Paper BB-04-2015

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. With the right combination of fire tetrahedron components—including fuel, oxidizer, heat, and chemical reaction—it 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. Thus, SAS programs may operate correctly in an organization's original development environment, but may fail in its production environment until 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 one's own infrastructure. The lack of software portability and the complexities of initializing new code often compel development teams to build code from scratch rather than attempting to rehabilitate or customize existent code to run in their environment. A solution is to develop SAS code that flexibly builds and validates its environment and required components during execution. To that end, this text describes techniques that increase the portability, reusability, and maintainability of SAS code by demonstrating 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, taking into account 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 moreover, 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.

SAS data analytic development, conversely, 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.” Thus, because SAS users are more 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 users' environments before program execution can be successful, differing substantially from applications development in which portability is an expected and necessary performance requirement since code is not intended to be modified once 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, while operating systems and versions of SAS hopefully will not differ, 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 easily 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 from scratch.

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 help ensure 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. And, 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 achieved easily with the PATHNAME function, which returns the logical path of a directory. Below, the logical folder of the library HERE is printed.

```
libname here '/folders/myfolders/here';
%put %sysfunc(pathname(here));
```

To the extent that developers can ignore logical structure and utilize solely 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.

```
%let basedir=/folders/myfolders/;
```

With the above line added in the AUTOEXEC.sas file, programs now can assign the SAS library HERE by executing more flexible code that references the global macro variable &BASEDIR as opposed to '/folders/myfolders'.

```
libname here "&basedir.here";
```

The above 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 some aspect of the root path structure changes, this change can be made once and it will be replicated throughout all derivative SAS code. In addition to being more flexible, this code also is more portable to new environments, again because the path only needs to be modified within AUTOEXEC.sas. In a large organization operating 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, the library HERE could initially be set to the directory /folders/myfolders/here/development during development and to /folders/myfolders/here/production once development, testing, and validation had completed and the program was operational. This methodology allows the code to remain static throughout development and production phases while the single modification is made within the AUTOEXEC.sas or other configuration file.

In attempting to create the HERE library, the above LIBNAME statement assumes that the logical location /folders/myfolders/here already exists. If it does not, the following output is produced and the library is not created.

```
libname here "&basedir.here";
NOTE: Library HERE does not exist.
```

Within an exception handling framework, the success of the above library assignment can be tested automatically 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. And, if a library is not being assigned but a developer still wants to test its existence, the %SYSFUNC(LIBREF(here)) function can be utilized within a macro-based exception handling framework.

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

%test;
```

Output from the above code depicts the exception caused when the logical location referenced does not exist.

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

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 above code failed because the logical location /folders/myfolders/here did not exist, but so long as the superordinate location /folders/myfolders 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 above LIBNAME statement now both creates the logical location /folders/myfolders/here (referenced in the first note) and assigns the HERE library (referenced in the second note.)

```
options DLCREATEDIR;
libname here "&basedir.here";
NOTE: Library HERE was created.
NOTE: Libref HERE was successfully assigned as follows:
      Engine:        V9
      Physical Name: /folders/myfolders/here
```

Introduced in SAS v9.3, 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 /folders/myfolders exists and is set to the macro variable &BASEDIR, one might imagine that the following code would create the directory /folders/myfolders/lev1/lev2. However, the following errors are received.

```
options DLCREATEDIR;
libname lev2 "/folders/myfolders/lev1/lev2";
ERROR: Create of library LEV2 failed.
ERROR: Error in the LIBNAME statement.
```

Thus, if the folder LEV1 does not first exist, the subordinate folder LEV2 will not be created. To recursively create both folders, two LIBNAME assignments must be made.

