Rapid Applications Development Using the SAS® System

Neil Davis, Manager Personnel Systems, ANZ Banking Group (New Zealand) LTD

Abstract

This paper discusses techniques for reducing the development time of applications, including, to the point where SAS/AF® frames can be built without writing any code. This paper will also discuss the importance of creating consistent user friendly interfaces and how this aids rapid application development. Techniques discussed will include creating sub-classes, composite widgets, and macros.

Introduction

Although there is a number of aspects involved in the rapid development of applications this paper will concentrate on:

- Minimizing the code that a developer must write. This includes minimizing it to point where Frame entries can be developed without writing any code at all.
- Using features within the SAS system that allow developers to find and utilise information about the local environment and dynamically adapt to that environment. Thus, making code easier to reuse and transport between operating systems.
- How consistent user friendly interfaces aid the rapid development of applications.

Topics discussed to reduce code include:

- Writing reusable blocks of code - Macros and Methods.
- Using System Files and Commands to gather Information.
- Consistent Friendly User Interfaces.
- Creating sub-classes of objects.

Some of the topics discussed will be useful in normal SAS Base® programs as well as SAS/AF applications. While the examples in this paper show specific ways to use the techniques discussed, it is important to remember that the concepts can be adapted to perform many other useful functions. The example code in this paper has been developed in an OS/2® environment however the concepts and examples discussed can be adapted to run in other environments with minor or no changes.

Reusable Code

As a developer you will invariably find sections of code that perform useful functions are required in many different programs and applications. Many of the functions that these blocks of code perform are usually quite simple or repetitive. Developing reusable code to perform these functions may take longer to initially develop it will pay off in the long run. Considerably reducing th development time of future applications because they can utilise the same block of code, and there is only one block of code maintain. This section will discuss two of the most useful ways to create reusable code in the SAS system.

Macros

Macros are probably one of the best known ways to store commonly used blocks of code for future referencing or inclusion in programs. Macros can be used in both SAS Base and SAS/AF programs.

Blocks of code can be written, then stored in special locations known as Autocall Librarys. Code stored in these locations can be quickly and simply referenced or included in other programs. Alternatively macro code can be compiled and stored in permanent catalogs for future referencing.

Additionally, the macro facility has a number of useful features that allow you to perform functions not possible in Base or SCL programs. Particularly in regard to reducing repetitive code and the conditional processing of code.

The macro facility also allows access to a number of “Automatic Macro Variables”. These variables contain useful system information about the environment you are operating in. For example, &SYSSCP returns the operating system being used, or to find the date the current SAS session started use &SYSDATE. The macro facility also allows developers to store information in macro variables for future reference within the SAS session. (Macro variables can be particularly useful for storing strings more than 200 characters long.)

Below are two examples that utilise the SAS Macro facility to demonstrate the features discussed and that make simple repetitive tasks, simple and efficient to perform.

For more information on the macro facility refer to : *SAS Guide to Macro Processing.*

Example 1

This example demonstrates the use of Automatic Macro Variables, developer created Macro Variables and conditional processing of code.

This code would typically be found in a DDE application. It creates a macro “startxl” and permanently stores it in the catalog “SASUSER.SASMACR”.

This macro determines if Excel is already running. If it is not, it locates “excel.exe” and executes it, if it was running it closes all the open workbooks. It then opens the workbook passed as a parameter to the macro. If no parameter was passed a new blank workbook is opened. It also defines a fileref ”xlsys” which can be used to pass system commands to excel. *(For details on the %findfile macro refer to the section on Unnamed Pipes).*

Once compiled and stored this code can be quickly utilised in any program by simply entering:

%startxl(<*filename*>);

