SCSUG 2015 – Paper CE05

Ushering SAS® Emergency Medicine into the 21st Century: Toward Exception Handling Objectives, Actions, Outcomes, and Comms

Troy Martin Hughes

ABSTRACT

Emergency medicine comprises a continuum of care that often commences with first aid, basic life support (BLS), or advanced life support (ALS). First responders, including firefighters, emergency medical technicians (EMTs), and paramedics, are often the first to triage the sick, injured, and ailing, rapidly assessing the situation, providing curative and/or palliative care, and transporting patients to medical facilities. Emergency medical services (EMS) treatment protocols and standard operating procedures (SOPs) ensure that, despite the singular nature of every patient as well as potential complications, trained personnel have an array of tools and techniques to provide varying degrees of care in a standardized, repeatable, and responsible manner. Just as EMS providers must assess patients to prescribe an effective course of action, software, too, should identify and assess process deviation or failure, and similarly prescribe its commensurate course of action. *Exception handling* describes both the identification and resolution of adverse, unexpected, or untimely events that can occur during software execution, and should be implemented in SAS® software that demands reliability and robustness. The goal of exception handling is always to reroute process control back to the "happy trail" or "happy path"—i.e., the originally intended process path that delivers full business value. When insurmountable events do occur, however, exception handling routines should instruct the process, program, or session to terminate gracefully to avoid damage or other untoward effects. Between the opposing outcomes of a fully recovered program and graceful program termination, however, lie several other exception resolution paths that can deliver full or partial business value, sometimes only with a slight delay. To that end, this text demonstrates these paths and discusses various internal and external modalities for communicating exceptions to SAS users, developers, and other stakeholders.

INTRODUCTION

Treatment protocols for EMS providers vary by jurisdiction but uniformly serve to ensure that patients receive rapid, reliable care that meets or exceeds national, state, and local standards. Before an ambulance ever arrives on scene, however, untold hours and decades of applied medical research have been invested into composition and testing of protocols, as well as training of first responders to ensure they understand and strictly follow these protocols. Moreover, due to the advancement of medical research and EMS equipment, as well evolving legal authorizations, EMS protocols and standards represent living, breathing entities that will continue to improve and advance the standard of pre-hospital care.

Exception handling routines—like EMS providers—rely on complex business rules to prescribe courses of action to ensure functioning not only under *normal* conditions, but also under *exceptional* or extraordinary conditions. For example, if SAS code is executing and suddenly runs out of memory, what should happen to the process? What should happen to dependent processes that are expecting and require the first process to complete? And, is the failure immediately communicated to users and developers, or do they have to painstakingly parse through a blood-stained log following the exsanguination of their program? Thus, "SAS emergency medicine" describes the rapid triage, assessment, and road to restoration that begins when SAS code hits a bump…or a jersey wall head on.

Under ideal circumstances, exception handling occurs natively without users even being aware that anything unusual has occurred. For example, SAS code might dynamically assess it is executing in a UNIX environment and perform certain functions that it would not within a Windows environment. While portability to a different operating system (OS) might not seem very *exceptional*, certain SAS statements and functions are OS-specific and must be dynamically coded. A large body of exception handling thus represents conditional logic statements that flexibly allow code to follow different paths back to the happy trail and toward the same eventual outcome. Other exception handling paradigms,

however, are less straightforward and represent convoluted pathways through complex logic that may deliver varying degrees of ideal or not-so-ideal software functionality and performance.

EMS protocols and SOPs, in prescribing courses of action, can vary the level of patient care based on the severity of illness or injury. A sprained ankle might be treated with a lower level of care (like an ice pack) than an obviously broken ankle requiring transport to a hospital. However, because the underlying etiology often cannot be determined during a rapid trauma or medical assessment, care sometimes can be reflective of symptomology alone rather than cause. Thus, if a patient can't walk on or even put pressure on an ankle without agonizing pain, he'll be transported to a hospital regardless of whether the ankle is sprained, broken, or suffering from some other malady. Software exception handling is no different, in that while some causes can be easily diagnosed, in other cases, an unknown error occurs and developers must instead follow business rules that identify and treat the symptoms. For example, a process may have failed due to memory limitations or to a lost network connection but, if the undesirable outcome is similar, the same exception handling routine may handle these very different events.

The ultimate goal of exception handling is to maintain or restore functionality and performance and thus deliver the intended business value. In many failure patterns, however, only partial functionality (e.g., only creating four out of five anticipated HTML reports) or partial performance (e.g., a runtime that is lengthened by 45 minutes) can be delivered or worse, the program must be terminated because no output or outcome will be of value. The following exception handling paths and outcomes vary in technique as well as degree of business value they provide.

1. **Undetected Success –** The exception is detected by the software but not the user, reflecting that execution continues unobstructed without detriment to functionality or performance. This is the most desirable outcome.

2. **Rerouted Success** – The software handles the exception by dynamically redirecting the process flow to a different path that eventually produces the desirable outcome. Functionality is undiminished, but performance may be compromised if the rerouted path is significantly slower.

3. **Reattempted Success** – The software detects the exception, waits a specified amount of time, and reattempts the initial process. This path often occurs in a busy-waiting loop that continues to reattempt the process for either a specified duration or a specified number of attempts but, because success cannot be guaranteed, this path must additionally include a secondary exception path in the event that reattempts fail. Thus, functionality may or may not be reduced, but performance is likely diminished due to wait times that are introduced.

4. **Partial Success** – The exception may or may not be detected by the user during execution, but some degree of functionality is by definition compromised. In data analytic development, this often occurs when injects (i.e., values, observations, data sets, or files) must be deleted, imputed, or reverted back to default values because they are missing, corrupt, or otherwise invalid. Although the software completes and delivers some business value, developers must often later modify the injects, software, or both to facilitate future, fuller functionality.

5. **Process Termination –** The process is skipped (if a priori requirements are not met) or is terminated prematurely (if post hoc failure is detected), but program control is maintained which shifts immediately to other processes that are independent of the failed process. Loss of functionality is proportionate to the failed process.

6. **Program Termination –** The entire program is not executed (if a priori requirements are not met) or is terminated prematurely (if post hoc failure is detected) but, unlike process termination, program control shifts nowhere except to graceful program termination, which implies no negative affect to the execution environment as well as prompt communication of the failure to stakeholders. All functionality is lost at the point of failure, but some functionality may have been delivered before the catastrophic failure.

The power of exception handling lies in its ability to alter program flow dynamically either to prevent failed states and deliver full or partial functionality, or to allow code to fail gracefully. Despite the previous array of exception handling tools and techniques, SAS literature too often depicts exception handling routines that only terminate the program while providing "exception reports" that do communicate invalid data or runtime errors but which fail to fuel further data-driven processing. In fact, SAS exception handling implementation is often absent or so archaic that it mimics medieval medical practices. SAS code failing because a data set is locked? *Terminate the program and notate in the log*. SAS code failing because it's running in UNIX? *Terminate the program and notate the log*. SAS code failing because invalid

data are ingested? *Terminate the program and notate the log*. Ankle swollen from a sprain? *Put leaches on it*. Ankle swollen from deep vein thrombosis (DVT)? *Put leaches on it*. Ankle swollen from a compound fracture and tibial protrusion? *Amputate…and put leaches on it!*

Because this resemblance is both striking and horrifying, this text seeks to demonstrate the broader panoply of exception handling pathways that can recover full or partial functionality and performance, ushering SAS emergency medicine into the 21st Century. It aims to supplant the outmoded practice of log parsing to validate program success with software development best practices that can facilitate reliable, robust execution. For an introduction to SAS exception handling terminology and techniques, a separate text by the author is recommended: *Why Aren't Exception Handling Routines Routine? Toward Reliably Robust Code through Increased Quality Standards in Base SAS*[i].

