Secret Menu Tricks: Using Lists to Manipulate Objects

James Glidden, Trilogy Consulting Corporation

ABSTRACT

The following paper describes the use of lists in manipulating objects on an SAS/AF® frame. Beginning with Version 6.11 of SAS®, it became possible to create new objects on the screen, or terminate them any time during a frame session. This is far more useful than the more traditional approach of simply hiding or unhiding an object. In addition, all of the information about the screen display, including type, color, width, and so forth, can be loaded into a list. This can be used to either change a current frame or one in another session. This paper shows how to load all of the information about a frame into a list, store it, and subsequently recreate it at runtime. In addition, we will see how to expand the perimeters of a frame to fit the entire monitor without using attachments and ultimately, how to sub-class the frame class to include these changes. It is intended for developers who understand basic SAS/Frame concepts and would like to become more proficient.

INTRODUCTION

To begin, let us go over some of the basic features in SCL that will be needed to accomplish this task.

One of the basic building blocks of SCL is the list, which is a collection of different kinds of information, such as character values, numeric values, and even sub-lists. During a Frame session, data can be taken in and out of the list. A powerful feature is the ability to put the list into a catalog as a stored list, or s-list. Information from the stored list may be used in another session. Common examples of s-lists include user information and report requests passed from session to session.

The concept of classes is also very important. Essentially, every object on the screen is a class. All of these classes share a Superclass known as the Widget class, which is why objects on the frame are called 'Widgets'. The frame itself is a Superclass of both the Widget class and their smaller sub-classes. There is even a Class Class, which is the Superclass of all of these. Nearly every class has a set of methods, or subroutines, which allow the SCL program to communicate with the object. Each time a notify command appears in a program, it is calling an object through a method.

The _GET_WIDGETS_ method of the frame class and the _GET_REGION_ method of the Widget class are the methods that will be used in putting the frame into a list. The _NEW_ class of the Class Class will be used later to place the widgets onto a new screen.

CREATING THE SLIST

The very first step is to build the menu itself. The screen is designed as any normal frame, but SCL will be used to save it rather than operate it. The following command loads information about every widget on the screen into a list:

```
call notify('.','_get_widgets_',widget_list);
```

This list contains a series of sub-lists, each of which contain information about a widgets's name, color, class, and so on. Unfortunately, what these lists do not contain is any information about the location of the widget on the screen. There is a missing piece, the region list, that exists now as an empty sub-list of the widget list. To recreate this widget elsewhere, we are going to need this information. The solution is the _GET_REGION_ method, which generates a list containing the following four items: ULX (Upper Left X Axis), ULY (Upper Left Y Axis), LRX (Lower Right X Axis), and LRY (Lower Right Y Axis). To transfer the region list into the widget list, use the following:

```
do i=1 to listlen(widget_list);
   a object_name = nameitem(widget_list,i);
    object_list = getiteml(widget_list,i);
    call notify(object_name,'_get_region_',
                object_region_list);
    rc = setniteml(object_list,object_region_list,
                '_region_');
    rc = setiteml(widget_list,object_list,i);
end;
```

The widget list is now ready for storing. To create the s-list, compile the frame, and run Testaf once. As you might expect, a copy of the frame should be saved as well as the s-list because s-lists are difficult to recreate or alter.

BRINGING BACK THE SLIST

To bring back the s-list in another session, simply load it into a list:

```
search_list = makelist();
```

```
rc =fillist('catalog',{filename},search_list);
```

Then create a loop that takes each sub-list and creates a new class using the information in the list as a feed-in.

```
do i=1 to listlen(search_list);
         this_list = getiteml(search_list,i);
        classname = getnitemc(this_list,
                        '_classname_');
         thisclass =loadclass('SASHELP.FSP.'
                                ||scan(classname,1));
        call send(thisclass,'_new_,new_obj_id,
                    this_list);
end;
```

All of a frame can be brought back as a s-list or a portion of one. If this frame is being placed onto a screen with existing widgets, however, an object may not appear if it clashes with one already on the screen. Caution should be used in such situations. SAS provides other capabilities, such as composite classes, that may be preferable.

