PharmaSUG 2022 - Paper AD-184

# SAS® Spontaneous Combustion: Securing Software Portability through Self-Extracting Code

Troy Martin Hughes

## ABSTRACT

Spontaneous combustion describes combustion that occurs without an external ignition source. The right combination of fire tetrahedron components—including fuel, oxidizer, heat, and chemical reaction—can be a deadly yet awe-inspiring phenomenon, and differs from traditional combustion that requires a fire source, such as a match, flint, or spark plugs (in the case of combustion engines). SAS® code as well often requires a "spark" the first time it is run or run within a new environment. For example, SAS programs may operate correctly in an organization's original development environment, but may fail in its production environment until necessary folders are created, SAS libraries are assigned, control tables are constructed, or configuration files are built or modified. And, if software portability is problematic within a single organization, imagine the complexities that exist when SAS code is imported from a blog, white paper, textbook, or other external source into a new environment. The lack of software portability and the complexities of initializing new software often compel development teams to build software from scratch rather than attempting to reuse or rehabilitate existent code. One solution is to develop SAS programs that flexibly build and validate required environmental and other components during execution. To that end, this text describes techniques that increase the portability, reusability, modularity, and maintainability of SAS code, and demonstrates self-extracting, spontaneously combustible code that requires no spark.

## INTRODUCTION

Software file extractors are commonly used to facilitate the automatic and efficient installation and initialization of applications. Compressed file formats such as ZIP and RAR help ensure that software is installed and initialized, incorporating environmental attributes such as the operating system, user permissions, and directory structure. During installation, user preferences often also can be selected and captured in a configuration file that can be modified as needed over time. Thus, a goal for applications development is not only to deliver software that operates reliably, but also to ensure a painless, reliable, and replicable installation experience. This simplicity is especially critical because most software applications are designed to be used by end-users who may have little to no technical expertise to troubleshoot deviations that might occur.

Conversely, SAS data analytic development often supports requirements and users *within* the developing organization rather than consumers purchasing external, third-party software. In many cases, the SAS practitioners themselves are both the authors and users of their code in a development model known as end-user development. Because SAS users are better equipped to troubleshoot code and because the code typically is unencrypted and freely viewable, software portability historically has not been a focus of SAS literature and is rarely discussed as a best practice. Instead, published SAS code more commonly instructs users to make specific modifications to the code or to the user environments before program execution can be successful, thus differing substantially from applications development in which portability is an expected and necessary performance requirement because code is typically encrypted and not intended to be modified once it is released.

Sometimes the distance that code must be ported is inescapably small—such as the few feet that separate an organization’s SAS development server from its production server. Yet, despite this proximity, many development teams still experience difficulty porting and implementing production code from development and test environments due to subtle variations in the architecture landscape. For example, although the operating systems and versions of SAS may be consistent, file structures, metadata libraries, SAS data sets, and user permissions may be disparate. Rather than recreating these environmental attributes manually, automated processes can detect their absence or deviation and facilitate their prompt creation or modification. This ensures a more reliable, repeatable process that can be ported more readily to other environments, including those outside the developing organization. Moreover, this reusability can ensure

that in the unfortunate corruption or contamination of the SAS infrastructure, self-extracting code can automatically recreate data sets, control tables, configuration files, and other components.

This text describes methods that can facilitate SAS code that runs "out of the box" without painful preparation during installation. Examples discussed include automatically creating logical folders, assigning SAS libraries, building control tables, and initializing configuration files. Where possible, dynamic, modifiable components of programs are maintained external to SAS code, such as in the AUTOEXEC.sas file, other configuration files, or control tables, thus facilitating code that can remain static even when ported to a new environment. In attaining portable code that can be reused both inside and outside the developing organization, SAS practitioners not only deliver quality but also help ensure that their work can be more easily maintained as well as adapted for future use by a broader audience.

## INITIALIZING SAS LIBRARIES AND LOGICAL FOLDERS

SAS data sets do not simply exist in the ether; they need SAS libraries to call home. Because libraries must reference a logical location in the directory structure, some knowledge of the SAS environment is required during initialization of a new program that references permanent libraries. SAS practitioners may not be aware of the logical structure or location of libraries or data sets, but this information can be acquired with the PATHNAME function that returns the logical path of a directory.

