Chapter 1
Screen Design for FRAME Entries

1.1 Overview

This chapter presents some ideas for designing screens used with FRAME applications. These ideas are a combination of tried and tested techniques, my ideas and those of others that I have found work in practice. The code in this chapter allows you to use a set of tools and ideas immediately in your applications.

GUI development is different from the old menu-driven "select a number and be branched off somewhere to fill out a couple of fields" system. A GUI is not just a pretty face; it is also the first and often only thing that forms a user's opinion of a system.

Here is a statement worth considering when building a GUI screen:

Users don't care what happens behind the scenes. They don't care about case tools, traditional structured programming, or object-oriented techniques. As far as they are concerned, the system is the GUI that they interact with.

Few developers are trained to develop screens that are ergonomic. We know that if the user clicks a certain box, certain events must happen, and we code those events with relish. How much thought do we put into whether the user can easily use the screens we create? Who uses our systems, and what right have we to dictate to the end user how their system interface will look? When developing GUI applications in an organization, we should work to accepted organization standards and involve users in determining the look and feel of the screen.

1.2 Widgets

We can add extra widgets at run time by creating dynamic objects using the CLASS class _NEW_ method. The ability to swap regions in and out also allows additional widgets to be viewed in an application without having them on screen (hidden, grayed, or visible). From Release 6.12 on, the TAB LAYOUT object allows us to place practically any number of objects in a FRAME without making the screen cluttered.

As GUI designers, we need to maximize visual impact as well as provide users with a screen that suits their work habits and methodologies. If a user requires many fields on a screen and regularly works with all that data, then we maximize the user's ability to carry their job out by having all those fields on screen.

If they regularly work with a few fields that are physically divorced from an aspect of their job that uses other fields, then putting all the fields together may be confusing.

There's only one hard and fast rule. Get the users to help design (or completely design) the GUI. We are just here to make their job a bit easier, not to impose our interpretations of how we think they work. At the very least, create a prototype and have the user review it.

1.2.1 The Types of Widgets

An object class is a template like a push button. We create a region using the mouse to drag out a rectangular box and fill it with a particular object class. These regions are called widgets.

A widget is a type of object. The other type of object is called non-visible or non-widget. Non-visible objects are not seen by the user and do their work behind the scenes. These are not referred to as widgets.

Push buttons take a bit of extra space but are a fairly standard way of presenting any sort of clickable selection. They can look at bit disjointed when you have different sized ones together, consequently it's not unusual to see buttons of a standard size that contain varying amounts of white space. Note that push buttons require an extra character on all sides of the visible button. Don't use push buttons to display non-clickable information.

Icons are considered more graphically pleasing because they include a picture. You are limited in your selection of icons. You can define your own, but it is a lengthy task and defeats the point of rapid applications development. Consult "Creating User-Defined Icons to Use with the SAS System" in *Observations* (Second Quarter 1994). You can only use SAS fonts with this object.

Text entries aren't intended for command driving, however, you can associate a command that is executed when the field is modified with a text entry object. You can make text entries look like buttons. I often use a border around them, as the region is not always obvious, especially if the pad character is blank. This object is irritating with its insistence on not permitting use of the first and last apparent character position. You can only use SAS fonts with this object.

Graphics text can be used just like text labels, but you can change the type font to make them look nicer, and they are graphical rather than character. You can also cram different lengths of text into the same size boxes. Try to avoid this as it can look awful! You can only use SAS/GRAPH fonts for this object class.

SAS/GRAPH and IMAGE widgets let you use as small a region as you like because the picture is scaled to fit. Be careful — you can display an unintelligible picture: something created at 640X480 and displayed at about 40X40 can't maintain all the information it started with. The more pictures you use, the longer it will take for your application window to open and redraw. You can use region attributes to make GRAPH and IMAGE widgets resemble a button. Image objects maintain the picture better than GRSEGs, however, the image will not necessarily fill the whole region if you use the KEEP ASPECT RATIO attribute; if you don't use that attribute, the image may be distorted.

Image icon is an image with text attached. It is different from an icon because you can combine a picture with text. You can use operating system fonts for the text.

Toolbars are a collection of images or text all under one region. I like using these as I can define a whole set of buttons easily. Toolbars allow you to mix text and graphics. You can use operating system fonts for the text.

Extended text entries are a graphical form of text entry. They accept characters as input, but the object is inherently graphical, allowing it to be moved on a pixel basis rather than a character basis. Extended text entries only accept character input and can use operating system fonts rather than SAS system fonts. This object is powerful in its ability to allow the entry of a character (without an ENTER) to be an event that drives a method called _FEEDBACK_. You can have code running as the user types. For example, you could ask the user to enter some text and have the _FEEDBACK_ method display a listbox that displays only the records in a dataset that match the currently entered text.

Input fields are a simple input mechanism. They are a composite widget, utilizing an extended text entry for the actual data input. They offer the ability to use formatted input and display.

Composite widgets are a special type of widget that is based on multiple classes. An example is the image icon, which is built from more than one object class. If you use cursor tracking with composite widgets, you may need to switch on the push button region attribute because the region may only track on the border otherwise.

TAB LAYOUT objects arrived with Release 6.12. They have the inherent ability to place many objects in a FRAME, while only displaying the ones on the currently selected tab. I think this object should be used extensively; it allows an application to group tasks simply and elegantly.

1.2.2 How Many Widgets Can Fit on a Screen?

FRAME entries make it easy to forget that we can call up other FRAME entries to do additional tasks. Its easy to see some spare space and put another field in it. Question whether the overall impact of the screen is improved by doing that or whether calling another entry would be better. My experience is that I often get carried away with the ability to get everything in one place, and I have to go back and re-examine the screen when I run TESTAF on it. However, with the TAB LAYOUT object my whole approach to screen design has changed, and the cluttering that can result in FRAMES is minimized.

A related issue is the choice of which screen resolution to use during development. Use the lowest common denominator. If your site has predominantly 640X480 screens with 16 colors, don't develop at 1024X780 with 16 million colors. 640X480 with 256 colors is a good size. Be aware of and guided by your client's hardware.

Attachments are a feature of FRAME entries intended to give developers control over the resizing of objects in relation to other objects and the FRAME master region. You can use attachments to maximize your ability to run an application on different screen sizes.

I find attachments work best when all your objects are graphic based. You can resize a FRAME master region so that character-based objects disappear off the edge of the FRAME because they cannot be shrunk beyond a certain size. In particular with push buttons, you may find that graphic image icons work better than attachments.

1.2.3 Listboxes

Make extensive use of listboxes to aid users with data entry. Listboxes used in this manner are often known as a component of data driven software because, instead of making the user enter a value and possibly get it wrong, the user is given a list of items to select from.

Remember that a LISTBOX object contains the entire set of listbox items in the object. The listbox items exist in memory while the listbox exists. You can quite easily run out of memory if you use listboxes that have a lot of items.

Don't give users listboxes with so many items that they become counter-productive. As a rule of thumb, I try to avoid using listboxes with more than three to four screens of scrolling.

Revised Stanley Chapter 1 10/19/98 6:03 PM Page 6

1.2.4 Prompting Widgets

A prompting widget is one that contains a message such as 'Please Enter Your Name'. Instead of using a prompting widget, the title area of a widget can provide prompting if the widget is wide enough. You can use the entry field with the title 'Enter Name' at the top, with an offset of three to position the title. By not using that prompting widget, you gain a lot more usable screen real estate. From Release 6.11 on, you can use operating system fonts in the title area.

If you position the title as I have suggested in the title area, the pixels between the title and the region are not part of the region. If you click on these pixels, nothing happens. If you are using cursor tracking, the pointer is in the FRAME master region, not the widget region. It isn't clear from documentation that the area between the region and the title is not part of the region, but that is the case. The title itself is part of the region, and if you enable cursor tracking or define tool tips, then these will function when the pointer is on the title.

1.2.5 Objects from Other Applications

With OCX and OLE, FRAME entries can use widgets that are external to the SAS software base classes. OLE was in Release 6.10; OCX is in Release 6.11 onward. You can use an existing object stored in a Visual Basic object library as if it were part of the SAS tool set. You don't have to re-invent the wheel, but you may have difficulties in migrating code because more than just the SAS application will need to be tracked and moved.

Do not expect to achieve portability with OCX and OLE objects. You will not in general be able to build an application using Windows objects and then port it to UNIX or other non-Windows based systems.

1.2.6 Popmenus

You need to consider placement of popmenus, particularly where a selection leads to another popmenu. I often code the popmenu start position to ensure that I'm not too near the screen bottom, and also to ensure that the popmenus that follow will appear in a logical place.

You can jazz up popmenus a little by judicious use of the SETLATTR function to gray items. This allows information items to appear in the popmenu. For example, I had a popmenu that could be selected by clicking on a filename in an extended table field. The popmenu was to provide the three options Add, Edit, and Delete. I added a header 'Select an option' followed by a blank item; both the header and the blank item were set to inactive by using the SET-LATTR function. At the end, I added a blank and a "do nothing" option; the blank was inactive. It's far from obvious to many users that the popmenu will disappear just by clicking somewhere off the popmenu or by using the escape key. I prefer to provide a "do nothing" option that closes the popmenu.

In SCL, you can use code like the following:

```
SELECT (popmenu(<listid>)) ;
when (1) ...
when (2) ...

otherwise ;
END ;
```

The OTHERWISE, which does nothing in this example, executes when the "do nothing" item is selected.

1.2.7 Modifying Objects at Run Time

You can change some widget attributes at run time. For example, suppose you have a screen that requests a user to enter something. On one invocation it may be a SAS dataset name. On another it may be a SAS variable name. The two have different lengths, the first being up to 17 characters, the second being up to 8 characters.

It is not necessary to have different FRAME entries to enter the different types of fields, nor is it necessary to use the same FRAME entry with extra spaces. You can use the _RESIZE_REGION_ method at run time to set the size of the field. For extended text entries, you could use the _SET_MAXCOL_ method, which will always leave the widget the same physical length on screen. However, it will restrict the number of characters that the user can type in.

