Paper 1548 - 2014

A Framework Based on SAS® for ETL and Reporting

Kevin Chung, Fannie Mae, Washington DC

ABSTRACT

Nowadays, most corporations build and maintain their own data warehouse and an ETL (Extract, Transform, and Load) process plays a critical role in managing the data. Some people might create a large program and execute this program from top to bottom. Others might generate a SAS® driver with several programs included and execute this driver. If some programs can be run in parallel, then developers must write extra codes to handle these concurrent processes. If one program fails then users can either rerun the entire process or comment out the successful programs and resume the job from where the program failed. Usually the programs are deployed in production with read and execute permission only. Users do not have the privilege of modifying codes on the fly. In this case, how do you comment out the programs if job terminated abnormally?

This paper illustrates an approach for managing ETL process flows. The approach uses a framework based on SAS, on a UNIX platform. This is a high-level infrastructure discussion with some explanation of the SAS codes that are used to implement the framework. The framework supports the rerun or partial run of the entire process without changing any source codes. It also supports the concurrent process, and therefore no extra code is needed.

INTRODUCTION

A framework is a software platform which provides generic functionalities for jobs running on the platform. The framework discussed in this paper is implemented by SAS/Base and Korn shell script. If a database is used in the back-end, the SAS/Access product is also required. The use of UNIX script is inevitable since the script is used to submit the SAS program. Suppose a large SAS job should be run every day on a UNIX platform. The program may be implemented in either ways below.

```
option . . .;
data tmp;
  infile "...";
  input ...;
  . . .
run;
proc sort data=tmp;
  by key;
run;

<< more statements... >>
```

Table 1

```
option . . .;

%inc "&src_dir/pgm1.sas";

%inc "&src_dir/pgm2.sas";

<< more statements... >>
```

Table 2

One day the job failed or part of the job may need to be rerun. One might have to search the log file and find out the location where error occurred. If the program is coded like the style in Table 1, one has to comment out several lines in the program based on the error found in the log, and rerun. If the program is written like the one in Table 2, it might be a little bit easier than previous case because one will only need to add one or more asterisks at the beginning of the `%inc` statement and rerun the job. Think of the programs in a production environment. You may not be able to modify the codes if you need to rerun. The purpose of this paper is to show a way to control the SAS job submissions using a meta data file which is editable by users. Simply speaking, the approach presented in this paper is a job submission application but it provides several features for jobs running on the framework.

The features provided by the framework are:

- Easy to perform the job execution
- Flexibility to rerun part of the jobs without changing the codes
- Support concurrent process and therefore no extra codes are needed
- Confirmation email is sent when job starts
- Users receive final/result email with time statistics if job completed successfully
- Each SAS program creates an individual log file with timestamp appended at the end of the log file.
- Log file is scanned to capture the WARNING and ERROR message
- Users get notified if any WARNING and/or ERROR occurred during the process
- Job is terminated automatically if ERROR identified
- Time statistics is provided at the end of the process via email, with this user can monitor the performance.

All data provided in this paper is fabricated for demonstration purpose. The terminology **job** and **program** are interchangeable in the paper.

ENVIRONMENT

To support the framework, the following UNIX directories are created as the foundation. We assume the source codes and data are stored in different file system and the project name is xyz. This framework is designed for jobs running in a production environment and we assume the source codes are deployed and owned by CM (Configuration Management) team and therefore users do not have the permission to modify the codes.

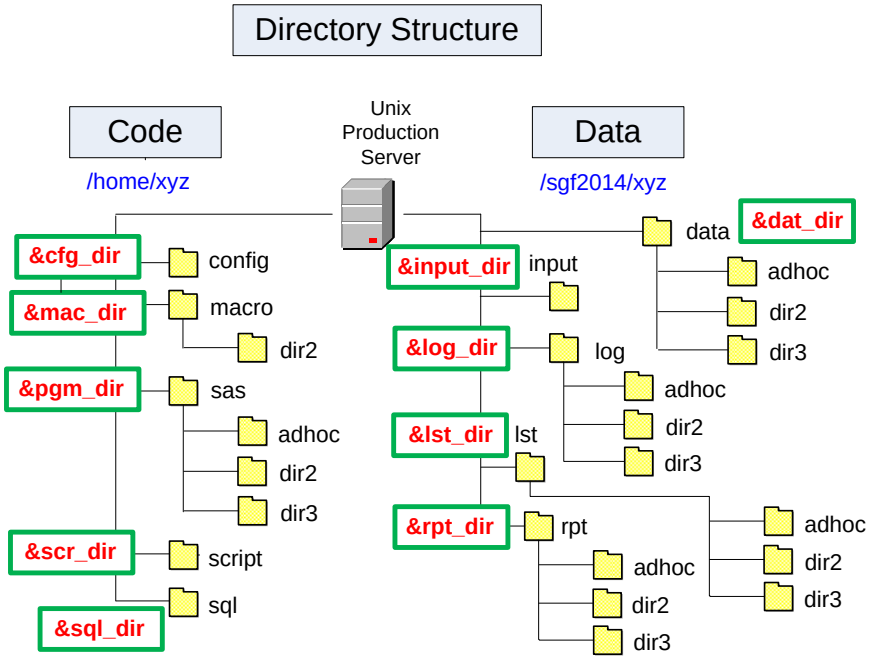

Table 3

The descriptions of each directory are listed below.

Code

- config – store sas config file and autoexec.sas. Password files can be saved in this directory as well.
- macro – store all supporting sas macros
- sas – each process stores its own sas programs in one directory
- script – store Korn shell scripts
- sql – store all SQL scripts

Data

- data – store SAS data sets in each directory based on a process
- input – all input data (SAS data sets or flat files)
- log – SAS log files or user-created log files
- lst – SAS output files
- rpt – any user-created output files.

If the code can be deployed in the UNIX file system, usually there must have the home directory associated with the user id. Those macro variables inside a green box in Table 3 are derived by framework automatically and are available in the entire process.

ASSUMPTION

Several assumptions are made as follows:

- The framework should be run on a UNIX platform.
- Users have the WRITE permission on the Data part of the directories structure in Table 3.
- The home directory exists for a UNIX id running the jobs in framework.
- SAS programs use `.sas` as the file extension and sql scripts use `.sql`. The `.ksh` is used as the file extension for Korn shell script.

HOW DOES THE FRAMEWORK WORK

Input File

An input file is associated with a particular process in the framework. Since the input file is not a program and it's saved under `input` directory, we assume users have the permission to edit this file. An input file is a meta data file which gives the instructions to framework.

A sample input file is listed below.

```
#
# updt_sale
#
# This is a comment
```

Comment Section

```
process_name=Update Sales Data
log_lst_day_retention=7
```

Variable Section

```
RUN_STEP=1  S   env_setup
RUN_STEP=1  Q   crt_stag_table stg_table
RUN_STEP=1  PB  backup_table
RUN_STEP=1  PW  pull_source_data
RUN_STEP=1  PW  crt_stag_data
RUN_STEP=1  PW  prep_master_data
RUN_STEP=1  S   combine_data
RUN_STEP=1  S   insert_data
RUN_STEP=1  S   updt_stag_table
RUN_STEP=1  S   compare
```

Job Section

Run Mode

Comment Section:
Start with a # sign, which is treated as a comment by framework. All contents after the # sign are ignored.

Variable Section:
Any variable defined in this section will be converted to a SAS global macro variable by framework and is available for all jobs during the process.

Job Section:
The keyword `RUN_STEP` only can accept values either 1 or 0. Job with `RUN_STEP=1` is executed.
The second column is the run mode which directs the framework how the job is submitted.

Table 4

The framework supports the following run modes:

SAS	SQL	Korn Shell
S	Q	KW
PW	QW	KB
PB	QB	KWG

S – single sas job in **S**equential
PW – group sas jobs in **P**arallel and **W**ait
PB – **P**arallel sas job in **B**ackground

Q – single S**Q**L job
QW – group S**Q**L jobs in parallel and **W**ait
QB – parallel S**Q**L job in **B**ackground

KW – a **K**orn shell script and **W**ait
KB – a **K**orn shell script in **B**ackground
KWG – a **G**roup of **K**orn shell jobs and **W**ait

NOTE:

- W refers to a synchronous job
- B refers to an asynchronous job
- Run mode starts with Q and K may accept arguments

The input file, `updt_sale.dat`, is read by framework and then each program with `RUN_STEP=1` is converted to a macro based on the run mode. A seq number is automatically added by framework to indicate the order in the execution sequence. Then all macros are written to a dummy SAS program and saved in SAS WORK for submission. This dummy SAS program can be considered as a driver of the process. The SAS WORK can be identified by `%sysfunc(pathname(work))`.