Consider a SAS practitioner familiar with the MINE library, but perhaps unfamiliar with its logical location. The following PATHNAME function (invoked with %SYSFUNC) displays the logical location of MINE:

```
libname mine 'D:\sas\myfolder';
%put %sysfunc(pathname(mine));
```

The path is printed to the log:

```
D:\sas\myfolder
```

To the extent that developers can ignore logical structure and utilize only SAS libraries, code will remain more flexible as well as portable to other systems and environments. Often, a single reference to the logical location can be included in the SAS AUTOEXEC.sas file or elsewhere in code referencing the root SAS directory of the logical drive, such as the following initialization of the &YOURS location:

```
%let yours=D:\sas\;
```

With this line added in the AUTOEXEC.sas file, programs can assign the SAS library MINE by executing more flexible code that references the global macro variable &YOURS as opposed to the static reference to the D drive and its SAS folder:

```
libname mine "&yours.myfolder";
```

This library assignment is a simpler solution than including the actual path. Moreover, since some logical locations include complicated directory structures, IP addresses, or other dynamic components such as a server name that can change over time, it's more efficient and effective to specify the root logical location only once within the AUTOEXEC.sas file. Thereafter, if the root path changes, this modification can be replicated throughout all derivative SAS code. In addition to being more flexible, this code is also more portable to new environments because the path needs to be modified only within AUTOEXEC.sas. In a large organization that operates separate SAS development, test, and production servers, the respective AUTOEXEC.sas files might differ, thus reflecting the respective server locations, but all SAS code maintained on those servers could remain identical.

Rather than incurring the additional cost and complexity of multiple SAS environments, many organizations instead separate development from production environments by distinct SAS libraries rather than distinct servers. In this more simplified structure, separate &PROD and &DEV macro variables could be initialized to represent these two environments, respectively:

```
%let prod=D:\sas\prod\;
%let dev=D:\sas\dev\;
```

These macro variables, &PROD and &DEV, could now be utilized throughout a program to reference the respective “environments” (which are really separate folders). Or, a layer of abstraction can be added, so &YOURS can be dynamically initialized based on whether a program is run in production versus development:

```
%macro init(env=DEV /* DEV or PROD */);
%global yours;
%if %upcase(&env)=PROD %then %let yours=&prod;
%else %let yours=&dev;
%mend;

%init();
%put &=yours;
```

Because the INIT macro is invoked but passes no arguments, the ENV parameter is initialized to its default value (DEV), which they initializes &YOURS to the value specified by &DEV:

```
YOURS=D:\sas\dev\
```

In other cases, separate SAS servers might be utilized for PROD and DEV environments, in which case the &SYSSITE automatic macro variable can be evaluated to determine dynamically in which environment a program is running. The &SYSSITE macro variable is unique to SAS instance, so it will be unique for each installation.

For example, the following INIT macro evaluates the value of &SYSSITE at a company to determine whether a program is running in the PROD or DEV environment, after which it initializes macro variables to be displayed on reports (&ENVIRONMENT), to define SAS libraries (YOURS), and to set logging levels (&LOGGING):

```
%macro init();
%global environment yours logging;
%if &syssite=1234567 %then %do;
   %let environment=Production;
   %let yours=F:\sas;
   %let logging=0;
   %end;
%else %if &syssite=2345678 %then %do;
   %let environment=Development;
   %let yours=H:\sas;
   %let logging=3;
   %end;
%mend;
```

Setting report titles, library locations, and logging levels are just a few of the many aspects of an environment that can be customized (and dynamically applied) through initialization macros such as INIT.

## INITIALIZING A SAS LIBRARY WHEN A FOLDER DOES NOT EXIST

In the previous section, the MINE library was initialized to the Myfolder folder, but what happens if the D:\sas\myfolder folder does not exist? Consider running the following code:

```
%let yours=D:\sas\;
libname mine "&yours.myfolder";
```

The log provides a note reflecting this failure:

```
NOTE: Library MINE does not exist.
```

Within an exception handling framework, the success of this library assignment can be tested programmatically to ensure that if a library could not be assigned, later code that relies on it will be skipped rather than execute and fail. The SAS automatic macro variable &SYSLIBRC (system library return code) will equal zero for all successful library assignments, or other values if the assignment was unsuccessful. The following macro TEST implements &SYSLIBRC within an exception handling framework that exits the SAS program if the library was not successfully assigned:

```
%macro test;
libname mine "&yours.myfolder";
%put &=syslibrc;
%if not(&syslibrc=0) %then %do;
    %put EXITING DUE TO LIBRARY ERROR;
    %return;
    %end;
data mine.mydata;
    set mydata;
run;
%mend;


%test;
```

Output from this code depicts the exception caused when the logical location referenced does not exist:

```
%test;
NOTE: Library MINE does not exist.
SYSLIBRC=-70008
EXITING DUE TO LIBRARY ERROR
```

If a library is not being assigned but a developer still wants to test its existence, the %SYSFUNC(LIBREF(mine)) function can be utilized within a macro-based exception handling framework. In many instances, code may be created on one system and subsequently ported to another. The first system will have the required logical structure, often created manually, to which SAS libraries are assigned. To run code in the second environment, however, that logical directory structure must be recreated. The previous code failed because the logical location D:\sas\myfolder did not exist, but as long as the superordinate location D:\sas exists, SAS offers a straightforward and portable method to create the subordinate logical directory in situ.