You can also move regions and change the layout of a FRAME entry screen to suit different circumstances. For example, you could allow your users to define at install time whether they want EXIT/CANCEL/HELP buttons at the bottom or arranged vertically down one side. Store the information in a SASUSER profile, and access the button placement in your FRAME entry's SCL INIT section.

You can create widgets at run time by using a feature of the CLASS class called run-time instantiation. This uses the _NEW_ method. You can also create non-widget classes in this way.

1.2.8 Using the _FEEDBACK_ Method with Extended Text Entries

The following example creates a listbox at development time but hides it until the user wants to use it. The listbox contains a list of items that match the data typed into an extended text entry. There is a checkbox that the users check if they require assisted entry in an extended text entry. A listbox appears, and the extended text entry's _FEEDBACK_ method is overridden to assist with the data entry.

The BUILD mode screen is shown in Figure 1. It consists of an extended text entry (called REQUEST) surrounded by a container box that provides a border. Within the container box, under the extended text entry, is a listbox (called PROMPTR). Note the checkbox with the text 'Assisted Entry On'. The checkbox is called ASSIST.

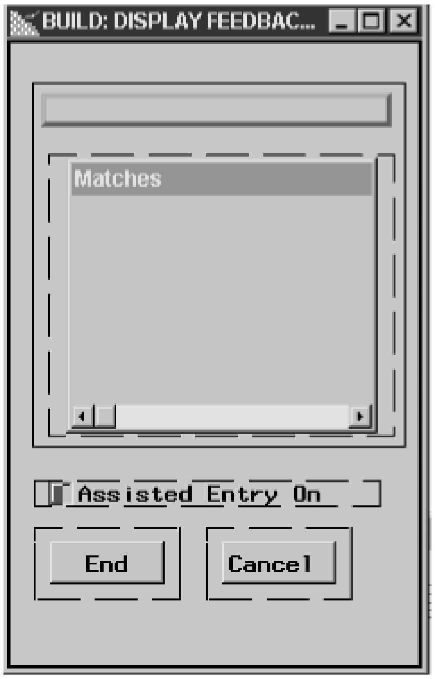


Figure 1: Development Screen with Listbox

When the application starts, the listbox is hidden. If the user checks the 'Assisted Entry On' check box, the listbox appears and displays the data where the characters typed into the extended text entry match the first letter of the listbox items. The assisted entry uses a dataset called SASHELP.COMPANY.

A sample screen is shown in Figure 2. In this figure, the user has switched on assisted entry and has entered an S in the extended text entry. The listbox displays only the items that start with an S.

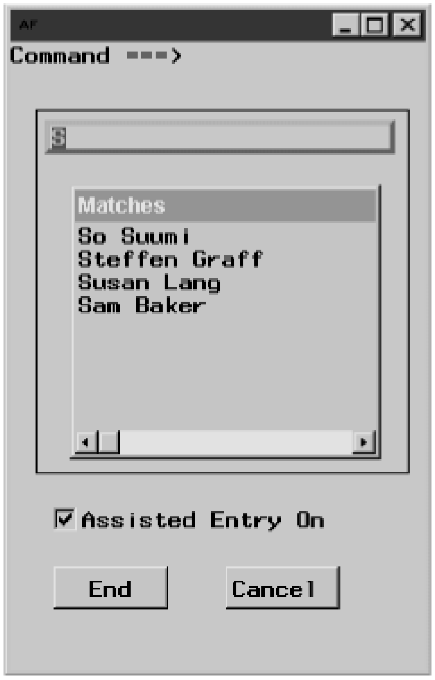


Figure 2: Run-Time Screen with Listbox

When the user selected the check box, the code allowed the _FEEDBACK_ method to execute, causing each keystroke to run the method. Checking it again will switch off the keystroke feedback.

The checkbox label has changed in Figure 2. It now includes a tick (✓) in the box to indicate that it is presently selected (switched on). Checking it now would switch off the assisted entry. The tick is platform dependent. Under UNIX Solaris, for example, it is a filled in diamond shape.

The SCL code associated with this FRAME entry is as follows:

```
length rc 3 text $ 30 ;
```

Place the cursor in the extended text entry, hide the list box that is used for prompting, and switch off feedback mode.

```
init:
  call notify('request','_cursor_') ;
  call notify('promptr','_hide_') ;
  call notify('request','_set_mode_','NONE') ;
  call notify('request','_set_instance_method_','
         _feedback_',
         'sasuser.book16.methods.scl','fback') ;
return ;
```

This is the labeled code for the checkbox. Start by finding the current state of the checkbox.

```
assist:
```

If the assisted entry is being switched off, hide the prompting object, switch off feedback mode, place the cursor in the extended text entry and close the data set used for prompting.

```
  if assist eq 'OFF' then do ;
     call notify('promptr','_hide_') ;
     call notify('request','_set_mode_','NONE')
     call notify('request','_cursor_') ;
1     if dsid('sashelp.company') then
        rc=close(dsid('sashelp.company')) ;
     return ;
  end ;
```

Switch on assisted entry. Open the dataset that the listbox is based on. Then switch on feedback mode and display the listbox.

```
  dsid = open('sashelp.company','i') ;
  call notify('request','_set_mode_','ALWAYS') ;
  call notify('promptr','_unhide_') ;
```

Place the cursor back in the extended text entry.

```
  call notify('request','_cursor_') ;
```

Check if there is already text in the extended text entry. If there is, force execution of the feedback method to set the listbox up correctly. Otherwise, populate the list box with all possible data.	`  call notify(‘request’,’_get_text_’,text) ;`
	`  if text ne _blank_ then`
	`     call notify(‘request’,’_feedback_’,’ ‘,.,.) ;`
	`  else call notify(‘promptr’,’_repopulate_’) ;`
	`return ;`
	
	`term:`

Close the dataset if it is open.	❶ `if dsid (‘sashelp.company’) then`
	`        rc=close(dsid(‘sashelp.company’)) ;`
	`return`

When a listbox item is clicked, the contents of the listbox appear in the extended text entry	`promptr:`
	`  call notify(‘promptr’,’_get_last_sel_’,rc,`
	`              rc,text) ;`
	`  call notify(‘request’,’_set_text_’,text) ;`
	`return ;`

❶ This is an unusual way to close a dataset. In this example, the dataset is closed each time the user switches off assisted entry. By the time we get to TERM, the dataset may or may not be open. However, the CLOSE function only removes the association between the DSID and the dataset; it does not set DSID back to missing or zero. Thus a statement such as

```
rc = close(dsid) ;
```

or

```
if dsid then rc = close(dsid) ;
```

is likely to fail at some point because the DSID may not be associated with the dataset. A sure way to find out if the dataset is still open is to use the DSID function as I did above.

Now code the _FEEDBACK_ override. Given the above instance method definition, this code would go into SASUSER.BOOK16.METHODS.SCL.

```
length text $ 200 varname $ 8 rc 3 ;
```

This statement stops the compiler from issuing warning messages.

```
_self_ = _self_ ;
```

To override a method, follow the directions in the *SAS/AF Software: FRAME Entry Usage and Reference.*

```
fback:
method event $ 20 line 8 offset 8 ;
```

Find out what the user has entered so far.

```
  call send(_self_,'_get_text_',text) ;
```

Get the listbox object ID so that methods can be sent to it from here.

```
  call send(_frame_,'_get_widget_','promptr',
            list box) ;
```

Find the data set that is the source of the listbox.

```
  dsid = dsid('sashelp.company') ;
```

Build a WHERE clause to restrict the listbox to only the items that match the text already entered. Note that this implies that the input entry matches the dataset text, i.e., no formatting.

```
  rc = where(dsid,'level5' !! ' like "' !!
             text !! '%"') ;
```

Repopulate the list box to show just the restricted records.

```
  call send(list box,'_repopulate_') ;
endmethod ;
```

Users do not see the listbox until they switch on assisted entry by clicking the checkbox. If they click on a listbox item, that item is moved into the extended text entry.

1.3 Screen Layout and Data Flow

One of the first factors to consider when planning your GUI screen is the logical flow of data entry. Does TAB always take you to a sensible place? Does ENTER cause processing that triggers errors in fields not yet filled in? Should pressing ENTER automatically end the window and return to a parent or should it spawn a child?

There are certain generally accepted rules of screen layout and processing flow:

- TAB should proceed through a group of related fields before going back to unrelated fields. In FRAME, you achieve this by placing the related objects inside a container box object.
- Fields that cannot be selected should be hidden, grayed, or protected. In FRAME, you use the _HIDE_, _GRAY_, and _PROTECT_ methods to do this.
- The cursor should be placed in the first data entry field on screen when the GUI starts. However, on return to the window from a second screen, the cursor should be where it was before the second screen was displayed.
- Don't do error processing for fields that haven't yet been filled in unless the user tries to advance to the next function. Display a message that the field must be filled in if the user tries to leave the screen.
- Tabbing to a clickable field and pressing ENTER should trigger the same processing as clicking on the field with the mouse.

1.3.1 Automatic Ending on Execution of Commands

In some applications, a button such as $\boxed{\text{OK}}$ or $\boxed{\text{END}}$ is highlighted when you access the window. This highlighted button is the default button. When the window opens, the cursor is placed in a different field. Pressing ENTER executes the command that is associated with the button.

Such a logic flow is so tightly ingrained in Windows and OS/2 GUI screen design and interface that it seems a glaring omission in FRAME entries. When you do see a SAS window in SAS that associates a command with a highlighted default button, it is usually an operating system dialog called from inside SAS, not a FRAME feature. An example is when you select the FILE → NEW → CATALOG option from the BUILD window.

You can create a pushbutton that executes a default command in FRAME entries, but you cannot highlight the border. Push buttons actually have a much larger region than is visible on screen at run time. When you modify a border, the region border, not the border of the visible pushbutton, is altered.

To create a FRAME entry that contains a default button and an associated command, follow the layout in Figure 3 and add the SCL below.