```
options DLCREATEDIR;
libname lev1 "/folders/myfolders/lev1";
NOTE: Library LEV1 was created.
NOTE: Libref LEV1 was successfully assigned as follows:
      Engine:        V9
      Physical Name: /folders/myfolders/lev1
libname lev2 "/folders/myfolders/lev1/lev2";
NOTE: Library LEV2 was created.
NOTE: Libref LEV2 was successfully assigned as follows:
      Engine:        V9
      Physical Name: /folders/myfolders/lev1/lev2
```

This manual process could be nightmarish and error-prone if a developer had to recreate a complex, hierarchical file structure on a new system. The following straightforward macro RECURSDIR handles this limitation, by recursively creating 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);
    %let nextdir=&nextdir./%scan(&dir,&i,/);
    libname temp "&nextdir";
    %let i=%eval(&i+1);
    %end;
%mend;
```

Now, when invoked and facing the challenge of /folders/myfolders/lev1/lev2, both the directory lev1 and subordinate directory lev2 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, and finally comes to reference the entire folder /folders/myfolders/lev1/lev2.

```
%recursdir(dir=&basedir/lev1/lev2);
NOTE: Libref TEMP was successfully assigned as follows:
      Engine:        V9
      Physical Name: /folders
NOTE: Libref TEMP was successfully assigned as follows:
      Engine:        V9
      Physical Name: /folders/myfolders
NOTE: Library TEMP was created.
NOTE: Library TEMP was created.
NOTE: Libref TEMP refers to the same physical library as LEV1.
NOTE: Libref TEMP was successfully assigned as follows:
      Engine:        V9
      Physical Name: /folders/myfolders/lev1
NOTE: Library TEMP was created.
NOTE: Library TEMP was created.
NOTE: Libref TEMP refers to the same physical library as LEV2.
NOTE: Libref TEMP was successfully assigned as follows:
      Engine:        V9
      Physical Name: /folders/myfolders/lev1/lev2
```

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, code validation efforts 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 since validation. In these extreme cases, even modification to a SAS library reference would be disallowed, thus all dynamic aspects of code must be derived from external files, either AUTOEXEC.sas or other 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'; /* MUST BE MODIFIED */
    %end;
```

While the above code is effective in preventing process failure and clearly indicates to SAS practitioners that the line must be modified with their specific logical folder location, it violates the principle that code should be static because it requires modification that could introduce error. Moreover, if the logical location changes, the program will cease to function until the code is modified. Thus, in a strict environment like the one described above 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 development and test cycles before production can be resumed. A common solution to this problem, as well as a method to make SAS code even more flexible, is to ensure that all dynamic code elements (such as logical folder locations) are described in external configuration files or control tables rather than in SAS code.

INITIALIZING CONFIGURATION FILES

Configuration files provide instruction to software and can reflect user selections or preferences that are read during execution and which often dynamically alter program flow. 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. Some configuration files, however, additionally can be read after initialization as code is running, thus incorporating configuration changes in real- or near-real-time. For example, each time a loop completes, code might reread the configuration file and, if values have changed, incorporate those updates into the next loop iteration.

Just as the AUTOEXEC.sas file can customize, improve, and direct the SAS environment, SAS programs also can benefit from configuration files that enable more flexible, portable, and maintainable code. Configuration files typically are XML or ASCII text files that facilitate modification by users as well as access by SAS programs. 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 otherwise would be found inside SAS code.[ii] By removing logic from code, this allows other users to customize business rules for their own use without any modification to code, thus eliminating the error-prone nature of changing lines of code. 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 examples above create a SAS macro variable &BASEDIR 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 that program or others that explicitly import the file. The following configuration file, Config.txt, is intended to assign two SAS libraries to dynamic logical locations that may differ across SAS environments.

```
<LIBRARY>
lockntrk: /folders/myfolders/lockntrk
temp: /folders/myfolders/temp/archive
<OTHERSTUFF>
blah blah blah
```

The following code reads the configuration file (located in a folder TEST) 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, if a different header were encountered, lines that follow that header could perform other actions, not included in this example.

```
options dlcreatedir;
data _null_;
    length tab $100 category $12 lib $8 loc $32;
    infile "&basedir.test/config.txt" truncover;
    input tab $100.;
    if upcase(tab)="<LIBRARY>" then category="lib";
    else do;
        if category="lib" then do;
           lib=scan(tab,1,":");
           loc=scan(tab,2,":");
           call execute('libname ' || strip(lib) || ' ' || '"' || strip(loc)
                  || '";');
           end;
        end;
    retain category;
run;
```

The above code produces the following abbreviated output.