The SAS system option DLCREATEDIR is the companion to the SAS default option NODLCREATEDIR. Enabling DLCREATEDIR permits SAS to create logical folders during the LIBNAME library assignment. For example, rather than failing, the previous LIBNAME statement now creates the logical location D:\sas\myfolders (referenced in the first note) and assigns the MINE library (referenced in the second note):

```
options DLCREATEDIR;
libname mine "&yours.myfolder";

NOTE: Libref MINE was created.
NOTE: Libref MINE was successfully assigned as follows:
      Engine:        V9
      Physical Name: D:\sas\myfolder
```

Introduced in SAS 9.3, DLCREATEDIR is a powerful ally, considering that some common SAS techniques for logical folder creation are restricted or are not portable across environments. For example, the SAS X command, which facilitates DOS commands such as MKDIR (make directory) to be run within a DOS shell from the SAS system, is disabled in many environments. Many X commands also are not portable between

Windows and UNIX environments. A limitation of DLCREATEDIR, however, is its inability to recursively create folders.

For example, given that the logical location D:\sas exists and is set to the macro variable &YOURS, and given that the previous LIBNAME statement created the D:\sas\myfolder folder, you might want to create two levels of folders (if they do not already exist):

```
* attempts to initialize LEV2 library to D:\sas\myfolder\lev1\lev2;
%put &=yours;
libname lev2 "&yours.myfolder\lev1\lev2";
```

The log demonstrates the failure, however, because the Lev2 folder cannot be created if Lev1 does not already exist—even with DLCREATEDIR enabled:

```
%put &=yours;
YOURS=D:\sas\
libname lev2 "&yours.myfolder\lev1\lev2";
ERROR: Create of library LEV2 failed.
ERROR: Error in the LIBNAME statement.
```

Thus, to initialize the LEV2 library, you would first need to initialize a LEV1 library to the Lev1 folder—and not only is this horribly inefficient and error prone, but this second library might not even be required:

```
options DLCREATEDIR;
libname lev1 "&yours.myfolder\lev1";
libname lev2 "&yours.myfolder\lev1\lev2";
```

The log now demonstrates that LEV1 and LEV2 libraries were successfully assigned:

```
libname lev1 "&yours.myfolder\lev1";
NOTE: Library LEV1 was created.
NOTE: Libref LEV1 was successfully assigned as follows:
      Engine:        V9
      Physical Name: D:\sas\myfolder\lev1

libname lev2 "&yours.myfolder\lev1\lev2";
NOTE: Library LEV2 was created.
NOTE: Libref LEV2 was successfully assigned as follows:
      Engine:        V9
      Physical Name: D:\sas\myfolder\lev1\lev2
```

This manual process could be nightmarish if a developer had to recreate a complex, hierarchical file structure on a new system.

To overcome this limitation of DLCREATEDIR, the following RECURSDIR macro recursively creates all superordinate logical folders if they do not exist:

```
%macro recursdir(dir);
options DLCREATEDIR;
%let i=1;
%let nextdir=;
%do %while(%length(%scan(&dir,&i,\))>1);
   %if &i=1 %then %let nextdir=%scan(&dir,&i,\);
    %else %let nextdir=&nextdir.\%scan(&dir,&i,\);
```

```
   %put &=nextdir;
    libname temp "&nextdir";
    %let i=%eval(&i+1);
    %end;
%mend;
```

For example, consider a new requirement to initialize the NEW2 library to a folder location that has not yet been created (D:\sas\myfolder\new1\new2). Given that the New1 folder does not exist, the out-of-the-box LIBNAME statement fails when attempting this initialization:

```
* this fails;
libname new2 "&yours.myfolder\new1\new2";
```

However, passing this folder instead to RECURSDIR does initialize the NEW2 library while creating both the New1 and New2 folders in the process:

```
%recursdir(&yours.myfolder\new1\new2);
```

