Helpful Hints on Developing a User Friendly Database with SAS/AF

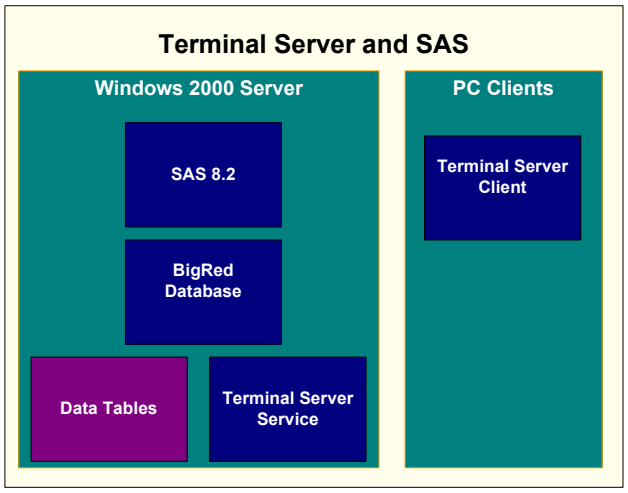
Sy Truong, Meta-Xceed, Inc, Fremont, CA

Abstract

Developing an effective database application requires an interface that is easy for the user. This paper will explore the features of SAS/AF in SAS version 8.2 and methodologies of building a successful database. It combines user interface suggestions for the front end while also suggesting back end SCL, SQL and data step logic that makes the software efficient to program and to operate. The majority of the examples are technical tips but there are also shared lessons learned from collaborating with end users which prove to be very important in creating an effective application.

Introduction

There are many solutions for creating a data entry system ranging from a simple Excel spreadsheet to a sophisticated Oracle database. Each set of technologies works well for a specified task. This paper will explore a database containing clinical information used in regulatory submission. SAS/AF is very suitable for this since all analysis work for clinical data requires SAS. The scenario of this particular project involved a pharmaceutical company developing a custom database. They had already licensed BASE SAS with very little additional modules. One advantage of SAS/AF is that it can be rolled out to clients that do not have SAS/AF during execution. In this scenario, SAS was installed on a Windows 2000 server.



SAS was delivered to users on their desktops through terminal services. This was economical since there was only one SAS license required. Even though SAS/AF was the main software used during development, the programming was not all in SCL, but also involved SQL and data step logic. It is useful to use SQL and data step logic where ever possible, so that more SAS programmers can understand and maintain the system in the future. The main reason the client chose a customized solution was because of unique requirements. It would have made an off-the-shelf system, such as Oracle Clinical, a large investment in infrastructure and operational procedure change. It was therefore more effective to develop tools specific to the specific requirements. Even though the data entry system, named "BigRed", was a custom effort, modularization and the use of a data driven approach made the process much more efficient.

Data Driven with Data Step

One of the biggest time savings in development was to drive the behavior of the data entry system through SAS datasets. The data tables which contain the actual clinical data being entered were more than just data repositories. They also acted as a collection of metadata which drives the labels on the screens. This is a data step example which was used to create the initial data.

```
*** Define the Treatment Center ***;
data dbdata.trtcent (label="Blood
Collection at Treatment Center"
read=&password genmax=11);
   attrib srcloc label="Collection
   Source Location" length=8
   format=LOCFMT.
   unitid   label="Blood Unit ID"
   length=$20
   grprh    label="Blood Group and Rh"
   length=8   format=BLOODFMT.
   pcode    label="Test or Control"
   length=8   format=PCODEFMT.
   coldate  label="Collection Date"
   length=8   format=mmddyy6.
   usrname  label="User Name"
   length=$20
   datetime label=
   "Date Time of User Interaction"
   length=8   format=DATETIME13.;
run;
```

The data table was defined with a label of "Blood Collection at Treatment Center". This label was later used in a selection list for table selection during data entry. Each variable also contained labels. These were used for data entry screen variable selections and labels. The user defined formats were stored in a formats catalog. The values of these were used as coded values for pull down menu selections. At first glance, the data being defined appears to be purely for storing information. For efficiency, these same attributes also affected the user's selection choices on the data entry screens.

One additional layer of security that was easily implemented was to password protect the data table. In this example, it prevented users from reading the data directly by the option (`read=&password`). It forces the users to use the data export or other reporting engines through the system. This allowed better control over what variables were delivered to users such as excluding unnecessary administrative variables.