```
NOTE: CALL EXECUTE generated line.
    + libname lockntrk "/folders/myfolders/lockntrk";
NOTE: Library LOCKNTRK was created.
NOTE: Libref LOCKNTRK was successfully assigned as follows:
     Engine:        V9
     Physical Name: /folders/myfolders/lockntrk
    + libname temp "/folders/myfolders/temp/archive";
ERROR: Create of library TEMP failed.
ERROR: Error in the LIBNAME statement.
```

Note that while the LOCKNTRK library was successfully assigned (and its logical location created), creation of the TEMP library failed because, as described in the previous section, the DLCREATEDIR option can only create a single folder and cannot recursively create superordinate folders. One solution is to refine the RECURSDIR macro from above so that it accepts parameters for both a SAS library name and corresponding logical location, and to replace the LIBNAME statement in the DATA step with an invocation of RECURSDIR.

```
%macro recursdir(dir=, loc=);
options DLCREATEDIR;
%let i=1;
%let nextdir=;
```

```
%do %while(%length(%scan(&loc,&i,/))>1);
    %let nextdir=&nextdir./%scan(&loc,&i,/);
    libname &dir "&nextdir";
    %let i=%eval(&i+1);
    %end;
%mend;

data _null_;
    length tab $100 category $8 lib $8 loc $32;
    infile "&basedir.test/config.txt" truncover;
    input tab $100.;
    if upcase(tab)="<LIBRARY>" then category="lib";
    else do;
        if category="lib" then do;
           lib=scan(tab,1,":");
           loc=scan(tab,2,":");
           call execute('%recursdir(dir=' || strip(lib) || ', loc=' || strip(loc)
                  || ');');
           end;
        end;
    retain category;
run;
```

The improved code now correctly creates two libraries, LOCKNTRK and TEMP, prescribed through the external configuration file. And, when 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 actual SAS code.

In production code intended to be robust and reliable, implementation of the above solution moreover would 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 window 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 unacceptable one), that the entire configuration file is overwritten and reverts to a default configuration file with all default values, or that the process or program is terminated with a message indicating the specific failure.

With the SAS environment specified and established via a configuration file, the SAS program can commence, with aspects of performance also customized through 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 the above 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. While this does not diminish the efficiency of the code as it executes, it does diminish its maintainability—thus, the developer's efficiency in maintaining or making subtle modifications to existent code. This complexity and lack of flexibility can cause developers to be hesitant to implement published code, or to make modifications to the code to further customize it.

The following updated configuration file now includes not only a reference to a logical folder location but moreover includes color values to be implemented in two levels of stoplight reporting.

```
<LIBRARY>
reports: /folders/myfolders/reporting
<STOPLIGHT>
lev1: green
lev2: very light red
```

The RECURSDIR macro is again referenced but remains unchanged, although the DATA step is modified to include business rules that reflect how to interpret values under the STOPLIGHT heading of the configuration file.

```
data _null_;
    length tab $100 category $8 lib $8 loc $32;
    infile "&basedir.test/config.txt" truncover;
    input tab $100.;
    if upcase(tab)="<LIBRARY>" then category="lib";
    else if upcase(tab)="<STOPLIGHT>" then category="stop";
    else do;
        if category="lib" then do;
           lib=scan(tab,1,":");
           loc=scan(tab,2,":");
           call execute('%recursdir(dir=' || strip(lib) || ', loc=' || strip(loc)
                 || ');');
           end;
        else if category="stop" then do;
           if strip(lowcase(scan(tab,1,":")))="lev1"
                 then call symput("lev1",strip(scan(tab,2,":")));
           if strip(lowcase(scan(tab,1,":")))="lev2"
                 then call symput("lev2",strip(scan(tab,2,":")));
           end;
        end;
    retain category;
run;
```

The above code does not validate entries in the configuration table, which is critical in production code to ensure unnecessary failures do not occur. However, it does import dynamic values LEV1 and LEV2 that can be utilized in stoplight reporting to vary the color scheme, as depicted below. In revised code (not shown), PROC REPORT now can be utilized to create an HTML report saved to the logical location /folders/myfolders/reporting that the code creates, and which contains two values for color coding, implemented with macro values for LEV1 and LEV2. Most importantly, the SAS environment (i.e., folder structure) and report presentation (i.e., color scheme) are now dynamically controlled from outside the code, meaning the code can remain static while only the configuration file is modified.

But what if components of a configuration file are missing or erroneous or, possibly worse, the entire configuration file itself is missing? To be portable to a new environment, not only the SAS code but also all required files—such as configuration files—must be included. This can be accomplished by manually copying the file from one environment to another, by creating the file from scratch or, preferably, by creating the file inherently inside the code. For example, the following SAS code creates the above configuration file (if it is missing from the current logical path from which the code is executed) to ensure it is available and correct for program initialization.