Now, when invoked and facing the challenge of D:\sas\myfolder\new1\new2, both the folder New1 and subordinate directory New2 are successfully created. Note that in this example, because the objective is solely to create logical folders and not corresponding SAS libraries, the library TEMP is repeatedly assigned to each newly created folder:

```
NEXTDIR=D:
NOTE: Libref TEMP was successfully assigned as follows:
      Engine:        V9
      Physical Name: D:\
NEXTDIR=D:\sas
NOTE: Libref TEMP was successfully assigned as follows:
      Engine:        V9
      Physical Name: D:\sas
NEXTDIR=D:\sas\myfolder
NOTE: Libref TEMP refers to the same physical library as MINE.
NOTE: Libref TEMP was successfully assigned as follows:
      Engine:        V9
      Physical Name: D:\sas\myfolder
NEXTDIR=D:\sas\myfolder\new1
NOTE: Library TEMP was created.
NOTE: Libref TEMP was successfully assigned as follows:
      Engine:        V9
      Physical Name: D:\sas\myfolder\new1
NEXTDIR=D:\sas\myfolder\new1\new2
NOTE: Library TEMP was created.
NOTE: Libref TEMP was successfully assigned as follows:
      Engine:        V9
      Physical Name: D:\sas\myfolder\new1\new2
```

Whatever method is utilized to create logical folders and to assign SAS libraries, the process should rely solely on repeatable code. This improves portability and, in the event that code fails and corrupts the environment or data, the corrupted files or folders can be deleted with the confidence that they can be recreated by again executing the code.

However, in some environments, software quality control standards require that SAS code be validated during testing and not modified thereafter in production. In extreme examples, which can be warranted in certain regulated industries or for critical infrastructure, hash checksum values of SAS code are created at validation and used during production to ensure that no changes have been made to the code after validation. In these environments, even modification to a SAS library reference would be disallowed, thus all dynamic aspects of code must be derived from external files such as AUTOEXEC.sas, configuration

files, or control tables. Consider an exception handling snippet from the author's text that first determines if the library LOCKNTRK exists and, if it does not, creates and assigns the library[i]:

```
%if %sysfunc(libref(lockntrk))^=0 %then %do;
    libname lockntrk '/folders/myfolders/lockandtrack'; /* USER MUST CHANGE */
    %end;
```

Although this code is effective in preventing process failure and patently indicates to SAS practitioners that the logical folder location must be modified, this violates code stability because it requires modification that could introduce error. Moreover, if the logical location changes, the program will cease to function until the code is corrected. Thus, in a strict environment in which code cannot be modified in production, if the location /folders/myfolders/lockandtrack needs to be modified, developers will need to return to the software test phase before production can be resumed. One solution to this problem that improves code flexibility is to ensure that all dynamic code elements (such as logical folder location) are described in external configuration files or control tables rather than within the code itself.

## INITIALIZING CONFIGURATION FILES

Configuration files provide instruction to software and can operationalize user preferences that are read during execution to alter program flow dynamically. For example, the SAS AUTOEXEC.sas file provides instruction to the SAS system, prescribing what should occur when SAS is run. Like many configuration files, AUTOEXEC.sas is read only once as SAS loads, thus changes to AUTOEXEC.sas during a SAS session will not take effect until SAS is closed and restarted.

Less commonly, however, configuration files can be read after initialization as software is running, thus incorporating configuration changes in near-real time. For example, each time a loop completed within some program, a macro might execute to reread the configuration file and, if values had changed, incorporate those updates into the next iteration within the software.

Just as the AUTOEXEC.sas file can customize, improve, and direct the SAS environment, SAS programs can also benefit from configuration files that enable more flexible, portable, and maintainable code. Configuration files can be XML, text, Excel workbooks, or other file formats that facilitate modification by users and access by SAS software. Some configuration files prescribe business rules and logic that control program operation. For example, one text by the author demonstrates how configuration files can be used to clean and standardize categorical data, replacing complex conditional logic or formatting commands that would otherwise be found inside SAS code[ii]. By removing logic from code, this allows users to customize business rules and decreases error-prone modifications to software. Moreover, in a robust system, these business rules can be further validated before execution to help ensure they conform to established requirements.

More central to the installation and initialization of SAS programs, however, configuration files are used to make modifications to establish the SAS environment in which code will operate. The previous examples create a SAS macro variable &YOURS in the AUTOEXEC.sas file and use this to reference all subsequent SAS library assignments, thus promoting flexibility and portability of code. An alternative method to achieve the same result is to include this information in a configuration file accessed only by programs that explicitly import the file. The following configuration file (D:\sas\myfolder\config.txt), is intended to assign two SAS libraries to dynamic logical locations that may differ across SAS environments:

```
<LIBRARY>
lockntrk: d:\sas\myfolder\lockntrk
temp: d:\sas\myfolder\temp
<OTHERSTUFF>
blah blah blah
```

The following code reads the configuration file with the INFILE statement and, once the header <LIBRARY> is observed, lines that follow represent library names and logical locations. Once the library and location are parsed from the configuration file, the CALL EXECUTE function calls the LIBNAME statement to assign the library and, if the logical location does not exist, the DLCREATEDIR option allows it to be created.