I. UNDETECTED SUCCESS

When the fentanyl and hydromorphone start dripping en route to the hospital, that's usually about the time the skateboarder with the compound tib/fib fracture begins to relax and joke about his gnarled leg. His body is no less injured, but its suppressed pain response eases suffering while allowing paramedics to better provide care. He probably won't notice—or remember—many details about the ride, but hopefully a friend is capturing the whole hilarious ordeal on video. While software exceptions are no laughing matter, exception handling routines should also aim to be both painless and unnoticeable. Thus, "undetected" paths reflect that while the exception was detected by software, it was handled in a manner undetectable to the user which degraded neither functionality nor performance. Thus, like those of covert operatives, the failures of exception handling routines will often be public and messy, while their successes will be largely private and unknown.

Because variability is the nemesis of reliability and reliable software, an intent of exception handling is to make software more robust to dynamic injects, environments, states, and other events. Before software design or development can begin, however, business rules should model how software will respond to specific anticipated exceptions, as well as unanticipated ones. Only by incorporating exception handling into the entire software development life cycle (SDLC) can it be seamlessly integrated throughout business logic. Thus, before even the cornerstone SAS semicolon is laid, business requirements should describe the quality demanded of software, and should prescribe the nature and degree of performance requirements that will be facilitated in part through an exception handling framework.

The vast majority of SAS code operates equivalently in both Windows and UNIX environments; however, some statements, procedures, functions, call routines, system options, and system variables do differ between OSes. Some of these subtleties are found in the SAS® 9.4 Companion for UNIX Environments, Fifth Edition.[ii] To give one example, the SLEEP function suspends execution of a SAS program for a specified period of time, but only operates within a Windows environment. To gain equivalent functionality in a UNIX environment, the CALL SLEEP function must be invoked. The SAS automatic macro variable SYSSCP is set to the runtime OS, allowing OS-specific code to execute dynamically. The following macro causes the processor to wait 10 seconds, and functions seamlessly in both Windows and UNIX environments.

```
%macro test();
%if &SYSSCP=WIN %then %let sleeping=%sysfunc(sleep(10));
%else %if %sysfunc(substr(&SYSSCP,1,3))=LIN %then %do;
    data _null_;
        call sleep(10,1);
    run;
    %end;
%else %put System is neither Windows nor UNIX;
%mend;

%test;
```

In this example, the user is blissfully unaware that an exception—recognition of a different operating system—has occurred, and the code reliably functions in Windows or UNIX. However, the previous code handles only these two OSes, while several others are recognized by SAS. Thus, if even more portable code is required, additional systems

such as HP, AIX, or SUN could be incorporated into the business rules. The SAS Macro Language Reference lists all operating environments and corresponding SYSSCP values.[iii] As this example demonstrates, while delivering higher performance, exception handling comes at the cost of increased code length and complexity, thus it should be implemented only to meet specific business objectives and software requirements.

A host of other portability considerations that can affect SAS environments and software are described in *Portable SAS: Language and Platform Considerations*[iv]. While software portability should be designed into robust SAS software intended for cross-platform use, other non-environmental exceptions can occur when software fails to respond flexibly to dynamic injects and states. As one example, conditional logic statements routinely parse categorical data and standardize acronyms, abbreviations, synonyms, spelling errors, and other variations that can obfuscate data analysis. The following code fragment bins vegetable names, accounting for the exception that occurs when spelling deviations are encountered. Without the following logic, subsequent analyses might underrepresent artichokes.

```
data perm.choke;
    set temp_choke;
    if veg in('artichoke','articoke','choke') then veg_clean='artichoke';
    else if veg in('spinach','nach') then veg_clean='spinach';
    else veg_clean='UNK';
run;
```

Rather than choking, this code accounts for exceptional artichoke values seamlessly and the data are cleaned without user awareness. While users should likely be aware of the underlying business rules that prescribe how data standardization occurs, they should not have to perform these rudimentary tasks manually. The ELSE statement additionally functions to alert users that values outside the model were observed, which might reduce business value as well as require model (and code) refinement to incorporate these values in the future. Thus, while the IF and ELSE IF lines represent the undetected exception path, the ELSE statement invokes a route that delivers some business value (i.e., demonstrating that an unknown value has been encountered), but fails to inform which vegetable, if any, was detected. In this way, exception handling routines—especially those that operate as data quality control gates—often grow over time to identify a wider array of divergent data and exceptions.

In addition to exceptional values, data sets and other files can cause exceptions or errors when they are missing, corrupt, or otherwise invalid. In a separate text, the author demonstrates how the initialization of SAS libraries, logical folders, control tables, and configuration files can be handled reliably behind the scenes without involving or alerting the user[v]. For example, given a program that requires a control table to operate, absence of this table would cause program failure. To remedy this situation, an exception handing routine can first check for the existence of the control table and, if it does not exist, create it dynamically. The following code creates the control table WORK.etl if it does not exist.

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

%build_control;
```

When exception handling is undetectable, users will be unaware when exceptions have occurred, but this is not to say the exceptions are hidden. In the first example, a user parsing the SAS log would have been able to identify which path—Windows or UNIX—of the SAS code executed. Similarly, in the second example, the user would have noticed that the WORK.etl data set had been created by the process. Thus, while operating in an undetected manner, the intent is to provide reliability and robustness rather than covert operation.

While an undetectable exception path is always the preferred outcome because business value is undiminished, this cannot always be achieved. In other cases, a delay in program completion could result, either from a rerouted or reattempted process. In more extreme examples, the process or program might need to be terminated, causing further diminished functionality and business value. The remaining sections describe other exception paths that can provide varying degrees of functionality and performance with varying degrees of user awareness.

II. REROUTED SUCCESS

EMS protocols and standards reflect the sometimes complex nature of successive steps that must be taken to provide care or save a life. When failure is not an option, and in the event that the first course of action is unsuccessful, a second course is sure to follow. For example, according to the American Heart Association, abdominal thrusts should be applied to an adult patient who is choking and, if successful, treatment often ends there. If the patient becomes unresponsive, however, cardiopulmonary resuscitation (CPR) is begun immediately[vi]. In this sense, the nature and severity of the patient will determine the ultimate course of actions that are taken but, however divergent these treatment paths may be, they will always follow the prescriptive business rules of EMS protocols.

In the era of big data, a painful reality is that SAS and other analytic software can run out of memory when executing even simple processes. Given the case in which the SORT procedure runs out of memory and fails, this runtime error can cascade and cause subsequent, dependent processes to fail as well. The following code, for example, could execute unreliably because it neither anticipates nor accounts for a memory failure. Not only would the user be faced with the initial memory error (during the SORT procedure), but moreover, the DATA step would also fail because the BY statement would produce an error the first time it encountered an observation that was not sorted.

```
data mydataissobig;
    length fname $20 lname $20 dob 8 servicedate 8;
run;

proc sort data=mydataissobig;
    by fname lname dob servicedate;
run;

data temp;
    set mydataissobig;
    by fname lname dob servicedate;
    * additional logic and transformations here;
run;
```

Several solutions exist to this problem, such as increasing memory on the SAS server, implementing an index on the data set, using divide-and-conquer distributed processing, or simply removing unnecessary fields or observations before the sort to reduce its memory consumption. The ultimate program objective as well as systems limitations should determine which methodology is implemented. A reliable method that the author demonstrates in a separate text invokes the SAFESORT macro that first attempts to sort a SAS data set using the SORT procedure and, if the procedure fails due to a memory (or other) runtime error, subsequently reroutes program flow to a less memory-intensive sorting procedure. This logic of implementing SAFESORT is simulated in the following code, while the logic of SAFESORT itself is detailed in the separate text[vii].

