Paper 228-2009

Power You Didn't Know You Had: Using PSAX to Create Self-Generating Web Libraries with SAS®

Turner Bond, Missouri Department of Transportation, Jefferson City, MO

ABSTRACT

Just as the Ajax development style revolutionized the Web with Web 2.0, SAS® has innate strengths that make it ideal for building drill-down libraries of Web information. Using the "PSAX" development style—pre-synchronous SAS and XML—you can push a Web-based information channel out to the Web, often with more primitive IT architecture than you would use for a Web 2.0 application. Whether you need an online information system to track highway construction bidders, or a modern, Web-based skin to present standardized pharmaceutical testing, you can use PSAX to push your analysis to a user-friendly Web space. PSAX involves the smart use of SAS tools you probably already know. These tools range from simple, computed links in PROC REPORT, to SAS code that writes HTML directly, to a Java script generated in SAS that pulls in JSON or XML data and harnesses Ajax technology. The strength of PSAX is using the self-aware, code-generating tools inherent in SAS to integrate analysis and Web generation in a way that other tools simply cannot.

INTRODUCTION

In 2005 Jesse James Garrett wrote an article about an emerging software development concept that was changing the face of the web-based world – web applications that run a server in the background to share the load, without waiting on the user. He called this output development style – Ajax – Asynchronous Javascript And XML. This article is about a programming style that may do the same for your work in the world of SAS. I call this style PSAX – Pre-synchronous SAS and XML. PSAX involves doing things hours or days ahead of time in a synchronized way that fits together as each process completes. While PSAX is applicable to many 4GL languages and enterprise information systems, it is particularly well suited for SAS.

Using the PSAX strategies outlined below an organization can use SAS to leverage some very limited resources into a comprehensive, professional online library of information, rich with interactive links between reports.

WHY PSAX ?

The power of Ajax was in its ability to manage time while the user was visiting the site. By initiating processes milliseconds ahead of time and letting them run independently on a remote server, Ajax conquered the human barrier of impatience. Before Ajax, online trip-maps were possible, but crawled along too slowly to match the speed of human affairs. With Ajax, a Yahoo ™ map is surprisingly fast and agile because it's always working in the background to bring in the map panels it thinks you'll need next. As you pan to the right, small image panels download in the background, ready to feed the eastern edge of your screen. Because Ajax takes advantage of the XMLHttpRequest function to initiate complex routines in the background, the map images you'll need are pulled from a library of millions without bogging down your browser.

So how clever was the Ajax idea? The XMLHttpRequest function was always there. Someone just needed to hook it up to meaningful routines on a server and let them run on their own. It's no mystery that a server can pull data faster than your PC's browser … right? Multi-tasking is a skill that any home-maker, executive or parent can appreciate. So how smart was that? In some ways it wasn't very smart at all. It didn't use new tools. It just used old tools to make things that perform in a new way … like the first person who used woodworking tools to carve a wheel.

PSAX may not be as clever as inventing the wheel, but, like Ajax it involves recognizing that we have more choices than we think we do with regard to timing. "Presynchronous" is the critical word in PSAX. We often use tools like SAS IntrNet, Ruby, etc. to create, interactive pages that chew away and – moments later – update our web-page with a momentary report. What we don't think to do is combine these two approaches into something new. PSAX involves creating pages, often static pages, but creating them ahead of time. Thousands of them. Thousands of pages generated automatically and automatically linked together ahead of time by the machine. By building them in advance you can create an interactive web library that is as fast or faster than an Ajax application, but was prepared slowly enough to allow involved analysis ahead of time. This approach wouldn't work for an on-line checking account, but it does work for far more applications than you might think. Sometimes you need immediate, real-time data, but for many applications data that is 24-hours old will do.

