Paper 021-30

More Than Batch – A Production SAS® Framework

Denis Cogswell, Quality Planning Corp., San Francisco, CA

ABSTRACT

How can we automate running a large number of SAS steps on the Windows platform? Windows doesn't provide a robust batch facility equivalent to JCL in z/OS. This paper presents a framework that will:

- Run a series of SAS steps with little or no analyst intervention
- Save all SAS LOG and PRINT files to provide an audit trail
- Catch SAS errors and prevent SAS from continuing after an error
- Notify staff of errors, including the failing step's LOG as an email attachment
- Notify staff of successful completion of the job
- On rerun, restart automatically at a previously failed step
- Require no editing of SAS source for regular processing – SAS source can be read-only
- Leverage staff familiarity with SAS – no new scripting language

INTRODUCTION

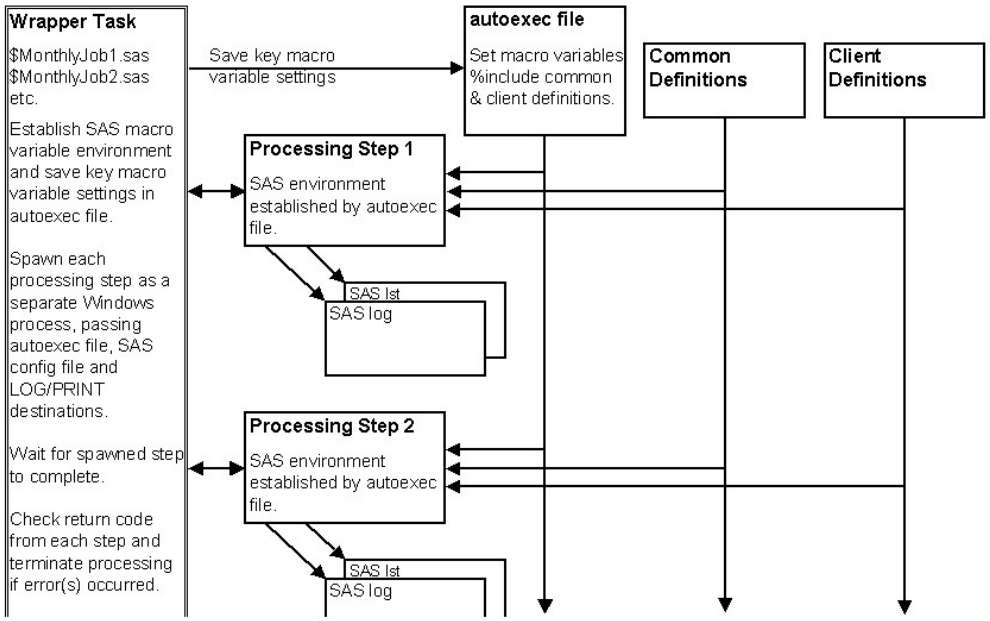
Monthly processing for each of our clients consists of fifty or more SAS steps and takes an analyst two or more days, running each step interactively. We couldn't bring in new clients without also adding staff. A production framework was needed to better utilize our staff and insure that client processing was completed successfully. In the process, we developed a consistent structure for our SAS processing environment and cleaned up error-prone code.

A SAS "wrapper" job acts as the batch executive, running each SAS step as a separate Windows process using the SAS SYSTASK statement. With ERRORABEND set in the processing steps, an error will terminate the step, but the controlling process remains alive and can notify staff, with the failed step's LOG as an email attachment. After corrective action is taken, the entire job can be rerun and the framework will automatically resume with the previously failed step.

Revealing my background in the IBM mainframe world, I call the "wrapper" code a *production job*, or the main task. Each processing step can consist of hundreds of individual SAS data steps and procedures. This figure shows the relationship between the wrapper (main) task and the individual processing steps. An autoexec file passes environmental information to the processing steps, including both common and client-specific definitions.

The SpawnTask macro does most of the work, and the production job is, for the most part, just a series of SpawnTask macro calls. With its simpler cousin DoTask, these macros are very helpful in developing a production framework for our day-to-day processing.

We'll now develop the framework in a few easy steps, starting with the SYSTASK statement. Along the way, we'll introduce additional features of the framework that provide a robust production environment.