```
%macro safesort(dsn=, sortby=);
%global safesortrc;
proc sort data=&dsn;
    by &sortby;
run;
%if &syserr=0 %then %let safesortrc=;
%else %do;
    %let syscc=0;
        * logic that performs sort using less memory-intensive methods;
```

```
        * code found in referenced text;
    %if &syscc=0 %then %let safesortrc=;
    %else %let safesortrc=&syscc;
    %end;
%mend;

%macro test();
%safesort(dsn=mydataissobig, sortby=fname lname dob servicedate);
%if %length(&safesortrc)=0 %then %do;
    data temp;
        set mydataissobig;
        by fname lname dob servicedate;
        * additional logic and transformations here;
    run;
    %end;
%else %return;
%mend;

%test;
```

The code is now more robust because the SORT procedure has been wrapped in an exception handling framework that includes the SAFESORT macro. The SAS automatic macro variable &SYSCC (system current condition), which records the highest warning or error code encountered, is used to test the completion status of SAFESORT, which is passed to the global macro variable SAFESORTRC. The length of SAFESORTRC is assessed and, if zero, the DATA step confidently runs knowing that either the SORT procedure or SAFESORT macro were successful. And, even if the more reliable SAFESORT macro fails, a second exception path is invoked with the %RETURN function that terminates the SAS macro and process. Thus, because even the rerouted path cannot guarantee process success, an additional exception path is required to either terminate the process or program in the event that this is warranted. Additionally, because SAFESORTRC is a global macro variable, If the TEST macro were called as a child process from some other parent process or program, that parent process could utilize the value of SAFESORTRC to drive further dynamic processing.

III. REATTEMPTED SUCCESS

CPR is considered to be a single procedure that involves chest compressions and, in some EMS protocols, rescue breathing. However, CPR can also be viewed iteratively as comprising discrete cycles of compressions, breathing, defibrillation, and assessment of patient vitals. In cases of cardiac arrest, while CPR is performed during the entire pre-hospital transport in the hopes of restoring cardiac rhythm, it cannot be performed indefinitely thereafter. Thus, based on factors such as lack of a heartbeat, other vital signs, patient age, health, and nature of arrest onset—and in concert with hospital protocols—at some point CPR is ceased and the arresting patient expires.

If the first round of chest compressions is unsuccessful in restoring a heart rhythm, first responders don't give up, but immediately reattempt a second round. Some software exceptions occur similarly, in that an exception may have occurred the first time a process was attempted, but would not occur if the process were reattempted. Or, the process might fail repeatedly for the first 30 minutes but, following that substantial delay, eventually succeed and revive program execution. Despite representing only a small subset of potential exceptions, these temporal exceptions should be handled because their underlying processes can be automatically and repeatedly reattempted, sometimes still delivering full functionality and business value.

Temporal exceptions can be handled with at least two different recovery methods. The process can enter a busy-waiting (or spinlock) cycle in which SAS is temporarily put to sleep for a parameterized amount of time until the process is reattempted. Once in the busy-waiting cycle, and unless the process succeeds, business rules should prescribe how long to continue to reattempt the failed process until eventually giving up. The second method requires a flag (i.e., typically a SAS macro variable) that records the event of the failed process, after which program flow is passed to subsequent, independent processes. Periodically thereafter, the flag is queried and the failed process is reattempted.

If successful, the flag is lowered and, if continually unsuccessful, business rules again should prescribe when to stop reattempting. Thus, with both methods, a sentinel value—typically either the maximum elapsed time or maximum number of attempts—is required to prevent an infinite loop and deadlock.

A notable example of a temporal exception solved with the busy-waiting method is a file access collision that occurs when one user or process attempts to access a SAS data set already exclusively locked by another user or process. A straightforward solution to this problem is introduced by the author in another text that describes the LOCKITDOWN macro which tests repeatedly for data set availability and either receives permission to access the data set or times out and prevents dependent processes from executing and failing.[viii] The following invocation of LOCKITDOWN tests the existence and availability of the data set PROD.Thebigtamale and, only if it is unlocked within the first ten minutes (i.e., MAX=600 seconds), continues to perform the dependent DATA step. The return code LOCKERR represents a SAS global macro variable that is initialized inside the LOCKITDOWN macro and whose length will be at least one character if LOCKITDOWN failed to gain a file lock after attempting every second for ten minutes.

```
%macro test();
%LOCKITDOWN (lockfile=prod.thebigtamale, sec=1, max=600, type=W, canbemissing=Y)
%if %sysevalf(%length(&lockerr)>0) %then %return;
data prod.thebigtamale;
    set blah;
run;
&lockclr; * use described in LOCKITDOWN text;
%mend;

%test;
```

Even repeatedly reattempting a process, however, in no way guarantees its success. After running the previous code for 10 hours (i.e., MAX=36000), Thebigtamale might still be locked by a process. In other cases, performance requirements might prescribe that the program only wait a few minutes for a locked data set before conceding defeat. Thus, code that implements LOCKITDOWN, LOCKANDTRACK[ix], or other busy-waiting methodologies must also include an exit strategy in the event that the file lock still cannot be obtained. In the previous example, the first exception path is to reattempt the process for five minutes but, if functionality still has not been restored, the second exception path relies on the SAS macro %RETURN statement to terminate the macro TEST because the return code LOCKERR indicates that the necessary file lock could not be achieved.

The second method to address temporal exceptions is, rather than sitting around idly waiting, to go do some other task whose execution does not rely on success of the first. First and foremost, this requires subsequent, independent processes to which the program can shift control through fuzzy logic or other business rules. Moreover, a flag is required to indicate that a process has failed and, finally, one or periodic checkpoints must exist at which point the flag is reevaluated and the original failed process is reattempted. In the following example, if the macro DOSTUFF fails because it cannot gain exclusive access to Somedata, the flag (macro variable RESIDUE) is set equal to the macro name, thus the macro DOSTUFF is reattempted later after other processes have completed. And, in this example, business rules prescribe if DOSTUFF fails a second time, a second exception path is initiated and the process is terminated with %RETURN statement.

```
%macro dostuff();
%global residue;
%let residue=;
data somedata;
    set mydataissobig;
run;
%if %sysevalf(&syserr>0) %then %let residue=dostuff;
%mend;

%macro dootherstuff;
%mend;
```

```
%macro control();
%dostuff;
%dootherstuff; * a macro not dependent on success of DOSTUFF macro;
%if %sysevalf(%length(&residue)>0) %then %do;
    %&residue;
    %end;
%if %sysevalf(%length(&residue)>0) %then %return;
%put Yay it worked!;
%mend;

%control;
```

In this example, the macro variable RESIDUE is a toggle that represents the status of the completion of the DOSTUFF macro. However, flags can also be space- or comma-delimited lists that contain numerous toggle values, or they can be updated with performance metrics that track reattempts or error codes. For example, logic might instead prescribe that RESIDUE contain a space-delimited list of all macros that have not yet been completed. Similar to the busy-waiting method, because there is no guarantee that a process will succeed even after numerous reattempts, a second exception path should exist to ensure that a deadlock is not created.

IV. PARTIAL SUCCESS

Sometimes in emergency medicine, while the patient may be saved, lasting disfigurement, disability, or ailments may persist. This occurs in the stroke patient who never regains use of his left arm, or the auto accident victim who is pulled from the wreckage but who loses an index finger. Software can encounter similar circumstances in which some segment of business value is lost while the remainder can be saved. Common patterns occur when missing, corrupt, or otherwise invalid values, observations, data sets, or files are encountered. For example, a SAS program that ingests Oracle tables over a network connection might ingest 15 of 17 correctly but have unspecified issues with the remaining two tables. If business rules dictate that subsequent processing should continue on those 15 data sets then, while business value and functionality are reduced, they are not eliminated. Thus, losing a digit isn't the end of the world, especially if the hand can be saved.

As a data analytic development application, Base SAS is much more likely to rely on data injects than other inputs. Thus, in robust infrastructures, quality control constraints should be applied to values, observations, data sets, and other ingested files. Where possible, business rules should be encoded to enforce the structure, format, content, completeness, quantity, and other aspects of quality for all dynamic injects into a robust system. Moreover, business rules additionally should prescribe how deviations from the expected norms will be handled, with common outcomes including data deletion, imputation, or modification to a default value. Thus, when encountering a missing value, exception handling routines could delete, impute, or modify its value—or delete, impute, or modify the entire observation—or delete, impute, or modify the entire data set, and so forth.

A missing value, observation, data set, or file typically imparts no value or functionality. When values are missing or invalid, statistical imputation methods can replace the value with a likely one. SAS literature is replete with different methodologies for imputation and, while not demonstrated in this text, these techniques can be woven into exception handling routines to ensure that maximum utility is garnered from data. In other cases, rather than dynamically or statistically imputing a new value, for simplicity, the value is deleted or set to a missing or unknown value. Similar to the previous artichoke example, the following code validates the categorical variable Food against a list of acceptable values. If Food is invalid, it is set to missing using conditional logic.