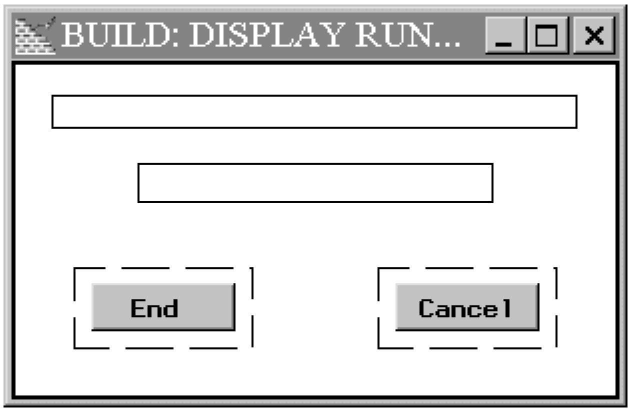


Figure 3: Sample FRAME with Autorun End Button

You call this FRAME using CALL DISPLAY.

The fields and their non-default attributes for this screen are as follows. The field listing follows the fields in the screen from top to bottom and left to right.

Field name	*Description/Attributes*
MESSAGE	This displays a prompt message from the calling program. ● Extended text entry ● Protected ● Blue text ● 40 characters ● Centered
USERFLD	This is the text that the user enters. ● Extended text entry ● 200 characters length

Field name	***Description/Attributes***
END	This will end the screen when selected. ● Push button ● Label End ● Command processing END
CANCEL	This will end the screen when pressed and always pass blank back to the caller. ● Push button ● Label Cancel ● Command processing CANCEL

The SCL for this screen is as follows:

These are called ENTRY parameters and are the means of communication between this screen and the caller. The fields are as follows: PROMPT gives this program a prompt to display. USERTEXT allows the entered data to be given back to the caller. INPUT_LENGTH is the maximum number of characters that the user can enter. EMPTY_ALLOWED determines whether a normal (non CANCEL) exit should allow a blank value to be passed back.	`entry prompt $ usertext $ input_length empty_allowed 3;`
Define a field to tell us whether various widgets have been changed or clicked.	`length select1 select2 3 ;`

Display the prompt.
 	
Set maximum entry length.
 	
Default user entry to blank.
 	
Set cursor on user entry.
 	
Allow user entry to be tabbed to.
	`init:`
	` call notify(‘message’,’_set_text_’,prompt) ;`
	
	` call notify(‘userfld’,’_set_maxcol_’,`
	`              input_length) ;`
	
	` call notify(‘userfld’,’_set_text_’,’ ‘) ;`
	
	` call notify(‘userfld’,’_cursor_’) ;`
	
	` call notify(‘userfld’,’_set_tabbable_’,’on’) ;`
	`return ;`

If a CANCEL, just exit after emptying the user entry field.
 	
Get entered text. If it is blank and the caller has specified it must not be, issue an error and continue.
	`term:`
	`  if _status_ eq ‘C’ then do ;`
	`    usertext = _blank_ ;`
	`    return ;`
	`  end ;`
	
	`  call notify(‘userfld’,’_get_text_’,usertext) ;`
	`  if not empty_allowed & usertext = _blank_ then do ;`
	`    _status_ = ‘R’ ;`
	`    _msg_ = ‘ERROR: Data Entry Required Before END’ ;`
	`    return ;`
	`  end ;`
	`return ;`

Check if the user pressed END or CANCEL. If not, force an END command. This permits the ‘autoend’ when they press ENTER on the USERFLD.	`userfld:`
	` call notify(‘end’,’_is_modified_’,select1) ;`
	` call notify(‘cancel’,’_is_modified_’,select2) ;`
	` if not select1 and not select2 then call execcmd(‘end’)`
	`;`
	`return ;`

You can place this FRAME and code in any catalog and catalog entry you desire.

In the above TERM code, I check if the field is blank and the EMPTY_ALLOWED parameter indicates it should be filled in. Then I issue a message to tell the user to fill it in. If you are happy to use the default SAS/AF message when a required field is left empty, change the INIT section to add the following:

```
  if not empty_allowed then
       call notify(‘userfld’,’_set_required_’,’Y’) ;
```

This allows SAS/AF to track that the required field is entered and issue its own message if it is not entered when you try to exit the FRAME. In this case, add the following in the TERM section:

```
if _status_ eq ‘C’ then usertext = _blank_ ;
```

This FRAME entry is a called program. This is an example of using OOP tools (the NOTIFY routine and methods) in FRAME to implement a standard SCL programming task. You do not have to create objects, override methods, or understand all about instance variables to use FRAME entries.

An example program that uses this FRAME follows. Place the program in an SCL entry, compile, and execute.

```
length paper $ 25 ;

init:
  call display(‘runok.frame’,’Enter paper type’,paper,
               8,0) ;
  sysrc = sysrc(1);
  put paper= sysrc=;
return ;
```

The value of zero for the EMPTY_ALLOWED parameter means that the user cannot END from the screen without entering some text in the USERFLD. The user can, however, CANCEL. Any other value for EMPTY_ALLOWED will allow an empty string to be passed back even if END is selected or if ENTER is pressed with the cursor on USERFLD.

The SYSRC(1) function is a special case of the SYSRC function. It permits the caller to determine whether the FRAME executed in the CALL DISPLAY was exited via END or CANCEL. If SYSRC(1) returns 0, CALL DISPLAY was exited via END; if –1 is return, the exit was via CANCEL.

It looks as though it would be simpler to have an END in the Command Processing attribute on USERFLD. However, if the user clicks the END or CANCEL button, two commands would be executed. The first one will be the USERFLD command, due to the processing order of fields. That would END this FRAME. However, the second END or CANCEL is still waiting for an opportune moment to execute. That moment will be when the next screen kicks back into life, and the screen will end itself! That is why the END command is pushed using EXECCMD only if neither the END nor CANCEL buttons were selected.

Now you have a very useful tool for your development tool box, an automatically ending entry that simulates the way many Windows and OS/2 dialogs work.

1.3.2 Navigating Around Objects on FRAME Entries

By default, when the user tabs between fields, the cursor moves from top to bottom and left to right. With FRAME entries, you can create groups of fields. When the user tabs to a grouped set of fields, the tabbing order follows the same rule within the group. Tabbing from the last element of the group moves to the next logical element on screen. Because the next logical element may be above the last element of the group, care should be taken to ensure that the user is not confused.

If a group of fields has a border, users will think that it is a grouped set of fields, and they will process these fields as a group. To create a group of fields, you must enclose the fields in a widget, such as the container box object. A container box object creates a visual border and forces objects within that border to all be tabbed before a tab takes you back to outside objects.

When you view a screen in development mode, using MAKE GROUP places a border around widgets in a similar manner to a container box. When you execute the screen, you see the border around the fields. However, only the container box will cause the tabbing and processing order to change from the default. The reason is that to change that order requires that the fields to be processed as a group be enclosed within a widget. MAKE GROUP does not create a widget, it is an editing tool only.

Try to avoid allowing users to tab to widgets that are not obviously selected. For instance, if a user places a cursor on a SAS/GRAPH object or an IMAGE object, the object may not appear to be selected. A GUI region should scream “here I am” at the user. Changing the color and width of the border is an ideal way to show where the selected graphics region is.

The _TAB_IN_ method executes whenever the user tabs to a widget. You can use this method to change the borders of a graphic. The method is also executed when you click on a widget and when you execute _CURSOR_ on a widget.

Use the _TAB_OUT_ method to override the default tabbing order. The _TAB_OUT_ method is executed whenever the user moves out of a widget. You can use this method to send a _CURSOR_ method to another widget; the user will be moved to that widget.

Suppose you have a FRAME entry that contains just two objects; each object is an image. You can show the user clearly which image is active by using the following code stored in a catalog entry named SASUSER.TABOUT.OVERRIDE.SCL.

```
tabout:
  method:

  /*
  set object border color to background and then
  execute the SAS/AF _tab_out_ code. Use background
  for the color so that the border cannot be seen.
 */

 call send(_self_,'_set_border_color_','background') ;
 call super(_self_,'_tab_out_') ;
endmethod;

tabin:
  method:

    /*
     change the border color and then execute the
     SAS/AF _tab_in_ default
    */

  call send(_self_,'_set_border_color_','pink') ;
  call super(_self_,'_tab_in_') ;
endmethod ;
```

Each labeled section is a method override. Method overrides allow us to replace or enhance the default processing performed by SAS/AF software when an object is selected. See the *SAS Institute FRAME Application Development Concepts* manual for a discussion of methods and method overrides. What method overrides essentially provide is a mechanism for adding our own functionality to that already supplied (either by SAS Institute or through another override).

By itself, the above code does nothing. Image objects must be linked with the code so that an event triggered on one of the images causes the code to execute. When a user causes a _TAB_OUT_ or _TAB_IN_ method to be triggered by SAS/AF, the code will run when the code is linked to an object.

The method override uses the CALL SUPER method, as will most of the overrides you build. CALL SUPER is used to execute the code that is supplied with SAS/AF to provide the functionality of the object. Sometimes we must call the SUPER method because it does something that FRAME cannot do without (e.g., an _POSTINIT_ override will program halt if no CALL SUPER is executed).