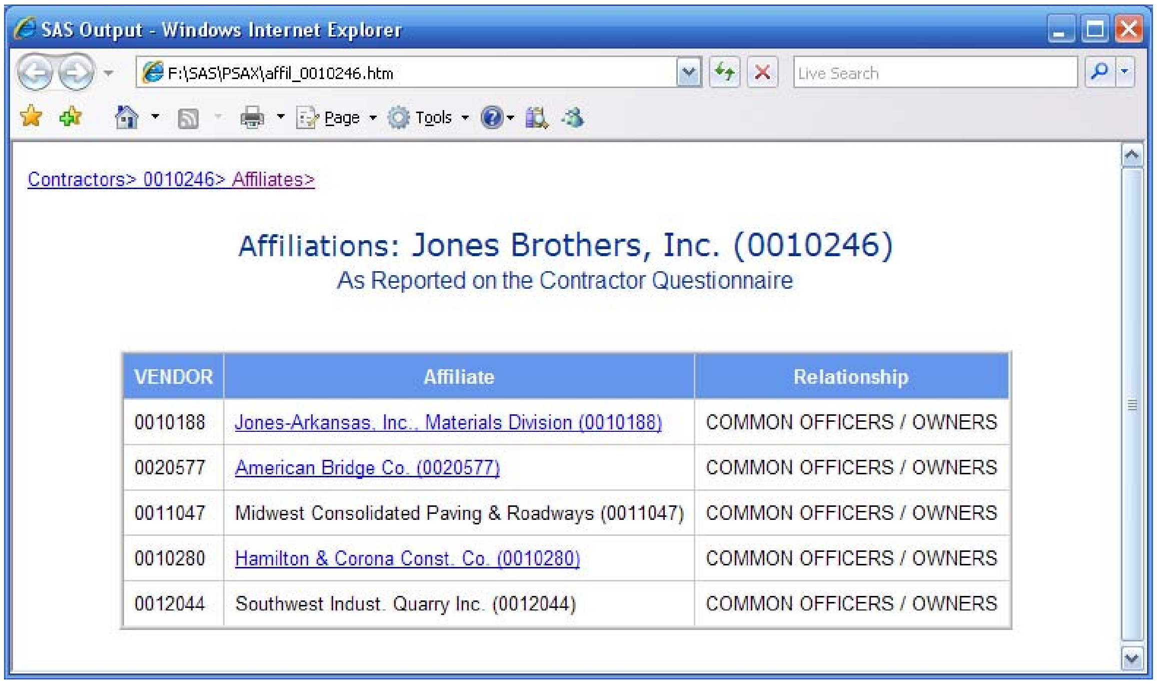
So PSAX embodies the following elements:

1. Recognizing situations where day-old data will do.
2. Building things ahead of time rather than trying to manage time while the user is on line.
3. Thinking through a predictable system of links that can be deduced and automatically created by a machine.
4. Thousands of reports that would take too long to wait for from interactive routines.
5. A user that wants to rapidly browse through a library of information at will.
6. Scheduled sessions running a self-aware programming language such as SAS.
7. Creating an open, seamless information channel that is easily linked to by approved users through normal working papers and reports.

Like Ajax, PSAX requires a paradigm shift. Where Ajax got it's power from sharing the load in real time, PSAX gets it's power from sharing the load *across* time. PSAX involves building the user experience ahead of time. PSAX is the art of preparing data, datamarts and screens ahead of time in a way the user is likely to want and then linking them together. PSAX places greater value on the analysis of data than the instantaneousness of data. Information that is 24 hours old, but processed into something more meaningful is worth it, if you can flip through it fast. While screens that were refreshed the night before would not be helpful in ordering online from Amazon™, they can be very useful in powering an analysis-based system where you need more than field values replicated to a screen.

CREATING LINKED REPORTS IN SAS

Most SAS programmers know how to output to a web-page format by wrapping their reporting routines in two lines of ODS code. What many don't realize is that SAS provides equally rich tools for embedding links to other pages. From custom templates or custom code, to the html= option in graphics procedures, to computed links in proc report there are many linking options in SAS. The simplest might involve using proc report and defining the links with compute variables. Example given below:


SAS Output - Windows Internet Explorer
F:\SAS\PSAX\affil_0010246.htm
Live Search
Page
Tools

Contractors> 0010246> Affiliates>

Affiliations: Jones Brothers, Inc. (0010246)
As Reported on the Contractor Questionnaire

VENDOR	Affiliate	Relationship
0010188	Jones-Arkansas, Inc., Materials Division (0010188)	COMMON OFFICERS / OWNERS
0020577	American Bridge Co. (0020577)	COMMON OFFICERS / OWNERS
0011047	Midwest Consolidated Paving & Roadways (0011047)	COMMON OFFICERS / OWNERS
0010280	Hamilton & Corona Const. Co. (0010280)	COMMON OFFICERS / OWNERS
0012044	Southwest Indust. Quarry Inc. (0012044)	COMMON OFFICERS / OWNERS

The report values are linked to other pages in the library and a title statement in the upper left corner creates a three-stage series of "bread crumbs" you can click on to return to previous spots in the library.