1. Run Individual Steps as Separate Batch Tasks - SYSTASK

The SAS SYSTASK statement (available in Windows and Unix environments) executes an operating system command as a separate task, for example:

SYSTASK command "C:\SAS\SAS –input c:\SUGI30\Clients\XYZ\step1.sas" wait;

SYSTASK command "C:\SAS\SAS –input c:\SUGI30\Clients\XYZ \step2.sas" wait;

This will run step1.sas and step2.sas in succession as separate tasks. The default is NOWAIT, which would run both steps simultaneously. Each processing step begins with a clean environment with no residual work files or macro variable settings from the prior steps.

SAS isn't normally installed as C:\SAS\SAS, and no reference has been made to a CONFIG file, so the above code won't work as shown. We'll fix that in a bit.

2. Save the Batch Step LOG and PRINT Files

To provide an audit trail, we need to save the LOG (and PRINT) output for each batch step. The LOG is crucial if we have to debug a problem. We save each step's LOG file using the -LOG command line option:

SYSTASK command "C:\SAS\SAS –input c:\SUGI30\Clients\XYZ\step1.sas
–log c:\SUGI30\Clients\XYZ \data\History\log\step1.txt" wait;

SYSTASK command "C:\SAS\SAS –input c:\SUGI30\Clients\XYZ\step2.sas
–log c:\SUGI30\Clients\XYZ \data\History\log\step2.txt" wait;

This will save each step's LOG output. Add the PRINT option to save the reports also. We give the LOGs a txt file type to enable browsing via a simple text editor such as Notepad. This is also helpful when the LOGs are included as email attachments, where you may not get a choice of programs to browse them. We also save output using the –PRINT command line option, but I'm omitting that here for simplicity.

3. Check the Ending Status of the Batch SAS Tasks

A batch SAS process sets a return code (e.g. DOS batch ERRORLEVEL) that indicates the severity of problems it encountered:

Condition	Severity	Return Code
All steps terminated normally	SUCCESS	0
SAS System issued warning(s)	WARNING	1
SAS issued error(s)	ERROR	2
ABORT statement	INFORMATIONAL	3
ABORT RETURN statement	FATAL	4
ABORT ABEND statement	FATAL	5
ABORT ABEND nnn statement	FATAL	nnn

The ERRORABEND option prevents SAS from continuing after an error by internally issuing ABORT ABEND. This will end a batch SAS step with return code 5. If logical errors are detected (e.g. out of range values), we can also issue the ABORT ABEND statement to terminate the step. If your analysts are commonly ignoring SAS errors (e.g. invalid substr() arguments), you may need to instill some better programming practices.

The SYSTASK STATUS option saves the batch step's return code in a macro variable:

SYSTASK command "C:\SAS\SAS –input c:\SUGI30\Clients\XYZ \step1.sas
–log c:\SUGI30\Clients\XYZ \data\History\log\step1.txt" wait **status=Taskrc**;

%if &Taskrc > 1 %then %put Batch SAS return code &Taskrc – ERROR;

If we wrap a SAS macro around the SYSTASK statement, we can monitor the ending status of each batch step and take appropriate action if the return codes are higher than acceptable. Thus began the development of our *SpawnTask* and *DoTask* macros.

Note: SAS 9.1 has a bug (scheduled to be corrected in 9.3) that gives an error message for the status variable:
`ERROR: Attempt to %GLOBAL a name (TASKRC) which exists in a local environment.`
Just globally define the macro variable before the SYSTASK statement to circumvent this problem.

4. Detect Timeout of the Batch Step - WAITFOR

The main task should regain control if the batch step fails to complete in a reasonable length of time. The production job may run unattended and we don't want it to hang if a batch step can't complete. To do this, replace the SYSTASK WAIT option with MNAME to save the SAS-assigned task name and use the SAS WAITFOR statement to wait for the task to complete:

```
SYSTASK command "C:\SAS\SAS –input c:\SUGI30\Clients\XYZ \step1.sas
    –log c:\SUGI30\Clients\XYZ \data\History\log\step1.txt" mname=Taskname status=Taskrc;

%if &SYSRC > 0 %then %do; %put Batch SAS could not be started: SYSRC=&SYSRC;
                          [fail the job]
                      %end;

/* task was started successfully – wait for it to complete */
WAITFOR &Taskname TIMEOUT=120; /* Number of seconds to wait before assuming timeout */

%if &SYSRC ne 0 %then %do; /* Task timed out */
                           %put Batch SAS step timed out;
                           SYSTASK KILL &Taskname;
                           [fail the job]
                         %end;

%if &Taskrc > 1 %then %do; /* Task ended with errors */
                           %put Batch SAS return code &rc – ERROR;
                           [fail the job]
                         %end;
```