To create the FRAME entry screen in Figure 4, the two objects are defined as follows:

Attribute	*Object 1*	*Object 2*
Name	IMAGE1	IMAGE2
Image Name	SASHELP.C0C0C. HELP.IMAGE	SASHELP.C0C0C. DISKLIST.IMAGE
Source	Catalog entry	Catalog entry
Region Attributes	Set outline to simple	Set outline to simple

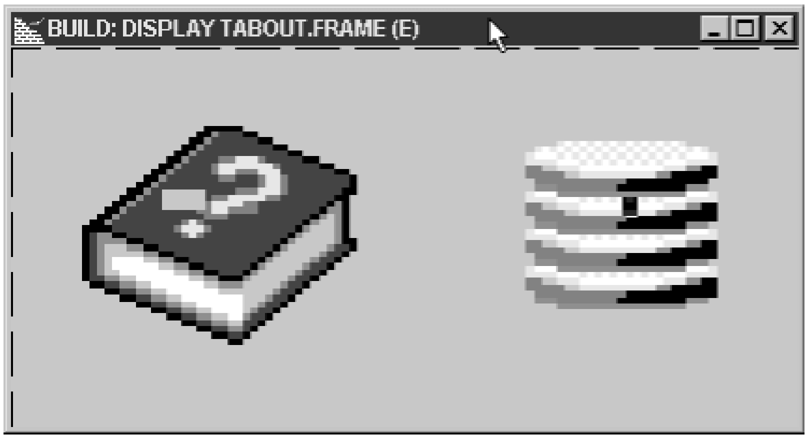


Figure 4: Two Image Objects in a FRAME

The SCL associated with the FRAME is as follows:

	`init:`

Make the images able to be tabbed to. By default, you cannot tab to graphical objects.	`call notify('image1','_set_tabbable_','on') ;`
	`call notify('image2','_set_tabbable_','on') ;`

These next four lines of code define the method overrides that will run when	`call notify('image1','_set_instance_method_',`
	`'_tab_out_',`
	`'sasuser.tabout.override.scl',`
	`'tabout') ;`

the code is linked to an object. An instance override is a mechanism that assigns to a specific object at run time the code to run when a method executes.	`call notify('image2','_set_instance_method_',`
	`'_tab_out_',`
	`'sasuser.tabout.override.scl',`
	`'tabout') ;`
	
	`call notify('image1','_set_instance_method_',`
	`'_tab_in_',`
	`'sasuser.tabout.override.scl',`
	`'tabin') ;`
	
	`call notify('image2','_set_instance_method_',`
	`'_tab_in_',`
	`'sasuser.tabout.override.scl',`
	`'tabin') ;`

Position the cursor on the first image when the screen starts. This also triggers the _TAB_IN_ override.	`call notify('image1','_cursor_');`
	
	`return`

Running the method overrides is automatic once the instance methods are defined. The above SCL assumes that the override code created above is in catalog SASUSER.TABOUT, but it can go any desired catalog and catalog entry. Change the INIT section above to point the instance methods to the correct entry.

Instance overrides occur at run time and are defined to an object in the FRAME. By contrast, class overrides are defined at the time a class is built and apply to every object belonging to that class that you place on a FRAME. An instance override executes the code found at a labeled section. That code does not need to be a method block – unless you wish to pass parameters. The _TAB_IN_ override above could have been coded as follows:

```
tabin:
   call send(_self_,'_set_border_color_','pink') ;
return ;
```

1.3.3 Initial Field Placement

To avoid confusion, place the cursor on the first field that requires some data entry.

If you are creating a data entry screen that is intended to have data entered from a form such as a questionnaire, design the screen to resemble the form so that users do not get confused when they look back and forth between the paper and the online form.

An example of poor initial field placement occurs in the Region Attributes → Set Title popup prior to Release 6.11. The cursor is not placed on the first field, which is for text entry; the cursor defaults to the OK push button. Most of the time, the text entry field is the reason for accessing this window. No description is highlighted to show where the cursor is placed.

1.3.4 Insert Mode

Under program control it is sometimes possible to switch insert mode on or off. I prefer to turn insert mode off when a screen starts, but there is no obvious way to do this in SAS if the user has pressed INS earlier in a SAS session. On some platforms, the insert key can be turned off easily by issuing the following command in the INIT section of the SCL:

```
call execcmd('winsert off') ;
```

Refer to the SAS host companion for your platform for more information.

Use WINSERT OFF or equivalent functions if it exists on the platform you are developing on. It seems pointless to allow typists and developers to access a screen, with the cursor positioned ready to start typing, and be inhibited immediately or a few characters into the text. Note that in Release 6.12 FRAME does not provide the ability to determine whether insert mode is one or off (ditto nums lock and caps lock)

1.3.5 Hidden, Grayed, and Swapped Out Fields

FRAME allows you to control the fields that are available to users. You can mask fields from users until the fields are needed. Likewise, you can gray fields so that they can be seen but can't be tabbed to or selected.

Don't leave fields on screen and selectable if they are not relevant to the current status of the screen. I prefer graying to hiding because it doesn't leave the screen with chunks missing.

You can swap regions in and out. Swapping is similar to the concept of virtual memory. With virtual memory, when an application needs more memory, it moves something that is not being used out of the way. In a FRAME entry's SCL, you can swap out an unused object and reuse the space with another object.

Swapping has some distinct advantages over hiding or graying:

- You are not using screen real estate for a widget that has no meaning in some contexts.
- You can swap a non-graphical region out and place another non-graphical region in its place.
- You do not have to overlay regions to achieve the same result as you had to in earlier releases. I have one application that displays either a listbox or a graph, depending on user selection. The two regions physically overlap, and the development environment looks messy and is hard to work with. Swapping allows one object to exist on screen and the other to be swapped in as needed.

If you swap regions that are character based rather than graphical, you must instantiate at least one region at run time. You cannot create both regions at development time because they will use the same space, and you cannot overlap text regions. Use the _SWAP_IN_ and _SWAP_OUT_ methods to swap a widget between the physical and virtual frames.

In the following example, a FRAME contains a text entry (which is a non-graphical widget) and a push button (which is also a non-graphical widget). We want to display the push button and, if it is clicked, replace it with the text entry. We cannot create them in the build environment because text regions cannot overlap.

To accomplish the display and swapping of text entries, this example creates one of the objects at build time and the other at run time. The run-time code is quite complex because it explicitly tells SAS/AF the position, size, name, and other attributes of the object.

First, create a FRAME with a push button. Name the push button object CLICKME and assign to it some text.

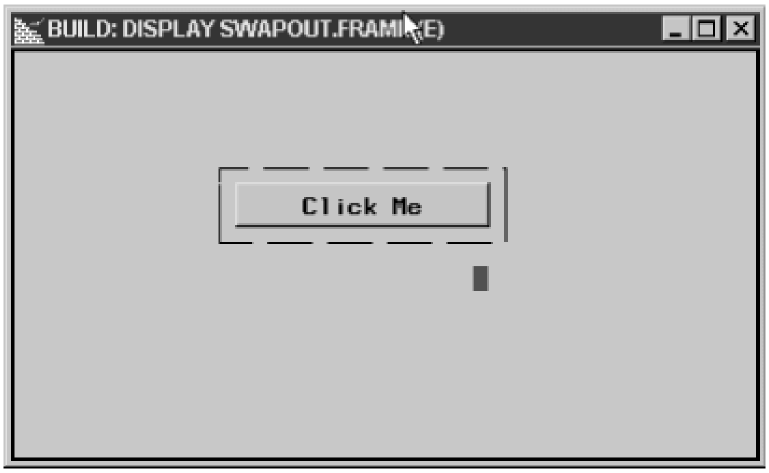


Figure 5: Push Button on a FRAME

Add the following SCL. In this example there are no overrides needed, as all the code will be in the FRAME SCL entry.

Text field to store user input in text entry.	`length mytext $ 30 ;`
	`init:`
The REG list will hold the description of where the text entry will be positioned.	`reg  = makelist();`
It draws its position from the initial position of the push button.	1 `call notify('clickme','_get_region_',reg,'c');`

ATTR is the object list that will contain the data about the text entry object. Here the list that is needed to position it on screen is defined.	`attr = makelist();`
	`rc   = setniteml(attr, reg, '_region_');`
	2 `rc   = setnitemc(attr, 'tbox', 'name');`
	
	`return ;`

When the FRAME finishes, you need to explicitly remove the text entry object (if it exists) and the lists associated with it.	`term:`
	` if textbox then call send(textbox,'_term_') ;`
	` rc = dellist(reg) ;`
	` rc = dellist(attr);`
	`return ;`

If the button is clicked then swap it out, check if the text entry exists (create it is it doesn't),and display the text.	`clickme:`
	` call notify('clickme','_swap_out_') ;`
	` if textbox le 0 then`
	❸ `    textbox = instance(`
	`            loadclass('sashelp.fsp.efield.class'),`
	`            attr);`
	` call send(textbox,'_swap_in_') ;`
	`return ;`

If the text entry labeled section is driven (i.e., the widget text is modified), then display what the user entered, swap the text entry out, and redisplay the push button.	`tbox:`
	` call send(textbox,'_get_text_',mytext) ;`
	` put mytext= ;`
	` call send(textbox,'_swap_out_') ;`
	` call notify('clickme','_swap_in_') ;`
	`return ;`

❶ The _GET_REGION_ method is extremely useful anytime you want to replace an object with another one or modify the size of a field. It returns the region co-ordinates in the list passed as the first parameter after the method name. An interesting practical example I have of this is where I have several thumbnail-size graphics on screen. When the user selects one, it explodes to full screen. I load the thumbnail region co-ordinates to a list first and use them to restore the thumbnail to the original size.

❷ The text entry is given a name (i.e.,TBOX) when it is created simply so that a labeled section can exist in the FRAME SCL code. You do not have to give a name to an object created at run time. The INSTANCE function will cause a unique object identifier to be generated, which is how FRAME uniquely identifies all objects.

❸ The text entry cannot be created by the INSTANCE function in the INIT section. It is not possible to define a new text-based region, which will occupy space used by an existing text-based region, until the existing region is swapped out. Because it is not desirable to swap the push button out in the INIT section, the creation of the text entry is left until the push button has explicitly been swapped out.

1.3.6 Placement of Action Widgets on the Screen

Some functions, for instance EXIT (or END or OK), CANCEL, and HELP, appear repeatedly in applications. In mainstream Windows applications, the only standard for placement of these functions is in pull-down menus. EXIT and CANCEL usually are found under the far left menu, which is often a FILE menu; HELP is usually the far right menu. Some products also place these functions on screen as push buttons, but position, text, and order tend to be inconsistent. In FRAME system windows, you will find these functions at the bottom or the right of the window.