Another simple yet powerful option is the (`genmax=11`) option. Each time the data is updated or modified, a backup is created. In this case, there are eleven such backups made. The twelfth data table is removed so there is always eleven backed up versions of the data. The underlying tables can be rolled back by actually copying the physical table into the current version. This, however, does not capture an audit trail so a roll back interface was created to make it easier for the administrator to maintain an audit trail.

There are two administrative variables that exist in almost all data tables created through the system. This included `usrname and datetime`. These two fields capture the user name along with the date time of their last interaction in relation to the table. This may appear to be insignificant, but it is essential for identifying who and when problems are caused.

Within the same data table creation program, a sort is applied to key fields.

```
*** Create key fields by sorting ***;
proc sort data = dbdata.trtcent
  (read=&password);
   by unitid;
run;
```

This adds extra metadata to the table in a similar way that a data set label does. That is, the data table contains an attribute specifying what variable it is sorted by. This key is used in the following ways:

- Resorting – After the data is updated, the data is re-sorted by the appropriate key.
- Reporting – Many reports are resorted before a report is generated.
- Merging – The key is used for merges for cross table edit checks or reports.
- Keys – Key fields are also used to drive pre-populate values for some fields and selection boxes.

A template accompanies each data table. This contains the values to which the data entry screen defaults. This can be updated at any time and is a big time saver for many data entry tasks.

```
*** Define Treatment Center Template ***;
data dbdata.t_trtcent (label="Blood
Collection at Treatment Center");
   attrib srcloc label="Collection
   Source Location"
   length=$200
   unitid   label="Blood Unit ID"
   length=$20 ...
run;
```

The only difference between the template and the main table is that the table name for the template starts with a "t_" and it is not password protected. This is a simple way to manage default values of all data entry screens.

In a similar approach to the template, there is an associated audit trail data table for every data table that contains clinical information.

```
*** Define Treatment Center Audit Trail ***;
data dbdata.a_trtcent (label="Blood
Collection at Treatment Center"
   read=&password);
   attrib srcloc label="Collection
   Source Location"  length=8
   format=LOCFMT.
   unitid   label="Blood Unit ID"
   length=$20 ...
   action   label="User Interaction"
   length=$100
   usrname  label="User Name"
   length=$20 ...
run;
```

Each time a user updates, inserts or deletes a row from the main table, the row is inserted into the accompanying audit trail data table. There is an additional action variable in the audit trail which documents what type of action was taken to distinguish events such as an insert versus a delete. This data table differs from the main table in that rather than keeping the most up-to-date data, it keeps a complete historic record of all interactions. Since it is a separate table, it is easy to search or generate reports from. This also meets regulatory requirements that require a complete audit trail of the data.

Another example of using data to drive the behavior of the system is to store all system configurations in a data table. This captures settings such as location of the system data or password expiration duration time. These parameters are usually set by an administrator during installation and configuration.

```
data dbsystem.config (write=&password
alter=&password);
   attrib param length=$100 label =
   "Parameters";
   attrib value length=$200 label =
   "Value";
   attrib usrname length=$20 label =
   "User Name";
   attrib datetime length=8 label =
   "Date Time" format=datetime13.;

   *** Define configuration ***;
   usrname = "admin";
   datetime=datetime();

   param="Data Location";
   value = "C:\Cerus\data";   output;
   param="Password Expiration Period";
   value = "90";   output;
   param="Password Expiration Warning";
   value = "20";   output;
   param="Sasroot";
   value = "c:\sas"; output;
   param="Sastemp";
   value = "c:\temp"; output;
   param="SAS Configuration";
   value = "c:\sas\config.sas"; output;
run;
```

There are several advantages to storing this information in a data table rather than a plain ASCII file.

- Audit Trail – Similar to other tables, a user name and date time is captured during updates. It is easy to determine what was changed last and by whom. Updates to the configuration require a SAS program which also creates a log file. This log file can be saved to retain a more detailed audit trail.
- Security – The data table can be password protected against unauthorized users' update. This can be restricted to administrators only.
- Reporting – Reports can be easily generated with tools such as PROC REPORT or PRINT. The report displays all the current configuration settings.

Even though there are many fancy object oriented features within the SAS Component Language (SCL), data step is still a very useful tool for setting up and working with certain aspects of a clinical database system.