DIFFERENT MONITOR SIZES

One problem that application developers run into, especially in the Unix environment, is how to deal with frames on different size monitors. I once spent several weeks building an SAS/AF system, not realizing I was working on one of the smallest terminals in the company. When I tested my programs on another employee's machine, I would have a frame that only covered 35-40% of the monitor.

I talked to a fellow programmer, and he suggested that 3 different size screens, small, medium, and large, be set up, which had been done on a SAS/AF application written several years before. Unfortunately, this was no solution either. Not only were the monitors different but so was the resolution. One monitor might be set at 1230x1076, another at 640x480. On the first monitor tested with a larger frame, the Pushbuttons on all of the frames were below the bottom of the monitor.

The next step was to try attachments, and while this was more viable, it was also very time-consuming.

The different size screens described above were created based on the number of columns and rows found at login. Given the ability to store information about widgets, the question now was the possibility of changing the items in the region sub-list to fit proportionally to the monitor size.

Before the widgets can be resized, the size of the current monitor has to be determined by using the following commands:

```
call notify('.','_winfo_','numcols',numcol);
call notify('.','_winfo_','numrows',numrow);
size = numcol/85;
rsize = numrow/33;
```

The numbers, 85 and 33 are defaults that represent the size of a smaller monitor. Your default numbers may be different. In this instance, If the number of rows on a monitor is 125, the size from left to right would be approximately 1.5. And while it is possible to take a large screen and shrink it, it is preferable to generate the frame from a smaller size and grow it.

To change the size of a widget, its sub-list must be taken out of the widget list, its region list extracted, and the ULX, ULY, LRX, and LRY items changed to fit the new screen size:

```
region_exists = nameditem(this_list,'_region_');

if region_exists and size ne 1 then do;
      this_region = getniteml(this_list,
                       '_region_');
      rc = setnitemn(this_region,
            getnitemn(this_region,'ulx')*size,
            'ulx');
      rc =  setnitemn(this_region,
             getnitemn(this_region,'lrx')*size,
             'lrx');
      original_uly = getnitemn(this_region,
            'uly');
      new_uly = original_uly*rsize;
      original_lry = getnitemn(this_region,'lry');
      new_lry = original_lry*rsize;
      rc = setnitemn(this_region ,new_uly,'uly');
      rc = setnitemn(this_region,new_lry,'lry');
      rc = setniteml(this_list,this_region,
            '_region_');
 end;
```

Not every widget will size correctly. The extended table widget requires its code to be performed in a GETROW or PUTROW, and cannot be included as part of a saved frame. Some widgets, such as Pushbuttons and Icons, on the other hand, do not grow horizontally. To accommodate such special cases, code is required which measure the approximate location on the screen, then grows the widget vertically but not horizontally:

```
if getnitemc(this_list,'_classname_') in
   ('ICON.ICON','EFIELD.EFIELD',
    ''TLABEL.TLABEL',
    'PBUTTON.PBUTTON' ) then do;
        midpoint = mean(new_uly,new_lry);
        widget_position = (original_lry-
                            original_uly)/2;
        new_uly = midpoint-widget_position;
        new_lry = midpoint+widget_position;
end;
```

Also, stored Container boxers must be reset on the new screen. Widgets identified as Container Boxes should be stored in a separate list, and that list be used to resize all of the container boxes at the end of the process.

```
if getnitemc(this_list,'_CLASSNAME_') eq
              'CONTAINR.CONTAINR'
    then rc=insertc(container_list,name,-1);

(Later)
do l=1 to listlen(container_list);
    call  notify(getitemc(container_list,l),
                 '_snug_fit_',1);
end;
```

DEALING WITH MONITOR SIZE FOR AN EXISTING SCREEN

So far, we can resize a monitor based on information stored in a s-list, which would not be an appealing solution for an already existing application. The trick with a frame that already contains widgets is to load that frame into a list, delete everything out of it, and recreate the widgets using sizes based on the existing monitor.

```
 search_list = makelist();
 new_list = makelist();

call notify('.','_get_widgets_',search_list);

 do ii=1 to listlen(search_list);
    object_name = nameitem(search_list,ii);
    object_list = getiteml(search_list,ii);
    object_region_list = makelist();
    call notify(object_name ,'_get_properties_',
                object_region_list);
    object_region_list =
      getniteml (object_property_list,'_region_');
    rc = setniteml(object_list,
                   object_region_list,'_region_');
    rc = setiteml(search_list,object_list,ii);
 end;
```