Where *filename* is the <*drive:*<*path*>>*name* of the excel workbook you wish to open.

```
*************************************************************************
* Title         : Start Excel.                                          *
* Stored In     : D:\ANZHRMIS\EIS\MACROS\ESTARTXL.SAS                   *
* Parameters    : Workbook to open (xlfile)                             *
*-----------------------------------------------------------------------*
* Description   : This macro finds excel, starts it, creates a fileref to  *
*                 pass commands, and optionally opens a work book.      *
*************************************************************************;

options mstored sasmstore=sasuser;

%macro startxl(xlfile) / store;

   options noxwait noxsync;

   filename xlsys dde 'excel|system' notab;

   data _null_; file xlsys; run;

❶  %if &syserr %then %do;
       %findfile(excel.exe,cd);   * Find and start Excel;
❷      %if '&fileloc' ne %then %do;
           %put Starting Excel located at &fileloc;
❸          %if '&sysscp' = 'OS2' %then x 'start /win /b &fileloc';
                                 %else x "&fileloc";
           data _null_; x=sleep(30); run;
           data _null_; file xlsys ; run;
       %end;
   %end;

❹  %if &syserr %then %put Excel could not be started;
   %else %do;
       %put Opening worksheet : &xlfile;

       data _null_; file xlsys;

         length xyz $200;

         if '&xlfile' = '' then xyz = '[new(1)]';
                           else xyz = '[OPEN(''||'&XLFILE'||'')]';

         put '[error(False)]';
         put '[close.all()]';
         put xyz;
       run;
   %end;

%mend;
```

❶ Use the automatic system variable (SYSERR) to determine if an error occurred while trying to access Excel|System topic.

❷ Check the macro variable (FILELOC) created by the *findfile* macro to see if Excel was found. If it did execute code to start Excel.

❸ Check the what the operating system is to determine how best way to start Excel.

❹ Re-check the system variable (SYSERR) to see if Excel was started and then open the required workbook.

Warning there are a number of issues (bugs) to be aware of with the OS/2 "START" command (most of which have been acknowledged by IBM® as problems) which can result in the server application not starting properly. A fix for this problem should soon be available. For more information or work arounds please do not hesitate to contact me.

Example 2

A common problem with producing graphs from the SAS system is the 16 character limitation on axis values.

The following graph is generated by the code that follows it. As you can see the *x* axis values have been truncated at the 16th character. This problem is easy enough to overcome by overriding the tickmark values.

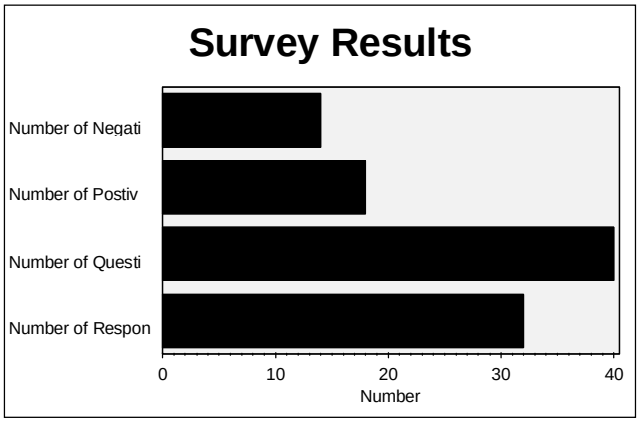

Figure 1

```
data temp;
  input xvar $34. yvar 3.;

cards;
Number of Questionaire Distributed 40
Number of Responses to Survey      32
Number of Postive Responses        18
Number of Negative Reponses        14
;

goptions device=OS2 cback=WHITE ctext=BLACK
         htitle=6   htext=2     ftext=swissl
         gunit =pct rotate=landscape;

title 'Survey Results';
footnote;
axis1 label=(height=4 'Number') value(h=4);
axis2 label=none color=BLACK value=(h=4);
pattern1 value=solid color=red;

proc gchart data=temp;
  hbar xvar / discrete       coutline = black
              raxis = axis1  maxis = axis2
              sumvar = yvar  sum
              width=20       nostats
              frame          cfr=gray;
  format xvar $34.;
run;
quit;
```

However, if you create a lot of graphs, a more permanent solution that will dynamically adjusts the axis values is required. So what do you do..... Create a macro that will dynamically override the tickmark values for you.