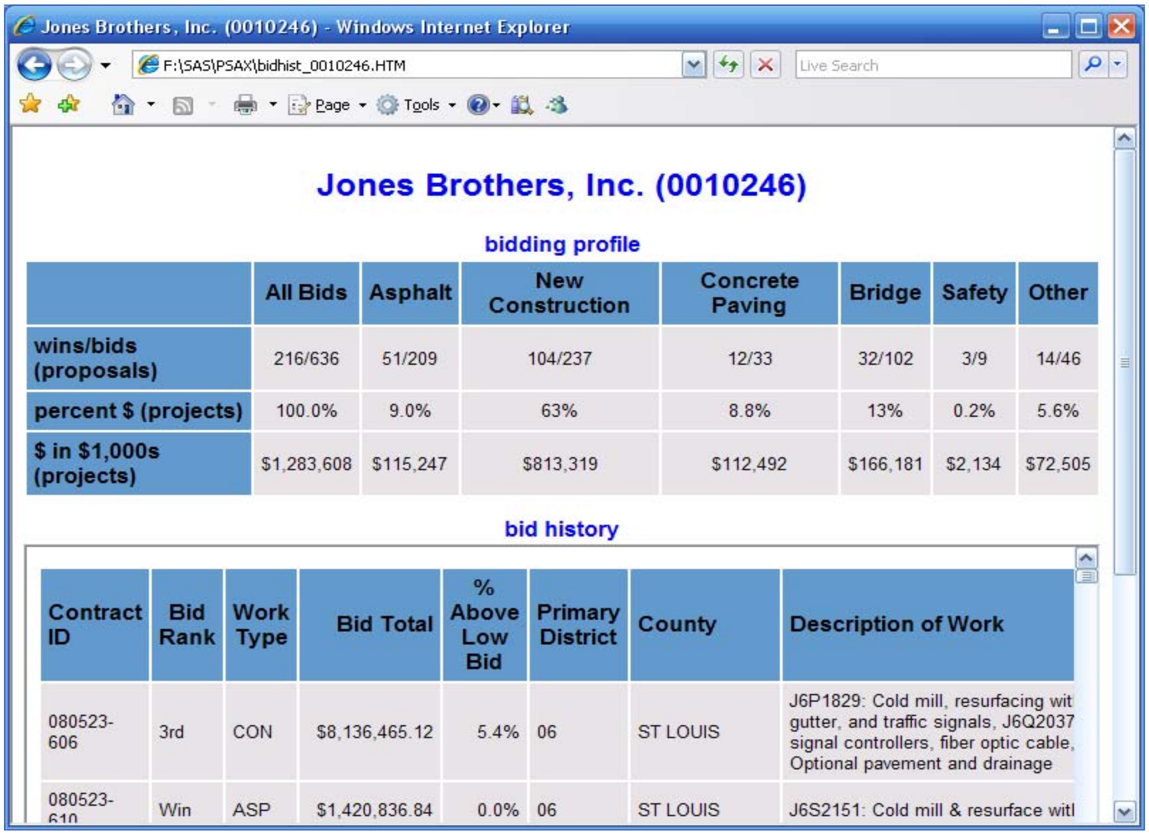
```
   /*** define HTML link for vendor **/
compute afflname ;
  if siteflg = 1 then do ;
    /* construct tag name */
    href = "..\" ||  trim(left(affil)) || "\V" || trim(left(affil))
|| ".htm" ;
    call define(_col_, "URLP", href);
  end ;
endcomp;
```

Code example 1: Computing the links in proc report

```
title1 j=l font='Arial' h=10pt color ='ltgray'
link = "../VLIST.htm"   "Contractors> "
link = "V&vendor..htm"   "&vendor> "
link = "affil_&vendor..htm" "Affiliates> "
;
```

Code example 2: Creating "bread-crumbs" at the top of the page with a title statement.

More complex uses are possible too. The following web page is created using "put" statements to push pure html code out to a file:


Jones Brothers, Inc. (0010246) - Windows Internet Explorer
F:\SAS\PSAX\bidhist_0010246.HTM
Live Search
Page
Tools

Jones Brothers, Inc. (0010246)

bidding profile

	All Bids	Asphalt	New Construction	Concrete Paving	Bridge	Safety	Other
wins/bids (proposals)	216/636	51/209	104/237	12/33	32/102	3/9	14/46
percent $ (projects)	100.0%	9.0%	63%	8.8%	13%	0.2%	5.6%
$ in $1,000s (projects)	$1,283,608	$115,247	$813,319	$112,492	$166,181	$2,134	$72,505

bid history

Contract ID	Bid Rank	Work Type	Bid Total	% Above Low Bid	Primary District	County	Description of Work
080523-606	3rd	CON	$8,136,465.12	5.4%	06	ST LOUIS	J6P1829: Cold mill, resurfacing wit gutter, and traffic signals, J6Q2037 signal controllers, fiber optic cable, Optional pavement and drainage
080523-610	Win	ASP	$1,420,836.84	0.0%	06	ST LOUIS	J6S2151: Cold mill & resurface witl