```
data food;
    length food $20;
    food="burritto"; *represents invalid data due to misspelling;
    output;
    food="chimichanga";
    output;
run;
```

```
%let validlist="burrito","enchilada","taco","chimichanga";
data final_food;
    set food;
    if lowcase(food) not in (&validlist) then food="";
run;
```

If business rules instead dictate that a missing or invalid value necessitates deletion of the entire observation, the following modified DATA step can be implemented.

```
data final_food;
    set food;
    if lowcase(food) not in (&validlist) then delete;
run;
```

Thus, very simple conditional logic that prescribes quality control techniques can often be considered a form of exception handling. While these examples demonstrate the business rules in the code itself, in some instances, this logic can be extrapolated to a hash object, lookup table, or an external control table to accomplish the same result. In a separate text, the author demonstrates external control tables that drive conditional logic to validate and standardize categorical values.[x]

In some cases, not just an observation but the entire data set is missing or corrupt. This often spells disaster and the process or program must terminate. However, as in the previous example in which business value can still be conferred by 15 out of the 17 total Oracle tables, processing can continue so long as no statements or processes are executed subsequently that are dependent on the invalid or missing tables. Consider the following initial example in which no exception handling routines are applied. Three data sets Perm_One, Perm_Two, and Perm_Three are to be created from three temporary data sets One, Two, and Three, respectively. Yet, an exception is encountered because the Two data set is missing, a common exception that should have been predicted and handled in more robust code.

```
data One;
    x=1;
run;
data Three;
    x=3;
run;

%macro wrapper(filelist=);
%let i=1;
%do %while(%length(%scan(&filelist,&i))>1);
    %let fil=%scan(&filelist,&i);
    data Perm_&fil;
        set &fil;
    run;
    %let i=%eval(&i+1);
    %end;
%mend;

%wrapper(filelist=One Two Three);
```

When executed, because the Two data set does not exist, the following log and error are produced.

```
NOTE: There were 1 observations read from the data set WORK.ONE.
NOTE: The data set WORK.PERM_ONE has 1 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.02 seconds
```

```
ERROR: File WORK.TWO.DATA does not exist.

NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.PERM_TWO may be incomplete.  When this step was stopped
there were 0 observations and 0 variables.
WARNING: Data set WORK.PERM_TWO was not replaced because this step was stopped.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

 NOTE: There were 1 observations read from the data set WORK.THREE.
 NOTE: The data set WORK.PERM_THREE has 1 observations and 1 variables.
 NOTE: DATA statement used (Total process time):
       real time           0.01 seconds
       cpu time            0.01 seconds
```

Moreover, the log indicates that despite this error, a dummy data set Perm_Two was still created, although it contains 0 observations and 0 variables. In some situations, this could cause subsequent, dependent processes to fail because they might attempt to utilize Perm_Two, believing it to be a valid data set. Thus, if the previous error is produced, the Perm data sets would most likely need to be deleted before reattempting to execute the program with improvements that included additional exception handling to account for missing data sets.

The following improvements now predict that under exceptional circumstances a data set could be missing, thus the process—the second iteration of the DATA step—is skipped by a priori exception handling if prerequisite requirements are not met.

```
data one;
run;
data three;
run;

%macro testdata(lib=, dsn=);
    %global err;
    %let err=library or data set does not exist;
    %if (%sysfunc(libref(&lib))=0 and
          %sysfunc(exist(&lib..&dsn))^=0) %then %let err=;
%mend;
%macro newwrapper(filelist=);
%let i=1;
%do %while(%length(%scan(&filelist,&i))>1);
    %let fil=%scan(&filelist,&i);
    %testdata (lib=work, dsn=&fil);
    %if %length(&err)=0 %then %do;
          data Perm_&fil;
              set &fil;
          run;
          %end;
    %let i=%eval(&i+1);
    %end;
%mend;

%newwrapper(filelist=);
```

In Section One, an example demonstrated the creation of a control table from scratch if the file was missing. This is not only seamless but also an ideal way to initialize control tables when they are first utilized (e.g., perhaps in a new

environment) and, because the file never existed before, no business value is lost. However, if a file becomes corrupted either through invalid user entry or process failure and needs to be deleted, this represents a loss of functionality, especially where process metrics or user customizations had been recorded and are now lost. A goal thus becomes to attempt to salvage these files automatically if some (but not all) of their components are invalid.

The following configuration file represents the text file config.txt designed to alter the values (LEV1 and LEV2) dynamically in a SAS HTML stoplight report (not shown). For example, the value LEV1 (i.e., green) will be ingested and instantiate the macro variable LEV1 so that it can dynamically color the report.

```
<STOPLIGHT>
lev1: green
lev2: very light red
<OTHERSTUFF>
blah blah blah
```

Config.txt is ingested with the following code and, if the values for LEV1 or LEV2 cannot be validated, they are overwritten with acceptable default values (i.e., green and red). User error could also have omitted (or misspelled) the STOPLIGHT header, in which case LEV1 and LEV2 would not have been assigned values but could be assigned default values. Thus, business value may be slightly diminished (because the user-defined values are lost) but the program nevertheless continues undaunted and produces the stoplight reporting.

```
%global lev1;
%global lev2;
data _null_;
    length tab $100 category $8 lib $8 loc $32 lev1 $20 lev2 $20;
    infile "&basedir.test/config.txt" truncover; /* &basedir must be set first */
    input tab $100.;
    if _n_=1 then do;
        lev1="";
        lev2="";
        end;
    if upcase(tab)="<STOPLIGHT>" then category="stop";
    else do;
        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;

%macro validate();
%let colorwheel=green, orange, red, very light red;
%let i=1;
%let lev1valid=0;
%let lev2valid=0;
%do %while (%length(%scan(&colorwheel,&i))>1);
    %if &lev1=%scan(&colorwheel,&i) %then %let lev1valid=1;
    %if &lev2=%scan(&colorwheel,&i) %then %let lev2valid=1;
    %let i=%eval(&i+1);
    %end;
%if &lev1valid=0 %then %let lev1=green; * sets to default if invalid or missing;
%if &lev2valid=0 %then %let lev2=red; * sets to default if invalid or missing;
%mend;
```