The following code creates a macro "tickmark" and permanently stores it in the "SASUSER.SASMACR" catalog.

This macro dynamically determines the appropriate tickmarks for each of the discrete axis values and creates a macro variable "&tickmark" which contains the tickmark overrides.

```
*************************************************************************
* Title         : Tickmark Macro.                                       *
* Stored In     : D:\ANZHRMIS\EIS\MACROS\TICKMARK.SAS                   *
* Parameters    : Dataset Name, Variable Name, Variable Format, Text Height *
*-----------------------------------------------------------------------*
* Description   : This macro determines the discrete values for a variable  *
*                 in a dataset and creates a macro variable "TICKMARK" into *
*                 which it put the statements required to override the      *
*                 default tickmarks for the graphs axis.                    *
*************************************************************************;

options mstored sasmstore=sasuser;

%macro tickmark(dataset,variable,format,height) / store;

  option nosource nosource2 nonotes;

❶ %if &dataset ne and &variable ne and &height ne %then
      %do;
❷        data tickmark(keep=desc t);
           set &dataset;

             length x y t $200 z $16 ;

             %if %quote(&format) ne %then %str(desc=put(&variable,&format););
                                  %else %str(desc=&variable;);

             do until(desc=' ');
                x    = substr(desc,1,indexc(desc,' '));
                y    = substr(desc,1,indexc(desc,' '));
                desc = substr(desc,indexc(desc,' ')+1);

                if length(x) > 16 then do until(x=' ');
                   y = substr(x,1,length(scan(x,1,',.-/'))+1);
                   x = substr(x,length(y)+1);

                   if length(y) > 16 then do;
                      t = left(trim(t)!!' '!!trim('j=1 ''!!trim(z)!!''''));
                      do until(length(y)<16);
                         t = left(trim(t)!!' '!!trim('j=1 ''!!
                                  substr(y,1,17)!!''''));
                         y = substr(y,17);
                      end;
                   end;
                   link tick;
                end;
                else link tick;
             end;

             t = compbl(left(trim(t)!!' '!!trim('j=1 ''!!trim(z)!!'''')));

           return;

           TICK:
             if length(left(trim(z)!!' '!!y)) > 16 then do;
                t = left(trim(t)!!' j=1 ''!!trim(z)!!'''');
                z = left(trim(y));
             end;
             else z = left(trim(z)!!' '!!y);
           return;

           proc sort data = tickmark nodupkey; by &variable

❸          data _null_;
             set tickmark end=done;
             file 'tickmark';
             if _n_ = 1 then put '%let tickmark= ' 'h=&height ';
             put 'tick=' _n_ t;
             if done then put ';';
      %end;
  %else
      %do;
❹        data _null_;
           file 'tickmark'; put '%let tickmark=;';

           %put ************************************************************;
           %put Tickmarks were not adjusted because: ;
           %if &dataset  eq %then %put A valid dataset name is required;
           %if &variable eq %then %put A valid dataset variable name is required;
           %if &height   eq %then %put A value for axis text height is required;
           %put provided to the tickmark macro.;
           %put ************************************************************;
      %end;

  run;

  option source notes;

%mend;
```

❶ Check to ensure that required parameters were passed to the macro.
❷ Determine each of the discrete axis values and divided into parts no longer than 16 characters.
❸ Write tickmark statements to an external for later inclusion into SAS/Graph® axis statement. Using this method to create the macro variable allows it to be longer than 200 characters.
❹ Write messages to the log if the required parameters were not past to the macro.

Once this macro code is compiled and stored it can be quickly utilised in any graph program by adding the shaded lines of code into the program.

```
%tickmark(temp,xvar,$34.,2);
%inc 'tickmark';

title 'Survey Results';
footnote;
axis1 label=(height=4 'Number') value(h=4);
axis2 label=none color=BLACK value=(h=4 &tickmark);
pattern1 value=solid color=red;

proc gchart data=temp;
  hbar xvar / discrete       coutline = black
              raxis = axis1  maxis = axis2
              sumvar = yvar  sum
              width=20       nostats
              frame          cfr=gray;
  format xvar $34.;
run; quit;
```

The graph output by the above code now contains the full *x* axis values (figure 2). Additionally, you have one block of code that can be simply and efficiently utilised in any number of programs to perform the same function without modification.

