.                          **/
datetime = int(datetime());     /** Now make it just the integer      **/
                                /** portion.                          **/
today    = datepart(datetime);
time     = timepart(datetime);
```

FIRST VERSION

The first attempt was extremely verbose:

```
length yyyymmdd $ 08;
yyyymmdd = compress(put(today,yymmdd10.),'-');
length hhmmss $ 06;
hhmmss   = compress(translate(put(time,time8.),'0',''),':');
length date_time $ 15;
```

```
date_time = yyyymmdd || '_' || hhmmss;
```

This created a macro variable name $date_time in the format of *yyyymmdd_hhmmss*. The following cade illustrates its use:

```
%make_date_time;
data mylib.myclass_&date_time;
  set sashelp.class;
run;
```

This creates in the library mylib a SAS table named, say, myclass_20160813_132701. (The exact name of the output file will vary each time the program is run and will depend on the date and time of the run.)

Similarly, the process can be used for any external file that is created, such as this proc export:

```
%make_date_time;
proc export
  data = sashelp.class
  outfile = "c:\mydir\myclass_&date_time..xlsx";
run;
```

This creates an Excel spreadsheet on the folder c:\mydir with the name of, say, myclass_20160813_132701.xlsx. Notice that double quotes are necessary to properly expand the macro variable name. And two periods are needed: one to terminate the macro variable name and another as actually part of the spreadsheet name.

ANALYSIS OF FIRST VERSION AND RE-THINKING PRIORITIES

We placed the first version in production, started using it, and received feedback. The feedback was pretty much unanimous: the concept was great, the macro was useful, but the generated file names were too long. In our zeal to make the macro variable as obvious as possible, we sacrificed to the “Gods of Verbosity”.

It was back to the drawing board. We realized that our some of guidelines were mutually exclusive. We needed to determine which ones were really important and which of them could be compromised.

We soon came up with three more guiding principles to help us design a second version of the macro:

1. The length of the resulting macro variable was more important than we first believed. We determined it would be best to create a variable of approximately eight bytes. This should be short enough for most applications, and long enough to contain all the information we needed.
2. We determined that the variable must be intuitively readable and unambiguously sortable. For example, a completely serial variable might be small and sortable, but would give no indication as to its meaning. Conversely, a completely random value might be compact, but would not be sortable and would certainly not be intuitive in meaning.
3. Although we initially wanted accuracy of one second or better, we might be willing to compromise this in the interests of compactness. Since the primary use of the variable was to construct file names, and since most file names take at least a few seconds to create, perhaps a macro variable that had an accuracy of a few seconds or even to one minute might be sufficient.

SECOND VERSION

The second attempt turned out to be the best for our purposes. We settled on a nine-byte version.

We determined that, with a little sacrifice, we could accurately depict the year portion of the value in two bytes. But, with a nod to the recent Y2K debacle, we did not want to simply use the last two digits of the year, because that would reset at the next century.

For that reason, we decided to use a base-24 representation of the year using the values 'A-Z', omitting the letters ‘I’ and ‘O’. Since there was no reason to create variables for dates in the past, we used (somewhat arbitrarily) the year 2000 as our base. So the year 2000 was represented by 'AA', the year 2001 by 'AB', and so on.

First, a 24-byte alphabet data vector variable was initialized:

```
alphabet = 'ABCDEFGHJKLMNPQRSTUVWXYZ'; /** Missing 'I' and 'O' **/
```

The following code creates the first byte of a base-24 year with an offset of 2000:

```
substr(alphabet,abs(mod((year(today)-2000)/24,24)+1),1)
```

To create the second byte, we simply remove the initial division of 24:

```
substr(alphabet,abs(mod((year(today)-2000),24)+1),1)
```

Notice that the code uses the variable for TODAY that we created previously, rather than the TODAY() function. This was to guarantee that the value of the date/time did not change during the run of the data step (which could happen if the job was run close to midnight).

The month can easily be represented by one byte, simply the nth byte of the previously-defined 24-byte alphabet:

```
substr(alphabet,month(today),1)
```

Since the day of the month can be 01 through 31, it can’t be represented by one byte in a base-24 system, so we must use two bytes for it. We decided to simply use the numeric two-byte representation of the day of the month. This provided two serendipitous benefits:

1. Having numerics in the middle bytes established a recognizable pattern of character-numeric-character, which is more visually appealing than all-numerics or all-characters.
2. A two-byte day-of-month indicator in the middle of the string satisfied the requirement of making the variable intuitively readable. For example, it is now possible to glance at a file name and know instantly that is the file that was created on, say, the 13th of the month.

The code for these two bytes is pretty straightforward:

```
put(day(today),z2.)
```

Since there are 24 hours in the day, the hour to a single letter of the alphabet was a no-brainer (after adjusting for the zero offset):

```
substr(alphabet,hour(time)+1,1)
```

There are 60 minutes in each hour, so the minute value is best represented strictly by the numeric values '00' through '59':

```
put(minute(time),z2.)
```

And it is here that we had to get creative and sacrifice accuracy in the interests of succinctness. At this point, we are up to eight bytes. We have a pretty good aesthetic and intuitive mixture of numerics and characters. But there are 60 seconds in an hour, so we can’t just use a base24 character representation. If we use a '00'-'59' numeric representation, it will require two bytes instead of one.

It was at this point that we decided to sacrifice accuracy. By dividing the minute into approximately 24 intervals, we can save a byte and use the 24-byte alphabet and only one byte. Each interval would be 2.5 seconds long instead of one second long. Ultimately, this is what we decided to do for the last byte:

```
substr(alphabet,int(second(time)*(24/60))+1,1)
```

By putting all those together, we end up with the following code:

```
length date_timex $ 9;
alphabet = 'ABCDEFGHJKLMNPQRSTUVWXYZ'; /** Missing 'I' and 'O' **/
date_timex =
```

```
    substr(alphabet,abs(mod((year(today)-2000)/24,24)+1),1) ||
    substr(alphabet,abs(mod((year(today)-2000),24)+1),1)    ||
    substr(alphabet,month(today),1)                         ||
    put(day(today),z2.)                                     ||
    substr(alphabet,hour(time)+1,1)                         ||
    put(minute(time),z2.)                                   ||
    substr(alphabet,int(second(time)*(24/60))+1,1)
    ;
```

Running that code on August 13, 2016 at 4:12:32 PM generates a macro variable named date_timex with a value of 'ASH13S12N'.

Since most dataset names in Windows are represented with lower case letters, we also created a lower case variation named date_timexl, which would be equal to 'ash13s12n'.

The lower case 9-byte variable ultimately became the de facto standard for suffixes for external datasets when a unique name was desired.

The following demonstrates the usage:

```
%make_date_time;
data mylib.myclass_&date_timexl;
  set sashelp.class;
run;
```

This creates a SAS table named, say, myclass_ash13s12n in the mylib library (i.e., a file named myclass_ash13s12n.sas7bdat). The file name will have a unique suffix every time the program is run.

OTHER VERSIONS

At this point, we had pretty much fulfilled the assignment. We created a macro that could create three different variables: an extremely verbose date/time stamp (some people liked that level of detail) and a slightly more succinct variation -- nine bytes long with a combination of alphabetics and numerics that made it somewhat intuitive once you knew the code. And it was available in both an upper and a lower case version

But we didn't stop there. We kept experimenting, just in case we might discover something that will suit another purpose.

For example, we discovered that we could create a 7-byte version. In this case, we used only characters -- no numerics. It is simply a base26 representation of the SAS date/time stamp -- the number of seconds since January 1, 1970. A loop that converts any integer into base26 does the job. Ambiguity wasn't an issue, since the entire string is only alpha, so there is no need to skip the letters "I", "L", or "O". Both an upper case and lower case version were created:

```
    length date_timea  $ 7;
    length date_timeal $ 7;
    do i = 1 to 7;
      substr(date_timea,7-i+1,1) =
        substr('ABCDEFGHIJKLMNOPQRSTUVWXYZ',
               (int(mod(datetime,(26**i))/(26**(i-1))))+1,
                1);
      end;
    date_timeal = lowcase(date_timea);
```

On January 13, 2017 at 12:32 PM, this will create a macro variable named &date_timea with the value of "FVMUKER".

But here's a danger to consider: Since this version isn't a combination of numerics and characters, literally any 7-byte character string is possible – including some that will coincidentally contain offensive

4-byte combinations. If you use this version, you would need to consider the consequences of creating such a character string.

There was no reason to stop with that. If we remove the requirement of any given position being either alphabetic or numeric, we could reduce the length to six bytes by simply adding ten bytes of numeric digits to the possible characters. The logic is virtually the same, just with base36 instead of base26:

```
length date_timean  $ 6;
length date_timeanl $ 6;
do i = 1 to 6;
  substr(date_timean,6-i+1,1) =
    substr('0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',
          (int(mod(datetime,(36**i))/(36**(i-1))))+1,
           1);
  end;
date_timeanl = lowcase(date_timean);
```

On January 13, 2017 at 14:27:09 PM, this will create a macro variable named &date_timean with the value of “TRMW59”.

The main advantage is that it is one byte smaller. The disadvantage is that we have now given up on unambiguous sorting because it is now possible for any one byte to be either alphabetic or numeric. And it is still possible to have accidental offensive words in the string. But this may be a good solution if the size of the string is more important than the readability.

Finally, there was a request to create a version that was accurate to more than one second. This might be necessary for applications that create unique keys for records (as opposed to file names, which was the original purpose of this project). This was achieved by simply adding a millisecond portion to the previous variable:

```
datetime_decimal = round(datetime_decimal,0.001);
datetime_decimal = datetime_decimal * 1000;
datetime_decimal_char = put(datetime_decimal,z3.);
length date_timeanm $ 9;
date_timeanm = trim(date_timean) || datetime_decimal_char;
date_timeanml = lowcase(date_timeanm);
length date_timeam $ 10;
date_timeam = trim(date_timea) || datetime_decimal_char;
date_timeaml = lowcase(date_timeam);
```

On January 13, 2017 at 12:45:51 PM, this will create a macro variable named &date_timeam with the value of “FVMULLV550”.

FINAL VERSION

Following is the complete source code for the macro:

```
/*********************************************************************/
/* make_date_time.sas --                                             */
/* Create a macro variable named &date_time, which contains a date/time  */
/* stamp in the format of yyyymmdd_hhmmss, which can be used to create a */
/* unique file name.                                                 */
/* This macro needs to run as a stand-alone data step.  It accepts no    */
/* arguments.  The only output is the creation of the macro variable and */
/* a note in the log.                                                */
/*********************************************************************/

%macro make_date_time;
data _null_;
  datetime = datetime();          /** Freeze the time and date, so it's **/
                                  /** consistent throughout the data    **/
```

```
                                /** step.                              **/
  datetime_decimal =            /** Strip off just the decimal         **/
    datetime - int(datetime);   /** portion.                           **/
  datetime = int(datetime);     /** Now make it just the integer       **/
                                /** portion.                           **/
  today    = datepart(datetime);
  time     = timepart(datetime);
  length yyyymmdd $ 08;
  yyyymmdd = compress(put(today,yymmdd10.),'-');
  length hhmmss $ 06;
  hhmmss   = compress(translate(put(time,time8.),'0',''),':');
  length date_time $ 15;
  date_time = yyyymmdd || '_' || hhmmss;
  length date_timex $ 9;
  /** Define the alphabet minus "I" and "O", so they won't be confused  **/
  /** with "1" and "0"                                                  **/
  alphabet = 'ABCDEFGHJKLMNPQRSTUVWXYZ';
  date_timex =
    substr(alphabet,abs(mod((year(today)-2000)/24,24)+1),1) ||
    substr(alphabet,abs(mod((year(today)-2000),24)+1),1)    ||
    substr(alphabet,month(today),1)                         ||
    put(day(today),z2.)                                     ||
    substr(alphabet,hour(time)+1,1)                         ||
    put(minute(time),z2.)                                   ||
    substr(alphabet,int(second(time)*(24/60))+1,1)
    ;
  date_timexl = lowcase(date_timex);
  length date_timea  $ 7;
  length date_timeal $ 7;
  do i = 1 to 7;
    substr(date_timea,7-i+1,1) =
      substr('ABCDEFGHIJKLMNOPQRSTUVWXYZ',
            (int(mod(datetime,(26**i))/(26**(i-1))))+1,
             1);
    end;
  date_timeal = lowcase(date_timea);
  length date_timean  $ 6;
  length date_timeanl $ 6;
  do i = 1 to 6;
    substr(date_timean,6-i+1,1) =
      substr('0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ',
            (int(mod(datetime,(36**i))/(36**(i-1))))+1,
             1);
    end;
  date_timeanl = lowcase(date_timean);
  datetime_decimal = round(datetime_decimal,0.001);  /** In Windows, it's **/
                                                     /** accurate only to **/
                                                     /** three places     **/
                                                     /** anyway...        **/
  datetime_decimal = datetime_decimal * 1000;        /** Make an integer. **/
  datetime_decimal_char = put(datetime_decimal,z3.); /** Make a string.   **/
  length date_timeanm $ 9;                           /** With milli-      **/
                                                     /** seconds added.   **/
  date_timeanm = trim(date_timean) || datetime_decimal_char;
  date_timeanml = lowcase(date_timeanm);
  length date_timeam $ 10;   /** With milliseconds added...   **/
  date_timeam = trim(date_timea) || datetime_decimal_char;
  date_timeaml = lowcase(date_timeam);