Example of the conversion

updt_sale.dat

```
RUN_STEP=1  S   env_setup
RUN_STEP=1  Q   crt_stag_table stg_table
RUN_STEP=1  PB  backup_table
RUN_STEP=1  PW  pull_source_data
RUN_STEP=1  PW  crt_stag_data
RUN_STEP=1  PW  prep_master_data
RUN_STEP=1  S   combine_data
RUN_STEP=1  S   insert_data
RUN_STEP=1  S   updt_stag_table
RUN_STEP=1  S   compare
```

driver.sas

```
%sequential(env_setup,seq=1)
%run_sql(crt_stag_table 'stg_table',seq=2)
%parallel(backup_table,mode=B,seq=3)
%parallel(pull_source_data crt_stag_data
          prep_master_data, mode=W,seq=4)
%sequential(combine_data,seq=7)
%sequential(insert_data,seq=8)
%sequential(updt_stag_table,seq=9)
%sequential(compare,seq=10)
```

Table 5

updt_sale.dat

```
RUN_STEP=1 S  env_setup
RUN_STEP=0 Q  crt_stag_table stg_table
RUN_STEP=0 PB backup_table
RUN_STEP=0 PW pull_source_data
RUN_STEP=0 PW crt_stag_data
RUN_STEP=1 PW prep_master_data
RUN_STEP=1 S  combine_data
RUN_STEP=1 S  insert_data
RUN_STEP=1 S  updt_stag_table
RUN_STEP=1 S  compare
```

driver.sas

```
%sequential(env_setup,seq=1)
%parallel(prep_master_data,mode=W,
          seq=2)
%sequential(combine_data,seq=3)
%sequential(insert_data,seq=4)
%sequential(updt_stag_table,seq=5)
%sequential(compare,seq=6)
```

Table 6

The `driver.sas` in Table 5 is created based on the input file. If in case part of the jobs need to be rerun, those programs that have been run successfully can be set to 0 in `RUN_STEP` and the SAS driver is created as Table 6.

To create `driver.sas`, the first step is to read input file, `updt_sale.dat`, and create a SAS data set. And then use the data set to aggregate the job by `run_mode`. Remember to apply the NOTSORTED option in BY statement because the order in the input file must be maintained.

```
data input_data2;
  length run_mode $3 list program $1024;

  retain run_mode list ' ';
  set input_data;
  by run_mode notsorted;
  if (first.run_mode) and (run_mode in: ('P','KWG')) then list=' ';
  else if (run_mode not in: ('P','KWG')) then output;

  if (run_mode in: ('P','KWG')) then list=catx(' ',list,program);

  if (last.run_mode) and (run_mode in: ('P','KWG')) then do;
    program=list;
    output;
  end;
  keep run_mode program;
run;
```

The listing of the data set created by above data step is displayed below.

```
 run_mode    program

    S        env_setup
    Q        crt_stag_table stg_table
    PB       backup_table
    PW       pull_source_data crt_stag_data prep_master_data
    S        combine_data
    S        insert_data
    S        updt_stag_table
    S        compare
```

Sample codes for generating %sequential, %parallel, and %run_sql are listed below.

```
data _null_;
  length pgm $32 parm $100;
  set input_data2;
  file "&wk_main/driver.sas";

  indx+1;
  if (run_mode='S') then                                          /* Sequential */
    put '%sequential(' program +(-1) ",seq=" indx +(-1) ")";
  else if (run_mode =: 'P') then do;                              /* Parallel */
    ch=substr(run_mode,2,1);
    n_of_pgm=length(compbl(strip(program)))-length(compress(strip(program),' '))+1;
    put '%parallel(' program +(-1) ",mode=" ch +(-1) ",seq=" indx +(-1) ")";
    indx+n_of_pgm-1;
  end;
  else if (run_mode =: 'Q') then do;                       /* run_sql */
    pgm=scan(program,1);
    parm=" '"||trim(substr(program,length(pgm)+2))||"'";    /* run_mode=Q can    */
    put '%run_sql(' pgm parm +(-1) ",seq=" indx +(-1) ")";  /* accept parameters */
  end;
run;
```

The above data step generates the following macros in &wk_main/driver.sas.

```
    %sequential(env_setup,seq=1)
    %run_sql(crt_stag_table 'stg_table',seq=2)
    %parallel(backup_table,mode=B,seq=3)
    %parallel(pull_source_data crt_stag_data prep_master_data,mode=W,seq=4)
    %sequential(combine_data,seq=7)
    %sequential(insert_data,seq=8)
    %sequential(updt_stag_table,seq=9)
    %sequential(compare,seq=10)
```

You might ask a question why the seq= is required? The &seq is used as a suffix in the file xxx.timestamp.&seq saved under main SAS WORK, where xxx is the SAS program or sql script. The run_mode Q family can accept parameters, there might have two or more sql scripts in the input file with different parameters. So, the &seq is used to uniquely identify each job in the entire process. Please refer to Table 8 through 11 for more detailed information.

Please note that if the words “main SAS WORK” is mentioned in the paper, it is referred to the SAS temporary WORK library that associated with a SAS job submitted by a Korn shell script and it’s referenced as &wk_main by framework in entire process.

Program Flow

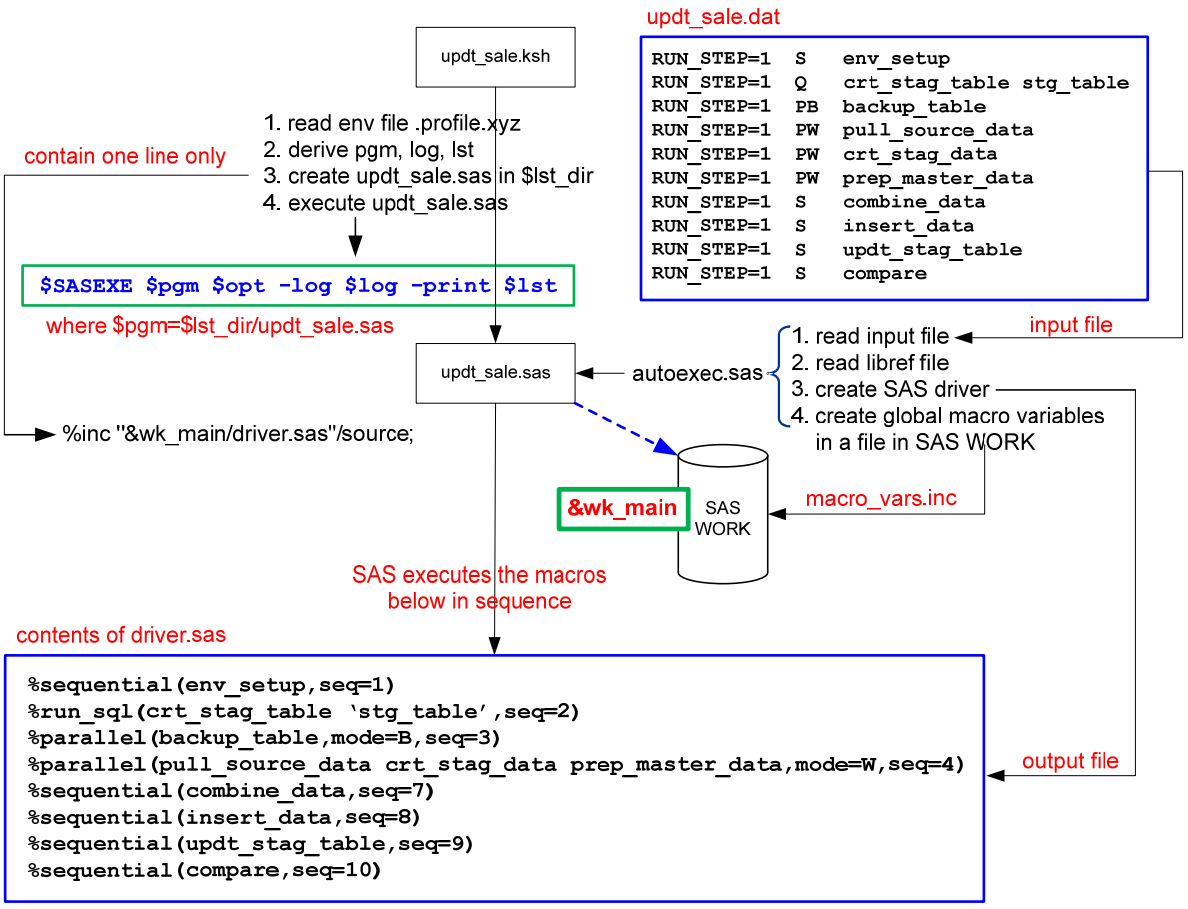

Table 7

Let's start from the UNIX shell script updt_sale.ksh and trace the program flow one by one to see how the framework works. The purpose of the UNIX shell script is to submit a SAS job. Appendix A shows the contents of the Korn shell script. After the shell script is kicked off, a dummy SAS program containing only one line statement,

```
%inc "&wk_main/driver.sas"/source;,
```