Without the WAIT option, SYSTASK initiates the batch step asynchronously and the main task continues processing. A nonzero value for &SYSRC from SYSTASK indicates that the command could not be executed (e.g. syntax error or insufficient resources). The SYSTASK MNAME option passes the task name to WAITFOR, which waits for completion of the batch task. The TIMEOUT value (in seconds) can allow the main task to regain control if the batch task fails to complete within the allotted time. If the batch task times out, the SYSTASK KILL statement terminates it.

5. Setup a SAS Macro Variable Environment

In the examples above, specific path names were used (e.g. C:\SUGI30\Clients\XYZ). As much as possible, we want to use common processing code among our various clients, so we need to externalize such things as path names and processing options. All items that may vary between clients are specified as SAS macro variables. Processing code uses these global symbols in the rest of the code so that only the macro variable environment needs to change to run the process for another client, or on a development server:

```
%let Client = XYZ;                                  /* Client (directory) name */
%let ServerPath = C:\SUGI30;                        /* Path to server */
%let CommonPath = &ServerPath.\Common;              /* Elements common to all clients */
%let ClientPath = &ServerPath.\Clients\&Client;     /* Path to current client's components */
%let ClientDataPath = &ClientPath.\data;            /* Current client's data */
%let ClientHistoryPath = &ClientDataPath.\History;  /* Current client's saved LOG/PRINT files */

SYSTASK command "C:\SAS\SAS –input &ClientPath.\step1.sas –log &ClientHistoryPath.\log\step1.txt"
    mname=Taskname status=Taskrc;
```

6. Use an Autoexec File to Set the Environment

A SAS macro variable environment established for the main task won't automatically be passed to the processing steps since they have their own unique environment. SAS WORK files created in the main task also won't be available to these steps. To solve this problem, we create a SAS *autoexec* file at the start of the main task and pass it to each batch step

using the AUTOEXEC command line option. The autoexec file sets key macro variables to establish the processing environment and performs any other initialization functions, such as including common and client-specific definitions.

Creating the autoexec file in the main task:

```
filename autoexec "&ClientPath.\autoexec.sas";
data _null_;
 file autoexec;
 put  "%let Client=&Client;"                          /* expand Client value */
    / "%let ServerPath = &ServerPath;"                /* expand ServerPath value */
    / '%let CommonPath = &ServerPath.\Common;'        /* don't expand ServerPath...*/
    / '%let ClientPath = &ServerPath.\Clients\&Client;'
    / '%include "&CommonPath.\CommonDefinitions.sas";'
    / '%include "&ClientPath.\ClientDefinitions.sas";';
run;
```

If the *Client* macro variable has a value XYZ, and ServerPath is C:\SUGI30, the resulting autoexec file would look like:

```
%let Client = XYZ;
%let ServerPath = C:\SUGI30;
%let CommonPath = &ServerPath.\Common;
%let ClientPath = &ServerPath.\Clients\XYZ;
%include "&CommonPath.\CommonDefinitions.sas";
%include "&ClientPath.\ClientDefinitions.sas";
```

Add the AUTOEXEC command line option in the SYSTASK command string:

```
SYSTASK command "C:\SAS\SAS –input &ClientPath.\step1.sas –log &ClientHistoryPath.\log\step1.txt
    –autoexec &ClientPath.\autoexec.sas" mname=Taskname status=Taskrc;
```

Key macro variables set in the main task are then passed to the spawned tasks. In this example, the values for *Client* and *ServerPath* were passed and the spawned tasks inherit the same settings. As the autoexec file is processed, other global symbols are defined (e.g. *ClientPath*), then the rest of the environment is established by the common and client definitions files. When each processing step starts, it will have the environment established, including global macro variables, processing options, librefs and filerefs. Note that the autoexec file doesn't have to be named autoexec.sas.

7. Dynamically Set ERRORABEND or NOERRORABEND

Previously, I urged the use of ERRORABEND toi terminate a batch step if an error was encountered. But ERRORABEND can be a nuisance if you're trying to debug an individual step interactively – an error will shut down your entire SAS session. We can set ERRORABEND or NOERRORABEND dynamically in the autoexec file, taking advantage of the fact that the AUTOEXEC option is used in the SYSTASK command string. When running interactively, the autoexec file is run manually and the AUTOEXEC option isn't used.