```
%macro makeit();
%if &SYSSCP=WIN %then %do;
    %let path=%sysget(SAS_EXECFILEPATH);
    %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.txt))^=1 %then %do;
    data _null_;
          file "&path/config.txt" termstr=CRLF;
          put "<LIBRARY>";
          put "reports: &path/reporting";
          put "<STOPLIGHT>";
          put "lev1: green";
          put "lev2: very light red";
    %end;
run;
%mend;

%makeit;
```

In the above example, the path of the program is dynamically assessed (in Windows and UNIX environments) and, if a configuration file does not already exist in this folder, it is created. Thus, as the code is transferred to a new environment or location, the configuration file will be recreated. Moreover, the dynamic path is included in the REPORTS line of the configuration file, thus when the RECURSDIR macro later is run based on this configuration file, the Reporting folder will be created inside the current folder. Because the above SAS program was executed from the folder Test, the subsequently created configuration file references Reporting under the Test folder.

```
<LIBRARY>
reports: /folders/myfolders/test/reporting
<STOPLIGHT>
lev1: green
lev2: very light red
```

Configuration files are an essential component to creating dynamic code that flexibly adapts to both its environment and user customization, while allowing the code substrate itself to remain intact and unchanged. Rather than passing dynamic attributes through macro parameters as a macro is invoked, those dynamic attributes are read from an external file, hopefully validated in some manner, and interpreted by the SAS process to alter program control.

INITIALIZING CONTROL TABLES

Control tables operate similarly to configuration files in that through data-driven processing, each can alter program flow or program attributes based on values in the table or file. Their similar externality to code is also critical, because this facilitates static code that can flexibly operate and adapt based on dynamic, external injects. A key difference, however, is that while 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 be more interactive with processes as they are executing, in that data are retrieved from control tables just as often as they are written to control tables. Thus, control tables often are used to record process performance metrics, especially those metrics that in turn drive further actions in other programs or processes.

Especially in SAS programming, control tables enable two or more concurrent processes to speak to each other, essentially by taking turns accessing the same location (a SAS data set) and by leaving messages for each other. The following code excerpt from a separate text by the author creates a control a table (if it does not exist) that allows code in concurrent SAS sessions to coordinate with each other[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 control table Control.etl is initialized, ensuring that when ported to a new environment, the code will not fail because the control table does not exist. To ensure this initialization occurs, preceding code should first test for the existence of the CONTROL library and, if it does not exist, create and assign it. Once created, the control table is accessed for seconds only at a time by DATA steps that read its contents and possibly leave behind additional information before closing the table, thus allowing other processes subsequently to access the table to communicate. And, despite the few split-seconds that 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 often are corralled in separate SAS libraries to help ensure that they are only accessed automatically by SAS processes and not directly by users, this further validation of the control table library should be a required step in production code. 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 code—especially code 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 either do not exist), the portability of this tool is dramatically 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 will automatically re-spawn when the code executes again.

CONCLUSION

Portability of code 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 complex modifications or refactoring required. 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, macro-driven code can facilitate ease of installation as well as further customization, but a best practice is to include dynamic elements in 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 already 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