SCL Hints

SAS Component Language has features for system development that go beyond the data step. It is the main language for working with objects within the interactive environment of AF, including things such as pull down menus and buttons. It remains one of the most powerful tools available for developing a cross platform interactive application. The following examples are just a few hints compared to the vast set of functions and tools available.

Rather than having an administrator set up the system datasets during system installation, it is convenient to dynamically create the datasets as they are being used for the first time. In this example, the code is executed when an administrator logs into the system.

```
*** Create Concurrent User Dataset ***;
if exist('dbsystem.concurent') = 0 then do;
   *** Create a new reports data set
   ***;
   dsid = open('dbsystem.concurent
   (read=mypassword)','n');
   rc = newvar(dsid,'usrname','c',20,
   'User Name');
   rc = newvar(dsid,'datetime','n',8,
   "Date Time of User Interaction",
   "datetime13.");
   rdsid = close(dsid);
end;
```

There are no big differences between the SCL and the data step examples previously. The same variable attributes can be defined in both languages, including password protecting the dataset. However, the “exist” function of SCL is a useful function that checks to see if the data exists before it is created. This is an example where SCL functions help make it a powerful development environment.

It is a common convention to have users press the F1 key to ask for help. This exists in most Windows applications. There are several ways to deliver help content to users. One efficient example is to deliver HTML from an existing intranet or Internet site.

```
*** Set the help file ***;
webroot = "http://localhost/";
_frame_.help = webroot ||
"welcome.html";
```

This site may already contain the complete usage and reference manuals for the application. The link made with the user request points to a specific page that relates to the screen where the user is currently at. This is more efficient in that you don’t have to develop another set of content for the help screens which is different from the user documentation.

There are many good examples of how SQL is used within a SCL program which is further explored in the next section. This example shows how an SCL program captures all the dataset names and labels with the help of some SQL code. The first step is to narrow the list of all table names down to the specified libname.

```
*** Capture all data sets ***;
submit sql continue;
   create table work.datatab
   as select lowcase(memname) as
   memname, memlabel
   from sashelp.vtable
   where libname="DBDATA" and
   memtype="DATA"
   and index(memname,'#') = 0 and
   compress(memlabel) ne '';
endsubmit;
```

A temporary dataset is created capturing this information from an existing SAS system view. This same task can be done with either a data step or SCL, but SQL is very concise and elegant in these types of operations.

The SCL program now opens up the same temporary dataset that was just created to capture the dataset names and labels.

```
*** Capture names into a list ***;
datlst = makelist();
datnlst = makelist();
dsid = open('work.datatab','i');
if dsid = 0 then do;
   call display('message.frame',
   "ERROR: Database is not accessible.");
   _status_ = "H";
   return;
end;
```

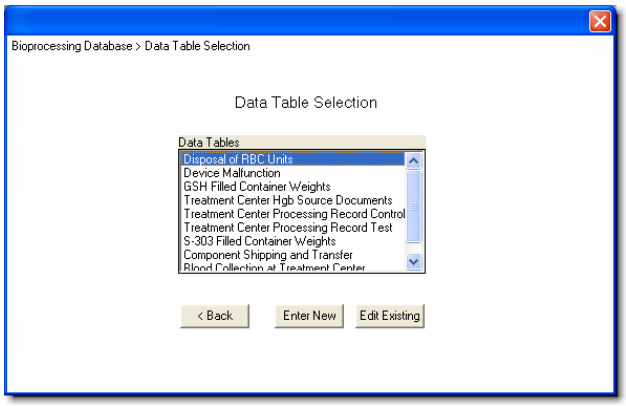
SCL has functions that allow you to verify if the dataset is available. You can therefore perform error checking to see if the data is available. It is possible that the previous SQL code did not work and therefore the data did not get created. In that event, the next step would crash without a check.