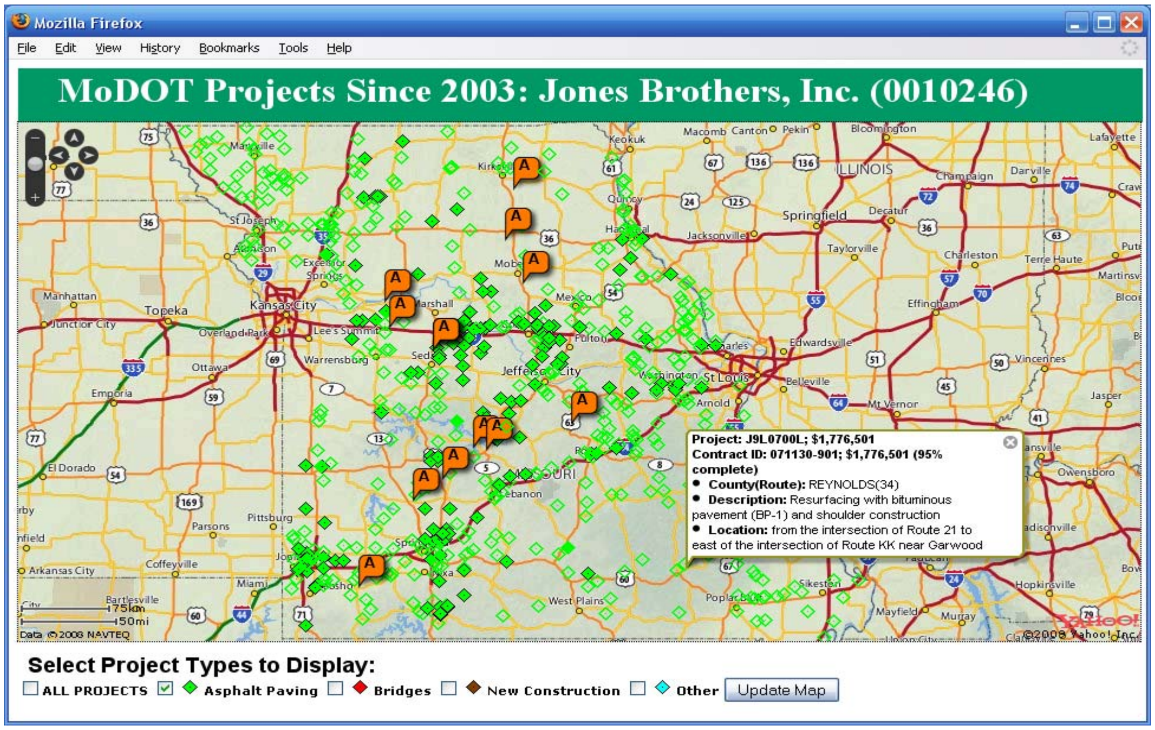
The report image is constructed using direct writes to two separate html files, a header file with summary analysis, and a scrolling table file of bid histories below. Both are generated automatically and are linked into a larger menu system. Below are snippets of the code that was used to write directly to the html file that constitutes the scrolling table.

```
put ' <TD ALIGN=LEFT bgcolor="#E7E3E7"><font  face="arial" size="2"
color="#000000">' &contid '</font></TD> ' ;
put ' <TD ALIGN=LEFT bgcolor="#E7E3E7"><font  face="arial" size="2"
color="#000000">' &bdbstat '</font></TD> ' ;
put ' <TD ALIGN=LEFT bgcolor="#E7E3E7"><font  face="arial" size="2"
color="#000000">' &cnprpwrk '</font></TD> ' ;
put ' <TD ALIGN=RIGHT bgcolor="#E7E3E7"><font  face="arial" size="2"
color="#000000">' &bdtotal '</font></TD> ' ;
put ' <TD ALIGN=RIGHT bgcolor="#E7E3E7"><font  face="arial" size="2"
color="#000000">' &pctover '</font></TD> ' ;
```

Code example 3: Put statements can be used to write directly to the html file.

As can be seen in code example 3, macro variables are used to carry the data values that will be written to the html file. The put statements, in a sense, serve as a static template that is refreshed and rewritten with new values over and over again to a new file for each company. In the case of this report it takes about 10 to 15 minutes for a very modest processor to write a couple of thousand reports. Because it is a direct write to the file, it runs more rapidly than a process managed by proc report or a graphics procedure.

A more challenging application involves combining Ajax methodology with PSAX. The example below is created by a SAS routine that assembles a datamart of project locations and uses SAS to generate the Javascript it would take to plot these on a Yahoo ™ map.