is created in `$lst_dir` by the shell script and then submitted by $SASEXE. ($SASEXE is defined in `.profile.xyz` and `$lst_dir` is derived in `derive_log_lst`). The following command submits a SAS job.

```
$SASEXE $pgm $opt -log $log -print $lst
where $opt is defined as
  opt="-unbuflog -noterminal -autoexec $autoexec -rsasuser -config $v9cfg"
```

`$autoexec` is defined in `.profile.xyz` and actually is the file `autoexec.sas` saved under config directory.

The SAS command line option **`–autoexec $autoexec`** is specified, the contents in `autoexec.sas` get executed first before `updt_sale.sas` is run. The major steps in `autoexec.sas` are

- read input file
  - ❖ Convert the sas jobs specified in to a macro, such as `%sequential`, `%parallel`, based on the run mode.
- read libref file
  - ❖ The contents of the file is shown in Appendix A. The purpose of this file is to avoid the same libref definition in every single SAS program. The framework reads the libref based on process and place the `libname` statement along with the macro variables in `macro_vars.inc`. (See the fourth bullet below)
- create a SAS driver under WORK
  - ❖ Write macros to a file `driver.sas` under `&wk_main`, where `&wk_main` is the SAS WORK directory associated with `updt_sale.sas` and derived in `autoexec.sas`.

- create global macro variables in a file, `macro_vars.inc`, under WORK.
  - ❖ The contents of the file, `macro_vars.inc`, are those UNIX environment variables, the libname reference such as `library` for format library and `stag` from `libref.dat`. It also defines the path for the SAS autocall library. The macro variable `&opt` is also defined in this file with the option **`-noautoexec`**.

Once the `autoexec.sas` is completed successfully, the one line statement

```
%inc "&wk_main/driver.sas"/source;
```

get executed and therefore the macros created in `driver.sas` are triggered from within the main SAS program `updt_sale.sas` in sequence. Each job is executed in an independent SAS session and each SAS program creates an individual SAS log in `&log_dir`. Please note that all SAS programs, either triggered by `%sequential` or `%parallel`, are read from their original location and some additional codes generated by framework are added to the top of each SAS program. Then each "revised" SAS program is written to the SAS WORK for execution.

The final email sent to recipients if the entire process executed successfully. A result email is shown in Appendix A. Let's take a look at some of the key components that implement the framework.

%sequential

The following is the major part of the macro `%sequential`. Since the run_mode is S for this job, the next job won't start until this job is finished successfully. The value of macro variable `&t1` is written to `env_setup.timestamp.1` and `&t2` is appended to the same file after the job `env_setup.sas` is done. The purpose of this is to gather the time statistics as shown in Appendix B. The key part of this macro is to submit the sas program by SYSTASK. The SYSTASK command is designed to execute asynchronous tasks. Since this is a sequential job, a WAITFOR statement is used to wait the specified task to finish executing. In this case, the SAS X statement or %sysexec work the same way as SYSTASK.

```
%macro sequential(pgm,dir=&wk_main,seq=);
  data _null_;
    infile "&pgm_dir/&pgm..sas";    ← original SAS program
    input;
    file "&dir/&pgm..sas";  ← new SAS program to be submitted
    if (_n_=1) then do;
      put '%inc '"'&dir/macro_vars.inc';"/  ← the file is created in autoexec.sas
          '%let t1=%sysfunc(round(%sysfunc(datetime())));'/  ← start time
          '%write_stat(&pgm,no=&indx)';    ← write &t1 to &pgm..timestamp.&seq
    end;
    put _infile_;
  run;

  systask command "&SASEXE &opt &dir/&pgm..sas -log &log -print &lst" taskname=job0;
  waitfor _all_ job0;

  %let t2=%sysfunc(round(%sysfunc(datetime())));  ← end time
  x "cd &wk_main; echo '%nrstr(%let t2=)&t2;' >> &pgm..timestamp.&seq";  ← write &t2
                                                                          to &pgm..timestamp.&seq
  %chk_log(&log,&pgm,dir=&wk_main,no=&seq)  ← check ERROR /WARNING in &log file
%mend sequential;
```

The following diagram shows the exection of the first job, `env_setup.sas`. Please note that each program is submitted from within the main SAS session, there always has a SAS WORK associated with each SAS job.

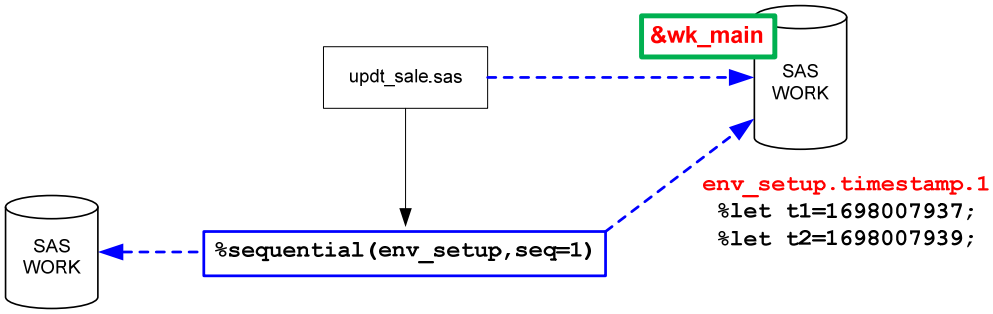

Table 8

Exception Handling for run_mode=S

What does framework do if there is an ERROR occurred in a sequential job? This functionality is handled by the macro %chk_log.

The key components for %chk_log are listed below. Macro variable &f is the log file.

```
%let word=uninitialized|^ERROR|^WARN|^SP2-|^ORA-|^PLS-|Invalid;
%let w1=Unable to copy SASUSER;
%let w2=Invalid \(or missing\) arguments to the DATEPART function;
%let w3=printed on page;
%let w4=SAS ended due to|ORACLE does not support SORTEDBY;

option nonotes nomprint nomlogic nosgen NOQUOTELENMAX;
filename in1 pipe "egrep -n '(&word)' &f|egrep -v '(&w1|&w2|&w3|&w4)'";
```

The values specified in macro variable word are the keywords used to identify the job status in the log file. If a job is finished successfully, there should not have any ERROR message in the log; but there might contain WARNING or Invalid data messages. In this case, WARNING/Invalid messages might be sent via email to recipients. If a job failed, the ERROR keyword can be found by the UNIX egrep command. The ^ sign immediately placed before the ERROR keyword is to locate the message with the ERROR at the first column of the log. Please note that the sas program env_setup in this example is submitted in a separate SAS session and a log file is created in the same session. The macro %chk_log is in the same SAS session as main program updt_sale.sas, so if a job failed and identified by %chk_log, the entire process can be terminated by framework and an ERROR email is sent to the recipients. The ENDSAS statement can be used to terminate a SAS job or session.

The contents defined in &w1 through &w4 are those WARNING or ERROR messages which can be excluded from the scan of the egrep command. The contents of &w1 through &w4 can be defined in an external file, say warning_excl.dat, and the UNIX command egrep -v '(&w1|&w2|&w3|&w4)' can be changed to egrep -v –f $input_dir/warning_excl.dat. This approach is better because it allows users to add some unwanted WARNING messages.