Now that the data has been confirmed, the next step is to capture the names of each table and insert them into an SCL list.

```
cnt = 1;
do while(fetchobs(dsid,cnt) = 0);
   curname =
   getvarc(dsid,varnum(dsid,'memname'));
   curlab =
   getvarc(dsid,varnum(dsid,'memlabel'));

   *** Skip over audit datasets ***;
   if (substr(curname,1,2) ne 'a_') and
   (substr(curname,1,2) ne 't_') then do;
      rc = insertc(datlst,curlab,-1);
      rc = insertc(datnlst,curname,-1);
   end;

   cnt = cnt +1;
end;
dsid = close(dsid);
```

SCL lists are a very useful and efficient construct. It is unique compared to other data structures in that you can insert both characters and numbers into the same list. The content of a list can have references to other lists. SCL lists are also used to store information for many AF objects. In this case, the pull down selection list is assigned and pre-populated with values of the list.

```
*** Update the list box with users ***;
l_data.items = datlst;

*** Preselect the first item ***;
l_data.selectedIndex = 1;
```



A good security measure is to expire the users' passwords after a certain time window. It is more polite to warn the user in such an event so a warning prior to the expiration is implemented.

```
*** Check for expiration beyond 90 days
***;
passdate2 = datepart(passdate);
if ((today()-(90)) > passdate2) then do;
   call display('message.frame',
   "WARNING: Your password has expired.");
   call display('_loginck.scl');
   return;
end;

*** Check for 14 days warning period ***;
if ((today()-(90-14)) > passdate2) then do;
   call display('message.frame',
   "WARNING: Your password is about to
   expire soon.");
end;
```

The user's password is modified in a separate table each time it is updated. The date is also captured through the variable PASSDATE. This is a simplified example where a 90 day expiration period is hard coded. The number of days can be configurable by the administrator to an appropriate time window. It can also be useful to have a grace period take effect after the user password expires before the user is locked out.

There are different areas where you can store user information within a session. The SASUSER is a predefined libname that is available for each user. This is usually where settings are stored for that specific user. Information stored in SASUSER is carried from one SAS session to the next. However, the system often needs to track a user within one login session. In this case, the work area is more suitable.

```
*** Update a data in the work signifying
current user ***;
dsid = open('work.user','n');
rc = newvar(dsid,'usrname','c',20,
'User Name');
rc = newvar(dsid,'datetime','n',8,
"Date Time of User
Interaction","datetime13.");
rdsid = close(dsid);
```

It is similar to using a macro variable but there is no conflict with local versus global. The work area dataset is a cleaner and more effective approach compared to macros in the case of tracking users within a session.

SQL Tricks

Similar to how BASE SAS integrates SQL through PROC SQL, SCL integrates SQL through a submit block. It does this seamlessly by preceding the SQL code with a "submit sql" statement. This opens the door for SCL programs to any SQL features. This offers more ways to accomplish the same tasks. When it comes to transactional data table manipulations, SQL is the way to go.

```
*** Insert an administrator new User ***;
submit sql continue;
   insert into dbsystem.users
   (read=&password)
   set usrname = "admin",
   datetime = datetime();
endsubmit;
```

This is a simple example of a record being inserted into the table which registers users to the system. The "admin" account is automatically created as the default first user. The administrator can then set up all other user accounts. You can queue up a series of SQL code through submit blocks. The "continue" option tells the system to actually submit and process the SQL code at that point. The submit block can act as a powerful code generator of SQL code that goes beyond what traditional SAS macros can do.

The SQL code that is submitted through SCL resolves variables noted through the '&' notation similar to how macros resolve variables. For example, if there is a SCL variable ONOTES, it would be referenced as &ONOTES in the SQL code. When it is submitted, it resolves to the value which ONOTES contains. It is recommended that whenever there is a character variable where the user enters the value, use *%NRBQUOTE*.

```
*** Update the audit trail ***;
insert into dbdata.a_disposal
(read=&password)
set unitid = "%nrbquote(&e_unitid)",
dispcom = "%nrbquote(&onotes)",
action = "Edit existing key: &keyvalue",
usrname = "%nrbquote(&ousrname)",
datetime = datetime();
```

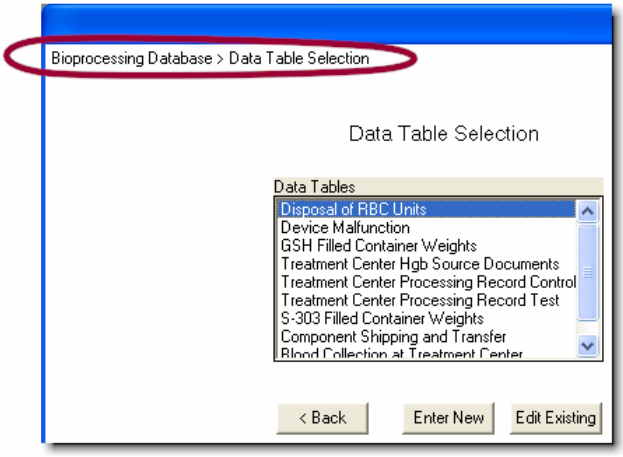
The *%NRBQUOTE* handles situations where the text contains a quotation or special characters. It is a good habit to place this around character variables since it will make the program much more stable.