MoDOT Projects Since 2003: Jones Brothers, Inc. (0010246)

Project: J9L0700L; $1,776,501
Contract ID: 071130-901; $1,776,501 (95% complete)
- **County(Route):** REYNOLDS(34)
- **Description:** Resurfacing with bituminous pavement (BP-1) and shoulder construction
- **Location:** from the intersection of Route 21 to east of the intersection of Route KK near Garwood

Select Project Types to Display:
☐ ALL PROJECTS ☑ ◆ Asphalt Paving ☐ ◆ Bridges ☐ ◆ New Construction ☐ ◆ Other [Update Map]

To do this once would be quite tedious, but because we're harnessing the automation abilities inherent in SAS, a single map drawing routine (Ajax) is created ahead of time and automatically reproduced over and over (PSAX) through a code generator inside a SAS data step. SAS provides an XML engine in the libname statement which can feed the map with XML data about points to plotted. If "object-oriented" data is something that excites you, can use JSON instead -- a beautifully streamlined, organic data structure. Unlike XML, you'll have to hand code the logic that converts tabular data into JSON. Again, because you'll be churning out hundreds of them, the initial effort can be worth it. You can also build the data into the web page itself by hard-coding the map points into Javascript functions in the header section of the file. While this would be clumsy in traditional Javascript programming, the automation abilities in SAS make it easy and even desirable. You could also use a single map template and build a PSAX-style routine to compute XML or JSON datasets for each new map image. It depends on your layout and design.

If the previous paragraph leaves you scratching your head …. don't worry. Plotting Yahoo™ or Google™ map web-page templates is definitely the outer edge of this technology. For those who come to this from object-oriented languages like Javascript or dataforms like JSON, however, it illustrates how SAS can provide an easily automated bridge between relational databases and these more organic forms.

For those who like digging around in Javascript, an excerpt is provided below of the SAS code that was used to write the Javascript which creates checkboxes at the bottom of the map. Macro variables are used to define the labels next to the check boxes. The check boxes activate the display of points that SAS has pulled out of the database with JSON or XML. In the case of the map above, "&uppr1_labl" resolves to "Asphalt Paving" and that becomes the analysis variable which is plotted on the map.

```
put "<span id=""loadtxt"" class=""pick"">Select Project Types to Display:
</span><br>                          " ;
 IF &allbox eq 1 THEN DO ; put "<input id=""cb_all"" type=""checkBox""
onclick=""javascript:maincheck();"">ALL PROJECTS              " ;  END ;
put "<input id=""cb_1"" type=""checkBox""
onclick=""javascript:whichcheck();"">&nbsp;<img
src=""../../../../../weblib/graphics/symbols/lime_win.gif"">&nbsp;&uppr1_labl
" ;
 IF &uppr2 ne 1 THEN DO ; put "<input id=""cb_2"" type=""checkBox""
onclick=""javascript:whichcheck();"">&nbsp;<img
src=""../../../../../weblib/graphics/symbols/red_win.gif"">&nbsp;&uppr2_labl
" ;  END ;
 IF &uppr3 ne 1 THEN DO ; put "<input id=""cb_3"" type=""checkBox""
onclick=""javascript:whichcheck();"">&nbsp;<img
src=""../../../../../weblib/graphics/symbols/brown_win.gif"">&nbsp;&uppr3_labl
" ;  END ;
 IF &uppr4 ne 1 THEN DO ; put "<input id=""cb_4"" type=""checkBox""
onclick=""javascript:whichcheck();"">&nbsp;<img
src=""../../../../../weblib/graphics/symbols/skblue_win.gif"">&nbsp;Other
" ;  END ;
put "<input type=""button"" value=""Update Map""
onmousedown=""javascript:dataWait();"" onmouseup=""javascript:mapPoints();""  >
" ;
put "</h4></form>          " ;
put "</body>               " ;
put "</html>               " ;
```

Code example 4: If-statements build checkboxes on a web page based on data values.

KNITTING IT ALL TOGETHER

SAS is particularly well suited to the work of knitting together an integrated library because it's easy to create code that adapts itself to conditions. In a sense SAS is "self aware." As in the previous code example SAS can adapt which lines of code get run based on macro variable settings and the content of the code itself can be derived from these variables on the fly. SAS can also use this self-awareness to knit pieces of a web library together on the fly. You can have SAS use the values of key variables to shape filenames, storage directories and names of links in consistent predictable ways. You can create storage directories and predictable file names using variable values. By integrating these values into the formation of your code SAS can morph itself to manage the seemingly impossible job of organizing thousands of files and reports without human intervention.