```
%validate;
```

In all cases, assessment of business rules followed by careful examination of program flow can elucidate process paths that can continue, even after software has encountered grave exceptions. Understanding of process prerequisites, requirements, and dependencies is key to designing and implementing models that can allow software to recover full or partial functionality. Notwithstanding, in some instances, a missing or invalid file or field or some other exception does necessitate that the process or program be terminated. Thus, the final two sections reflect methods that can facilitate graceful process and program termination.

V. PROCESS TERMINATION

Sometimes when all hope is lost for one process, this doesn't mean that hope in general is lost. When confronted with a patient presenting uncontrolled bleeding, the common mnemonic DEPT instructs that direct pressure, elevation, pressure points, and tourniquet be applied to stop bleeding. Thus, when transporting a trauma patient with severe brachial hemorrhaging, under many protocols, a tourniquet is quickly applied. The injury is not ameliorated, but the temporary solution allows EMS providers to redirect attention to assessment and treatment of other injuries. Thus, the tourniquet represents a temporary stopgap until additional emergent care can be provided at the hospital.

Process termination reflects that some module—one or a collection of DATA steps, procedures, or macros—must terminate prematurely and cannot complete, thus eliminating its business value. It also reflects that program flow continues to another independent process after the exception is encountered. Thus, the exception handling framework automatically places a tourniquet on the exceptional module, allowing focus and functionality to continue elsewhere. In procedural languages like SAS, process delimiters can be vague, undefined, or nonexistent but, to the extent that code can be organized and modularized by function or other attribute, modularity can be increased, thus helping isolate damage caused by exceptions and errors. This is akin to applying a tourniquet close to the wound to prevent unnecessary tissue damage elsewhere.

In Section Four, examples demonstrated that processing can often continue if a value, observation, data set, or file is missing, corrupt, or otherwise invalid. In other cases, however, a missing data set might require process termination. The exception can be handled through a priori methods that test prerequisite conditions for the process, in essence deducing whether it will succeed, or through post hoc methods that test error codes, return codes, or process outcome requirements after the process has completed. These methods are also differentiated in literature in that a priori testing is equivalent to "error proofing," whereas post hoc testing represents "error checking.[xi]" Often, a combination of both methods is required to produce software that both flexibly avoids and responds to exceptions.

The following example introduces a priori conditional logic that bypasses the macro EXITEARLY if the data set Churro does not exist, allowing program flow to continue to the NICEDAY macro. Functionality is diminished because Churro cannot be sorted, but at least the user receives a pleasant message in the log that brightens his day, since the NICEDAY macro can still complete.

```
%macro exitearly();
%if %sysfunc(exist(work.churro))^=0 %then %do; * if data set exists;
    proc sort data=churro;
        by cinnamon sugar;
    run;
    %end;
%else %put Something aint right!;
%mend;

%macro niceday();
%put have a nice day!;
%mend;

%exitearly;
%niceday;
```

This functionality could also be accomplished through use of the SAS macro %GOTO statement.

```
%macro exitearly();
%if %sysfunc(exist(work.churro))=0 %then %goto exit;
    proc sort data=churro;
        by cinnamon sugar;
    run;
%exit: %put Something aint right!; * incorrect logic here;
%mend;
```

Both examples produce a warning message to the log that "something ain't right" but, in fact, the second code itself ain't right! Because the SAS label EXIT lies in the path of the program flow—directly following the SORT procedure—it executes regardless of the value of the EXIST function, thus printing the warning message whether the SORT procedure is executed or not. For this reason, while %GOTO statements can be useful if not necessary in a procedural language like SAS, the assignment of process metrics (such as specific return codes or log messages) should typically be handled immediately after the exception is detected rather than in the subsequent label statement. The following code now correctly assigns a brief description of the exception—if the data set is missing, contains no observations, or the sort variable is missing—to the global macro variable AINTRIGHT, which can be parsed in subsequent processes.

```
data nochurro;
run;

data churro;
    length cinnamon $10;
    cinnamon="numnumnum";
    output;
    cinnamon="so sweet";
    output;
run;

%macro exitearly(dsn=, sortby=);
%global aintright;
%let aintright=;
%if %sysfunc(exist(&dsn))=0 %then %do;
    %let aintright=Missing data set;
    %goto exit;
    %end;
%let dsid=%sysfunc(open(&dsn, i));
%let varnum=%sysfunc(varnum(&dsid, &sortby));
%let num=%sysfunc(attrn(&dsid, nvars));
%let buhbye=%sysfunc(close(&dsid));
%if %eval(&num=0) %then %do;
    %let aintright=No Observations;
    %goto exit;
    %end;
%if %eval(&varnum=0) %then %do;
    %let aintright=Sort variable missing;
    %goto exit;
    %end;
proc sort data=&dsn;
    by &sortby;
run;
%exit:;
%mend;
```

```
%exitearly(dsn=churro, sortby=cinnamon);
%put RC: &aintright;
```

When executed, this code correctly sorts the Churro data set and produces an empty return code, the global macro AINTRIGHT. To be clear, this return code does not signify process success, but rather demonstrates that all listed a priori process prerequisites were met and that the process execution started.

```
NOTE: There were 2 observations read from the data set WORK.CHURRO.
NOTE: The data set WORK.CHURRO has 2 observations and 1 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time           0.01 seconds
      cpu time            0.02 seconds

%put RC: &aintright;
RC:
```

When the EXITEARLY macro is executed with the Churro data set but a misspelled sort variable CINNAMONNN, the macro correctly detects the sort variable CINNAMONNN is missing and exits with the appropriate return code.

```
%exitearly(dsn=churro, sortby=cinnamonnn);
%put RC: &aintright;
RC: Sort variable missing
```

And, when executed with the Nochurro data set, the macro correctly detects that no observations exist, and exits with the appropriate return code.

```
%exitearly(dsn=nochurro, sortby=cinnamon);
%put RC: &aintright;
RC: No Observations
```

Finally, when executed with a spurious data set Spurious that does not exist, the macro correctly detects that the data set is missing and exits with the appropriate return code.

```
%exitearly(dsn=spurious, sortby=cinnamon);
%put RC: &aintright;
RC: Missing data set
```

The previous examples demonstrate some of the exceptions that can be predicted if something is awry within the data set. In an actual production system, the AINTRIGHT return code would be used thereafter to alter program flow dynamically rather than being printed to the log as an exception report. But, thinking outside the box, what if something is wrong with the macro invocation itself? For example, if two tokens—CINNAMON and BUN—are supplied in the SORTBY parameter rather than just one, the code ultimately should fail because BUN is not in the data set. However, the code fails sooner because the VARNUM function is provided two variables but expects only one.

```
WARNING: Argument 2 to function VARNUM referenced by the %SYSFUNC or %QSYSFUNC
macro function is out of range.
NOTE: Mathematical operations could not be performed during %SYSFUNC function
execution. The result of the operations have been set
      to a missing value.
ERROR: Variable BUN not found.

NOTE: The SAS System stopped processing this step because of errors.
NOTE: PROCEDURE SORT used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

%put RC: &aintright;
```

```
RC:
```

This highlights the need to make macros that are as dynamic as possible, for example, accepting a space- or comma-delimited parameter for one (or more) variables, rather than assuming that the data set will only be sorted by a single variable. Exception handling routines must also especially guard against the possibility of unwanted variance when macro parameters are passed. The previous error also demonstrates that the return code AINTRIGHT, which subsequent processes might be using to determine if the macro succeeded, does not record this new, unhandled error. This highlights the need to include a final post hoc check of the automatic macro variable SYSCC to assess if *anything* went wrong in the program, at which point a generic error return code can be generated. Validation of macros and their parameters to facilitate robustness is discussed throughout SAS literature.

When the process does succeed—as in the first of the previous four examples—the empty return code should be used by other processes to demonstrate that predictable exceptions did not occur in this macro. However, because no post hoc testing ensured that the SORT procedure completed without error, the empty return code still does not demonstrate or guarantee process success. To accomplish this, an additional test of the SAS automatic macro variable SYSCC would need to occur *after* the SORT procedure. This allows the exception handling framework to account for exceptions arising from failures—from memory errors or other reasons—during the SORT procedure itself.

A benefit of a priori testing is that no error is produced; however, this also requires the identification of specific exceptions, so business rules should be established that specify minimum requirements to execute a process. Over time, developers may notice that they are reusing the same chunks of exception handling code throughout various processes and programs. This reuse is a good practice, as it allows for standardization of code but, to the extent possible, reusable code should be modularized into separate macros that can operate as functions. For example, in the previous scenario, because testing for a missing data set or a data set that has no observations is a common quality control, these two functions might be wrapped in a single macro that can be saved for posterity.

Post hoc exception handling involves the testing of SAS warning and error codes and messages that were generated by DATA steps, procedures, statements, or processes, as well as testing of return codes generated by a priori exception handling routines. Because an exception can occur unexpectedly at any moment, SAS software that demands high reliability should contain post hoc testing at the close of each boundary or process to convey whether it completed successfully or not. While post hoc testing is typically conceptualized to evaluate the *process* that completed, just as SAS users sometimes painfully parse the SAS log, it also can evaluate a *product* that was created. Thus, just as quality control routines were demonstrated in previous sections to evaluate the validity of data and other injects, similar quality control routines can evaluate data products for validity after a process has concluded. Thus, business rules should often specify both the prerequisite requirements that must be met to allow a process to initiate and a checklist that specifies what requirements must be met to signify process completion.

When a process must be terminated abruptly, cleanup of the environment may sometimes be required, although typically this cleanup can be minimized or eliminated—as well as automated—through development best practices. Cleanup can include such operations as deleting folders or data sets that were created, or reverting a control table back to pre-failure (or pre-exception) conditions. The goal of cleanup is to ensure that the environment was not harmed by the exception, as well as to ensure that when the software is executed again, it will not fail due to some lingering consequence of the exception. In a separate text, the author discusses best practices for developing SAS code that supports efficient recoverability and which does not require environment cleanup[xii]. Consider the following code in which the data set Churro is created and sorted inside the macro ETL, which simulates a much more complex ETL process.