Thereafter, were a different header encountered, lines that follow that header could perform other actions (not demonstrated):

```
options dlcreatedir;
data _null_;
   length tab $300 category $12 lib $8 loc $250;
   infile "&yours.myfolder\config.txt" truncover;
   input tab $300.;
   if upcase(tab)='<LIBRARY>' then category='lib';
   else if substr(tab,1,1)='<' then category='';
   else do;
      if category='lib' then do;
           lib=scan(tab,1,':');
              loc=substr(tab,index(tab,':')+1);
              call execute('libname ' || strip(lib) || ' ' || '"' || strip(loc)
                  || '";');
         end;
        end;
   retain category;
run;
```

Note that the ELSE IF line is required to account for undefined keywords (denoted with brackets, like <LIBRARY>) that might appear within a configuration file (and thus be unaccounted for within the program interpreting this control file). Thus, although the configuration file denotes an <OTHERSTUFF> entry that is undefined in the subsequent DATA step, the ELSE IF catches this exception and facilitates the successful completion of the library assignment.

The above code produces the following abbreviated output.

```
NOTE: The infile "D:\sas\myfolder\config.txt" is:
      Filename=D:\sas\myfolder\config.txt,
      RECFM=V,LRECL=32767,File Size (bytes)=105,

lockntrk d:\sas\myfolder\lockntrk
temp d:\sas\myfolder\temp
NOTE: 5 records were read from the infile "D:\sas\myfolder\config.txt".
      The minimum record length was 9.
      The maximum record length was 34.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

NOTE: CALL EXECUTE generated line.
1    + libname lockntrk "d:\sas\myfolder\lockntrk";
NOTE: Libref LOCKNTRK was successfully assigned as follows:
      Engine:        V9
      Physical Name: d:\sas\myfolder\lockntrk
2    + libname temp "d:\sas\myfolder\temp";
NOTE: Libref TEMP was successfully assigned as follows:
```

```
      Engine:        V9
      Physical Name: d:\sas\myfolder\temp
```

Both libraries are dynamically initialized, but consider the need to recursively assign a library (because of a missing superordinate folder). To facilitate this objective, the RECURSDIR macro can be modified so that both the library name and logical location thereof are parameterized:

```
%macro recursdir(lib, loc);
options DLCREATEDIR;
%let i=1;
%let nextdir=;
%do %while(%length(%scan(&loc,&i,\))>1);
   %if &i=1 %then %let nextdir=%scan(&loc,&i,\);
    %else %let nextdir=&nextdir.\%scan(&loc,&i,\);
    libname &lib "&nextdir";
    %let i=%eval(&i+1);
    %end;
%mend;
```

Thereafter, an updated DATA step can reread the configuration file, and instead call the updated RECURSDIR macro to initialize all libraries dynamically:

```
options dlcreatedir;
data _null_;
   length tab $300 category $12 lib $8 loc $250;
   infile "&yours.myfolder\config.txt" truncover;
   input tab $300.;
   if upcase(tab)='<LIBRARY>' then category='lib';
   else if substr(tab,1,1)='<' then category='';
   else do;
      if category='lib' then do;
           lib=scan(tab,1,':');
              loc=substr(tab,index(tab,':')+1);
           call execute('%recursdir(' || strip(lib) || ', ' ||
                 strip(loc) || ');');
        end;
      end;
   retain category;
run;
```

The improved code now calls RECURSDIR to ensure that superordinate folders will be created if they are evaluated to be missing. And if this code is ported to a new environment, it can remain unchanged while developers modify only the configuration file, thus preserving the integrity of the software.

In production code intended to be robust and reliable, implementation of the previous solution would moreover necessitate additional exception handling routines to validate the input of the configuration file. Especially where configuration files are intended to be modified manually by developers in a text editing application like WordPad or Notepad, input validation rules should ensure that the structure and content of the configuration file are accurate. And if some aspect of the configuration file is found to be erroneous, business rules can dictate that a specific value reverts to a default value (rather than the invalid one), that

the entire configuration file is overwritten and reverts to a default configuration file that specifies default values, or that the process or program is terminated with a message communicating the specific failure.