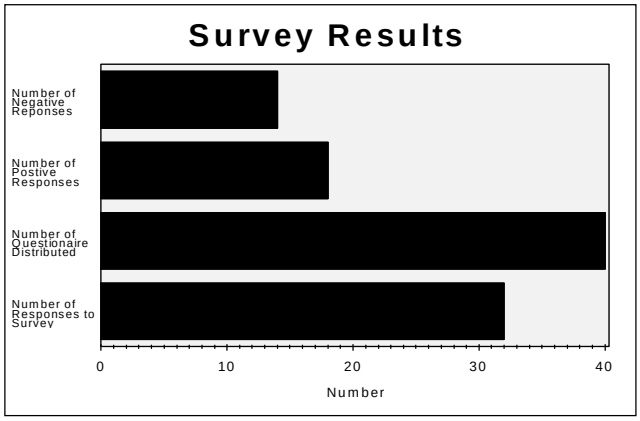

Figure 2

Methods

Methods are blocks of SCL code that perform common operations and are stored in SAS catalogs. They are generally not as flexible as macro code as they can only be used from SAS/AF, FSVIEW and FSEDIT applications.

The method example below is used to determine the existence and exact location of a file. When it finds the required file it creates a macro variable "&fileloc" that contains the drive/path/filename of the file. The next section on Host Commands has example of a macro which uses SAS Base code to perform a similar function.

```
FINDFILE:
  method file $;

    do drive = rank('C') to rank('Z');
       rc =
       filename('dir','dir '!!byte(drive)!!':\'!!file!!' /s /f /n','pipe');
       id = fopen('dir','s');
       rc = fread(id);
       rc = fget(id,text);
       id = fclose(id);

       if text ^= _blank_ then leave;
    end;

    if text ^= _blank_ then do;
       call symput('fileloc',text);
       put 'File Location = ' text;
    end;
    else do;
       call symput('fileloc',' ');
       put 'Error :  ' text ' could not be found !!';
     end;

  endmethod;
```

Once compiled and stored this code can be quickly utilised in any program by simply entering:

call method('pipe.scl','findfile','*filename*');

Where *filename* is the *<path>name* of the file you wish to locate.

Using System Files and Host Commands to gather Information

There are many system files and commands which can be used to gather useful information about the environment in which the application is running. This information can then be used by developers to make applications more functional, user friendly, and assist in the transportation of code.

Using the SAS system there are number of ways that this information can be accessed. This paper will discuss three of the possible methods Unnamed Pipes, Redirecting Output and Analysing System Files.

Unnamed Pipes

Unnamed pipes is a feature of OS/2 and Windows NT that allows you to redirect the output from programs external to the SAS system to the SAS system. Many host commands work in this way. This enables you to issue host commands then capture and analyse the output that they produce.

For more information on unnamed pipes refer to the host documentation for the version of SAS you are running.

Redirecting Output

For operating systems that do not support Unnamed Pipes other means must be employed to capture host command output. It is important that developers are aware of the facilities are available from host systems that are developing for so that they can improve the flexibility of their applications.

An alternative option available in most host systems is to redirect the output from a host command to a file and then read data in from the file.

Analysing System Files

Systems files such as configuration and ini files can be read to find information about the local environment. An example of how to use system files to gather information is shown in the sub-classing section of this paper.

Below is practical example of how to gather information from host commands using Unnamed Pipes and by redirecting Host Command Output.

Example

Like most organisations we have found that each user can have their computer setup differently and files can be located in different directories or even drives.

e.g. Microsoft Excel can be located in :

C:\EXCEL or
C:\MSOFFICE\EXCEL etc.....