%run_sql

Any sql script can be converted to a SAS job using SQL procedure with SQL pass-through facility. This example is to show the execution of a pure Oracle sql script by the framework. The following is the key component to kick off an Oracle sql script.

```
%sysexec &SQLPLUS &user/&pass@&path @&pgm &log &parm_list;
%chk_log(&log,&pgm,type=Q,no=&seq)
```

The Table 9 shows the execution of the sql script. Since this is a sql job triggered by a SAS process, the SAS WORK is still created for this job.

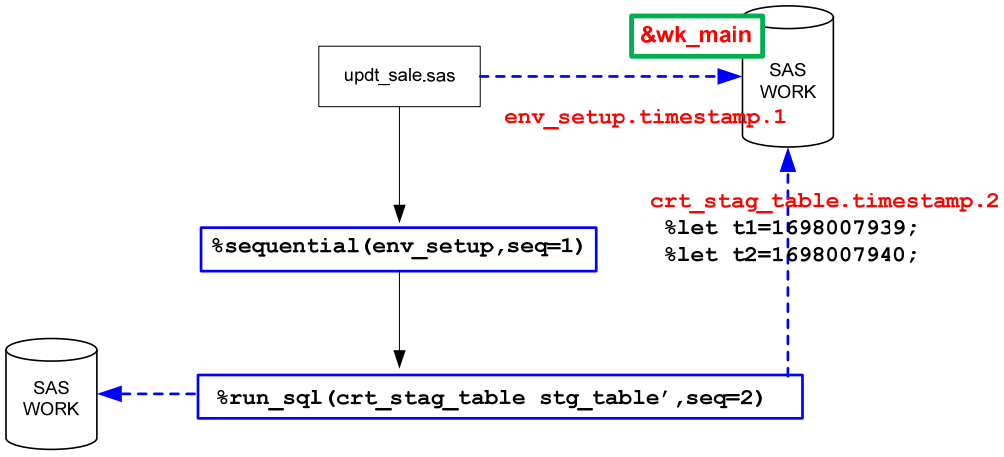

Table 9

We assume the username, password, and Oracle instance can be retrieved from the file saved under config directory. In the example, &pgm is crt_stag_table.sql and &parm_list is stg_table. The &log file is always placed immediately after the &pgm in the command line and sql output can be written to &log. This requires some extra codes added on the top of the sql script as follows:

```
set echo on
set timing on
set serveroutput on
spool &1    ← redirect the execution message to the &log
```

The &log is created due to the use of the Oracle sqlplus spool command. The following diagram shows the execution of the sql script. In this stage, the first job, %sequential, is completed successfully and it's associated SAS WORK is gone.

Exception Handling for run_mode=Q

This is not a SAS job, so there is no ERROR or WARNING messages. But the keywords SP2-, ORA-, and PLS- can be used to identify the job status.

%parallel with run_mode=PB

The key components are described below. Basically, we want to submit the job like the following:

```
%sysexec &SASEXE &opt &wk_main/&pgm..sas -log &log -print &lst &;
%chk_log(&log,&pgm,no=&seq)
```

The & at the end implies this is an asynchronous job. The %chk_log is to check the log file after the job is done. If the job backup_table.sas is triggered like this way, macro %chk_log gets executed immediately after %sysexec is run. At this time, the log only contains partial information because the job is still running. Obviously, this is not the right process we expect. To solve this issue, we can do something like this below:

Create a new SAS program, say &pgm..sas2, in &wk_main with the following codes:

```
%sysexec &SASEXE &opt &wk_main/&pgm..sas -log &log -print &lst;
%chk_log(&log,&pgm,no=&seq)
x "\rm &log_dir/&pgm._&dttm..log2";
```

Then, submit the new SAS program as below:

```
%sysexec &SASEXE &wk_main/&pgm..sas2 &opt -log &log_dir/&pgm._&dttm..log2 &;
```

It turns out that the new SAS program is run as an asynchronous job, but the behavior is identical to %sequential inside the new code. In this stage, the second job, %run_sql, is completed and its associated SAS WORK is gone.

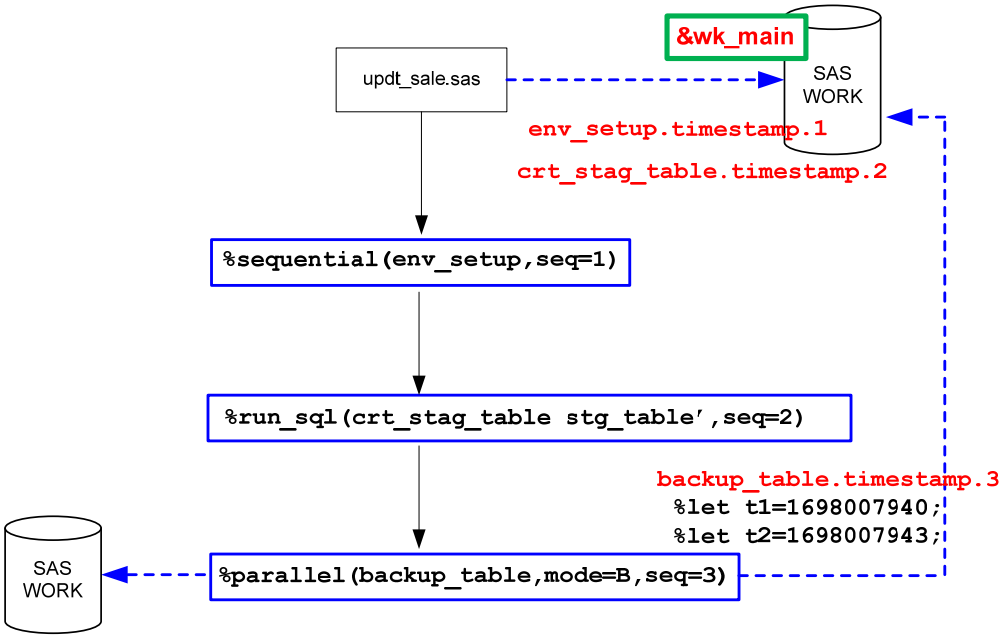

Table 10

Exception Handling for run_mode=PB

The job with `run_mode=PB` is run independently and it does not caused main process to be terminated even the asynchronous job failed. In other word, if a PB job is failed, the job is terminated by framework and the entire process keeps running.

%parallel with run_mode=PW

The key components for `%parallel` with `run_mode=PW` job is similar to that for PB job except each new SAS program is submitted by SYSTASK as follows:

```
%do i=1 %to &n_of_job;
  systask command "&SASEXE &opt &dir/&pgm..sas2 -log &dir/&pgm..log2" taskname=J&i;
  %let job_list=&job_list J&i;
  %let log_list=&log_list &pgm._&dttm..log;
%end;
waitfor _all_ &job_list;
%chk_all_log_err(&log_list)
```

In this example, three SAS jobs, `pull_source_data.sas`, `crt_stag_data.sas`, and `prep_master_data.sas` are kicked off one by one by SASTASK in a macro do loop. The tasknames are stored in a macro variable. After all three jobs are submitted, the `WAITFOR` statement suspends execution of the current SAS session until the specified three tasks finish executing. Table 11 below shows the execution of this PW job. In this stage, `%run_sql` is done and the statistics of this job has been written to main SAS WORK as `crt_stag_table.timestamp.2`. As you can see from the diagram, three PW jobs are running and each owns its SAS WORK.