Additional user-defined preferences can also be included in the same configuration file. For example, the file might additionally specify colors to be used in stoplight reporting. A separate text by the author demonstrates stoplight reporting as a quality control device but, if organizations want to implement the code and change something insubstantial like the coloring from "very light red" to "red", they are faced with this enigmatic, inscrutable code within the REPORT procedure[iii]:

```
if length (_c%sysevalf(5+((&b-1) * 7)+3+((&a-1)*&hrs*7))_)>1
    or length(_c%sysevalf(5+((&b-1) * 7)+5+((&a-1)*&hrs*7))_)>1 then do;
    call define("_c%sysevalf(5+(&b * 7)+((&a-1)*&hrs*7))_",'style',
          'style=[backgroundcolor=very light red flyover="'||
          strip(_c%sysevalf(5+((&b-1) * 7)+6+((&a-1)*&hrs*7))_)||'"]');
    end;
```

Modifying anything in this code quagmire can be somewhat intimidating and, because "very light red" was not encoded dynamically as a macro variable, the value must be manually changed throughout the code wherever it appears. Although this does not diminish the efficiency of the code as it executes, it does diminish its readability and maintainability, which are valued when a SAS practitioner must modify the code for any reason.

To overcome this issue and instead dynamically assign report color from a control file (like a configuration file), a second configuration file (D:\sas\myfolder\config_new.txt) can be created:

```
<LIBRARY>
reports: d:\sas\myfolder\reporting\misc_reports
<STOPLIGHT>
lev1: green
lev2: very light red
```

Note that a new keyword <STOPLIGHT> has been added, with LEV1 and LEV2 represented as subordinate key-value pairs. This new keyword requires subtle modification to the DATA step that ingests configuration files to create dynamic instructions. Moreover, additional configurability can be gained by parameterizing the configuration file name (and folder location), so the CONFIG parameter is now declared within the READ_CONFIG macro:

```
%macro read_config(config /* config dir and filename */);
data _null_;
   length tab $300 category $12 lib $8 loc $250;
   infile "&config" truncover;
   input tab $300.;
   if upcase(tab)="<LIBRARY>" then category="lib";
   else if upcase(tab)="<STOPLIGHT>" then category="stop";
   else if substr(tab,1,1)='<' then category='';
   else do;
      if category="lib" then do;
         lib=scan(tab,1,':');
            loc=substr(tab,index(tab,':')+1);
         call execute('%recursdir(' || strip(lib) || ', ' ||
            strip(loc) || ');');
            end;
      else if category="stop" then do;
```

```
         if strip(upcase(scan(tab,1,":")))="LEV1"
            then call symputx("lev1",strip(scan(tab,2,":")),'g');
         if strip(upcase(scan(tab,1,":")))="LEV2"
            then call symputx("lev2",strip(scan(tab,2,":")),'g');
         end;
      end;
    retain category;
run;
%mend;
```

READ_CONFIG can now be invoked, passing the configuration filename, and read the updated configuration file Config_new.txt:

```
%read_config(&yours.myfolder\config_new.txt);
%put &=lev1;
%put &=lev2;
```

A partial listing of the log demonstrates that the &LEV1 and &LEV2 macro variables were dynamically declared, as specified in the configuration file <STOPLIGHT> section, and can now be referenced with SAS reports in lieu of statically defined colors:

```
%put &=lev1;
LEV1=green
%put &=lev2;
LEV2=very light red
```

The configuration file data structure has been shown to be flexible, with a new keyword <STOPLIGHT> now interpreted and operationalized within the READ_CONFIG macro. However, with future changes expected in the configuration file—including modifications to both its structure and content—some method to validate a configuration file programmatically could be warranted.

For example, what if components of a configuration file are missing or erroneous or, possibly worse, the entire configuration file itself is missing? Thus, “portable” software describes not only the ability of SAS code to run in a new environment, but also that necessary control files are available, such as configuration files or control tables. Control files can be manually copied from one environment to another, created from scratch on the new system, or in some cases, generated by the code itself.

The following SAS program (saved as D:\sas\myfolder\makeit.sas) creates the previous configuration file if it is missing from the current logical path from which the code is executed:

```
* saved as d:\sas\myfolder\makeit.sas;
%macro makeit(config_file /* file to check */);
%if &SYSSCP=WIN %then %do;
   %let path=%sysget(SAS_EXECFILEPATH);
   %let path=%substr(&path,1,%length(&path)-%length(%scan(&path,-1,\)));
   %end;
%else %do;
   %let pathfil=&_SASPROGRAMFILE;
   %let pathno=%index(%sysfunc(reverse("&pathfil")),/);
   %let path=%substr(&pathfil,1,%eval(%length(&pathfil)-&pathno+1));
   %end;
%if %sysfunc(fileexist(&path.&config_file))^=1 %then %do;
   data _null_;
      file "&path.&config_file" termstr=CRLF;
      put "<LIBRARY>";
      put "reports: &path.reporting";
```

```
        put "<STOPLIGHT>";
        put "lev1: green";
        put "lev2: very light red";
     %end;
  run;
  %mend;

  %makeit(config.txt);
```

Thus, consider that Makeit.sas has been ported to a new environment in which the program expects to find the Config.txt configuration file in the same folder as the program file. Rather than having to statically specify the entire path, only the filename (config.txt) can be passed while the MAKEIT macro dynamically assesses in which folder the Makeit.sas program file is running.