```
%macro etl();
data churro;
    length cinnamon $10;
    cinnamon="numnumnum";
    output;
    cinnamon="so sweet";
    output;
run;
%if &syserr^=0 %then %return;
proc sort data=churro;
```

```
    by cinnamon;
run;
%if &syserr^=0 %then %return;
%put SUCCESS!;
%mend;

%etl;
```

The code includes no a priori exception handling routines, but does verify process success after both the DATA step and subsequent SORT procedure. And, in this example, the code executes perfectly.

```
NOTE: The data set WORK.CHURRO has 2 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.01 seconds

NOTE: There were 2 observations read from the data set WORK.CHURRO.
NOTE: The data set WORK.CHURRO has 2 observations and 1 variables.
NOTE: PROCEDURE SORT used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds

SUCCESS!
```

However, if the Churro data set is misspelled (Churrrro) in the SORT procedure, immediately after the procedure fails, the %RETURN macro function exits the macro, thus bypassing the proclamation of success.

```
NOTE: The data set WORK.CHURRO has 2 observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.01 seconds
      cpu time            0.00 seconds

ERROR: File WORK.CHURRRRO.DATA does not exist.

NOTE: The SAS System stopped processing this step because of errors.
NOTE: PROCEDURE SORT used (Total process time):
      real time           0.00 seconds
      cpu time            0.00 seconds.
```

In reality, an exception in a robust ETL infrastructure would more likely be caused by dynamic data injects rather than a misspelled data set name, but this error simulates the type of "leftovers" that failed SAS processes can create that can cripple or clog the environment. Subsequent procedures might expect Churro to be sorted, assess that the data set exists, and fail because it is not actually sorted. Thus, under certain business rules, one option would be to delete and recreate the Churro data set if the SORT procedure fails.

Also demonstrated in the previous example, the SAS macro %RETURN statement (and its equivalent non-macro RETURN statement) is commonly implemented to terminate processing and return functioning to a higher level. The %RETURN statement causes termination of the current macro[xiii], while the RETURN statement causes the DATA step to return functioning to the start of the DATA step or, if implemented, to the line after the LINK statement.[xiv] Because the %RETURN statement is easily recognizable and can be implemented effectively in a single line of code without the convolutions of %GOTO statements, it often is utilized in literature (and as in this example) to represent program termination within exception handling routines. However, because %RETURN exits the macro abruptly, return code assignment or environment cleanup must occur before its use.

Because the goal of exception handling is to maximize software functionality, performance, and value, modules unrelated to a failed process should usually continue to execute if they can continue to provide some business value.

As programs grow in complexity, it may become difficult to code business rules in a coherent or readable fashion, much less to organize rules that accurately reflect all requirements and dependencies of each. To further this goal, a separate text by the author demonstrates the use of control tables to drive fuzzy logic algorithms that automatically prescribe which processes can and should execute based on factors such as process priority or prerequisite process completion.[xv] Although outside the scope of this text, these methods provide for straightforward creation, tracking, modification, and implementation of complex business rules, facilitated through data-driven processing, thus ensuring the maximum amount of juice is squeezed from each SAS program, even when exceptions are encountered.

VI. PROGRAM TERMINATION

Not every patient can be saved. While no attempt is made to resuscitate a room temperature victim presenting rigor mortis, patient respect and protocols remain paramount. A sheet is pulled and various authorities are notified or summoned. Thus, even in death, EMS protocols specify final steps that must be taken. Program termination similarly differs from process termination only in that execution ceases when the exception is encountered. This usually represents the effort to fail gracefully when it becomes apparent that no further functionality or business value can be delivered. When prudent, the foregoing exception handling paths should have been considered before the decision to terminate the program. But, when termination is inevitable, attention should turn to ensuring that the failure will be graceful, not damage the execution environment, and will provide meaningful return codes and other diagnostics to aid developers in future investigation and resolution, especially where the exception was unpredicted or unknown.

The SAS system option ERRORABEND—counterpoint to the default system option NOERRORABEND—is the option most synonymous with program termination. SAS documentation states the following regarding its use: "Use the ERRORABEND system option with SAS production programs, which presumably should not encounter any errors ."[xvi]" The difficulty with implementing ERRORABEND in a single program is that because it abruptly terminates the program, any custom exception handling routines designed to guide the software to a graceful termination are never executed. Thus, checking for post hoc errors is pointless in an ERRORABEND environment because the SAS processor will detect the runtime error and terminate before SYSERR or SYSCC are read by exception handling routines.

The benefits of ERRORABEND, as detailed thoroughly in Denis Cogswell's text, are that in a production environment in which a controller program is spawning successive processes in other programs, the spawned programs will abort immediately following an error, but control will return to the controller.[xvii] Thomas Billings, a great proponent of infusing SAS software with exception handling, refers to use of ERRORABEND as "a high risk situation" when he describes exception handling implementation in both Enterprise Guide and Batch modes. Moreover, because significant differences exist in how the option functions in SAS Batch and other modes, he goes on to recommend using ERRORABEND only in Batch mode.[xviii] Thus, in some limited situations, the use of ERRORABEND is warranted; however, robust infrastructure in production environments can and should implement exception handling routines that far surmount the violent termination dictated by the ERRORABEND option.

The SAS macro statement %ABORT and its equivalent SAS statement ABORT are the statements most synonymous with program termination[xix]. Invoking %ABORT with optional arguments causes program termination and return to the SAS operating environment and, because %ABORT can be utilized in conditional logic statements, it can gracefully direct a program to terminate after identification of an exception. As a caution, and like the ERRORABEND option, both %ABORT and ABORT statements are environment-specific and thus can produce unintended and problematic results if not tested thoroughly in all intended SAS environments. Billings thoroughly summarizes the %ABORT arguments ABEND, CANCEL, and RETURN, and highlights the discrepancies that exist among these.[xx]

In the default (and generally recommended) NOERRORABEND environment, use of ABORT or %ABORT is required to stop a program because, despite encountering an error, the SAS processor will only terminate the offending DATA step, procedure, or macro, and will proceed to execute all subsequent processes. In Section Five, the EXITEARLY macro demonstrated the use of a priori exception handling to test for predicted exceptions (e.g., a missing data set, empty data set, or missing sort variable), but failed when an unpredicted exception—an extra token passed in the SORTBY parameter—occurred. Unpredicted exceptions are often best handled with use of %ABORT and, by modifying the EXITEARLY code by inserting a few lines between %EXIT and %MEND, this can be accomplished.

```
...additional code from above...
%exit:;
%if %eval(&syscc>0) %then %do;
    %let aintright=unanticipated exception;
    %abort abend;
    %end;
%mend;

%exitearly(dsn=churro, sortby=cinnamon bun);
%put RC: &aintright;
```

When executed, this code does produce an error but, because this exception is now adequately handled, it terminates the program before the %PUT statement, which in a realistic scenario would represent subsequent, dependent code that would have failed because of this error.