For Windows users, much of the screen design and layout standard has been taken from Microsoft applications and texts. Microsoft does not have a fixed standard for many screen attributes; rather, they impart guidelines. For example, action push buttons may be at the bottom or at the right of the screen. It seems to be fairly well accepted that action buttons should appear at the bottom of the screen. Don't try to define a completely different layout–follow acceptable guidelines as defined by other applications.

SAS Institute uses push buttons for END, CANCEL, and other functions in their development software, but our end-users don't necessarily use SAS, and we don't have to inflict buttons on them if they are used to pull-down menus in other applications. However, I believe that push buttons are one the best mechanisms for triggering action commands. They are intuitive and easy to set up.

If you use widgets to identify actions such as END, be consistent from screen to screen. Don't swap the position of the END and CANCEL buttons in different screens.

The developers of SAS/AF have extended the idea of a push button to an object called the command push button. This object looks a bit different from the usual push button, but behaves in the same way from the end user's perspective. The COMMAND PUSH BUTTON class is useful because it is already populated with commands, and you can select the appropriate command during development.

You can make use of methods defined in the development environment to assist with placement of standard buttons. The following is an adaptation of code in the SAS Institute FRAME documentation (*SAS/AF Software: FRAME Class Dictionary, Version 6, First Edition, pg. 11–12 of FRAME Class chapter*). This code automatically places $\boxed{\text{END}}$ and $\boxed{\text{HELP}}$ buttons on every screen whenever you enter a FRAME entry in build mode. It removes some of the tedium of placing the buttons on every screen. It also adds the commands for the buttons, so the entire creation of the objects is done automatically. The code forces the names of the buttons to be the same in every entry. Buttons will not be placed on the screen if any other widget is already in the FRAME entry. You also have the option not to have the $\boxed{\text{END}}$ and $\boxed{\text{HELP}}$ buttons.

Subclassing is a way of customizing a SAS/AF class that is supplied by SAS Institute. The customization could be very simple, like setting a certain color for the background, or very complex. SAS/EIS objects are complex examples of subclassed FRAME objects.

You can subclass any of the base FRAME classes, including the FRAME class itself. By subclassing the FRAME class, you can modify the development environment, as well as adding functionality to the run-time environment. Note that widgets, as well as FRAMEs, have the ability to customize their BUILD-time characteristics.

To use this code you first need to subclass the FRAME class. Follow these steps. Open a catalog in BUILD mode before you begin.

1. Issue the command EDIT.NEWFRAME.CLASS.
2. Make the parent class SASHELP.FSP.FRAME.
3. Make the description My FRAME With END/HELP Buttons.
4. Click the METHODS attribute.
5. Locate _BPOSTINIT_ method, click it once, and enter a source entry and SCL label to execute. Here I use NEWFRAME.SCL (you need to add the LIBNAME and catalog), and the label is BPOST.

Enter the following SCL program. You can END out of the edit of NEWFRAME.CLASS and enter the name of the SCL entry that you defined to store the method, or click on the Actions → Edit Source Entry popup. This SCL program uses instance variables to define attributes for an object created in SCL using the _NEW_ method. *SAS/AF Software: FRAME Class Dictionary* contains detailed information about instance variables, as well as detailed information about the _NEW_ method.

Set the label to what you called the override in the NEWFRAME.CLASS entry.	`bpost:`
	`  method ;`

Start by calling SUPER. It is done here to ensure the FRAME setup is complete.	❶ `call super(_self_,'_bpostinit_') ;`

If any widgets already exist, don’t do anything at all here.

```
call send(_self_,’_get_widgets_’,widget_list) ;

if listlen(widget_list) gt 0 then return ;
```

Create lists to be used to check if the user wants the fields.

```
endcheck = makelist() ;
rc=insertc(endcheck,’Insert An END Button’,-1) ;
rc=insertc(endcheck,’Do Not Insert END Button’,-1);

helpcheck = makelist() ;
rc=insertc(helpcheck,’Insert A HELP Button’,-1) ;
rc=insertc(helpcheck,
           ‘Do Not Insert HELP Button’,-1) ;
```

Find the FRAME co-ordinates.

```
call send(_self_,’_WINFO_’,’STARTROW’,sr) ;
call send(_self_,’_WINFO_’,’STARTCOL’,sc) ;
call send(_self_,’_WINFO_’,’NUMROWS’,nr) ;
call send(_self_,’_WINFO_’,’NUMCOLS’,nc) ;
```

Create the new co-ordinates for the END button. Set it just left of center and 1 row above the bottom.

```
center = (nc-sc)/2 ;
lry = nr - 1 ;
lrx = ceil(centre) - 2 ;
ulx = lrx - 10 ;
uly = lry - 4  ;
```

Initialize the instance variables needed to create the object.

```
attr = makelist() ;
reg  = makelist() ;
rc = setniteml(attr,reg,’_region_’) ;
rc = setnitemc(attr,’C’,’_justify_’) ;
rc = setnitemn(reg,ulx,’ulx’) ;
rc = setnitemn(reg,uly,’uly’) ;
rc = setnitemn(reg,lrx,’lrx’) ;
rc = setnitemn(reg,lry,’lry’) ;
```

Determine if the user wants an $\boxed{\text{END}}$ button. If so, give it a name of END (i.e., this will be the object name), create it, and use an SCL identifier of BUTTEND so that CALL SEND can be used. Give it a command of END.

```
select(popmenu(endcheck)) ;
  when (1) do ;
     rc = setnitemc(attr,’END’,’name’) ;
     end = loadclass(‘sashelp.fsp.pbutton’) ;
❷    call send(end,’_new_’,buttend,attr) ;
      call send(buttend,’_set_label_’,’End’) ;
     call send(buttend,’_set_cmd_’,’end’) ;
  end ;
  otherwise ;
end ;
```

Determine if the user wants a HELP button. If so, give it a name of HELP (i.e., this will be the object name), create it, and use an SCL identifier of BUTTHLP so that CALL SEND can be used. Give it a command of HELPMODE ON.

Give rid of the temporary lists.

```
  select(popmenu(helpcheck)) ;
    when (1) do ;
       rc = setnitemc(attr,'HELP','name') ;
       rc = setnitemn(reg,lrx+4,'ulx') ;
       rc = setnitemn(reg,lrx+14,'lrx') ;
       help = loadclass('sashelp.fsp.pbutton') ;
       call send(help,'_new_',butthlp,attr) ;
       call send(butthlp,'_set_label_','Help') ;
       call send(butthlp,'_set_cmd_','Helpmode on') ;
    end ;
    otherwise ;
  end ;
  rc = dellist(endcheck) ;
  rc = dellist(helpcheck) ;
  rc = dellist(attr) ;
  rc = dellist(reg)  ;

endmethod ;
```

Finally, exit back to the build environment.

❶ There are a number of methods in FRAME that are used at BUILD time. They allow us to modify the development environment. This one, _BPOSTINIT_, is the BUILD-time POSTINIT method. It runs when we open a FRAME for editing in the development environment, but after the FRAME display and widgets have been built. We could not run this code in a _BINIT_ override because the FRAME would not have completed building at that time.

❷ I use CALL SEND rather than CALL NOTIFY because this is an override and thus has no associated FRAME entry. CALL NOTIFY can only be used in SCL that runs in a FRAME. Although I created a name for each of the new objects (END and HELP), those names are for use in the application being built, not in the underlying development environment.

Create a new RESOURCE entry (named BUILD.RESOURCE) by copying from SASHELP.FSP.BUILD.RESOURCE to your development catalog (or add to an existing RESOURCE entry if you have one) and editing the RESOURCE entry. Then do the following:

1. Select Actions → Add and add your new FRAME class.
2. Click on the new class, select Actions → Set Active to make it the default FRAME class used to create new FRAME entries.
3. Exit the RESOURCE entry.

If you do not name a RESOURCE entry BUILD.RESOURCE, you need to use the RESOURCE command before editing a FRAME to specify the name of the RESOURCE entry to use.

In a RESOURCE entry, the word ACTIVE to the left of a class indicates that it is the FRAME class that every FRAME edited with this RESOURCE entry will use. Only one class will be marked active in a FRAME RESOURCE entry.

Now when you edit a FRAME entry, you will be prompted to indicate whether you want the END and HELP buttons. If yes, they will be placed for you.

As an exercise, modify the code to add a CANCEL and a PRINT button.

The ability to modify the development environment is one of the things that make SAS/AF FRAME entries so versatile. You can use FRAME entries and the object-oriented programming approach to add a lot of extra functionality so that your development environment becomes easier to manage.

1.3.7 Tool Tips

My favorite GUI feature in Windows products is the ability to display text that describes what a widget will do if you click on it. For example, in Microsoft Word, if you move the cursor onto a toolbar image and hold it there, a short piece of text appears telling you what will happen if the widget is clicked. This feature is called a tool tip.

You can create tool tips from Release 6.11 of SAS onward. Tool tips influence the design of GUI applications because you no longer have to mix icons and text. I find icons often take up too much space and are difficult to create because of this.

To add tool tips to most widgets, follow these steps:

- Select Region Attributes from the popup menu.
- Select the outline push button option at the left of the window.
- Set Outline Type to Button.
- Set button behavior to push button.
- Add tool tip text to the Description field. Start the text with \n. \n is the delimiter that SAS/AF uses to determine the start of the tool tip. The \n can be anywhere in the field; characters preceding it do not become part of the tool tip.

You cannot always create tool tips. For example, a text entry widget does not display the tool tip. On the other hand, an extended text entry will display the tool tip, but when Push button behavior is defined, you cannot do any data entry.

You can also use the _CURSOR_TRACKER_ method to create these short text descriptions that appear as you move over the object. The Release 6.12 FRAME documentation contains an example using a widget that appears and disappears as you move the pointer over the region that a tip is defined for. This example uses the description field that all objects have in the Object Attributes screen.