When developing a DDE application that uses Excel you must first determine exactly where the file (EXCEL.EXE) is. In an OS/2 or Windows NT environment this can be done using an "Unnamed Pipe" to execute an command and capture the data returned by that command. In an OS/2 environment the "DIR" command is used to obtain this information. In OS/2 it is recommended that you use the /s /f /n switches when using the "DIR" command:

/s causes OS/2 to search all sub-directories as well as the current directory.

/f causes OS/2 to output the full file path and name only. It also eliminates unnecessary data. ie header and trailer data details like volume label, available disk space etc. are not output.

/n ensures that the information is returned in HPFS format even if the disk is formatted for FAT.

In a Windows, Windows 95 or Windows NT environment the "DIR" command can also be used with the /s /b switches:

The /s equivalent to OS/2 /s switch.

The /b equivalent to OS/2 /f switch.

Note that the code has been developed to determine the most appropriate means for locating the desired information from the host system. Thus, making the code easily transportable between host systems.

When the location of the file is discovered the full path name of the file is stored in a macro variable. By checking the value of the macro variable "FILELOC" created by the following macro you can determine if Excel was found and use it or issue the appropriate warning message.

While this macro was originally written for use with locating applications for DDE applications, it can also be used to determine the location of any file.

The macro code below shows further examples of how Macro Variables, conditional, and repetitive processing can be achieved with the macro facility.

```
*************************************************************************
* Title         : Find File Macro.                                      *
* Stored In     : D:\ANZHRMIS\EIS\MACROS\FINDFILE.SAS                   *
* Parameters    : File name to find , drives to search                  *
*-----------------------------------------------------------------------*
* Description   : This macro seacrhs the system for the specified file and  *
*                 returns the full path name of the file.               *
*************************************************************************;

  %macro FindFile(parms) / parmbuff store;

    %global fileloc;

    %let fileloc  = %str();
    %let file     = %scan(&syspbuff,1,'(,)');
    %let drives   = %scan(&syspbuff,2,'(,)');

❶   %if &drives eq %then %let drives = CDEFGHIJKLMNOPQRSTUVWXYZ;

❷   %do i = 1 %to %length(%trim(&drives));

      %let drive = %substr(&drives,&i,1);

      %if %index('CDEFGHIJKLMNOPQRSTUVWXYZ',%upcase(&drive)) %then
          %do;
            %put Looking for &file on drive : %upcase(&drive);

❸           %if '&sysscp' = 'OS2' %then %do;
              filename dir pipe 'dir &drive.:\&file /s /f /n';

              data _null_;
                infile dir lrecl=80 pad;
                input file $80.;
❹               if file ne '' then call symput('fileloc',file);
              run;
            %end;
            %else %do;
              options noxwait noxsync;

              x 'dir &drive.:\&file /s /b >c:\fileloc.txt';
              quit;

              data _null_;
                infile 'c:\fileloc.txt' lrecl=200 pad;
                input @1 text $200.;
❹               if file ne '' then call symput('fileloc',file);
              run;

              x 'del c:\fileloc.txt';
              quit;
            %end;
          %end;
      %else %put WARNING : '&drive' is not a valid drive name.;

      %if &fileloc ne %then %let i = %eval(%length(%trim(&drives))+1);
    %end;

    %if &fileloc eq %then %put ERROR : &file could not be found.;
                     %else %put File Location = &fileloc;

  %mend findfile;
```

❶ If no drives have been specified as a parameter search all drives.
❷ Search each of the drives for the specified file.
❸ Use the automatic macro variable &SYSSCP to determine operating system type and the best way to find the file. In an OS/2 environment it uses an unnamed pipe, in other environment it redirects output to a file for analysis.
❹ Write file location the macro variable &FILELOC for future referencing.

Once compiled and stored this code can be quickly utilised in any program by simply entering:

%findfile(*filename, <drives>*);

Where *filename* is the *<path>name* of the file you wish to locate and *drives* is a string containing all the drives to be searched. If *drives* is not provided the macro will search drives C through Z.