This bit of code sets ERRORABEND or NOERRORABEND depending on the AUTOEXEC option:

```
%macro SetErrorabend;
 %if %sysfunc(getoption(AUTOEXEC)) eq %str()
    %then %str(NOERRORABEND);
    %else %str(ERRORABEND);
%mend SetErrorAbend;
options %SetErrorabend;
%put SAS option %SetErrorabend has been set;
```

In Windows, the SAS SYSENV automatic macro variable can't be used, since it has the value FORE, even in the spawned tasks unless the NOTERMINAL option is set.

8. Provide a CONFIG File

One weakness in SAS is that it doesn't, by default, know where to find a CONFIG file. If you launch the main task using your SAS desktop shortcut, the CONFIG file is specified in the shortcut's associated command string. We need to do the same for the batch SAS steps launched by SYSTASK. One option is to point to either the standard CONFIG file, or a copy of it in another directory.

```
%let SAScfg=&ClientPath.\Client.cfg; /* Client-specific copy of standard SAS CONFIG file */

SYSTASK command "C:\SAS\SAS –input &ClientPath.\step1.sas –log &ClientHistoryPath.\log\step1.txt
    –autoexec &ClientPath.\autoexec.sas -config &SAScfg" mname=Taskname status=Taskrc;
```

A problem with this is that it's difficult to run a processing step interactively if it relies on a customized CONFIG file. My suggestion is to use the default CONFIG file and set any required SAS options through your autoexec file.

The code above still won't work, since SAS normally isn't installed as C:\SAS\SAS.exe.

9. Dynamically Locate the SAS Executable & CONFIG File

Also, the SAS executable may not be installed in the same path on each system on which you plan to run your application. The following code, at the beginning of the main task, sets macro variables for the locations of the current SAS executable and its default CONFIG file:

```
proc sql noprint;
 select path into :R from dictionary.members  where libname='SASROOT';
run;

/* Set values for SASPath and SAScfg global symbols */
data _null_;
 length path $ 120;
 path = "'" || trim("&R") || "\sas.exe" || "'";                          /* Single quotes */
 call symput("SASpath",trim(path));                                      /* Path to SAS executable */
 path = "'" || trim("&R") || "\sasv" || scan("&sysver",1,'.') || ".cfg" || "'";  /* Single quotes */
 call symput("SAScfg",trim(path));                                       /* Default SAS CONFIG file */
run;
```

We can use these macro variables for the location of the SAS executable and the SAS default CONFIG file. Note that the *SASpath* and *SAScfg* values are contained in single quotes, since they may contain blanks and will expand within a string enclosed in double quotes:

```
SYSTASK command "&SASpath –input &ClientPath.\step1.sas –log &ClientHistoryPath.\log\step1.txt
    –autoexec &ClientPath.\autoexec.sas  -config &SAScfg" mname=Taskname status=Taskrc;
```

At this point, we have implemented most of the features of our production framework. By now, the SYSTASK command string has grown long enough that we receive warnings, but the commands are successfully executed. I know of no way to suppress these warnings:

```
WARNING: The quoted string currently being processed has become more than 262 characters long.
         You may have unbalanced quotation marks.
```

10. Support Restart

Not only do we want to stop processing a batch step if an error is encountered, we also want all subsequent steps to be skipped. ERRORABEND assures that an individual step will terminate on an error, but we need to add logic to our main task to assure that it doesn't continue with subsequent steps. Once the cause of the error has been corrected, we would also like to be able to rerun the original job (the main task) and have it automatically skip down to the previously failed step to resume processing. This is done by maintaining a SASRestart macro variable in the main task, and a ClientRestart.sas

file that saves the SASrestart value for the next run. Initially, the SASRestart macro variable has a null value and the ClientRestart.sas file looks like:

%let SASRestart=;

We insure that each step has a unique name (e.g. all the step1.sas, step2.sas, etc. file names are unique). If, based on our analysis of return codes, we determine that a step has failed, the SASRestart macro variable is set to the step name (e.g. *step2.sas*, or just *step2*) and saved in the ClientRestart.sas file, which might then look like this:

%let SASRestart=step2;

The SpawnTask macro code checks the value of the SASRestart macro variable. If it is not null, the current step is run only if its name matches the SASRestart value. If the step completes successfully, SASRestart is again set to a null value and the ClientRestart.sas is rebuilt to reflect this change. In this way, after a previously failed step runs successfully, all subsequent steps are again allowed to run.