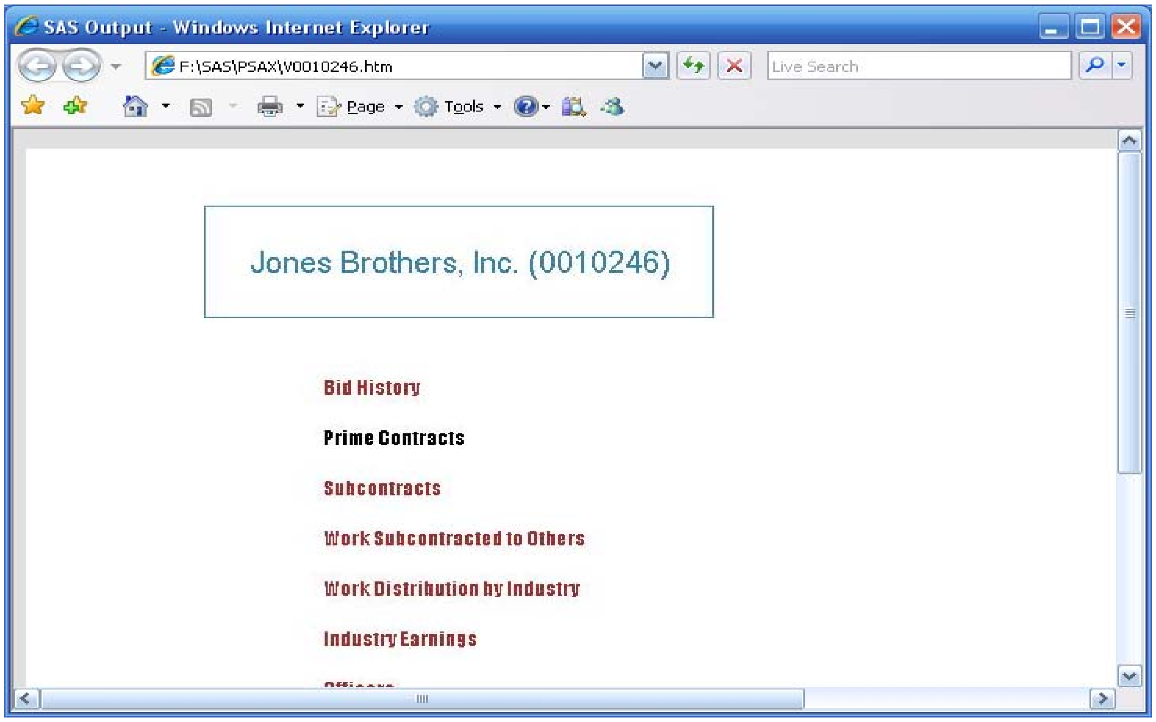
Of course you can't physically knit things together in your site without creating navigational devices that link to your various reports. Reference links in a reporting page are fine but you also need organizational pages, panels or drop downs whose sole purpose is to present a list of links to other pages. For traditional HTML devices you could write a SAS program to hard code these features in HTML, but you can adapt traditional SAS procedures to do this as well.

On the following page is an excerpt taken from a gslide procedure, in which a title statement was used to list reports and create links to them when they are available. Reports that are available are displayed in brown and provided with an active url to the report. Reports which are not available to this section (due to a lack of data) are colored black with no underlying hotlink. This is one of many ways to create a navigational page in SAS.

```
title1 font='Arial Narrow' j=l color ='steel' move=(1.5in,85pct) box = 1
h=22pt "%superq(contrname2)" ;

ODS html path = "&dataroot\&qviewnam\vendor\&vendor"  FILE="V&vendor..htm"
gtitle  ;
proc gslide
description = "<CLICK> on BROWN titles to access reports" ;
title2  height = 14pt font='Impact' color='black'
%IF &bidflg = 1 %THEN %DO ;
link = "bidhist_&vendor..htm"  color='brown'
%END ;
%ELSE  link = " "  ;
j=l  move=(2.0in,70pct) "Bid History" color='black'

%IF &primeflg = 1 %THEN %DO ;
link = "primes_&vendor..htm"  color='brown'
%END ;
%ELSE  link = " "  ;
j=l move=(2.0in,&vertspace) "Prime Contracts" color='black'

%IF &sub4flg = 1 %THEN %DO ;
link = "sub4_&vendor..htm"  color='brown'
%END ;
%ELSE  link = " "  ;
j=l move=(2.0in,&vertspace) "Subcontracts" color='black'
```

Code example 5: Building titles with links in proc gslide to create a navigational page.

Below is a screen shot of a page produced by this technique. The example below has been pared down to illustrate the basics of code in example 5 above. In actual use, a variety of logos, images, colors, fonts and styles are added based on the needs of the organization. SAS provides tools for this as well.



Jones Brothers, Inc. (0010246)

Bid History

Prime Contracts

Subcontracts

Work Subcontracted to Others

Work Distribution by Industry

Industry Earnings

The program in this example has to be self-aware enough to only create navigational links to reports that are actually available. In reviewing the code on the previous page you'll see a series of macro variables which trigger whether or not to create a link: &bidflg, &primeflg, &sub4flg, &sub2flg, etc. The page drawing routine is housed in a macro and the trigger variables are fed in as parameters when the macro is called.