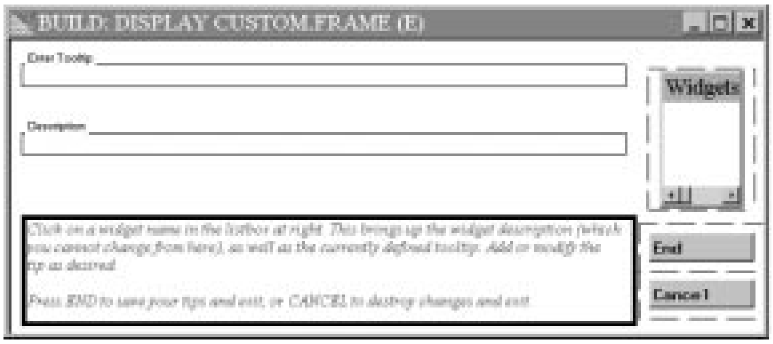
An alternative technique is to use the message area rather than a FRAME widget (similar to the SAS BUILD directory and other parts of SAS, or WordPerfect). This example uses features to add the ability to specify the tip at development time. This means that the Description field can be different than the tip. Because the message area is available, this method can use longer tips than the Description field allows. This example also demonstrates how you can override the FRAME class, override methods for that class, and create an additional attribute screen. So as well as a different way of seeing tool tips, you are also going to learn about many aspects of the FRAME environment.

The example in this section illustrates the following:

- cursor tracking, i.e., the ability to detect mouse pointer movement and take action dependent on the position of the pointer
- overriding methods in the FRAME class
- creating additional FRAME class methods
- adding a custom attribute screen in the FRAME class
- per instance methods, i.e., methods that we define to an object at run time.

Follow these steps (using an appropriate catalog; DGS.TOOLBOX is used here):

- Subclass the FRAME class. Call the new class NEWFRAME.CLASS.
- Override the _POSTINIT_ method. The override is a labeled section called POSTINIT in DGS.TOOLBOX.TOOL TIPS.SCL. Note that this section will run when the FRAME is created at run time, not development time.
- Override the _CURSOR_TRACKER_ method. The override is a labeled section called FTRACK in DGS.TOOLBOX.TOOL TIPS.SCL. This is the override for the FRAME, and we will also create a cursor tracking override for the widget class later.
- Add the new class to the relevant resource entry. Make the new class the active FRAME class by selecting the Actions → Set Active options with the class highlighted. Exit back to the BUILD directory. If the resource entry is not called BUILD.RESOURCE, issue the RESOURCE command so that FRAME entries built hereafter use the correct entry.



Click on a widget name in the listbox at right. This brings up the widget description (which you cannot change from here), as well as the currently defined tooltip. Add or modify the tip as desired.

Press END to save your tips and exit, or CANCEL to destroy changes and exit.

Figure 6: New Attribute Screen for Adding Tool Tips

- Create the custom attribute screen. Edit a FRAME entry called CUSTOM.FRAME, as shown in Figure 6.

The fields on CUSTOM.FRAME are as follows. In this table, the fields are listed in standard FRAME screen order, top to bottom, left to right.

Field name	***Description/Attributes***
TOOL TIP	● Extended text entry ● Line length: 70 ● Region title = Enter Tooltip
WIDGETS	● Listbox ● Sourced from SCL list named WIDGET ● Single-click selects an item
DESCR	● Extended text entry ● Protected ● Line length: 70 ● Region title = Description
PROMPT	● Extended text entry ● Number of rows = 6 ● Line length: 200 ● Protected ● Color: red ● Region outlines are SIMPLE with a width of 6
END	● Push button ● Command = END ● Label = End
CANCEL	● Push button ● Command = CANCEL ● Label = Cancel

CUSTOM.FRAME is a FRAME entry in its own right, but you should understand that it is intended to become part of our FRAME development environment, not part of the run-time environment that your application users will see. It will be called from the Custom Attribute screen in the FRAME development environment.

- Add the SCL for CUSTOM.FRAME. This is stored in CUSTOM.SCL.

Custom attribute screens have an optional parameter, LISTID, which contains data about the FRAME that called the custom attribute. This is needed because in the custom attribute, _FRAME_ refers to the custom FRAME, not the original caller.

```
entry optional= listid 8 _uattr_ $ class 8 ;

length text $ 8 tip $ 70 rc 3 ;
```

Get the calling FRAME entry's object ID and extract a list of widgets on the FRAME.

```
init:
  frameid = getnitemn(listid,'_frame_') ;
  widget_list = makelist() ;
  call send(frameid,'_get_widgets_',widget_list) ;
```

Create lists to store tips and widget data.

```
  tipslist = makelist() ;
  widget = makelist() ;
```

Loop through the widget list, storing the widget name and SCL ID in a list. Also, look at the widget and see if a tool tip already exists. Place the tip in the tips list if so.

```
  do i=1 to listlen(widget_list) ;
    rc = setnitemc(widget,
                   nameitem(widget_list),
                   popl(widget_list)) ;
    widget_id = nameitem(widget,i) ;
    if nameditem(widget_id,'TOOLTIP') then
       rc = setnitemc(tipslist,
            getnitemc(widget_id,'TOOLTIP'),widget_id);
    else rc = setnitemc(tipslist,' ',widget_id) ;
  end ;


return ;
```

This is the code that is executed when a widget is selected from the listbox.

```
widgets:
```

This finds which widget was selected, and return if it is a deselection.

```
        text) ;
  if issel eq 0 then return ;
```

Get the widget description.

```
  call notify('descr','_set_text_',
             getnitemc(nameitem(widget,rn),'DESC'));
```

Place the cursor on the tool tip entry.

```
  call notify('tooltip','_cursor_') ;
```

Get the widget ID and load the current tool tip into the tool tip field.

```
  widget_id = nameitem(widget,rn) ;
  call notify('tooltip','_set_text_',
              getnitemc(tipslist,widget_id)) ;
return ;
term:
```

The TERM section causes an exit if the user enters a CANCEL.

```
  if _status_ = 'C' then return ;
```

If a normal END, it moves any non-blank tool tips from the TIPSLIST list into the widget that the tip belongs to. If the TIPSLIST entry is blank and a tip exists in the widget, the widget tip is deleted. Finally, TERM removes the lists used in this SCL.

```
  do i=1 to listlen(tipslist) ;
     widget_id = nameitem(tipslist,i) ;
     if getitemc(tipslist,i) ne ' ' then do ;
        tip = getitemc(tipslist,i) ;
        rc = setnitemc(widget_id,tip,'TOOLTIP') ;
     end ;
     else
        if nameditem(widget_id,'TOOLTIP') then
           rc = delnitem(widget_id,'TOOLTIP') ;
  end ;

  rc = dellist(tipslist) ;
  rc = dellist(widget) ;
  rc = dellist(widget_list) ;
return ;
```

The tool tip section executes when a tip is altered. It moves the tip from the Input field to the tips list.

```
tooltip:
  call notify('tooltip','_get_text_',tip) ;
  rc = setnitemc(tipslist,tip,widget_id) ;
return ;
```

Note that the TOOLTIP entry field appears on the FRAME entry before the list of widgets. If it didn't, a user could edit a tip, click on the widget list, and the widget listbox processing would remove the tip, as the TOOLTIP labeled section of code would find the tip for the newly selected widget, not the one just edited. This is due to the left to right, top to bottom processing order. You need to be constantly aware that field positioning and order of processing are related.

Now add the custom attribute to NEWFRAME.CLASS. To do this, edit the class, select the Add Custom Attributes option, and enter CUSTOM.FRAME as the custom attributes. Use the Display From Custom Attribute Button option.

At this stage you have completed the steps needed to integrate the custom tips entry screen into the development environment. To test that it works, edit a new FRAME entry called TEST.FRAME. Add some widgets, fill them as desired, and select the General Attributes popup option. Then select the Custom Attributes option. You should see the new screen, complete with the widget list. Click on a widget to see its description. The cursor will be placed in the TOOLTIP field, where you can add a tip for this widget. When you have added all the tips (as many as you want — you do not need to add tips for all fields), click on END to save the tips.

Now we create the method overrides needed to support the tips in the ***run-time*** environment. This is what will happen here:

1. Code a _POSTINIT_ method to perform initialization of the tips.
2. An override to the _CURSOR_TRACKER method is required for the FRAME, plus an override to the same method for widgets. The widget override will display the tool tip, the FRAME override will remove it.

Code the methods in TOOL TIPS.SCL, which follows. The sections in this code are as follows:

- tracker overrides the widget _CURSOR_TRACKER_ method as a per-instance method
- ftrack overrides the FRAME _CURSOR_TRACKER_ method as a per-instance method
- postinit overrides the FRAME _POSTINIT_ method.

The _POSTINIT_ method override is used to run code automatically each time the FRAME is executed. That code runs after the FRAME has been created (which is when _POSTINIT_ always automatically runs) and is used to define the _CURSOR_TRACKER_ overrides to all the widgets that have a tool tip defined, as well as defining the FRAME _CURSOR_TRACKER_ method. The code is as follows:

This is the override for the FRAME _POSTINIT_.	`_self_ = _self_ ; _frame_ = _frame_ ;`
	
	`postinit:`
	`method ;`

Call the SUPER method first.	`  call super(_self_,’_postinit_’) ;`

Get a list of widgets in the FRAME.	`  widget_list = makelist() ;`
	`  call send(_self_,’_get_widgets_’,widget_list) ;`

Initialize TRACKON to zero. It gets set to one, which is used as a flag to switch on FRAME level cursor tracking, if a widget is found in the FRAME with a tool tip.	`  trackon = 0 ;`

Loop through the list of widgets, switch on cursor tracking for any that have an item named TOOL TIP in the list, and set the flag to switch on FRAME cursor tracking if needed.	`  do i=1 to listlen(widget_list) ;`
	`    widget_id = popl(widget_list) ;`
	`    if nameditem(widget_id,’TOOLTIP’) then do ;`
	`       call send(widget_id,’_set_instance_method_’,`
	`                 ‘_cursor_tracker_’,`
	`                 ‘dgs.toolbox.tooltips.scl’,`
	`                 ‘tracker’) ;`
	`       call send(widget_id,’_cursor_tracking_on_’) ;`
	`       trackon = 1 ;`
	`    end ;`
	`  end ;`

Finished with the widget list now, so delete it.	`  rc = dellist(widget_list) ;`

Switch on FRAME cursor tracking if required.	`  if trackon eq 1 then`
	`      call send(_frame_,’_cursor_tracking_on_’) ;`
	`endmethod ;`

Widget cursor tracking. Just display the tool tip in the message area.	`tracker:`
	`method x y 8 ;`
	`  call send(_frame_,’_set_msg_’,`
	`           getnitemc(_self_,’TOOLTIP’)) ;`
	`endmethod ;`

FRAME cursor tracking. Blank out the message area.	`ftrack:`
	`method x y 8 ;`
	`  call send(_frame_,’_set_msg_’,’ ‘) ;`
	`endmethod ;`

Having completed the above steps, you should now have a functional system for writing tips to the message area. Here is how to check that it all works.