Consistent Friendly User Interfaces

The development of applications can be sped up considerably by having consistent screen layouts and using the same objects on different screens to perform the like functions. When you have a function that is required on many screens or in different applications, you can build objects or write reusable code (macros or methods) that can be used in many applications.

The screen below *(figure 3)* contains a number of objects that can be used perform exactly the same function on many different screens.

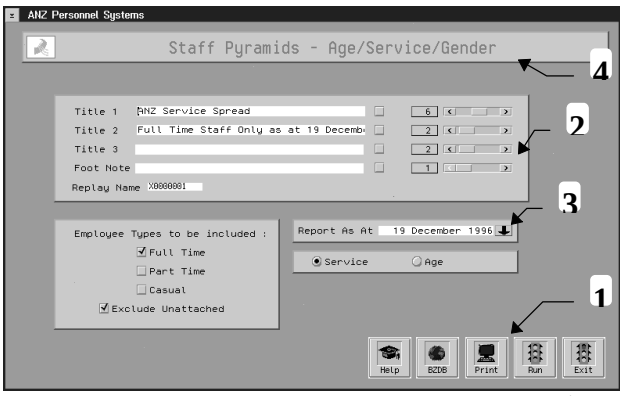

Figure 3

1. A generic command bar, which is a composite widget made up of Image Icons. It issues simple commands such as end, cancel and help but also performs more complex functions such as subseting datasets and redirecting output and printer setup.

2. A generic composite widget, which allows users to alter the appearance and contents of graph titles. It also allows the user to specify the name of graphic output.

3. A generic composite widget that allows users to selection a valid pay date or displays the current pay date.
4. A generic full screen composite widget that is used as a title bar and sets frame colors and setting to the defaults required for the user interface.

Once you have decided on a screen layout or an object's functionality, an object can be built that will instantly create that raw layout or functionality. These object can then be placed on new frames to instantly create the same look and functionality.

By far the biggest advantage in having consistent layouts and objects is that users only need to be shown once what the object does. Despite the fact that the objects may appear on many frames with differing functionality the individual objects will always look and function the same.

Sub-classing

Sub-classing is a feature of SAS/AF which allows you to enhance the standard objects that are shipped with the SAS system. The new object can be as simple as a button which issues a specific command or a collection of different objects (composite widget) which all work and communicate with each other to perform complex functions. Because the look of an object is controlled by the data within it, the appearance of an object can be changed by altering its data elements only. For example, a Push Button can display different text by simply modifying the text in the objects "Label" variable.

The objects functionality is controlled by code stored in method blocks. Different methods are executed when different events occur. eg. when the user presses a push button the _SELECT_ method runs. You can override an object default data elements and methods to create a new object which has all the functionality of it's parent plus any additional functionality you wish it to have by default.

Example

The following example shows how to create a simple sub-class of the "Push Button" object. This object is used to determine the default output device for Win OS/2 sessions, the available Win OS/2 output devices and allows the user to redirect the output to one of them.

Figure 4 below shows an example of the "WINDEV" push button class with the users default windows output device showing.

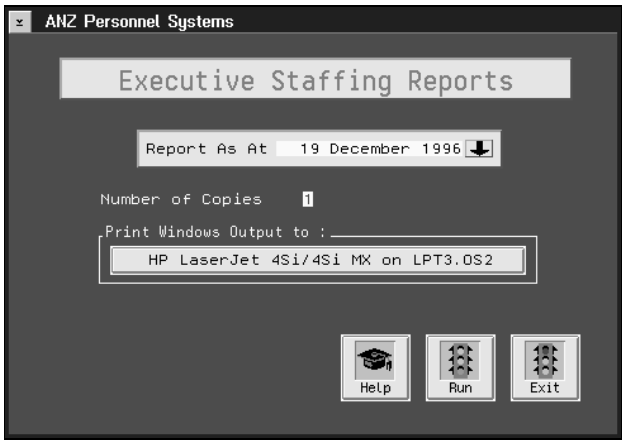

Figure 4

This sub-class creates a macro variable (WINDEV) which contains the standard printer setup command for Microsoft applications :

printer.setup("HP Color LaserJet on LPT1.OS2:")

To do this, it does as follows:

- after a frame containing this sub-class completed execution of the frames INIT section the systems WIN.INI is searched to determine the default output device. If the WIN.INI could not be accessed or the default device could not be found the push button label is set to "- Default Printer Only -". If the default device was found, the name of the device is displayed as the button label. The macro variable WINDEV is defined as _BLANK_.

