MULTIPLE APPLICATIONS IN SAS® SOFTWARE
Jonathan M. Siegel, Trilogy Consulting Corporation, Kalamazoo, MI

ABSTRACT

SAS Software allows users to open multiple application windows at the same time, giving users power and flexibility to extend and combine applications. However, developers should provide safeguards when you design applications which may be open in conjunction with others. This paper illustrates the benefits of having multiple applications. It explains why safeguards are needed and how to provide them. Examples emphasize the IBM OS/2® operating system, but apply generally.

INTRODUCTION

In many windowing environments, users are accustomed to having several applications open on their workstations at once, and freely moving back and forth among them. The SAS system has several ways to permit multiple SAS applications to be open at once.

• In many operating systems, multiple SAS sessions can be run at once, each containing its own application.

• Since Version 6.07, the SAS System has provided the AFAPPLICATION (AFA) command to permit multiple applications to be open at once, each in its own window.

• The SAS Display Manager contains numerous windows which can be used as tools to get a better look at data, each of which can be open simultaneously.

In practice, the current SAS environment does not fully support multiple applications. Under some circumstances, application can "crash" into each other, affecting each other's behaviour in unexpected ways. This paper illustrates some techniques developers can use to build protection into their applications, and to prevent difficulties from occurring.

BENEFITS OF MULTIPLE APPLICATIONS

• Users engaged in several different activities can have applications for each activity available quickly, without having to navigate complex menus, or wait for applications to be repeatedly invoked and shut down. Users can switch tasks in a natural, irregular order.

• Background tasks can execute continuously, while users are interacting with the system. This enables the SAS System to be used in applications to monitor and control processes.

• A variety of SAS tools can be used from within applications to examine and correct data.

AN EXAMPLE

Our first example is a SAS/AF® frame, containing a graphics object which displays Sales by Region, using SAS/GRAPH® software. The graph uses dataset GRAPHDAT.SALES. You wish to update the graph. Here's how:

1. Type FSEDIT GRAPHDAT.SALES directly on the command line (or use the PMENU).
2. Using SAS/FSP® software, edit relevant data points in the FSEDIT window.
3. Return to the FRAME and click the mouse on the graph. The graph will update.

This example helps illustrate the power of multiple windows. Knowledgeable users can combine developed applications with built-in SAS tools to extend the power of their applications without additional programming.

The frame requires a single line of SCL code per object to permit dynamic updating.

```sas
GRAPH1:
   call notify('GRAPH1','_update_');
   return;
```

WHY IS CAUTION NEEDED?

The example below shows what can go wrong when two apparently unrelated SAS/AF applications are run in the same SAS session.

The first application shows total sales by region and lets the user select a region. The second application shows the employees under each manager. When run separately, both applications work flawlessly. Yet if the second application is invoked from within the first, the first one will fail. The on-screen demonstration illustrates what happens.

Why did they fail? The answer can be found by looking at each application's SCL code. If we look inside the Sales Total application, we see the following:

```sas
INT:
   submit continue sql;
   create table temp as
   select REGION, sum(sales) as TOTAL
   from TEST.SALES
   group by REGION
   endsuite;
   return;
```
Note that the INIT section creates and uses a temporary data set called TEMP.

The SCL code for the second SAS/AF application looks like this:

```sas
INIT:
submit continue sql;
create table temp as
select emp.empid as empid, emp.name as empname,
emp.title as emp-title, man.empid as
manid, man.name as manname, man.title as
mantitle
from test. employee as emp
left join test. employee as man
on emp.manid eq man.empid
endsubmit;
RETURN;

SELECT:
tempid = open('TEMP');
empname • datalistc(tempid,
'EMPNAME ENPTITLE MANNPNAME',
'Please Select Employee', 'Y');
rc • close(tempid);
call notify('EMPLOYEE', '_set_text_·, empname);
call notify('MANAGER', '_set_text_·, manname);
RETURN;

VIEW:
call fsview('TEMP');
RETURN;

TERM:
RETURN;
```

Looking closely at both applications, we see that both begin by creating a data set called WORK.TEMP. This data set is used in both applications. This is our problem. When we invoke the first application, WORK.TEMP is created in the INIT section. Then, when we invoke the second SAS/AF application, another data set with the same name is created, overwriting the first one. When we return to the first window, variables which the first application expects to find in the data set are no longer there. The first application fails. While each application, taken separately, ran smoothly, the combination resulted in a fatal crash.