Create a new FRAME entry using the resource entry that contains NEWFRAME.CLASS as the active FRAME class. Add some widgets to the new entry and fill them as desired. Use the General Attributes option to access the custom tips screen and add some tips. Now add the following SCL:

```
init:
return ;
```

Now compile and testaf your entry. You should see your tips appearing in the message area as you move the cursor over the widgets in TESTAF.

For instance (oops, using that term in a book related to OOP seems a bit punnish), create the screen in Figure 7 in the FRAME development environment. The widgets are an image and three text entries. Add a tool tip of 'This Is An IMAGE' for the image object and 'This Is A TEXT ENTRY' for one of the text entries. Leave the other tips blank.

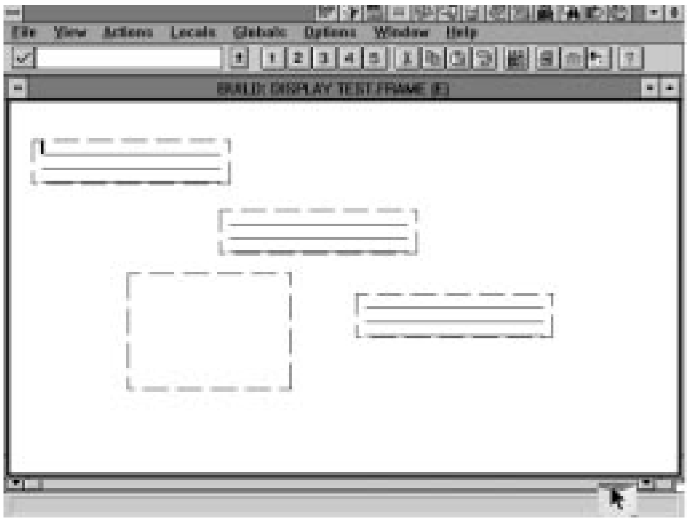


Figure 7: Sample Screen to Show Tool Tips

In run-time mode, with the cursor over the image, you see a screen like Figure 8.

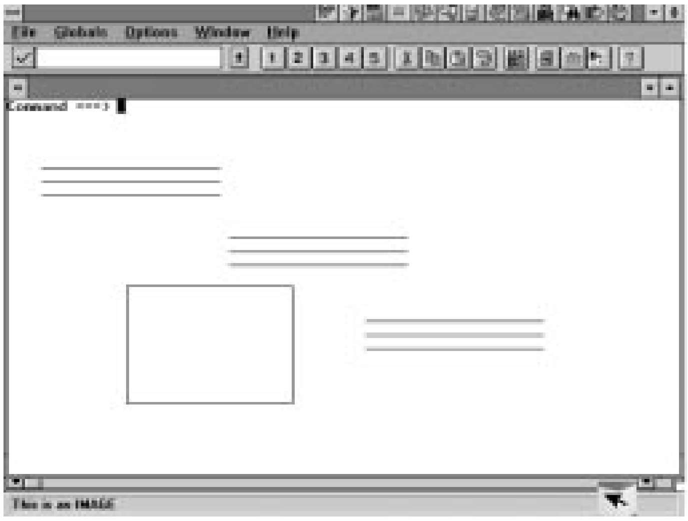


Figure 8: Sample Screen Showing Tool Tip In Message Area

Note the tip in the message area.

You could modify the code provided in *SAS/AF Software: FRAME Class Dictionary, Widget Class, pg. 73* to give yourself a system that used the custom attribute screen above to enter and store the tips and then presented them as boxes. You would need to alter the code to extract the TOOL TIP list item rather than the DESCR instance variable from the object list.

1.4 Screen-Related Issues

1.4.1 The Size of Our Screens

An application doesn't need to fill a whole screen. Sometimes the application looks and feels better by utilizing a small part of the screen. In general, PC users are confident with maximizing windows and seeing multiple windows from multiple products on screen. My feeling is why prevent them seeing and using other windows if I don't need the whole screen myself. On other platforms, you need to consider whether the platform is inherently window or full-screen oriented.

A related item is the use of AFA to start up a separate AF task stream. This allows multiple AF applications to be running and allows swapping between them.

Be warned that using AFA starts a new AF task, and it is not possible for different tasks to communicate using methods. It is even possible that a FRAME entry created in a new task could have the same frame-id as one in the calling task. I suggest that if you design applications that may benefit by using separate tasks, you think carefully about whether the tasks are intended to have inter-object communication.

1.4.2 Logos and Logo Placement

Wherever possible, replace the start-up SAS Institute logo with one that is more suited to the application. I don't think it is a good start-up procedure to tell the user all about SAS when they aren't using SAS — they're using Don's Financial Application, for example.

An invocation option to define which bitmap to use for the start-up logo is available from Release 6.11 TS040 on some platforms and generally available from Release 6.12. The option is -SPLASHLOC. Here is an example:

```
–SPLASHLOC c:\mybmp\mypic.bmp
```

1.4.3 What is a Menu?

A menu lists a set of alternatives from which one can be chosen for further processing. Usually a menu is displayed as a list, with a number or letter to be entered on a command line to choose an item.

GUI design adds to the menu concept by allowing us to logically divide screens so that previous menu items are on screen. It also allows us to display the mechanisms for selection in an intuitive manner.

A screen can be modified very quickly with FRAME entries. Once code is written and working, widgets can be changed with little code change, provided the names are retained. Sometimes methods depend on widgets, but in general, you can quite quickly alter method calls.

An interesting and intuitive menu is a bitmap with hotspots. SAS Institute in Australia wrote an EIS system that had a picture of an office as its front end. The office had a door. If you clicked on the door, you exited the application. The office had a filing cabinet. If you clicked on that, you saw a submenu of folders that you could investigate. The EIS system had other hotspots that corresponded to things in the real world: a sales chart on a wall that triggered a sales EIS; an IN tray that triggered e-mail. Users who don't like this graphical approach can switch to a more traditional menu.

A good rule to follow when designing menus or selection screens is do not allow the user to have access to menu items that are meaningless in the current context of the screen. Gray these items, or remove them completely. Switch off SAS software menus that the user doesn't need to see, for example, SAS default PMENUS, by issuing OPTIONS NOAWSMENUMERGE at start-up. This assists an application user who knows nothing about SAS, as the menu items that carry out tasks in the SAS environment, rather than the developed application environment, would often do things that we don't need an application user to do.

1.4.4 Fonts

When I design software that is to run on many machines on the same platform, I use the standard fonts that were supplied with the operating system. Many products stamp a sort of watermark on their software by using unique fonts. Because of this, I can't guarantee that a particular font will be available on another machine, even with a product like SAS.

Although all of the installed SAS software is theoretically available to all users in most applications, my experience is that different users install different features, and the supplied SAS MONOSPACE FONT is not always available. This gets further complicated when an application is shipped in run-time (trace-back) mode, as fonts are often not even installed on the target machine.

A missing font causes the operating system to try to place the nearest font, or a default, in its place. A carefully designed descriptive box suddenly loses its last few characters, or a title that just fits in a region is truncated. You can use the MULTENVAPPL option to cause FRAME entries to restrict their fonts to portable fonts. Portable fonts are system fonts that are used on many systems. They include the DMS font, times, helvetica, and courier. You can select the size, style, and weight of system fonts.

SAS/GRAPH software fonts, which are the same on every system, are also enabled by MULTENVAPPL. However, no other product uses SAS/GRAPH fonts, and using them removes the consistency of operating system fonts.

In general, there are two font types: serif and sans serif. A third type of font is the unusual ones such as *Script* type fonts, which are not discussed here. In sans serif fonts, such as this one, each character has no foot.

Here, you are reading times roman, a serif font. Note that each character has a foot. This is the main visual entity that defines a serif font. Note also that this font is thicker than a sans serif font.

With serif, the base of some characters can be in the way of region outlines. On pull-down menus, using serif can get in the way of the underline under shortcut characters. Sans serif characters tend to look better on screen.

Another criterion for deciding on a font is whether characters such as 'g' descend below the line or not. I prefer fonts where characters such as 'g' descend (this is from times new roman), as opposed to non-descending. For example, using antique olive, the 'g' is non-descending. I prefer descending fonts, as they are more natural to the eye. They reflect the way that most of us learned to write, and are most heavily used in the printing industry.

You can use different fonts to present different types of information. There are a number of categories for text information that goes onscreen.

Banners are headings such as vendor or developer company names. For banners, I use graphic text and a strong, filled font such as XSWISSB. I use a restrained color because a banner is not intended to be something that constantly grabs the eye.

Information fields are items telling you what to do on a screen. For information fields I use extended text entries because they are simple to spread across multiple lines and they permit the use of operating system fonts. Under Windows and OS/2, I use italic bold fonts such as arial.

Errors are program-generated notes that the user must see. For errors presented in the status line you must use the current SAS system font, but you can use SAS DMS colors. If I create an object to display error messages, I always use extended text entries.

Input fields are entered by the user. Input fields usually use a sans serif font. I often use extended text entries for these, but FRAME has many candidates for data entry, and often the text entry will provide the necessary functionality.

1.4.5 Storing User Choices

If an application field has a number of options, you can store each user's last choice in a list in SASUSER. For example, usually each user will use the same printer each time they print; you can store the user's printer choice from session to session rather than make the user re-enter it.