To test the functionality of the MAKEIT macro, first delete the Config.txt configuration file (from the D:\sas\myfolder folder), and subsequently rerun MAKEIT. The macro will have created the Config.txt file dynamically from code, which can be beneficial when porting software from one environment to another.

In this example, the path of the program is dynamically assessed (in Windows Display Manager and UNIX environments) and, if a configuration file does not already exist in this folder, it is created. However, if the code is run from SAS Enterprise Guide, the &PATH macro variable is not created, demonstrating a lack of portability of the code (within that specific environment). Moreover, because the functionality that assesses the path of the current SAS program could be something valuable in future software, code reusability can be improved by segmenting this module into a separate macro that can be called by other programs.

The following code now demonstrates the GET_CURRENTPATH macro, which sets the global macro variable &PATH to the path of the current (named) SAS program:

```
* saved as d:\sas\myfolder\get_currentpath.sas;
%macro get_currentpath();
%let syscc=0;
%global path get_currentpathRC;
%let path=;
%let get_currentpathRC=GENERAL FAILURE;
%if %symexist(_clientprojectpath) %then %do;
   %let path=%sysfunc(dequote(&_clientprojectpath));
   %let path=%substr(&path,1,%length(&path)-%length(%scan(&path,-1,\)));
   %end;
%else %if &SYSSCP=WIN %then %do;
   %let path=%sysget(SAS_EXECFILEPATH);
   %let path=%substr(&path,1,%length(&path)-%length(%scan(&path,-1,\)));
   %end;
%else %if &_CLIENTAPP=SAS Studio %then %do;
   %let pathfil=&_SASPROGRAMFILE;
   %let pathno=%index(%sysfunc(reverse("&pathfil")),/);
   %let path=%substr(&pathfil,1,%eval(%length(&pathfil)-&pathno+1));
   %end;
%else %do;
   %let get_currentpathRC= Environment Not Defined!;
   %put &get_currentpathRC;
   %end;
%if &syscc=0 and %length(&path)>0 %then %let get_currentpathRC=;
%mend;
```

Thereafter, the MAKEIT macro can be revised to rely on GET_CURRENTPATH to assess the current pathname, after which MAKEIT only is responsible for creating the configuration file if it does not exist:

```
* saved as d:\sas\myfolder\makeit.sas;

%include "&yours.myfolder\get_currentpath.sas";
```

```
%macro makeit(config_file /* file to check */);
%get_currentpath();
%put &=path &=get_currentpathRC;
%if %length(&get_currentpathRC)=0 %then %do; * validates GETCURRENTPATH
macro;
   %if %sysfunc(fileexist(&path.&config_file))=0 %then %do;
      data _null_;
      file "&path.&config_file" termstr=CRLF;
            put "<LIBRARY>";
      put "reports: &path.reporting";
            put "<STOPLIGHT>";
            put "lev1: green";
            put "lev2: very light red";
      %end;
   %end;
run;
%mend;

%makeit(config.txt);
```

The program now runs in Windows Display Manager, the UNIX-based SAS University Edition, and the SAS Enterprise Guide, demonstrating even more portability than before. Some exception handling is also included, now testing to ensure that a valid path is assigned as well as validating that the &SYSCC global macro variable remains 0, the no-error condition. Within the MAKEIT macro, the &GETCURRENTPATHRC return code is evaluated, and the configuration file will only be created 1) when a valid path is evaluated and 2) no configuration file exists in that folder.

Configuration files are an essential component to creating dynamic code that flexibly adapts to both its environment and user preference while allowing the code substrate itself to remain intact and unchanged. Rather than passing dynamic attributes through macro parameters as a macro is invoked, these dynamic attributes are read from an external file, validated for format and/or content, and interpreted by the program to alter program control.

## INITIALIZING CONTROL TABLES

Control tables operate similarly to configuration files in that data-driven design can alter program flow or program attributes based on control data rather than business logic statically defined within code. A key difference, however, is that although configuration files typically are modified directly by the user (or through initialization processes that recommend default values based on the system environment), control tables tend to interact with processes as they are executing. For example, a control table might be used to record process performance metrics, and those metrics might in turn drive subsequent processing.