SHARED ENVIRONMENTS: BASIC CONCEPTS

Some concepts, analogous to ones in SAS/SHARE® Software, can help us understand how multiple applications can bump into each other even though each one, taken separately, runs perfectly well by itself. We will use these concepts to explain how to design applications to enable them to coexist.

Each invocation of the SAS system can be thought of as a session. Some environments, like the IBM OS/2 Operating System, permit multiple sessions on the same workstation at the same time.

Each application running within a Display Manager window can be thought of as a task. In each of our two examples, two tasks were running within the same session. Tasks can include built-in SAS windows as well as enduser applications.

Every component of the SAS system which is used by a task -- data sets, the SUBMIT queue, the screen itself, anything which is available to tasks to do work, is a resource. In our second example, a crash occurred because two tasks needed write access to the same resource at the same time.

Every resource has a resource manager which works as an intermediary between tasks and the resource itself. The SAS system has many resource managers built into its design. Database engines handle data: Display Manager handles the screen; the Session Manager routes SUBMIT requests, etc.

Many SAS resource managers were designed to handle requests from multiple tasks. SAS/SHARE software, for example, was designed to handle contentions for the same dataset. Display Manager was designed for multiple windows. The resource managers for some SAS resources, however, were not designed with multiple tasks in mind.

Potential conflicts occur when a resource must be shared by two or more tasks, but its resource manager was not designed to recognize requests as coming from different places, and does not handle contention for the same resource.

SOLUTIONS

There are three basic strategies to avoid conflict:

(1) If the operating system permits, run applications in separate SAS sessions.

(2) Create your own resource manager when resources must be shared.

(3) Use defensive coding to prevent applications from interfering with each other.
SEPARATE SAS SESSIONS

A number of operating systems, such as OS/2 and Unix, permit more than one SAS session to run on the same workstation at the same time, and provide operating-system level protection for each session. In these environments, if sufficient memory is available on the workstation, each application can be run under its own SAS session, virtually independently of the others.

Putting each application in its own separate, independent SAS session is usually the simplest, least laborious, and most effective means of allowing multiple applications to run together without interfering with each other.

In addition, using independent SAS sessions is the only way to have background tasks running independently at the same time as a foreground task. Within a single SAS session, only one window can be active (actually executing code) at the same time: each application "sleeps" until the user activates it.

Here is how to set up SAS applications to run under separate SAS sessions in OS/2:

1. Create a separate AUTOEXEC file for each application, setting up appropriate LIBREVs and options, and invoking the application. The file does not have to be called AUTOEXEC.SAS: it can (and should) have a different name for each application.

2. Create a separate SAS configuration for each application by copying and customizing the standard CONFIG.SAS file. The configuration file need not be called CONFIG.SAS: it should have a different name for each application. Each application's SAS configuration file should contain the following:

   a. A -WORK option which does not contain the USE suboption, so that the SAS system will create a separate subdirectory for each application. (Since SAS will take care of generating unique subdirectories, all applications can point to a common work directory.)

   b. A -SASUSER option pointing to a unique SASUSER directory for each application. This enables function keys, windows, etc. to be changed in one application without affecting any others.

   c. An -AUTOEXEC option pointing to the autoexec file created in (1) above.

3. Create a separate icon on the OS/2 desktop or folder for the application. To do this:

   a. Right-click on the icon for the SAS system.

   b. Select "Create Another" from the popup menu.

   c. Select the target folder for the application.

   d. Under "New Name", give the application an appropriate name. Click the "Create" button.

   e. Under "Path and file name", enter the full path name of the SAS.EXE file. If SAS is installed on the C: drive with default options, this file is C:\SAS\SAS.EXE.

   f. Under "Parameters", enter -config <configfile>, where <configfile> is the full path name of the application's configuration file. Example:

   -config c:\myapp\programs\mycfg.sas

   g. An icon will appear on the screen which, by default, is the same as the one used to invoke the SAS system. If you wish, you can substitute an icon specific to your application.

Using this method puts SAS applications under separate icons, enabling users to access them the same way they access any other application on their PC. Applications are substantially protected from affecting each other in unexpected ways.

Putting applications in separate SAS sessions is the safest, most convenient, and most powerful way to let users run multiple SAS applications at the same time.