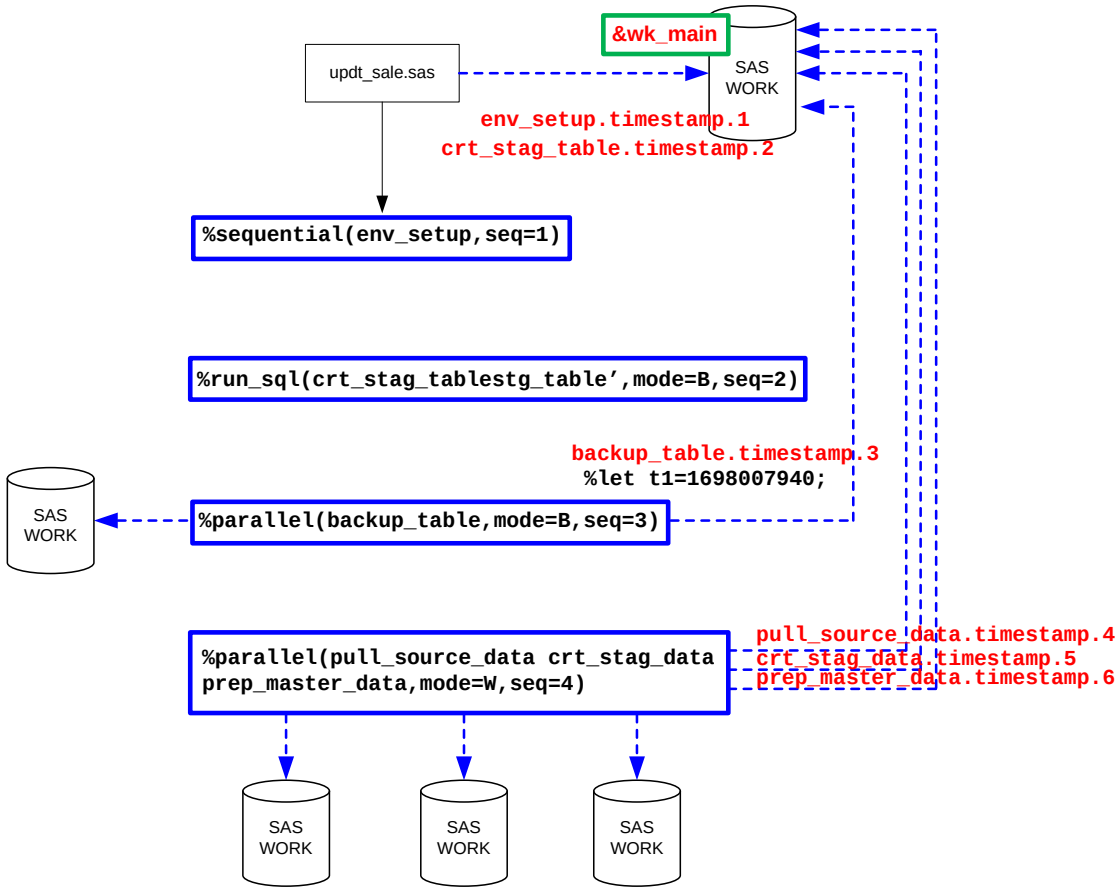

Table 11

Exception Handling for run_mode=PW

If any job gets failed during the process, that job is terminated immediately. All other PW jobs keeps running until they are complete successfully or terminated. Then entire process is aborted abnormally.

Aggregation of processes

We have demonstrated how a shell script triggers SAS programs or sql script in an input file. Can a shell script trigger an input file which contains shell scripts only? The answer is YES. Each UNIX shell script in framework is used to trigger one "process". In this section, a process stands for a collection of jobs/programs. We can think of each shell script is used to trigger a process. Several processes (shell scripts) can be grouped together and triggered by a shell script as if several SAS programs/sql scripts can be placed in an input file and triggered by a shell script. For example, let's assume there is a dayily ETL process started 7am every day. The contents of the 7am process are displayed below. All the Korn shell scripts are stored in `&scr_dir` with file extension `.ksh`. The sample Korn shell script `daily_7am.ksh` is listed in Appendix A.

daily_7am.dat

```
RUN_STEP=1   KW    pull_data
RUN_STEP=1   KW    step1
RUN_STEP=1   KW    step2
RUN_STEP=1   KWG   load_table1
RUN_STEP=1   KWG   load_table2
RUN_STEP=1   KB    sales_rpt
```

KW – Korn shell script with WAIT. The next process can not start until this process is done successfully. Only one shell script is submitted. The behavior is similar to run_mode=S

KWG – Same as KW but several processes can be run in parallel. G stands for Group. The process load_table1 and load_table2 can be run at the same time. The behavior is similar to PW.

KB – Korn shell script run in Background. The behavior is similar to PB.

The following contents are created in `driver.sas`.

```
%run_ksh(pull_data,mode=W,seq=1)
%run_ksh(step1,mode=W,seq=2)
%run_ksh(step2,mode=W,seq=3)
%run_ksh(load_table1 load_table2,mode=WG,seq=4)
%run_ksh(sales_rpt,mode=B,seq=6)
```

The following is an example in `%run_ksh` which creates a new shell script in `&wk_main` and run that script. The original Korn shell script is read and written to the `&wk_main`. The purpose is to add three UNIX environment variables to the original file immediately after `.profile.xyz`.

```
data _null_;
  infile "&scr_dir/&pgm..ksh" lrecl=512;  ← original Korn shell script
  input;
  file "&wk_main/&pgm..ksh" lrecl=512;  ← new Korn shell script
  put _infile_;

  if (index(_infile_,".profile.xyz")) then do;
    put "export BOX_WK_MAIN=&wk_main"/
        "export BOX_SEQ=%eval(&seq+&i-1)"/
        "export BOX_RUN_MODE=K%substr(&mode,1,1)"/
    ;
  end;
run;
x "chmod 700 &wk_main/&pgm..ksh";  ← change permission to executable
x "cd &wk_main; &pgm..ksh";   ← run the Korn shell script
```

Why does the framework do this way? At page 6, a shell script triggers a process by $SASEXE with the command line option –`autoexec $autoexec`. A statement `%let wk_main=%sysfunc(pathname(work));` in autoexec.sas gets the physical location of main SAS WORK. Several global macro variables are created in `macro_vars.inc` in `autoexec.sas` and `&wk_main` is one of them (Refer to Table 7). The file, macro_vars.inc, is added to the top of each program (Refer to page 7 for `%sequential`) so that all the child jobs inherit the environment settings from the parent job via macro variables. One of the macro variable in `macro_var.inc` is

```
%let BOX_WK_MAIN=/sastemp/ SAS_work03750_server1;
```

This implies every child process knows its parent's physical location of SAS WORK. This is important because if a child process with run_mode=KW failed (let's assume this is caused by a SAS job with run_mode=S) then the ERROR information can be escalated to it's parent and the ERROR message is written to `&BOX_WK_MAIN` so that the `%chk_log` in parent process identifies the ERROR status and the parent process is terminated. In this case, the child process has been terminated before parent process is terminated. The following diagrams show the program flow of the example `daily_7am.ksh`.

Program Flow

The shell script `daily_7am.ksh` is kicked off, a `driver.sas` is created in the dummy SAS program `group.sas`. The first KW job `pull_data.ksh` is run and triggers the dummy SAS program `pull_data.sas`. In this case, `group.sas` is the parent process with the SAS WORK `&BOX_WK_MAIN`, which is identical to `&wk_main` in its SAS session. When child process, `pull_data.sas`, is triggered, there is another SAS WORK associated with `pull_data.sas` with the same name `&wk_main`. This is ok because they are in different SAS session. Since parent's SAS WORK information has been passed to child process via `"export BOX_WK_MAIN=&wk_main"`, child process can recognize the parent's SAS WORK location and pass message back to parent if necessary.