Naturally, some coordination is required to make this work. Every time a reporting process is run, a list of the resulting reports is stored to disk for later use. Each reporting routine generates its own list and all lists are combined to shape which links are activated within the navigational page shown in the previous example. Because the PSAX concept is about leveraging time to update pages ahead of time, you'll find yourself running separate sessions scheduled according to when the data becomes available and when you're willing to dedicate processing resources to the job. Hence you'll need to identify organizing threads within the data, use consistent naming formats and design each reporting program to run as a freestanding module. If you're pulling from a relational database your key variables are often good candidates for organizing file names, reporting lists and even storage locations.

While PSAX adapts well to incremental growth and ad hoc additions, a certain amount of planning is required ahead of time and you'll want to think your basic organizational structure through carefully before planning any code.

STRATEGIES: MASS PRODUCTION AND RESOURCE MANAGEMENT

With a PSAX system it's not uncommon to generate 15,000 reports to fully update a web library and more are certainly possible with a popular, well-used library. This means you need to think through how you intend to do production on this scale. SAS enables several mass-reporting strategies but you'll find that code generators are the best for reliably handling the diversity and memory-management needs of a fully deployed PSAX operation. "By-variables" provide a common mass production strategy in SAS but they're not up to making independent free-standing web pages. If you're familiar with launching reporting macros from within a datastep using "Call Execute" this can be an excellent technique for repetitive reporting elsewhere, but the report–calling volume of a PSAX system can overwhelm memory as each Call Execute request stacks up in memory. Code generators work best.

Code generators employ the SAS technique of writing code directly to an external file and then pulling the file back into the program to run when needed. (Example below.).

```
/** Generate macro calls for an address report web-page **/
filename SCRATCH "d:\tempcode.sas" ;
data _null_ ;
  set VENDORS ;
file SCRATCH;
if &do_reports eq 1 and &do_addr_reprts eq 1 then
    put '%RPTADDR(' vendor +(-1) ',%nrstr(' contrname +(-1) ')) ; ' ;
run ;
%include SCRATCH ;
```

Code example 5: Simple example of a code generator

Code is generated based on values and settings found in the data. In a PSAX implementation you use this technique to generate macro calls based on parameters, fields, titles and settings that you feed in with a dataset.

As you develop these systems, you'll find that mass production at this level demands intentional memory management strategies to succeed. Even with the use of a code generator, you can get into memory problems if you're not careful. Many applications, including SAS, are not aware enough of the operating system to release the use of a file name once they have saved something under it. Normally this is not a problem, but with the thousands of files generated by a PSAX system, SAS can use up its file name storage space in the operating system within a few short minutes. This will show up in your SAS log as a Memory Allocation error and the source of this can be quite vexing to diagnose. Any one SAS session can only store a finite number of Windows file names and in the mass-production environment of a PSAX system you can actually hit the limit after generating a few thousand files.

The best approach to managing file name memory is to break your code into modules and run them under separate

SAS sessions. The simplest way to build modules is to break your reporting code into several different programs and call them separately with timed *.BAT files in Windows. Each batch file launches a fresh new session of SAS, which keeps things clean and it fits well with the fact that you'll find yourself wanting to launch different parts of your code on different time cycles depending on when the source data is updated. SAS's Technical Support Note 648, provides an exhaustive and highly readable description of creating Windows batch files that run independent SAS sessions. You'll want to master this technique anyway because, absent any other means of scheduling your reporting jobs, you may be using this technique to schedule your report updates.

Another approach to managing file name storage capacity is to change the machine's registry settings to expand how much memory the operating system sets aside for this task, but you have to be willing to mess with memory mapping settings in the registry and these may conflict with or be wiped out by enterprise-wide maintenance and upgrades of the operating system. You may also find the file name storage capacity to increase under installations of Windows such as XP or Vista.

Whatever approach you take, you'll need to develop some strategy for managing file name storage memory. SAS, as currently configured, doesn't know to release this memory and high volume applications like PSAX will overwhelm it.

With the high volume production of PSAX, the process of creating a log will also consume memory fast and at these volumes it can lock up your session in a couple of minutes. Creating a log also diverts SAS from other work and slows it down so, once you're satisfied with your code, you'll want to shut down all possible forms of messaging to the log. The following options settings will shut down all chatter except for errors and warnings:

```
options nosource nosource2 nonotes nosymbolgen nomlogic nomprint;
```

You may also want to offload your log to an external file on the disk just to be safe. Log files on disk don't have an upper limit. Even warnings can stack up in a PSAX log. SAS provides a "printto" procedure for off-loading the log. Example below:

```
filename SASlog TEMP ;
proc printto log=SASlog; run ;
```

Note that the example sends the log to the "TEMP" directory. This rarely used strategy lets SAS assign a random, unique filename in the hidden temporary directory that SAS creates to manage a SAS session. (The file is destroyed once the SAS session ends.) Using a unique filename is important in a PSAX system because you don't want two different SAS sessions reaching for the same log file at the same time. PSAX gets its power from leveraging processing over time and, as you keep adding things, it's not unusual to find your system is running overlapping sessions at the same time.