RESOURCE MANAGERS

Using multiple SAS sessions is not always possible. Some operating systems, like CMS, do not permit a single user to run multiple SAS sessions. In addition, there are resources which can be accessed by only a single SAS session. In such cases, there are programming techniques available to prevent specific types of interference when multiple applications are run in different windows in the same SAS session.

EXAMPLE: A DATA SET NAME MANAGER

The NAME method is a short SCL method that creates unique data set names. It will prevent the type of conflict discussed at the beginning of the paper.
The NAME method can be used in the following manner:

```sas
/* NAME Method */
NAME: method
   name $17 /* The name of the work data set */
   i = 1;
do while (exist("WORK.T"||trim(left(put(i, 7.,)"DATA")))
      i = i + 1;
end;
name = "WORK.T"||trim(left(put(i, 7.,))); endmethod;
```

The NAME method can be used in the following manner:

```sas
/* Get unique data set name */
length temp $D;
call method('methods.scl', 'name', temp);
submit continue;
data &:temp;
run;
endsubmit;
```

Observe that after the NAME method is invoked, the SCL variable TEMP contains the unique data set name.

**EXAMPLE: A REMOTE SESSION MANAGER**

One case where multiple sessions are impossible is when SAS applications must share a remote communication session under SAS/CONNECT® Software. Usually, multiple SAS sessions cannot share a remote host session.

This problem is not a trivial one. As client/server applications become more important in business information structures, it will be increasingly likely that data needed for a local application will be stored remotely.

If multiple SAS applications are run in a single SAS session, however, they can interfere with each other. For example, if one application signs off the host, it will leave the others hanging.

The following resource manager, an SCL method called REMOTE, can prevent this problem. Applications can use the REMOTE method to perform SIGNONs and SIGNOFFs. It maintains a count of the number of applications signed on to each remote session. If another application has already signed on, the REMOTE method simply increments a counter for that session. When an application signs off, it decrements the counter. It only issues a 'SIGNON' or 'SIGNOFF' for the first and last application to sign on or off.

The REMOTE method also takes care of selecting the script and other ancillary information SAS/CONNECT software needs to access remote sessions. This insulates applications from the details of session management.

This is another benefit of resource managers: in addition to protecting applications from interfering with each other, they can keep track of the physical details necessary to access a resource, simplifying applications programming.

The REMOTE method accepts three arguments: the command to execute (SIGNON or SIGNOFF), the session id to sign on or off to, and a variable to hold the status on return. The status will be 0 if the command executed successfully, nonzero if it didn't.

The REMOTE method updates a dataset (SYSMAN.REMOTE) which stores information about remote sessions. The data set has one observation per available host session. In this example, it has three variables: SESSION (the remote-session-id), SCRIPT (script file name), and USERS (number of tasks signed on). SYSMAN.REMOTE must be created in advance. The REMOTE method uses the USERS variable to keep count.

At the start, the data might look like this:

<table>
<thead>
<tr>
<th>SESSION</th>
<th>SCRIPT</th>
<th>USERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C:\SAS\CONNECT\SASLINK\HOSTA.SCR</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>C:\SAS\CONNECT\SASLINK\HOSTB.SCR</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>C:\SAS\CONNECT\SASLINK\HOSTC.SCR</td>
<td>0</td>
</tr>
</tbody>
</table>

Here is the SCL code for the REMOTE method:

```sas
/* Status is -1 during processing */
status = -1;
/* Make sure the command is in upper case */
command = upcase(command);
/* Set the libref if it hasn't been set */
if not(libref('SYSMAN') then
   rc = libname('SYSMAN', 'c:\user\sugi19\sysman');
/* Check to see if the dataset SYSMAN.REMOTE exists. If not, we cannot continue */
if not(exist('SYSMAN.REMOTE')) then do;
   status = 1;
   return;
end;
/* Open the dataset in update mode */
rdata_h = open('SYSMAN.REMOTE', 'U');
if rdata_h eq 0 then do;
   status = 3;
   return;
end;
/* Make sure session requested is on the dataset */
rc = where(rdata_h, 'session eq '||session||')';
if rc gt 0 then do;
   status = 3;
   call method('remote.scl', 'cleanup', rdata_h, status);
   return;
end;
/* Fetch the observation */
rc = fetch(rdata_h);
/* Make sure it was fetched */
if rc eq 0 then do;
   status = 4;
end;
```

```sas
length script $200 /* Name of script file */
users 8 /* Number of tasks on session */
```