When the production job is rerun, it restores the value of SASRestart via:

%include "&ClientPath.\ClientRestart.sas";

A little logic in the SpawnTask macro then uses the SASrestart value to determine if the current step is to be executed:

```
%let stepname = [name of step to be run by SYSTASK – the file name, minus the .sas suffix];
%if &SASRestart eq %str() or &SASRestart eq &stepname %then
    %do; /* Run the current batch step */
     SYSTASK .... status=Taskrc;
     WAITFOR ...
     %if &Taskrc > 1 %then
     %do; /* The batch SAS task failed – save SASRestart value for rerun, if necessary */
      %if &SASRestart ne &stepname %then
          %do; /* We need to update the SASRestart value */
           %let SASRestart = &stepname;
           [rebuild ClientRestart.sas with new value for SASRestart]
          %end;
     %end;  %else
     %do; /* The batch SAS task did not fail – nullify SASRestart, if necessary */
      %if &SASRestart ne %str() %then
           %do; /* This was the restart step and it ended successfully – nullify current & saved SASRestart value */
            %let SASRestart = ;
            [rebuild ClientRestart.sas with null value for SASRestart]
           %end;
     %end;
    %end;
```

On a restart, the main task will skip all batch steps until the step whose name matches the SASRestart value, restarting at that point. If the step subsequently completes successfully, the SASRestart variable is set to null and the ClientRestart.sas file is correspondingly updated.

11. Notification of Errors

Since the production job may run unattended, we need to notify a responsible analyst if a step fails so that diagnostic and corrective action can begin immediately. If the macro code that analyzes the SYSTASK and WAITFOR status codes detects that a step has failed or timed out, an email notification is sent to the responsible analyst, including the LOG of the failing step as an attachment. The analyst can browse the attached LOG, take appropriate corrective action, and schedule a rerun of the production job. Add a SASemail macro variable to the ClientDefinitions member and a little more logic to the SpawnTask macro:

%let SASemail = dcogswell@qualityplanning.com; /* in ClientDefinitions.sas */

```
%let log = &ClientHistoryPath.\log\&stepname..txt; /* Path name for saved LOG */

SYSTASK command "... -log=&log" status=Taskrc;
WAITFOR ...;

%if &Taskrc > 1 %then
    %do;
     filename notify email "&SASemail"
       subject="Monthly &Client client processing failed in step &stepname" attach="&log";
     data _null_;
      file notify;
      put "Step &stepname ended with return code &Taskrc.."
        / "Review the attached LOG file, make corrections, and rerun the production job."
     run;
    %end;
```

12. Notify Analysts of Successful Completion

Since a production job may run for several hours, and may even be scheduled to run overnight, a final step is added to the production job to send a notification when the entire process completes successfully:

```
%include "&ClientPath.\NotifyCompletion.sas";
```

The analysts can monitor several production jobs, taking action only when an error occurs or some manual step is needed between jobs.

Recent Outlook security enhancements create a dialog whenever SAS attempts to send email. I know of no way to disable this dialog, and the SAS step would timeout if run unattended. Fortunately, a little utility can auto-reply to this dialog – find it at: http://www.snapfiles.com/get/clickyes.html.

13. Use Autocall to Find Macros

At this point the SAS code implementing the production framework is nearly complete, but there are a few additional structural issues that should be mentioned.

SAS macros should be found via autocall, not hidden inside %included files. A client-specific macro library is placed ahead of the common macro library using the SASAUTOS option:

```
/* Setup autocall macro search path for current client */
options sasautos=("&ClientPath.\Macros"
          "&CommonPath.\Macros"
          SASAUTOS) ;
```

This allows a default macro from the common library to be over-ridden by a client-specific version in the client macro library. For example, a client may need additional lines in its autoexec file to pass additional global symbols, and a *ClientAutoexec* macro in the BuildAutoexec.sas code can insert them. Those clients not needing these extra definitions would use a dummy *ClientAutoexec* macro that would be found in the common library.