```
%exitearly(dsn=churro, sortby=cinnamon bun);
WARNING: Argument 2 to function VARNUM referenced by the %SYSFUNC or %QSYSFUNC
macro function is out of range.
NOTE: Mathematical operations could not be performed during %SYSFUNC function
execution. The result of the operations have been set
      to a missing value.
ERROR: Variable BUN not found.

NOTE: The SAS System stopped processing this step because of errors.
NOTE: PROCEDURE SORT used (Total process time):
      real time           0.01 seconds
      cpu time            0.02 seconds

ERROR: Execution terminated by an %ABORT statement.
```

Moreover, the revised code now creates a return code AINTRIGHT that accurately reflects both predicted and unpredicted exceptions, and which can be utilized in subsequent processes to drive program execution. And, if an unexpected error occurs that has not been uniquely handled, the program terminates immediately to allow developers to investigate the error. In the previous scenario, developers would quickly realize that robust code would additionally need to include exception handling routines that validate all parameters passed to the macro, and potentially also the capability to perform multi-variable sorts. Thus, as processes mature over time, they will predict and handle increasingly more types of exceptions that can occur. This continuous quality improvement (CQI) is a necessary component of any robust exception handling framework, and the reason it is a living, breathing entity.

COMMUNICATION

Exception handling routines require a mix of both internal and external communication to function. Internal communication includes warning and error codes that are generated by the SAS processor, SAS automatic macro variables, exception handling return codes, and, to a lesser extent, control tables or other external files that can foster communication between concurrent SAS sessions of SAS. Warning and error codes can be dramatically reduced by implementation of a priori testing described previously, but not always eliminated. They also can never guarantee that a process has succeeded, thus, if that assurance is required before passing program control to subsequent, dependent process, a post hoc test is also required.

The number and comprehensive nature of SAS automatic macro variables grow with each new SAS software release, allowing SAS software to be even more flexible and responsive to changing environments, system states, file states, and dynamic injects. Understanding these variables is invaluable to developing exception handling routines that can assess exceptional states. The authoritative list of SAS automatic macro variables is found at *SAS® 9.4 Macro Language: Reference, Fourth Edition* and its understanding and implementation are essential to developing reliable, robust software[xxi].

Control tables, configuration files, and other permanent files offer additional communication advantages in coordinating exception handling frameworks. While error codes and return codes can perish when a SAS process, program, or session is terminated, permanent files can save this information and transmit it across sessions. Thus, if the SAS server is disconnected suddenly, an exception handling framework will do little good because execution ceases when SAS is terminated. Even if SAS global macro variables had been tracking software performance, their values would be lost with the server crash. Tracking software performance—including both successes and failures—in a permanent location like a control table thus can offer a substantial advantage. These metrics can typically be accessed when the SAS server is restored and, in many cases, can improve performance during the recovery period as software begins to churn again. A second major benefit of using permanent files to record process and performance metrics is the ability for multiple users or programs to access those data simultaneously. Thus, a control table accessed by multiple SAS sessions effectively allows those sessions to concurrently speak to each other because they are able to pass messages.

Like the greater body of exception handling, return codes should also be used to drive dynamic functioning rather than simply print information to the log. In the final EXITEARLY macro example, the return code (AINTRIGHT) records the first (if any) exception encountered and can be used effectively in subsequent processes to assess completion status of this macro. To improve readability in this text, however, the only demonstrated subsequent use of AINTRIGHT was the %PUT statement that printed its value to the log; however, this statement is understood to simulate subsequent, dependent processes that would rely on AINTRIGHT to further drive program control. Thus, while SAS literature may often reflect return codes that are created only to be spat out to the log, this practice exists to provide less complex, more comprehensible examples. In production environments, return codes that are only printed to the SAS log are of little use and should instead be incorporated into data-driven processing routines.

The previous communication occurs—during both exceptional and normal processing—within SAS code and without the awareness of developers, users, or other stakeholders. While return codes and other process and performance metrics can be output to the log, this practice is best saved for development and test environments—not production. Thus, external communication encompasses information shared between software and stakeholders during and after program execution. External communication is typically unidirectional, conveying process and performance metrics to stakeholders, either in real time or after process or program completion or termination. In some cases, users are able to modify configuration files which in turn communicate data-driven "commands" to a SAS process, which can dynamically alter its execution in near-real-time. Most external communication, however, informs developers and users how or how well their code executed.

The log can be a strong ally to exception handling routines, as process and performance metrics can be passed. However, exceptions that can be predicted generally should not pass remarks to the log—after all, if an exception can be imagined, clearly defined exception paths should prescribe subsequent program control, so the log need not be parsed. Thus, in production systems, the only use of the log should be to drive CQI efforts because the log will have recorded events surrounding new, unpredicted exceptions that were either not handled or which were handled only through generic exception routines. For example, a primordial exception handling routine might detect only whether *any* error has occurred, but this routine when refined over time might later direct specific handling in the event that a *memory-related* error occurs. Thus, the log can be used to make exception handling routines more robust, specific, and meaningful. Several SAS texts demonstrate the benefits and ease of automated processes that save, parse, display, and archive logs for developers, and this automation of log activities should be considered the standard in any production environment.

Other advanced methods for communication of exceptions include automatically sending emails or even playing a tone or song, as demonstrated separately by David Fielding[xxii] and Ann Olmsted[xxiii]. As mentioned above, control tables also are an excellent method to archive process metrics (including warnings and errors) because they persist after the SAS session is terminated. Moreover, reports that can be automatically generated from control tables represent a proven method for tracking software functionality and performance. An added benefit of routines that print static reports from dynamic control tables is that this eliminates the possibility that a developer will accidentally modify or delete the control table, or that the table will be locked and prevent other processes from accessing it.

BUT DON'T INTRODUCE MORE EXCEPTIONS DURING EXCEPTION HANDLING!

Throughout this text, the complexity of business rules that prescribe exception handling routines has been demonstrated. Care must be taken to ensure that in this complexity, however, additional errors are not introduced. Testing represents a critical component of any exception handling framework and should accompany each new exception path as it is designed and developed. Smoke detectors are tested regularly to demonstrate they are functional and in fact providing their advertised and expected level of protection and, similarly, all exception handling routines must be demonstrated to be valid and effective if they are to be depended upon.

The following macro, reprised from the example in Section One, instructs SAS to sleep for ten seconds and functions in both Windows and UNIX environments—with one exception—the CALL SLEEP function has been misspelled this time.

```
%macro test();
%if &SYSSCP=WIN %then %let sleeping=%sysfunc(sleep(10));
%else %if %sysfunc(substr(&SYSSCP,1,3))=LIN %then %do;
    data _null_;
        call sleeeep(10,1); * oops somebody made a spelling error;
    run;
    %end;
%else %put System is neither Windows nor UNIX;
%mend;

%test;
```

The savvy SAS practitioner may have thought he was making his code more flexible and portable but, in releasing production code that contained this error, he succeeded only in demonstrating that his code had never been tested in a UNIX environment.

```
%test;
NOTE: Line generated by the invoked macro "TEST".
data _null_;     call sleeeep(10,1);   run;
                      _______
                      251
ERROR 251-185: The subroutine SLEEEEP is unknown, or cannot be accessed. Check your
spelling.
          Either it was not found in the path(s) of executable images, or there was
incorrect or missing subroutine descriptor information.