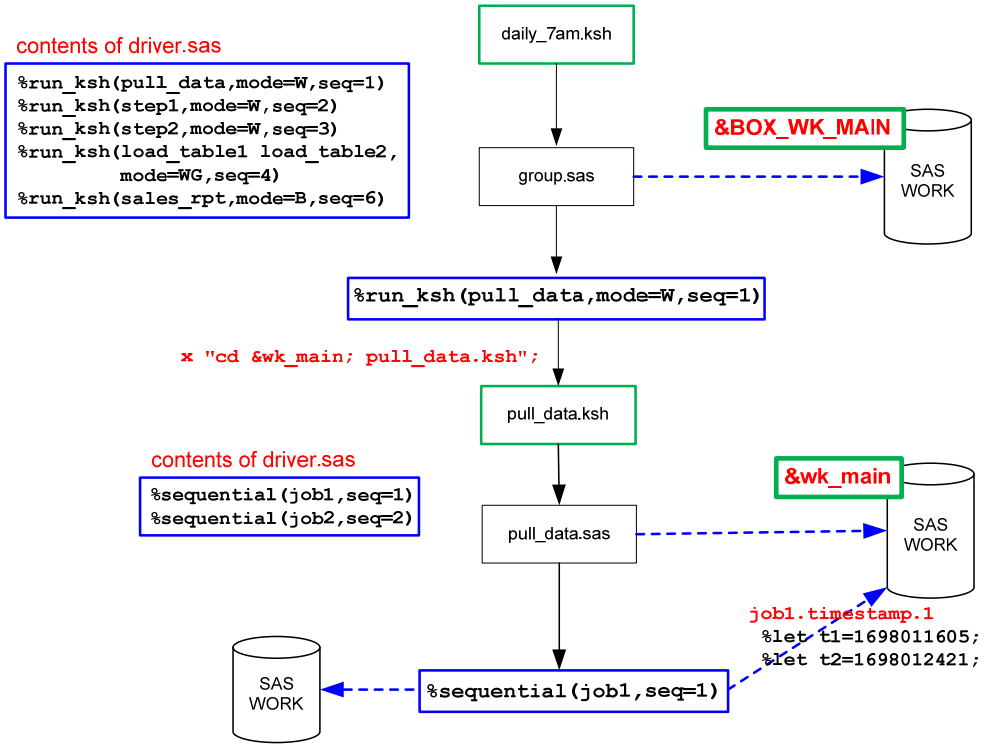

Table 12

Suppose first job in child process is done and the second job started. The diagram below only shows the child process. Parent process stays the same as Table 12.

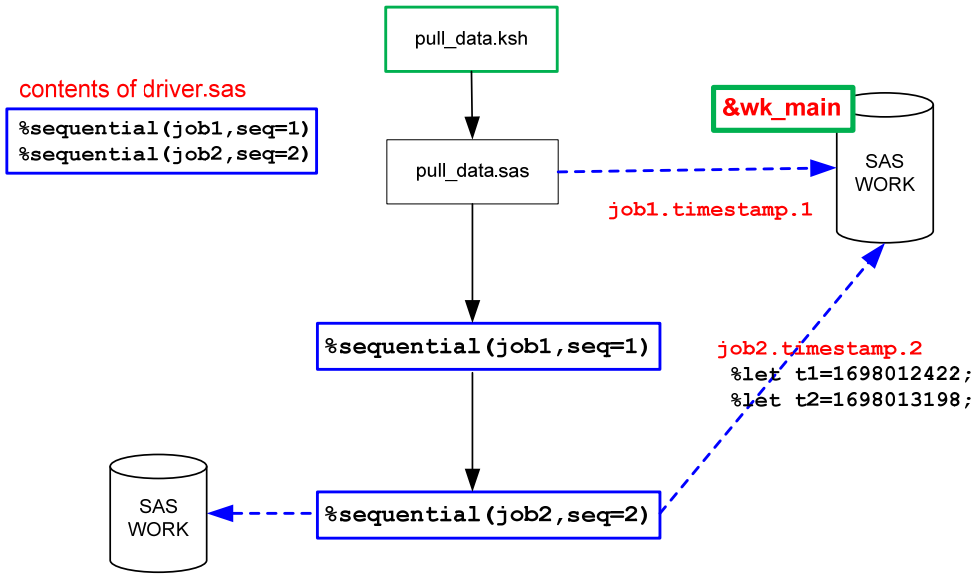

Table 13

Suppose two sequential jobs in child process have been completed successfully in Table 13. Then the time statistics information of the first job of parent process is written to parent's SAS WORK in Table 14.

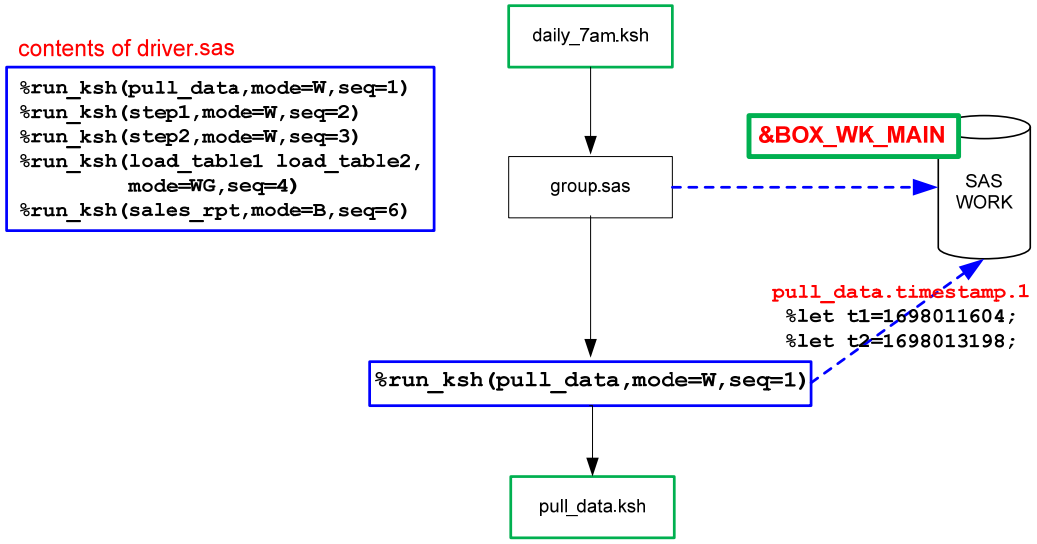

Table 14

Same procedure repeats until all jobs in parent's driver are processed.

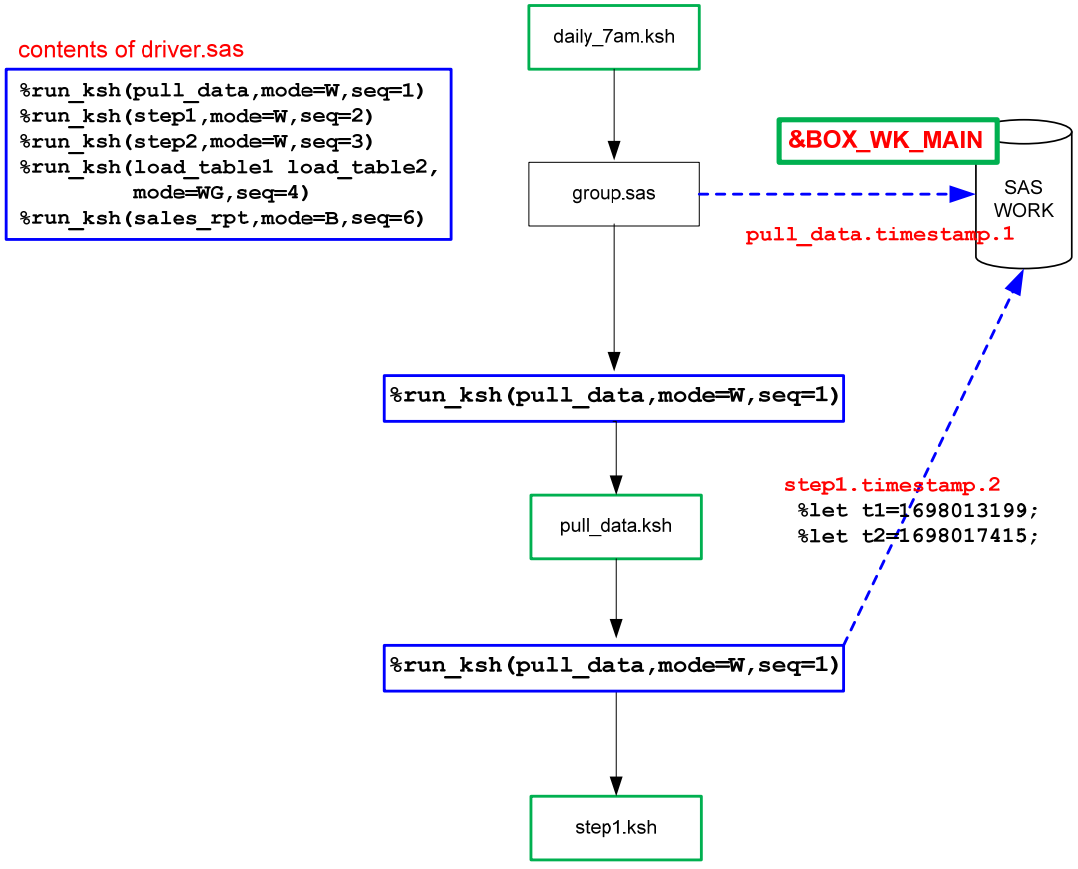

Table 15

Email Body

The email body in Table 16 is created by the HTML tags below. The content of part 1 is generated by `data _null_`.

```
<table>
  <td>
    <pre> <font face="Courier New" size="2">
      (1)
    </font></pre>
  </td>

  <td width=50/>   <-- (2)
    [ CSS Style ]

  <td>
      (3) (4)

  </td>
</table>
```