REMOTE:
method
   command $8 /* 'SIGNON' or 'SIGNOFF' */
   session $8 /* The remote session id */
   status 8 /* status on return */
   /* Status is -1 during processing */
   status = -1;
   /* Make sure the command is in upper case */
   command = upcase(command);
   /* Set the libref if it hasn't been set */
   if not(libref('SYSMAN') then
      rc = libname('SYSMAN', 'c:\user\sugi19\sysman');
   /* Check to see if the dataset SYSMAN.REMOTE exists. If not, we cannot continue */
   if not(exist('SYSMAN.REMOTE')) then do;
      status = 1;
      return;
   end;
   /* Open the dataset in update mode */
   rdata_h = open('SYSMAN.REMOTE', 'U');
   if rdata_h eq 0 then do;
      status = 3;
      return;
   end;
   /* Make sure session requested is on the dataset */
   rc = where(rdata_h, 'session eq '||session||')';
   if rc gt 0 then do;
      status = 3;
      call method('remote.scl', 'cleanup', rdata_h, status);
      return;
   end;
   /* Fetch the observation */
   rc = fetch(rdata_h);
   /* Make sure it was fetched */
   if rc eq 0 then do;
      status = 4;
   end;
```
When applications use REMOTE.SCL to sign on and off remote sessions, they will not disconnect each other from the host session.

DEFENSIVE CODING PRACTICES

The basic concept of defensive coding practice is to realize that at any time, the user may switch to another application which may do things which will affect yours. Because there is only one Session Manager (and one Remote Session Manager for each remote session), some other application's code might be submitted just before or just after yours is. Similarly, if users switch to other applications, they can change function keys, system options, and other parameters.

Here are some suggestions for coding practices which will make applications less likely to interfere with each other.

- Always associate a KEYS entry with each application window in your application. That way, you can be certain that function keys will remain the same even if another application changes them. Do not rely on defaults.

- Always submit a complete step when SUBMITting to the SAS Session Manager or Remote Session Manager. Always begin and end it with RUN; QUIT; statements. This will prevent isolated SAS statements from different applications from becoming entangled with each other. Each block of code you submit should begin and end on a step boundary.

Note: Each SAS task has its own preview buffer, which is not affected by applications. If you submit code to the preview buffer (plain "SUBMIT;" statement), you can submit partial blocks. However, if you submit code with a when option (CONTINUE, IMMEDIATE, PRIMARY, or TERMINATE), you should be sure that the statements in the preview buffer represent a complete step, starting and ending on a step boundary.

- Store titles, footnotes, options, etc. in SCL variables. Reissue them each time you submit code. That way, they will be correct for your application even if they were altered in another. Clear all title and footnote lines following the last one you issued. For example, if your application uses title lines 1 and 2, issue a TITLE3; statement.
CONCLUSION

As seen above, having multiple applications open at once involves potential difficulties. The best way to deal with them is to protect applications from each other by letting each application run in its own SAS session. This method also permits background applications to execute SAS code simultaneously with foreground tasks.

When multiple applications are running in the same SAS session, a number of coding techniques can be used to build partial protection into programs. These techniques require effort, and the advantages of allowing multiple applications to run simultaneously may or may not be worth the cost. The current AFA environment is not fully robust.

This paper has explained how to create a simple resource manager. In addition to ensuring that resources are used safely, resource managers can also insulate application programs from the details of managing the resource.

The ability to safely run multiple applications at the same time provides an important benefit to SAS users, particularly those used to windowing, multi-application environments. SAS Institute, and the SAS user community, should facilitate its safe and effective use.

REFERENCES


ACKNOWLEDGEMENTS

SAS, SAS/AF, SAS/GRAPH, SAS/FSP, SAS/SHARE, and SAS/CONNECT are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. OS/2 is a registered trademark or trademark of International Business Machines Corporation. ® indicates USA registration.

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

I am grateful to Raymond Barnes, Peter Lord, Andrew Norton, and John Steurwold for their contributions to this paper, and to Trilogy Consulting Corporation and The Upjohn Company for making this paper possible.

CONTACT

Jonathan M. Siegel
Quality Systems Developer
Trilogy Consulting Corporation
5148 Lovers Lane, Kalamazoo, MI 49002
Phone: (616) 344-4100 or (616) 384-2382
Fax: (616) 344-6849 or (616) 384-2295