14. Common and Client Formats

User-defined SAS formats are useful for translating or validating values in addition to the traditional value display. As with macros, we want to support a common set of SAS formats, plus a set of client-specific formats, some of which may override their common versions. The framework uses a common format library and a format library for each client. The FMTSEARCH option specifies the search order for locating SAS formats:

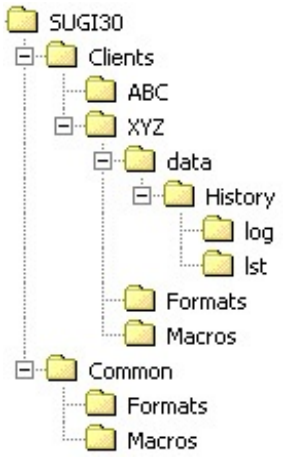
Search order:
1. Client-specific formats
2. Temporary formats created in a processing step
3. Common formats
4. SAS-provided formats

```
/* In CommonDefinitions.sas */
libname library "&CommonPath.\Formats";

/* In ClientDefinitions.sas */
libname formats "&ClientPath.\Formats";
options fmtsearch=(formats WORK library);
```

15. A Consistent Directory Structure

The SAS processing steps use macro variables to specify directory paths and processing options. Processing can be moved to another server or run in a test environment by changing only a few macro variable settings. Where appropriate, the production framework should have a consistent directory structure between clients. This simplified directory structure is used by way of demonstration:



Macro Variable	Sample Value	Description
Client	ABC, XYZ, etc.	Client (directory) name
ServerPath	C:\SUGI30	Path to server where applications reside
CommonPath	&ServerPath.\Common	Path to common components
ClientPath	&ServerPath.\Clients\&Client	Path to client directory on server
ClientDataPath	&ClientPath.\data	Path to client data directory
ClientHistoryPath	&ClientDataPath.\History	Path to client History directory

If time permits, a demonstration will show how a series of steps can be run in the framework, how step failures are handled, and how restart resumes at the failed step.

16. Put It All Together

The **SpawnTask** macro incorporates most of the functions described above. The production job just needs some preliminary setup code to make it ready to run the various processing steps:

```
/*-------------------------------------------------------------------------------------------------------------------------------------------*
* Production Framework – Main Task  - Client XYZ                                                                                             *
*-------------------------------------------------------------------------------------------------------------------------------------------*/
%let Client = XYZ;                                  /* Set Client (directory) name */ */
%let ServerPath = C:\SUGI30;                        /* Define path to server where components reside */
%let CommonPath = &ServerPath.\Common;              /* Path to common components */
%let ClientPath = &ServerPath.\Clients\&Client;     /* Path to current client's components */
%include "&CommonPath.\CommonDefinitions.sas";      /* Common definitions */
/* Run to here to run an individual step interactively ---------------------------------------------------------------------------------------*/
%include "&ClientPath.\ClientDefinitions.sas";      /* Additional client definitions, processing options, etc. */
%include "&ClientPath.\BuildAutoexec.sas";          /* Build the autoexec file */
%include "&ClientPath.\ClientRestart.sas";          /* Restore SASRestart macro variable */

%SpawnTask(file="&ClientPath.\step1.sas",...);      /* Run step1 as a batch task */

%SpawnTask(file="&ClientPath.\step2.sas",...);      /* Run step2 ... */

%SpawnTask(file="&ClientPath.\step3.sas",...);      /* etc... */

%include "&ClientPath.\ NotifyCompletion.sas";      /* Notify on completion of job */
```

A long series of SAS steps can be run in this way, leaving the analyst free to do other tasks.

17. Debugging

The individual SAS steps within this framework can't be run interactively by themselves since they expect an environment established by the autoexec file. At the beginning of each processing step, we add the following lines (comments):

```
/* Run the following line (excluding asterisk) to run interactively */
*%include "&ClientPath.\autoexec.sas";
```

To run a step interactively:

1. Load the production job containing the step to be debugged and run just a few introductory statements to set key macro variables (e.g. Client and ServerPath) and %include CommonDefinitions. See "*Run to here*…" comment, in the production job, above.

2. Load the step to be debugged and run the %include for the autoexec file. Normally this is just a comment, but for interactive debugging, we execute it manually. If you may need to edit the SAS source, reset the read-only attribute, if necessary, before loading.

3. The rest of the processing step can be run interactively. Since the autoexec file will not have been passed via the AUTOEXEC command line option, NOERRORABEND should be set so an error won't kill your SAS session.