/** Display all the values:                    **/

/** All numeric:                               **/
  call symput('date_time',date_time);           /** 20160512_152938 **/

/** Alpha/numeric, fixed positions:            **/
  call symput('date_timex',date_timex);         /** ASE12R29R       **/
  call symput('date_timexl',date_timexl);       /** ase12r29r       **/

/** Alpha/numeric (base 36):                   **/
  call symput('date_timean',date_timean);       /** TEZF1E          **/
  call symput('date_timeanl',date_timeanl);     /** tezf1e          **/
```

```
/** Alpha/numeric (base 36, with milliseconds: **/
  call symput('date_timeanm',date_timeanm);     /** TEZF1E387       **/
  call symput('date_timeanml',date_timeanml);   /** tezf1e387       **/

/** Alpha (base 26):                          **/
  call symput('date_timea',date_timea);         /** FTSHSOW         **/
  call symput('date_timeal',date_timeal);       /** ftshsow         **/

/** Alpha (base 26), with milliseconds:       **/
  call symput('date_timeam',date_timeam);       /** FTSHSOW387      **/
  call symput('date_timeaml',date_timeaml);     /** ftshsow387      **/

run;

%put NOTE: Macro variable %nrstr(&)date_time     created with a value of: &date_time..;
%put NOTE: Macro variable %nrstr(&)date_timex    created with a value of: &date_timex..;
%put NOTE: Macro variable %nrstr(&)date_timexl   created with a value of: &date_timexl..;
%put NOTE: Macro variable %nrstr(&)date_timean   created with a value of: &date_timean..;
%put NOTE: Macro variable %nrstr(&)date_timeanl  created with a value of: &date_timeanl..;
%put NOTE: Macro variable %nrstr(&)date_timeanm  created with a value of: &date_timeanm..;
%put NOTE: Macro variable %nrstr(&)date_timeanml created with a value of: &date_timeanml..;
%put NOTE: Macro variable %nrstr(&)date_timea    created with a value of: &date_timea..;
%put NOTE: Macro variable %nrstr(&)date_timeal   created with a value of: &date_timeal..;
%put NOTE: Macro variable %nrstr(&)date_timeam   created with a value of: &date_timeam..;
%put NOTE: Macro variable %nrstr(&)date_timeaml  created with a value of: &date_timeaml..;
%mend;
```

ADVANTAGES / DISADVANTAGES OF EACH METHOD

The macro creates six versions of date/time stamps, in both upper and lower case versions. The macro variables are of different sizes and have different characteristics, which gives them each a unique combination of advantages and disadvantages. To help you understand which variable is the best for your particular application, consult the following chart of advantages and disadvantages:

	Verbose numeric	Compact date/time	Base 36	Base 36 with ms	Base 26	Base 26 with ms
Variable name	`&date_time`	`&date_timex` `&date_timexl`	`&date_timean` `&date_timeanl`	`&date_timeanm` `&date_timeanml`	`&date_timea` `&date_timel`	`&date_timeam` `&date_timeaml`
Length	15	9	6	9	7	10
Mask	yyyymmdd_hhmmss	yymddhmms	xxxxxx	xxxxxx999	aaaaaaa	aaaaaaa999
Alpha/Numeric	Numeric	A/N	A/N	A/N	Alpha	Alpha
Example	20160512_152938	ASE12R29R	TEZF1E	TEZF1E387	FTSHSOW	FTSHSOW387
Accurate	✓ (1 second)	- (2.5 seconds)	✓ (1 second)	✓✓ (0.001 second)	✓ (1 second)	✓✓ (0.001 second)
Compact	-	✓	✓✓	✓	✓	-
Intuitive	✓✓	✓	-	-	-	-
Embarrassing / offensive combinations avoided	✓	✓	-	-	-	-
Case-agnostic	✓	✓	✓	✓	✓	✓
Un-ambiguously sortable	✓	✓	-	-	✓	✓
Each position alpha or numeric, but not both	✓	✓	-	-	✓	✓
Date should not reset itself too soon	✓✓✓✓ until Dec 31, 9999	✓✓✓ until Dec 31, 2575	✓ until Dec 23, 2028	✓ until Dec 23, 2028	✓✓ until Jul 8, 2214	✓✓ until July 8, 2214

EXAMPLES OF THE MACRO IN ACTION

Following are some examples of this macro and the macro variables in action:

EXPORTING A FILE TO A UNIQUELY-NAMED EXCEL SPREADSHEET:

```
%make_date_time;
proc export data = myfile outfile = '\mydir\myfile_&date_timexl..xlsx';
```

Result:

```
\mydir\myfile_ase12r29r.xlsx
```

ASSOCIATING TWO SAS FILES FROM THE SAME RUN:

```
%make_date_time;
libname mylib '\mydir';
data mylib.myclass_&date_timexl;
  set sashelp.class;
run;
data mylib.mycars_&date_timexl;
  set sashelp.cars;
run;
```

Result:

```
\mydir\myclass_ase12r29r.sas7bdat
\mydir\mycars_ase12r29r.sas7bdat
```

ASSOCIATING A SAS FILE AND AN EXCEL SPREADSHEET FROM THE SAME RUN:

```
%make_date_time;
libname mylib '\mydir';
data mylib.myclass_&date_timexl;
  set sashelp.class;
run;
proc export
  data = sashelp.class
  outfile = '\mydir\myclass_&date_timexl..xlsx';
```

Result:

```
\mydir\myclass_ase12r29r.sas7bdat
\mydir\myclass_ase12r29r.xlsx
```

A PREFIX, WHICH WILL SORT IDENTICAL RUNS TOGETHER:

```
%make_date_time;
libname mylib '\mydir';
data mylib. &date_timexl._myclass_;
  set sashelp.class;
run;
proc export
  data = sashelp.class
  outfile = '\mydir\&date_timexl._myclass.xlsx';
run;
```

Result:

```
\mydir\ase12r29r_myclass.sas7bdat
\mydir\ase12r29r_myclass.xlsx
```

OTHER POSSIBLE USES

We use it for suffixes for file names. But they could also be used for customer numbers, purchase order numbers, work order numbers, or any other identifier where it is necessary for unconnected sources to assign unique, discrete identifiers as identifiers for documents, file names, keys to specific records in a file, and many other purposes.

CONCLUSION

This macro is a flexible method of creating a variety of date/time stamps that can be used for many different purposes. The macro can easily be adapted to fit the particular needs of your application.

REFERENCES

Wikipedia articles on date/time stamps:
https://en.wikipedia.org/wiki/Timestamp
https://en.wikipedia.org/wiki/Timestamping_(computing)

University of Cambridge. “A summary of the international standard date and time notation” (ISO 8601) by Markus Kuhn. Accessed January, 2017. http://www.cl.cam.ac.uk/~mgk25/iso-time.html

The “other” way of doing this (SAS files only) with SAS generation datasets:
SAS. “Understanding Generation Data Sets”. Accessed January, 2017. http://tinyurl.com/sas-generation-datasets

CONTACT INFORMATION

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

Joe DeShon
joedeshon@yahoo.com