Once a widget is destroyed, the list from which it came is also destroyed. In that case, a copy of the list needs to be made prior to destroying it, and the new list used as the source of the frame.

```
rc = copylist(search_list,'Y',new_list);

do jj=1 to listlen(search_list);
   this_list = getiteml(search_list,jj);
   classid = getnitemc(this_list,'NAME');
   call notify(classid,'_term_');
end;
```

WHERE DOES ALL THIS CODE GO?

The first screens using this technique simply linked to another label in the SCL. While this worked, it also involved a lot of repetitive code in every SCL program. One solution was 2 macros:

%setwidge - which resized an existing monitor.
%addwidge - which created a screen from an s-list.

These certainly worked, but the macros had to be compiled prior to compiling the SCL, and not surprisingly, this was sometimes forgotten. The ultimate solution proved to be the incorporation of this code as methods in a sub-classed Frame Class.

SUBCLASSING THE FRAME CLASS

A new product called Classbrowser is available for displaying the hierarchy of classes available in SAS. One of its capabilities is sub-classing.

To start, enter classbrowser on the command line of display manager.

When the browser appears on the screen, the Entry Type should be changed to Class. The Frame Class is found in the SAS Help directory under the FSP catalog, and can be selected by clicking on the Class entry.

Sub-classing is available on the Locals section of the pmenu. When that is chosen, a Subclass screen appears that contains 2 entry boxes. One is for Class Entry, which includes the library, catalog, and name of the new sub-class. The other is for Description, which is a short English description describing the new class.

After pushing OK, a build screen appears for the new class. By clicking on the Methods selection under Additional Attributes, another screen appears which shows all of the available methods. To add a new one, click on the Action button and choose Add Mode On. In our example, we are planning to add the method that resizes existing widgets. The name of this new method will be Resize, and it is to be included in SCL stored under the same name and catalog as the class. By pushing the New button, the method is added to the class.

So far there is a new class which references a new method in SCL that does not exist. To rectify that situation, we need to exit the Class Browser, and enter the Build Screen. After a new SCL entry is created, the code for re-sizing widgets should be copied into the blank SCL.

Above the copied code, put

```
RESIZE:
 method;
```

Below the code, put

```
endmethod;
```

Also, if the code originated in a macro, remove the macro and mend statements. After the SCL is compiled, the method becomes part of the new class.

The final step is to add this class to the Resource list found in Profile Catalog of the SASUSER library, and change it to the Active Class.

The new class has all of the capability of the original frame class, plus an extra method, RESIZE, that changes the size of all of the objects on the screen to fit the current monitor. It can be called by including

```
Call send('.','resize');
```

at the beginning of the INIT section of the SCL code. Unfortunately, this method will not be available for frames built prior to its creation, but it can be used in all future frames.

CONCLUSION

This paper only scratches the surface in screen manipulation. SAS offers a number of other options, such as sub-classing the Frame class to create specific screens, or the use of composite classes. My experience since the development of the concepts described here is an increased flexibility in Frames applications. The ability to set up Frame programs with multiple screens allows the development to be quicker and at the same time more powerful. Also, once the programming to deal with monitor sizes was completed, it ceased to be an issue, and attachments were rendered unnecessary. I hope this paper provides ideas developers can expound upon.

REFERENCE

SAS/AF Software: Frame Class Dictionary, ver 6, 1st Ed, published by the SAS Institute, Cary NC

ACKNOWLEDGEMENT

The author would like to thank Don Clements
For providing some of the ideas used in this paper.