- during the initialization process a list of other available output devices is also built.
- after the frame has been initialized if the user presses the button a list of available devices is displayed for the user to select from. Shown in *figure 5*. When the user selects a device the WINDEV macro variable is updated. If no list was built at initialization time a warning message is displayed indicating that only the default device is available.

To build this object :

- save a copy of Push button object as "WINDEV.CLASS".
- override the _POSTINIT_, _SELECT_ and _PRETERM_ methods to use the code detailed below.
- create an automatic numeric instance variable (LST) which is the identifier of the list that contains the available windows output devices.
- set Custom attributes to blank, Replace supplied attribute window, an switch off Display attributes upon make/fill.
- add the new class to the appropriate resource file.

Note, that if you make changes to your Windows setup while the frame is active, they will not take affect until the frame is reinitialised.

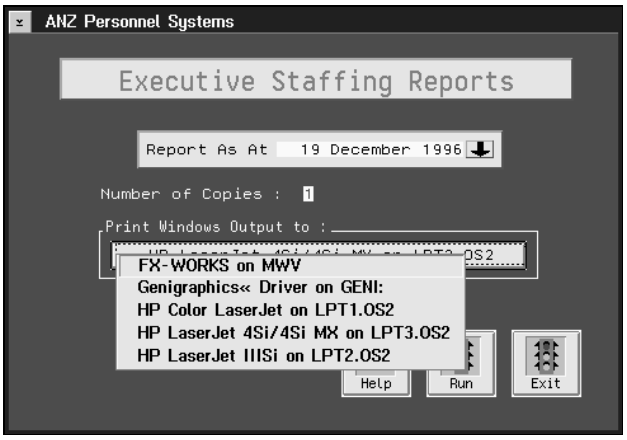

Figure 5

```
*************************************************************************
* Title         : Windows Output Devices Custom Class Methods.          *
*                                                                       *
* Program Group : UTILITYS                                              *
* Stored In     : EIS.UTILITYS.WINDEV.SCL                               *
* Called By     : EIS.UTILITYS.WINDEV.CLASS                             *
* Help Screen   : None                                                  *
*                                                                       *
*-----------------------------------------------------------------------*
* Description   : These methods are used by the WINDEV class to display a   *
*                 list of available windows devices and make selections.    *
*************************************************************************;

length text $50 defaultprt $40;

POSTINIT:    /* WINDEV.CLASS : _POSTINIT_ method override */

* Determine name of the default windows output device and store in macro
  variable 'WINDEV';

  method;

    * Create fileref and open win.ini;

    rc  = filename('INI','c:\os2\mdos\winos2\win.ini');
    fid = fopen('INI','i',0,'V');

    * If fileref cannot be established then warn user that default windows
      device will be used;

    if rc and not fid then
       do;
         _msg_ = 'WARNING : Only default Windows printer is available !!';
         alarm; refresh;
         call send(_self_,'_set_label_','- Default Printer Only -');
         return;
       end;
    else
       do;
       * Determine the default windows output device and create a list of
         available  windows output devices;

         lst = makelist();

         do while(^fread(fid));
            rc = fget(fid,text,200);
            if upcase(text) =: '[WINDOWS]' then winfound = 1;
            if winfound and upcase(text) =:  'DEVICE=' then
               do;
                 winfound   = .;
                 defaultprt = scan(text,2,'=,')!!' on '!!scan(text,4,'=,');
                 call send(_self_,'_set_label_',defaultprt);
               end;
            if upcase(text) =: '[PRINTERPORTS]' then prtsfound = 1;
            if prtsfound = 1 and text = _blank_ then prtsfound = .;
            if prtsfound and upcase(text) ^=: '[PRINTERPORTS]' then
            lst = insertc(lst,scan(text,1,'=')!!' on '!!scan(text,2,','),-1);
          end;

          rc = fclose(fid);
       end;

    call super(_self_,'_POSTINIT_');

    call symput('windev','');

  endmethod;


SELECT:      /* WINDEV.CLASS : _SELECT_ method override */

* Display a list of available windows output devices and stores the users
  selection in the macro variable 'WINDEV';

  method;

    call send(_self_,'_get_label_',text);

    if listlen(lst) then
       do;
         pm = popmenu(lst);

         if pm then call send(_self_,'_set_label_',getitemc(lst,pm));

         if getitemc(lst,pm)=: '- Default' then call symput('windev','');
         else
           do;
            call symput('windev',getitemc(lst,pm));
            call symput('windev','[printer.setup(''!!getitemc(lst,pm)!!'')]');
           end;
       end;
    else
       do;
         _msg_ = 'Default Printer Only is available';
         alarm; refresh;
       end;

  endmethod;


PRETERM:     /* WINDEV.CLASS : _PRETERM_ method */

* Delete the list of windows devices;

  method;

    if lst then rc = dellist(lst);

  endmethod;
```