4. Remember, however, that LOG and PRINT files won't automatically be saved and the default CONFIG file wil be used when you run individual steps interactively.

18. Better Than X – The DoTask Macro

It's occasionally necessary to execute other commands (e.g. PKZip, copy) and these have previously been implemented by the SAS X statement. However, X provides no feedback of the success or failure of the underlying process. Since a production platform must be able to monitor the results of each step, X is not adequate in this framework.

Most applications or operating system commands set a return code (a.k.a DOS ERRORLEVEL). Just as with our SAS steps, we can use SYSTASK & WAITFOR to process these commands.

DoTask, a simplified version of SpawnTask, can issue these commands and provide feedback so we can be assured that the command was successful (e.g. PKZIP successfully created the compressed file, COPY successfully copied the specified file). Since these would be part of a larger SAS processing step, DoTask has an ABORT ABEND statement to force termination of the processing step if the command failed.

19. Synchronizing Manual Steps

One final note on a structural issue with our production jobs:

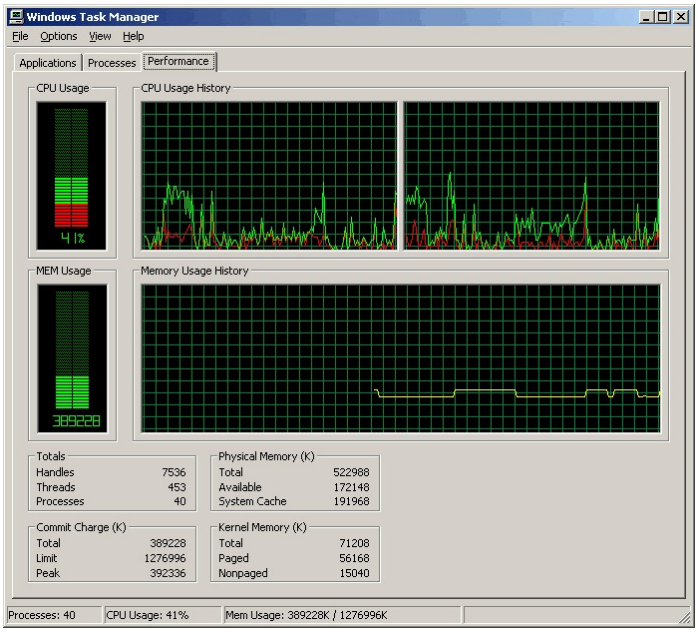
Our processing makes use of external data services such as Lexis Nexis. In most cases, data is sent to these services via FTP or email attachment. It may take a day or two after a file is sent for the results to appear on an FTP site or as an email attachment. Until we have a way of automating these processes, they remain as manual steps. Because of this, our production jobs usually split into two phases. Phase one reads client input, selects data worth analyzing, builds files for external data services and sends the files to them. When these services return their results, the files are saved in designated directories, and phase two of the production process can begin.

Trigger files are used to indicate that manual steps have been completed, one trigger file for each manual process such as retrieving MVR data from Lexis Nexis. Phase two of our production processing begins with a step that looks for these files and waits until they're all present. Phase two can be automatically scheduled, perhaps two days after phase one completes, but if one of the external services is late, the processing will wait until it is completed as indicated by the presence of its trigger file. Using a suitable timeout value for this initial step assures that phase two won't wait forever if a manual process has been forgotten.

20. Performance

Running two instances of SAS on a desktop PC is not a major performance problem. In the production framework, only one instance is active – the main task is normally waiting for each spawned task to complete. With SAS 9.1, memory usage is nearly flat, at least as far as Win 2K Performance Monitor reports it.

This Task Manager screen was captured and edited in Photoshop while the production framework was running, so it is not necessary to dedicate a system to the production framework.



Conclusion

We now have a framework that automates the task of running a large number of SAS steps, monitoring each step for successful completion and notifying an analyst if an error occurs. The analyst is freed from the tedium of running each step manually and only needs to respond to correct errors or check results at completion of the entire process.

The production framework requires better programming practice, eliminating all errors that might previously have been ignored, but that's a good thing anyway.

For a copy of the SpawnTask and DoTask macros, plus a sample production job, pick up a diskette or send me an email.

Contact Information

Denis Cogswell
Quality Planning Corp.
595 Market St. Suite 950
San Francisco, CA 94105
Work Phone: 415-281-2889
dcogswell@qualityplanning.com