The TEMP destination can also be a good place to send your text files when you're running a code generator. The previous example of a code generator sent code to a hard-coded file name but you can also avoid having two SAS sessions reach for the same file by sending these temporary scratch files to the hidden TEMP directory.

You'll also need to enact some memory management strategies for diskspace. Over consumption of disk space will slow and eventually sieze up your system and, of course, creeping disk consumption is a waste of organizational resources. Logs and scratch files are easily managed but it's easy to overlook the voluminous background graphics files needed to build images in a web page. If you don't take control of how these are named, SAS graphics procedures will name them randomly and thousands of obsolete files will build up over time as you continue to update the library.

To force background images to use the same name you need to employ the name= option SAS provides in its graphics procedures. Unfortunately, SAS only allows 8 characters for this name so you'll want to pick your naming convention carefully. It's also important to understand how SAS manages the names of graphics. SAS builds images in a universal memory location (the GSEG catalog) and then translates them to match the intended destination. If you send more than one copy of the same name to the graphics catalog, SAS won't overwrite the old graphic. It will just add a number to the end of the name. That new name is what gets written to your file. Hence, it's important to use each name only once in a sesssion. If you really want to repeat the use of a name you can use proc catalog to either delete the old file first or kill the entire graphics catalog first. Killing the catalog is the morgiving of these two options, because deleting a single entry will throw an error code in situations where the entry doesn't exist.

Last but not least, you may find your self making extensive use of proc datasets to delete old datasets in working memory as you build the data that feeds your web reports. There's no size limit, other than the size of the disk to limit temporary work space but, at these volumes, you may find it important to clean up your files as you go.

CONCLUSION

The PSAX concept of programming presents a new strategy for harnessing old ideas. By recognizing areas where day-old information is as good as real-time data you can use PSAX to leverage many of SAS's unique strengths. The intensely modular nature of SAS and the extensive automation capabilities built into SAS make it ideal for this kind of application. While a PSAX design would not be good for an online banking application or internet shopping, there are many applications, from NFL scores to customer histories where PSAX makes a lot of sense.

Obviously, programming a large coordinated PSAX system takes some experience and challenges you to have good organizational habits in your code. Standardized variable names, good notes and well-structured macros all become vital when you have dozens of pages of interdependent computer code.

If you have an application suited to a PSAX implementation there are several advantages to consider. It's not just that a mass-produced online library protects you from chewing up staff time on minor reports. PSAX installations allow incremental growth and experimentation from the ground up– as long as you do the initial planning. They're very adaptive to user demand. It's extremely frugal. You can have some very small machines running some very large web libraries. You don't even have to install a web-server. HTML web pages are inherently weak at conveying analytics and PSAX, through SAS, helps overcome that. Also, PSAX can serve as an intermediary for other information. If, for instance, you work in pharma and are limited to the last version of SAS approved by the FDA, you can still use the latest features of SAS for presentation, by reading results from earlier versions and presenting them through an online library generated behind the scenes. Best of all, PSAX deployments tend to stimulate innovation because a small core of experienced professionals can build one of these with limited resources.

ACKNOWLEDGMENTS

Like most of us, I owe any success I stumble upon in my SAS projects to Barri, Chevelle, Claire, Martin, Troy, Charlie, Janice, Jodi and all the other great staff at the SAS technical support team.

RECOMMENDED READING

- *HTML for the World Wide Web*, Fourth Edition, by Elizabeth Castro, Peachpit Press, October 30, 1999. A must-have, well-written HTML reference and a great place to start in understanding how HTML fits together and what web pages do. Owning this will empower you to make full use of the HTML linking functions in SAS.
- *Yahoo Maps Mashups*, by Charles Freedman, Wrox, February 12, 2007. If you liked the online maps in this paper, this book is a good place to get started in the bewildering but exciting world of "mashing" up your SAS or XML data points with Yahoo or Google maps.
- *Learning JavaScript* (2nd edition is best), by Shelly Winters, O'Reilly Media, Inc., December 26, 2008, A good book for hard-core learning about Javascript and how to build custom logic into web pages. Not required for most of the things in this article, but a good book if you want to complement your SAS repertoire with building custom logic into your web pages using JavaScript.
- *Pro JavaScript Techniques*, by John Resig, Apress, December 11, 2006. Great book on advanced techniques in JavaScript once you begin to grasp the basics.

CONTACT INFORMATION

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

Name: Turner Bond
Enterprise: Missouri Department of Transportation
Work Phone: (573)751-3993 or (573)268-8385
E-mail: turner.bond@modot.mo.gov or catdancer40s@aol.com