From within your frames SCL you can use the macro variable &windev to cause output to be sent to the desired device. e.g.

```
    submit continue;

          filename xlsys dde 'excel|system' notab;

          data _null_;
            file xlsys;

            Other commands.....

            put &windev;
            put '[print()]';
          run;

    endsubmit;
```

Composite Widgets

A composite widget is an object which is a collection of other objects that are always used together. The advantage of composite widgets is that when you have a group of objects that are always required together on a frame you can build a model of those objects. This model defines how the group of objects should be placed on the screen. Each of the objects within the composite widget will still function as it was originally designed to and you can communicate with the objects collectively or individually.

Below is an example of composite widget which can be used when editing a SAS dataset.

Considered the functionality required by a data editing application:

- Scrolling through observations.
- Adding, duplicating and deleting observations.
- Finding observations.
- Validating data.
- Auditing data modifications.
- Data security.
- Help.
- etc.....

These are all common functions required by any data editing application. So why not create objects that manages the entire process for any collection of data.

Figure 6, shows an example of a frame that uses such objects. To create such an application four objects had to be built. The main two objects were :

- a composite object made up of a frame class, Dataset Data Model and Dataset Data Vector. This object controls scrolling, searching, security, additions, deletions, etc... to the dataset and the displaying of data on the frame.
- the second object contains the basic screen layout for an editing application. It is made up of 6 command buttons, 4 control objects, 2 extended text entry objects, 3 contain boxes and an image object. The buttons on the bottom of the screen issue commands that the frame object intercepts and actions to prove the desired functionality.

The third and fourth objects are sub-classes of the extended text entry. The third is used to display the contents of each variable within the dataset. It contains the functionality required to control data validation, auditing of changes and variable help. The fourth is simply used to quickly create variable labels based on the datasets variable labels.

Because the object knows how to perform the above functions the developer does not need to be concerned with developing that functionality.

By simply placing objects on a frame and altering their data elements you have any application that performs all the functions required when editing datasets.

Figure 6

Building such an application will be demonstrated as part of this papers presentation at SUGI 22.

Conclusion

Using the techniques discussed in this paper (macros, sub-classing and composite widgets), complex and/or time consuming functions can be built into SAS applications can be developed extremely quickly. Writing quality reusable code that will meet many application development needs will save considerable time in the long run. Applications can instantly contain many user friendly features which would usually take considerable time to develop. The need to write code for SAS/AF frame applications can be totally eliminated with the development the appropriate objects.

Author Contact Details :

Author : Neil Davis, Manager Personnel Systems

Address : ANZ Personnel
PO Box 1492
Wellington
New Zealand

Telephone : 0064-4-496 8573

Facsimile : 0064-4-496 7355

Email : A676068@ANZ.COM