The macro `%bold` below can be used to set the font color and font weight.

```
%macro bold(txt,color,weight=B);
  %local w1 w2;
  %if (%upcase(&weight)=B) %then %do;
    %let w1=<b>;
    %let w2=</b>;
  %end;
  &w1<span style=%bquote('color:&color')
>&txt</span>&w2;
%mend bold;
```

To apply color on a line, just add the tags before and after the contents as follows:

```
<span style="color:blue">        Process: Update Sales Data </span>
```

To generate the table in part 3, the following CSS style should be added to [CSS Style]. The prupose of the style is to set the font weight and color for the contents in the table.

```
<style type="text/css">
  p {font-family:"verdana"; font-size:"12px"}
  p.f {color:black}
  p.g {font-weight:"bold"; color:green}
  p.r {font-weight:"bold"; color:red}
  p.p {font-weight:"bold"; color:purple}
</style>
```

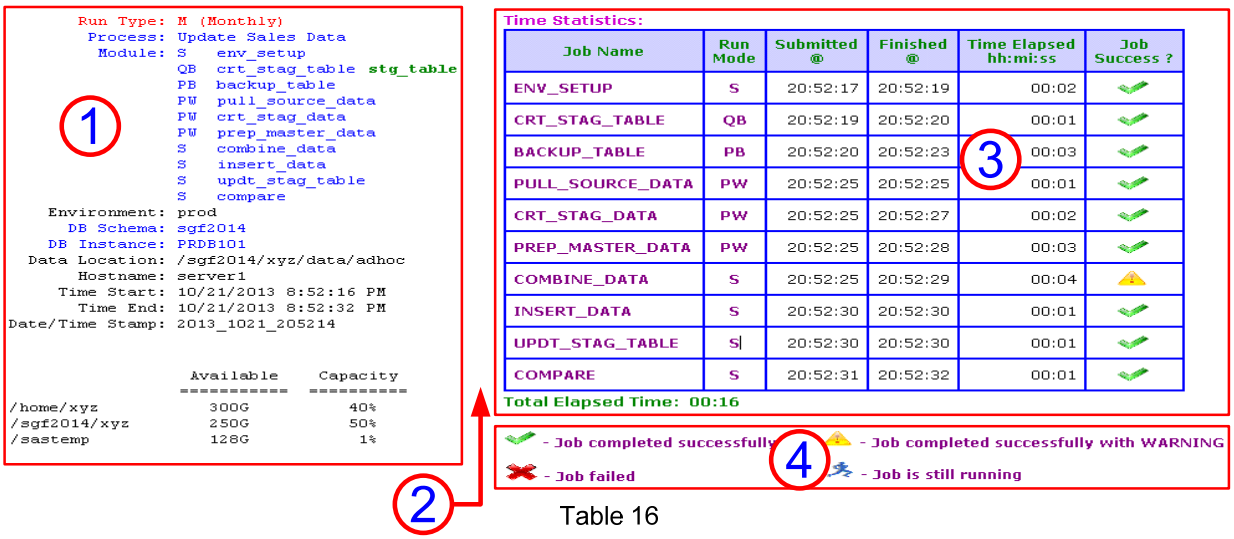

Table 16

The first two rows in part 3 of table 16 are generated by the following HTML tags.

```
<span style="color:#CC00CC; font-family:verdana; font-size:10pt; font-weight:bold">
Time Statistics:</span>
<table border=1 bordercolor=blue cellpadding=6 cellspacing=0>
<tr align=center bgcolor="#CCCCFF">
  <th ><p align=center class="g">Job Name</p></th>
  <th ><p class="g">Run<br>Mode</p></th>
  <th ><p class="g">Submitted<br>@</p></th>
  <th ><p class="g">Finished<br>@</p></th>
  <th ><p class="g">Time Elapsed<br>hh:mi:ss</p></th>
  <th ><p class="g">Job <br>Success ?</p></th>
</tr>
```

```
  <tr>
    <td align=left><p class="p">ENV_SETUP</p></td>
    <td align=center><p class="p">S</p></td>
    <td align=right><p class="f">20:52:17</p></td>
    <td align=right><p class="f">20:52:19</p></td>
    <td align=right><p class="f">00:02</p></td>
    <td align=center><p class="g"><img src= "http://sharepoint/success.png" width="27"
        height="18"></p></td>
  </tr>
```

And the last row in part 3 of table 16 is created by the following.

```
  <tr>
    <td align=left><p class="p">COMPARE</p></td>
    <td align=center><p class="p">S</p></td>
    <td align=right><p class="f">20:52:31</p></td>
    <td align=right><p class="f">20:52:32</p></td>
    <td align=right><p class="f">00:01</p></td>
    <td align=center><p class="g"><img src="http://sharepoint/success.png" width="27"
        height="18"></p></td>
  </tr>
  </table>
  <span style="color:green; font-family:verdana; font-size:10pt; font-weight:bold">
Total Elapsed Time:  00:16</span>
<br><br>
```

10 data rows in part 3 are generated by SAS macro `DO` loop below.

```
%let wh=%str(width=%""27%"" height=%""18%"");
%do i=1 %to &n_of_tm_stamp; /* prog_row */
  "<tr>"/
  "  <td align=left><p class=""p"">&&prog_row&i</p></td>"/
  "  <td align=center><p class=""p"">&&mode_row&i</p></td>"/
  "  <td align=right><p class=""f"">&&submit&i</p></td>"/   ← &t1
  "  <td align=right><p class=""f"">&&finish&i</p></td>"/   ← &t2
  "  <td align=right><p class=""f"">&&elapsed&i</p></td>"/  ← derived. &t2 - &t1
  "  <td align=center><p class=""g""><img src=""&location/&&status&i...png""
     %unquote(&wh)></p></td>"/
  "</tr>"/
%end;
```

The contents in part 4 are generated by the following.

```
%let location=http://sharepoint;
%let success=Job completed successfully;
%let fail=Job failed;
%let warn=Job completed successfully with WARNING;
%let running=Job is still running;

<table>
<td>
  <p class="p"><img src="&location/success.png" &wh> - &success</p>
  <p class="p"><img src="&location/error.png" &wh> - &fail</p>
</td>
<td width=30/>
<td>
  <p class="p"><img src="&location/warning.png" &wh> - &warning</p>
  <p class="p"><img src="&location/running.png" &wh> - &running</p>
</td>
</table>
```

Appendix A

Sample Korn shell script

```
#!/bin/ksh
#
# updt_sale.ksh
#

. ~/.profile.xyz      ← include framework environment settings
process=adhoc         ← the directory or process under sas
infile=updt_sale.dat  ← the input file
prog=updt_sale        ← a dummy sas program to be kicked off by shell script

derive_log_lst $prog $process    ← derive $pgm, $log, and $lst

echo "Job submitted at `date`"
$SASEXE $pgm $opt -log $log -print $lst
```

Sample Korn shell script for a GROUP process

```
#!/bin/ksh
#
# daily_7am.ksh
#

. ~/.profile.zeus
process=group          ← a dummy directory for a GROUP process
infile=daily_7am.dat   ← the input file
prog=group             ← a dummy sas program to be kicked off by shell script

derive_log_lst $prog $process    ← derive $pgm, $log, and $lst

echo "Job submitted at `date`"
$SASEXE $pgm $opt -log $log -print $lst
```