NOTE: The SAS System stopped processing this step because of errors.
NOTE: DATA statement used (Total process time):
       real time           0.04 seconds
       cpu time            0.04 seconds
```

This underscores the importance of testing every exception path that is developed to ensure that it does function as intended and does not accidentally introduce additional errors in the software. Because SAS data analytic development is data-centric, many of the complexities encountered in SAS that produce exceptions arise from dynamic injects, including variability observed in values, observations, data sets, and files. Thus, even when an exception path is tested, tried, and true, a slight variation in its throughput or injects can reveal logic errors or other code defects that had been hidden.

As one example, variations of the following code are found throughout SAS literature as a demonstration of how to determine if a data set has zero observations. And, typically this yields the correct observation count.

```
%macro empty(dsn=);
%global nothin;
%let nothin=;
proc sql noprint;
```

```
    select count(*) into :nobs from &dsn;
quit;
run;
%if %eval(&nobs=0) %then %let nothin=EMPTY;
%else %let nothin=FULL;
%mend;

data Somethinghere; * contains 1 variable;
    a=5;
run;

data Nothinghere; * contains no variables;
run;

%empty(dsn= Nothinghere);
%put NOTHIN: &nothin;
```

Given the existence of the data set Somethinghere which has one observation, this produces the correct output FULL, indicating that the data has at least one observation. But, when the EMPTY macro is executed to examine the data set Nothinghere (containing no observations or variables), the SQL procedure and subsequent statements crash unexpectedly.

```
%empty(dsn=Nothinghere);
ERROR: Table WORK.NOTHINGHERE doesn't have any columns. PROC SQL requires each of
its tables to have at least 1 column.
NOTE: PROC SQL set option NOEXEC and will continue to check the syntax of
statements.
NOTE: The SAS System stopped processing this step because of errors.
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.01 seconds
      cpu time            0.02 seconds

WARNING: Apparent symbolic reference NOBS not resolved.
WARNING: Apparent symbolic reference NOBS not resolved.
ERROR: A character operand was found in the %EVAL function or %IF condition where a
numeric operand is required. The condition was:
        &nobs=0
ERROR: %EVAL function has no expression to evaluate, or %IF statement has no
condition.
ERROR: The macro EMPTY will stop executing.
%put Nothin: &nothin;
Nothin:
```

Thus, not only did the SQL procedure fail but it caused the macro variable NOBS not be created, which in turn caused the subsequent macro %IF and %EVAL statements to fail. Failure begets failure and all because some SAS practitioner tried to do the right thing and implement exception handling routines to test process prerequisites—in this case, the common requirement that a data set contain at least one observation. This underscores the necessity that if exception handling routines are going to be implemented, they cannot be depended upon until all exception paths have been tested and additionally until all paths have been tested with the full spectrum of data variability they are intended to handle. Because the full spectrum of variation is likely unknown at project conception and throughout development, even high quality software invariably is released with defects that are overcome only through later discovery and resolution. Additional factors that should be taken into account when counting observations in SAS can be found in the seminal text *How Many Observations Are in My Data Set?*[xxiv]

CONCLUSION

Where reliable and robust code are warranted, exception handling routines can provide both functionality and peace of mind that code will not unexpectedly come to a tumultuous halt. Like emergency medical protocols, exception handling routines must follow established business rules that prescribe and sequence exception handling paths and responses. These routines often go undetected by users, facilitating functionality in dynamic environments that experience both predictable and unpredictable deviations from the norm. Moreover, in many circumstances, all or some intended business value can be achieved with exception handling implementation, possibly with only a delay in program completion. Finally, to further the recovery of future functionality and performance, exception handling should always capture new and previously undetected exceptions and errors in the hope that through CQI developers can incorporate these into future a priori and post hoc testing.

REFERENCES

[i] Hughes, Troy Martin. 2014. *Why Aren't Exception Handling Routines Routine? Toward Reliably Robust Code through Increased Quality Standards in Base SAS*. Midwest SAS Users Group (MWSUG).

[ii] SAS Press. *SAS® 9.4 Companion for UNIX Environments, Fifth Edition*. 2015. Retrieved from http://support.sas.com/documentation/cdl/en/hostunx/67929/PDF/default/hostunx.pdf.

[iii] SAS Press. *SAS® 9.4 Macro Language: Reference, Fourth Edition. SYSSCP and SYSSCPL Automatic Macro Variables*. Retrieved from http://support.sas.com/documentation/cdl/en/mcrolref/67912/HTML/default/viewer.htm#n0e7s8lf147kazn17u45trpwb6sw.htm.

[iv] Cruz, Robert A. 2009. *Portable SAS: Language and Platform Considerations*. Western Users of SAS Software (WUSS).

[v] Hughes, Troy Martin. 2015. *SAS Spontaneous Combustion: Securing Software Portability through Self-Extracting Code*. Western Users of SAS Software (WUSS).

[vi] American Heart Association. 2010. *American Heart Association Guidelines for Cardiopulmonary Resuscitation and Emergency Cardiovascular Care Science. Part 5: Basic Life Support: Relief of Foreign-Body Airway-Obstruction*. Retrieved from http://circ.ahajournals.org/content/122/18_suppl_3/S685.full.

[vii] Hughes, Troy Martin. 2015. *Sorting a Bajillion Records: Conquering Scalability in a Big Data World*. South Central SAS Users Group (SCSUG.)

[viii] 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).

[ix] 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).

[x] 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).

[xi] Nelson, Gregory; Zhou, Jay. 2012. *Good Programming Practices in Healthcare: Creating Robust Programs*. SAS Global Forum.

[xii] Hughes, Troy Martin. 2015. *When Reliable Programs Fail: Designing for Timely, Efficient, Push-Button Recovery.* Pharmaceutical Industry SAS Users Group (PharmaSUG).

[xiii] SAS Press. *SAS® 9.4 Macro Language: Reference, Fourth Edition. %RETURN Statement*. Retrieved from http://support.sas.com/documentation/cdl/en/mcrolref/67912/HTML/default/viewer.htm#p0qyygqt5a69xnn1rhfju3kjs8al.htm.

[xiv] SAS Press. *SAS® 9.4 Statements: Reference, Fourth Edition. RETURN Statement*. Retrieved from https://support.sas.com/documentation/cdl/en/lestmtsref/68024/HTML/default/viewer.htm#p0t9xi85l04wjtn1j31cx4j4kry7.htm.

[xv] 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).

[xvi] SAS Press. *SAS® 9.4 System Options: Reference, Fourth Edition. ERRORABEND System Option*. Retrieved from https://support.sas.com/documentation/cdl/en/lesysoptsref/68023/HTML/default/viewer.htm#n1w4v7ouc6vfhmn1cnsq5aunqk2j.htm.

[xvii] Cogswell, Denis. 2005. *More Than Batch – A Production SAS Framework*. SAS Users Group International (SUGI).

[xviii] Billings, Thomas. 2015. *Strategies for Error Handling and Program Control: Concepts*. SAS Global Forum.

[xix] SAS Press. *SAS® 9.4 Macro Language: Reference, Fourth Edition. %ABORT Statement*. Retrieved from http://support.sas.com/documentation/cdl/en/mcrolref/67912/HTML/default/viewer.htm#p0f7j2zr6z71nqn1fpefnmulzazf.htm.

[xx] Billings, Thomas. 2014. *Differences in Functionality of Error Handling Features, SAS Enterprise Guide vs. Batch*. Western Users of SAS Software (WUSS).

[xxi] SAS Press. *SAS® 9.4 Macro Language: Reference, Fourth Edition. Automatic Macro Variables*. Retrieved from http://support.sas.com/documentation/cdl/en/mcrolref/67912/HTML/default/viewer.htm#n18mk1d0g1j31in1q6chazvfseel.htm.

[xxii] Fielding, David. 2004. *Making Music in SAS: Using Sound to Alert Users of Errors and Data Discrepancies*. SAS Users Group International (SUGI).

[xxiii] Olmsted, Ann. 2007. *Wake Up and Look at the Log: How to Code a Nag Macro*. Western Users of SAS Software (WUSS).

[xxiv] Hamilton, Jack. 2001. *How Many Observations Are in My Data Set?* SAS Users Group International (SUGI).

CONTACT INFORMATION

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

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

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.