Interface Suggestions

The user interacts with the database mainly through the application screens. There may be some batch scripts that administrators use but interactive dialog boxes are the user's interface to most functions. It is therefore important to make them consistent and user friendly. These are some suggestions which can enhance the user's experience.

- Screen Title – Every screen and dialog box contains a consistent title centered at the top. This is a unique short one line of text describing the purpose of the screen.
- Navigation Path – At the upper left corner, there is a navigational path describing where the user is at. If the user drills down from one screen to another, this describes the path. For example, if the main screen is entitled "Bioprocessing Database" and the user drills down to "Data Table Selection", the path would show:

  Bioprocessing Database > Data Table Selection



- Consistent Buttons -- The names of buttons are consistently named. For example, "OK" and "Cancel" or "Back" is commonly used for most screens.
- Disable Objects -- If the user does not have access to a certain screen due to restricted access, disabling the objects is a good approach. The following example disables a couple of buttons.

```
*** Disable buttons ***;
if write = "No" then do;
   b_template.enabled = "No";
   b_ok.enabled = "No";
end;
```

Just as in a written document, there are styles pertaining to the use of fonts and rules of grammar. There is a style guide for user interface. No matter what style you decide to follow, make sure it is consistent throughout the application.

Collaboration with Users

The technical challenge of working on any project is only part of the challenge. The communication and collaboration with the users plays a significant role in the success of the project. For this particular project, the client was a couple hours away so there were creative solutions which assisted in our communications. The items below are examples from lessons learned.

- Extranet Portal – A secured website was created for the client to log in to obtain documents. This became a repository of documents capturing an audit trail of documents and also meeting discussions. The website became the glue of the collaboration efforts. The documents listed below were all posted on the extranet.
- Meeting Summary – Meeting minutes or even diagrams drawn from the discussions were captured as PDF and placed on the extranet site.
- User Requirements, Functional Specifications and Test Plan – Documents that require many iterations of review became very useful to store centrally. The older versions were also retained for reference.
- Time Line – At the beginning of the project, the time line helps the upper managers allocate their resources. It became a road map that drove the order of the work flow.
- Online Meetings – The use of Webex to hold online meetings was very efficient. It allowed the teams to hold teleconferences while, at the same time, seeing what was on each other's screen. This helped cut down wasted travel time.

There is nothing like face to face meetings. This is the cornerstone of making decisions and working effectively as a team. The suggestions above complemented the traditional meetings and allowed the team to work together even when not everyone was in the office at the same time.

Conclusion

A data entry system for a clinical database requires specialized functionality. SAS/AF and its related technologies are very suitable tools for the job. SAS Component Language (SCL) is the primary language used in AF but it is not the only one. It has the ability to integrate BASE data step and PROCs along with SQL. SCL has many of its own functions that are quite powerful. When everything is combined into one development platform, this creates a superset that is more powerful than any one language by itself. Even through SCL has many powerful functions, the use of simple data step and the storing of metadata inside a data table can be used for building a database. SQL contributes its strength in data manipulations. These sets of technologies really take care of the application itself. Collaborating with users during development requires another set of technologies which involve an extranet. These technologies are very helpful in complementing the communication between the developer and end users. However, nothing replaces the interactions of traditional face to face communication.

References

FDA, Guidance for Industry: Providing Regulatory Submissions in Electronic Format – General Considerations, 1999

Guidance for Industry: Providing Regulatory Submissions in Electronic Format - NDAs , January 1999

Data structures and software can be found at:
http://www.meta-x.com

About the Author

Sy Truong is a Systems Developer for Meta-Xceed, Inc. They may be contacted at:

Sy Truong
48501 Warm Springs Blvd. Ste 115
Fremont, CA 94539
(510) 226-1209
sy.truong@meta-x.com