Example of .profile.xyz

```
#
# .profile.xyz
#
dttm=`date +%Y_%m%d_%H%M%S`       # format: yyyy_mmdd_hhmmss
this_script=$0
env=prod
project=xyz
home=/home/xyz
data_dir=/sgf2014/xyz
cfg_dir=$home/config
SASEXE=/apps/sas_9.2/SASFoundation/9.2/sas
SQLPLUS=/apps/oracle/bin/sqlplus
FPATH=$home/scripts
autoexec=$cfg_dir/autoexec.sas
v9cfg=$cfg_dir/sasv9.cfg
BOX_WK_MAIN=""
BOX_SEQ=""
BOX_RUN_MODE=""
```

libref.dat

```
 P R O C E S S     libref                    F U L L   P A T H
================   ========   ==================================================
daily_2am          stag       &dat_dir
daily_7am          stag       &dat_dir
weekly             lib        &input_dir
monthly            in         /sgf2014/data_p1/in
monthly            out        /sgf2014/data_p2/out
adhoc              in1        &dat_dir
adhoc              in2        &dat_dir
adhoc              in3        &dat_dir
```

Example of autoexec.sas

```
option nocenter varlenchk=nowarn errorabend;
%let session_time_start=%left(%sysfunc(date(), mmddyy10.)) %left(%sysfunc(time(),
timeampm.));

%let dttm=%sysget(dttm);  /* format: 2012_0816_073025 used as suffix of log file */
%let email_list=%sysget(email_list);
%let wk_main=%sysfunc(pathname(work));    ← identify current SAS WORK
%let pid=&sysjobid;

%let env=%sysget(env);
%let home=%sysget(home);
%let data_dir=%sysget(data_dir);
%let project=%sysget(project);
%let process=%sysget(process);
%let process_name=&process;
%let infile=%sysget(infile);
%let SASEXE=%sysget(SASEXE);
%let SQLPLUS=%sysget(SQLPLUS);

%let BOX_WK_MAIN=%sysget(BOX_WK_MAIN);    ← main SAS WORK
%let BOX_SEQ=%sysget(BOX_SEQ);
%let BOX_RUN_MODE=%sysget(BOX_RUN_MODE);

libname inputs "&input_dir";
libname library "&input_dir";
libname wk_main "&wk_main";
filename source ("&pgm_dir");

%read_libref(&input_dir/libref.dat)  /* return: GLOBAL libref_list */
%read_input(&input_dir/&infile)
%crt_mac_var(&cfg_dir/autoexec.sas,var=&g_list,lib=&libref_list)
```

Appendix B

Confirmation email

```
          Run Type: M (Monthly)
           Process: Update Sales Data
            Module: S   env_setup
                    QB  crt_stag_table stg_table
                    PB  backup_table
                    PW  pull_source_data
                    PW  crt_stag_data
                    PW  prep_master_data
                    S   combine_data
                    S   insert_data
                    S   updt_stag_table
                    S   compare
       Environment: prod
         DB Schema: sgf2014
       DB Instance: PRDB101
        Process ID: 8388814
     Data Location: /sgf2014/xyz/data/adhoc
          Hostname: server1
        Time Start: 10/21/2013 8:52:16 PM
  Date/Time Stamp: 2013_1021_205214
             WORK: /sastemp/SAS_work3597008000CE_server1


                    Available     Capacity
                   ============  ==========
/home/xyz             300G          40%
/sgf2014/xyz          250G          50%
/sastemp              128G           1%
```

Result email

```
          Run Type: M (Monthly)
           Process: Update Sales Data
            Module: S   env_setup
                    QB  crt_stag_table stg_table
                    PB  backup_table
                    PW  pull_source_data
                    PW  crt_stag_data
                    PW  prep_master_data
                    S   combine_data
                    S   insert_data
                    S   updt_stag_table
                    S   compare
       Environment: prod
         DB Schema: sgf2014
       DB Instance: PRDB101
     Data Location: /sgf2014/xyz/data/adhoc
          Hostname: server1
        Time Start: 10/21/2013 8:52:16 PM
          Time End: 10/21/2013 8:52:32 PM
  Date/Time Stamp: 2013_1021_205214


                    Available     Capacity
                   ============  ==========
/home/xyz             300G          40%
/sgf2014/xyz          250G          50%
/sastemp              128G           1%
```

Time Statistics:

Job Name	Run Mode	Submitted @	Finished @	Time Elapsed hh:mi:ss	Job Success ?
ENV_SETUP	S	20:52:17	20:52:19	00:02	✓
CRT_STAG_TABLE	QB	20:52:19	20:52:20	00:01	✓
BACKUP_TABLE	PB	20:52:20	20:52:23	00:03	✓
PULL_SOURCE_DATA	PW	20:52:25	20:52:25	00:01	✓
CRT_STAG_DATA	PW	20:52:25	20:52:27	00:02	✓
PREP_MASTER_DATA	PW	20:52:25	20:52:28	00:03	✓
COMBINE_DATA	S	20:52:25	20:52:29	00:04	⚠
INSERT_DATA	S	20:52:30	20:52:30	00:01	✓
UPDT_STAG_TABLE	S	20:52:30	20:52:30	00:01	✓
COMPARE	S	20:52:31	20:52:32	00:01	✓

Total Elapsed Time: 00:16

✓ - Job completed successfully ⚠ - Job completed successfully with WARNING

✗ - Job failed - Job is still running

Result email with ERROR

```
        Process: Update Sales Data
         Module: env_setup
                 crt_stag_table
                 backup_table
                 pull_source_data
                 crt_stag_data (ERROR)
                 prep_master_data
                 combine_data
                 insert_data
                 updt_stag_table
                 compare
    Environment: prod
  ZEUS DB Schema: sgf2014
     DB Instance: PRDB101
      Process ID: 36110384
   Data Location: /sgf2014/xyz/data/adhoc
       Hostname: server1
     Time Start: 10/21/2013 9:26:02 PM
       Time End: 10/21/2013 9:26:21 PM
Date/Time Stamp: 2013_1021_212601


                  Available    Capacity
                 ===========  ==========
/home/xyz           161G          61%
/sgf2014/xyz        161G          61%
/sastemp            114G          12%


crt_stag_data_2013_1021_212601.log
    53:ERROR: File WORK.STG_DATA1.DATA does not exist.
```

Time Statistics:

Job Name	Run Mode	Submitted @	Finished @	Time Elapsed hh:mi:ss	Job Success ?
ENV_SETUP	S	19:26:04	19:26:07	00:03	✓
CRT_STAG_TABLE	QB	19:26:08	19:26:08	00:01	✓
BACKUP_TABLE	PB	19:26:08	19:26:13	00:05	✓
PULL_SOURCE_DATA	PW	19:26:09	19:26:16	00:07	✓
CRT_STAG_DATA	PW	19:26:09	19:26:14	00:05	✗
PREP_MASTER_DATA	PW	19:26:09	19:26:21	00:12	✓

Total Elapsed Time: 00:19

- Job completed successfully
- Job completed successfully with WARNING
- Job failed
- Job is still running

Summary of Job Termination

How is the entire process terminated by framework if ERROR occurred? The answer is it depends on the `run_mode`. The following table shows the actions the framework will take if a job (SAS program or SQL script) got an ERROR.

Run Mode	Action
S Q	job terminated immediately ERROR email sent entire process aborted
PB QB	job terminated immediately ERROR email sent entire process keeps running
PW QW	job terminated immediately ERROR email sent all other PW / QW jobs keep running entire process aborted after all PW / QW jobs are done

The following table shows the actions the framework will take if the Korn shell process fails.

Run Mode	Action
KW	child process terminated immediately ERROR email sent parent process terminated
KB	child process terminated immediately ERROR email sent parent process keeps running
KWG	child process terminated immediately ERROR email sent all other child processes keep running parent process aborted after all KW jobs are done

CONCLUSION

This is a high-level infrastructure illustration of a framework based on SAS. A real framework is running in our team for the daily, weekly, and monthly ETL processes. The framework is also suitable for reporting and modeling process. All the presentation materials can be downloaded from www.kevin-chung.com

REFERENCES

[1] SAS OnlineDoc® 9.2, SAS Institute Inc. Cary, NC.
http://support.sas.com/documentation/cdl/en/lrdict/64316/PDF/default/lrdict.pdf

[2] SAS 9.2 Companion for UNIX Environment
http://support.sas.com/documentation/cdl/en/hostunx/61879/PDF/default/hostunx.pdf

[2] HTML Reference
http://www.w3schools.com/html

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Feel free to contact the author at:

Kevin Chung
Fannie Mae
4000 Wisconsin Ave., NW
Mail Stop: 2H-4S/07
Washington, DC 20016
Work Phone: 202-752-1568
E-mail: kevin_chung@fanniemae.com
kchung01@hotmail.com