AUTHOR CONTACT INFORMATION

The author can be contacted at
Jbglidden@msn.com

APPENDIX

```
%macro setwidge;

* Creates lists needed for this process;
search_list = makelist();
container_list = makelist();
new_list = makelist();

* Determines rows and columns of this screen;
call notify('.','_winfo_','numcols',numcol);
call notify('.','_winfo_','numrows',numrow);

* Determines size proportional to original frame;
size = numcol/85;
rsize = numrow/33;

* Loads all of the widgets into a list;
call notify('.','_get_widgets_',search_list);

* Loads region information for each widget by using
the _get_properties_
  method and transferring the _region_ list into the
seach  list;
do ii=1 to listlen(search_list);
    object_name = nameitem(search_list,ii);
```

```
      object_list = getiteml(search_list,ii);
      object_property_list = makelist();
      call notify(object_name, '_get_properties_',
                  object_property_list);
      object_region_list =
              getniteml(object_property_list,'_region_');
      rc = setniteml(object_list,object_region_list,
                     '_region_');
      rc = setiteml(search_list,object_list,ii);
   end;

 * Makes a 2nd copy (new_list) of search_list;
 rc = copylist(search_list,'Y',new_list);

 * Erases all of the widgets from the screen;
 do jj=1 to listlen(search_list);
     this_list = getiteml(search_list,jj);
     classid = getnitemc(this_list,'NAME');
     call notify(classid,'_term_');
 end;

 * Now that the screen is clear, takes the 2nd copy of
 the widget_list (new_list),   (The first copy
 (search_list) was made unusable when the widgets
 were erased from the screen), and returns them to
 the screen resized to the proper proportion;
 do kk=1 to listlen(new_list);
   this_list = getiteml(new_list,kk);
   name = getnitemc(this_list,'NAME');
   region_exists = nameditem(this_list,'_region_');

 * Resets the size of this widget to the size of this
 screen by changing the ulx, uly, lrx, and lry
 values in the region sublist;
   if region_exists and size ne 1 then do;
     this_region = getniteml(this_list,'_region_');
       rc = setnitemn(this_region,
       getnitemn(this_region,'ulx')*size,'ulx');
       rc = setnitemn(this_region,
            getnitemn(this_region,'lrx')*size, 'lrx');
       original_uly = getnitemn(this_region,'uly');
       new_uly = original_uly*rsize;
       original_lry = getnitemn(this_region,'lry');
       new_lry = original_lry*rsize;

 * If this is a widget which cannot be increased
 vertically,  the uly and lry are reset so that the widget
 is placed in  the middle of the area it would have
 occupied if it could have been increased vertically;
       if getnitemc(this_list,'_classname_') in
         ('ICON.ICON','EFIELD.EFIELD',
          'TLABEL.TLABEL',
          'PBUTTON.PBUTTON') then do;
         midpoint = mean(new_uly,new_lry);
         widget_position =
         (original_lry-original_uly)/2;
         new_uly = midpoint-widget_position;
         new_lry = midpoint+widget_position;
       end;
       rc = setnitemn(this_region ,new_uly,'uly');
       rc = setnitemn(this_region,new_lry,'lry');
       rc = setniteml(this_list,this_region,'_region_');
   end;

  * Resets the length, row and column of this widget
 to the size of this screen;
   if nameditem(this_list,'length') then
     rc = setnitemn(this_list,
          getnitemn(this_list,'length')*size,'length');
   if nameditem(this_list,'row') then
     rc = setnitemn(this_list,
          getnitemn(this_list,'row')*size,'row');
   if nameditem(this_list,'col') then
     rc = setnitemn(this_list,
          getnitemn(this_list,'col')*size,'col');
   if nameditem(this_list,'_len_') then
     rc = setnitemn(this_list,
          getnitemn(this_list,'_len_')*size,'_len_');

  * If this is a container box, adds to container list;
   if getnitemc(this_list,'_CLASSNAME_') eq
     'CONTAINR.CONTAINR' then
   rc=insertc(container_list,name,-1);

  * Creates the widget by loading its class and
   running '_new_' method;
 classname = getnitemc(this_list,'_classname_');
     thisclass =
    loadclass('SASHELP.FSP.'||scan(classname,1));
     call send(thisclass,'_new_',new_obj_id,this_list);
   end;

  * Snug fits all container boxes by using the
 container list;;
   do l=1 to listlen(container_list);
     call notify(getitemc(container_list,l)
                 ,'_snug_fit_',1);
   end;
 %mend setwidge;
```