Critical to SAS programming, control tables enable two or more concurrent processes to speak to each other, essentially by taking turns accessing the same control file (like a SAS data set) and by leaving messages for each other. The following code excerpt from a separate text by the author instantiates a control a table (CONTROL.etl) if it does not exist:[iv]

```
%macro build_control();
%if %sysfunc(exist(control.etl))=0 %then %do;
    data control.etl;
        length process $32 dsn $32 date_complete 8;
        format date_complete date10.;
        if ^missing(process);
    run;
    %end;
%mend;
```

The BUILD_CONTROL macro ensures that when this software is ported to a new environment, the code will not fail if the control table does not exist. To ensure this instantiation occurs, preceding code should first test for the existence of the CONTROL library and, if it does not exist, initialize it. Once created, the control table is accessed for seconds only at a time by DATA steps that read its contents and update the table with data before closing the table, thus allowing other processes to access the table subsequently to communicate. Despite the split-seconds during which each process accesses the control table, to ensure that two concurrent processes do not attempt to access it simultaneously (which would cause a process failure), the LOCKITDOWN macro or similar control gate should be implemented, as described in a separate text by the author.[v]

Because control tables are often corralled in separate SAS libraries to help ensure that they are accessed only by SAS processes and not directly by users, this further validation of the control table library should be a required step in production software. This is demonstrated in the following excerpts from the macro LOCKANDTRACK, which first validates the SAS library LOCKNTRK and subsequently creates the control table Lockntrk.control if it does not exist:

```
%if %sysfunc(libref(lockntrk))^=0 %then %do;
    libname lockntrk '/folders/myfolders/lockandtrack';
    %end;

%if %sysfunc(exist(lockntrk.control))^=1 %then %do;
    data lockntrk.control;
        length lib $12 fname $32 dsid 8 program $32 process $32 sec 8 max 8
           type $2 dtg_request 8 dtg_timeout 8 dtg_lockstart 8
           dtg_lockstop 8 lockedby_program $32 lockedby_process $32
           lockedby_start 8;
        format dtg_request dtg_timeout dtg_lockstart dtg_lockstop
           lockedby_start datetime17.;
    run;
    %end;
```

Control tables offer a powerful advantage to SAS developers intent on creating reliable, robust software—especially software designed to run on concurrent SAS sessions. When implemented in a flexible manner with exception handling routines that validate library and data set existence (and which can create both if do not exist), software portability is improved. Moreover, in cases in which a control table becomes corrupt due to the entry of erroneous data, this flexibility allows the table to be deleted, after which it can automatically regenerate like a Phoenix when the code executes again.

## CONCLUSION

Portability of software reflects its ability to execute reliably and effectively within new and varying environments. Portable code is more likely to be widely used because it flexibly adapts to dynamic infrastructures without requiring complex modifications or refactoring. One aspect of portability often overlooked in SAS literature is the installation and initialization period in which SAS programs are imported from external sources and first executed in a new environment. Dynamic, data-driven code can facilitate ease of installation as well as further configuration, by maintaining dynamic elements within external files such as configuration files or control tables rather than in SAS code. Improving the installation and initialization experience not only aids development teams as they progress from development, to testing, to production phases, but also aids external organizations in the effortless adoption of previously validated and published, spontaneously combustible SAS code.

## REFERENCES

[i] Hughes, Troy Martin. 2015. Beyond a One-Stoplight Town: A Base SAS Solution to Preventing Data Access Collisions through the Detection, Deployment, Monitoring, and Optimization of Shared and Exclusive File Locks. *Western Users of SAS Software (WUSS).*

[ii] Hughes, Troy Martin. 2013. Binning Bombs When You're Not a Bomb Maker: A Code-Free Methodology to Standardize, Categorize, and Denormalize Categorical Data Through Taxonomical Control Tables. *Southeast SAS Users Group (SESUG)*.

[iii] Hughes, Troy Martin. 2014. Will You Smell Smoke When Your Data Are on Fire? The SAS Smoke Detector: Installing a Scalable Quality Control Dashboard for Transactional and Persistent Data*. Midwest SAS Users Group (MWSUG)*.

[iv]Hughes, Troy Martin. 2014. Calling for Backup When Your One-Alarm Becomes a Two-Alarm Fire: Developing Base SAS Data-Driven, Concurrent Processing Models through Fuzzy Control Tables that Maximize Throughput and Efficiency. *South Central SAS Users Group (SCSUG)*.

[v] Hughes, Troy Martin. 2014. From a One-Horse to a One-Stoplight Town: A Base SAS Solution to Preventing Data Access Collisions through the Detection and Deployment of Shared and Exclusive File Locks. *Western Users of SAS Software (WUSS)*.

## CONTACT INFORMATION

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

Name: Troy Martin Hughes
E-mail: troymartinhughes@gmail.com