1.5 Displaying Messages

In this context, a message is a response from the application that is displayed for the user's benefit. Usually the message conveys some information. SAS software uses Error, Warning, or Notes messages in an application, but there is nothing to stop us as SAS software users from using other titles.

Error messages should cause some action to prevent the application from continuing until the user corrects the problem. In SAS/AF, the ERRORON statement (and its equivalent method _ERRORON_) accomplish this. Warning messages suggest to the user that, although the data is valid, it may not be useful. For example, a report covering a period that is very old. Notes are purely informational, for example, 'Data Has Been Saved.'

This section addresses error messages, but many of the principles are applicable to all message types.

The user has a right to expect meaningful and concise error messages. All fields that can be in error should be able to display an error message. All error messages should be displayed in the same place. Error messages should follow the same format.

From Release 6.11 on, the _MSG_ area is available on some platforms regardless of whether a command line exists. The _MSG_ area is the most sensible standard position for display of single error messages. With platforms or applications that do not have a message line, you can create a message system or use the screen name area to display messages.

1.5.1 Designing Systems to Avoid Error Conditions

It is frequently possible to prevent error situations by forcing the application to take advantage of existing data. For example, a user may need to enter a project identification number. Rather than allowing users to enter an incorrect number, give them a list of valid projects from which to choose.

Sometimes this can cause more problems than it fixes. Suppose you have a very large number of projects that all start with a similar structure, for example, W02045, a letter followed by a sequential number. A listbox approach requires that all the listbox data be stored in the listbox object (i.e., in the object's list). You may find memory a problem. A bigger problem is likely to be that the user finds a long list very hard to use. Extended tables or other table objects have the same issues in terms of amount of data to scroll through, but using these objects will usually alleviate the memory problem because not all data is stored in memory at all times.

There are alternatives to listbox-based approaches, for example, remembering the last value entered by the user and re-entering it. This approach requires that a dataset or a list in a private library (usually SASUSER) has data stored for each field and that the data is reloaded the next time the screen is entered.

A way to use listboxes intelligently is to code the _FEEDBACK_ method of the extended text entry. You can allow a user to enter text and simultaneously have code update a listbox to show only data that matches the text already entered by the user. It is clear when an error occurs (there is no matching data) because the listbox will be empty. An example of this is presented on page 9.

Case Study

A good example of a system that could have bypassed some error conditions is a security request management system at a site where I have been working recently. All the demographic information about a user is available to the application, but the user is forced to enter a phone number. The phone number field is buried in the midst of other demographics; the application doesn't tab to the field. It is often a time consuming affair to exit this application, as it checks every field for errors when exiting, and then displays just the first error condition, which is often the phone number and it is often the only error.

The application displays a child window, which describes the error but doesn't allow it to be fixed. If you click OK, you return to the application, which takes you to the last field you were on. You have to scroll back up to the phone number field. Upon correcting the phone number, the application then goes back through the entire field error check. To cap it all off, the only exit button is at the bottom of the screen, so you need to scroll back down to it.

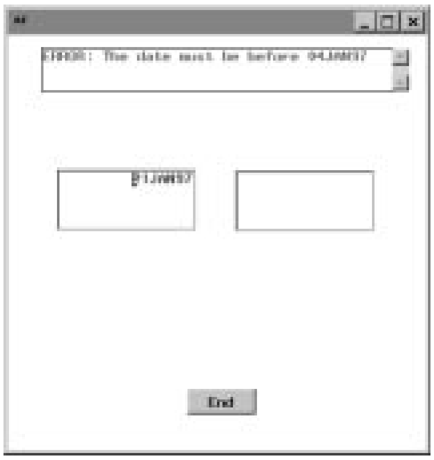
This situation would be made much simpler by loading the phone number from the known data, or at least by placing the user back in the phone number field and avoiding the child window.

1.5.2 Messages for Multiple Fields

From Release 6.11 on, you can create intuitive means of displaying messages. Figure 9 demonstrates these facilities.

- Switch ERRORON for each field in error.
- Enable cursor tracking.

- When the _CURSOR_TRACKER_ method detects the mouse moving over a widget that is in error, display a previously hidden or swapped out widget containing a context sensitive message. Hide, or swap out, the message widget when the user moves the cursor off the widget in error. When another widget in error is moved onto, change the message displayed in the widget.


ERROR: The date must be before 04JAN97

01JAN97

Figure 9: Error Message Display

This is quite a stunning way to present errors. You can present messages for every field in error simply by having the user move the cursor across or onto it.

I have implemented this under Release 6.12. The object pops up as above when a field is in error. Figure 9 shows what the screen will look like when the error message displays pops up. When the cursor moves off the field in error, the explanation box disappears again. The code for this is presented in Chapter 2.

1.5.3 Error Message Layout

Start each message with the word ERROR, WARNING, or NOTE. Make sure the user understands what the message means and what needs to be done.

Ensure that the display of messages is in sync with the placement of the corresponding fields on screen. Also, ensure that the error messages match up with their corresponding fields.

There is also an art to determining what information to place in error messages. My pet hate is software that dumps unintelligible messages, but as we all know, sometimes it is unavoidable. If you can trap an error condition and print a meaningful message, you should do so rather than leaving the system to dump its own internal messages.

Keep messages as short and succinct as you can. Consider the experience level and needs of your user base.

1.5.4 Unmodified Fields

For unmodified fields, you can supply a generic message issued in MAIN that the field should be filled in. You may find it easiest to issue such messages during the TERM processing. At that stage you can check for any required fields not being complete. If a required field is found empty, set _STATUS_ to 'R' and issue a RETURN from the TERM section, or set the REQUIRED attribute to ensure that information is entered in the field.

Using the REQUIRED attribute standardizes the message that the users sees to use that supplied by SAS Institute. You will need to supply your own message (using _MSG_ or the _SET_MSG_ method) if checking for required fields in TERM.

My preference is not to use the REQUIRED attribute because I usually want to issue a more context-specific message than the SCL default.

1.5.5 Difficulties with Error Processing and Screen Design

There is no hard and fast rule for how we cope with error processing that causes problems in the design of the screen. The literature and courses on screen design that I have attended simply don't address these issues.

It can sometimes be difficult to co-ordinate error messages. If a screen has a number of fields and the user places the cursor on the fifth field and enters an incorrect value, do you display messages for fields one to four that need to be populated, or do you display a message for the error in field five? How do you process errors for those first four fields, given that the labeled section will not run?

This is a design issue. The error for the fifth field is easy to display, because its labeled section will run. If you wish to display a message for the other sections, you could place code in MAIN and use CONTROL ALWAYS to trigger MAIN if no fields are entered.

The code that you need in MAIN is as follows:

```
if not modified('obj1') then link obj1 ;
if not modified('obj2') then link obj2 ;
```

In MAIN, determine if the user has modified the fields that you desire to issue messages for. If the fields are not modified, the labeled section will not have run, so you explicitly force it to run.

Suppose a user fills out the third and fifth field in error. You switch ERRORON for each field and display a message for the third field. The user corrects the fifth field but doesn't correct the third field. Because the label for the third field isn't carried out by default, no message is displayed even though the third field is still in error.

Again the answer is to make use of MAIN. However, because you don't want to run the labeled sections for sections that are not modified, use the ERROR function instead of the MODIFIED function in MAIN.

```
if error(objname) then link <label> ;
```

Use of ERROR can create another difficulty. If a section did execute this time through the SCL and switched errors on for an object, MAIN would cause the labeled section to run again. You can resolve this difficulty by bypassing the labeled section completely and always linking from MAIN. CONTROL ERROR does not help here because it refers only to the running of MAIN, not to object labeled sections. Using MAIN in this manner makes it difficult to bypass sections that should not execute, such as hidden fields.

If multiple sections find fields in error, by default the last one is the one that _MSG_ gets written for. Because the cursor will be placed on the first field for which ERRORON was switched on, the cursor positioning may be out of sync with the message. The message that displays for the first field physically in error on the screen may also be out of sync.

Change the logic flow to check whether _MSG_ contains a value and not to assign a value if it does. If you don't do this, the user sees error messages appearing in an illogical order.

For example:

```
label1:
  if <error condition> then do ;
    _msg_ = '....' ;
    erroron label1 ;
    return ;
  end ;
 ... non error processing ...
return ;

label2:
  if <error condition> then do ;
    if _msg_ eq ' ' then
       _msg_ = '....' ;
    erroron label2 ;
    return ;
 end ;
 .. non error processing ...
return ;
```

1.5.6 Informing Users about Long-Running Tasks

Some tasks take a long time to run. It is difficult for a user to know what is going on unless you provide some sort of message system to keep them informed. With SCL code that runs for a long time without user intervention, you can display a graphics text object that updates with a percentage of completion. Alternatively, the *FRAME Class documentation, pg. 35–36 of SAS/AF Software: FRAME Class Dictionary, Version 6, First Edition* gives an example of changing the cursor shape during a long-running SCL task. You can use either of these methods with submitted code if you are willing to split the SUBMIT block into several smaller blocks. Then you can update your graphic text or cursor shape in between the submit blocks.

1.6 The HELPMODE Command

1.6.1 What is HELPMODE?

HELPMODE places every field in a pseudo unprotected mode so that you can click on fields and display the object help entry for the field. By "pseudo unprotected," I mean that a field can be clicked on, but it cannot trigger its SCL section nor can data be entered into it. This mode exists purely to obtain help on the field.

When HELPMODE is switched on, the cursor changes to a help cursor shaped like a question mark. This cursor shape displays only until you select a widget. You must switch HELPMODE on for each widget that you require help on.

These overrides to FRAME processing are in existence only while HELPMODE is on. It is not possible for users to compromise the integrity of applications because a previously protected or grayed field cannot be selected except to access object help.

1.6.2 Starting HELPMODE

You use HELPMODE by issuing a HELPMODE ON command from a widget (e.g., a push button) that executes the command. You can also use a toolbar icon. You can assign the HELPMODE ON command to a function key or a pmenu or a popup. You can have users enter HELPMODE ON themselves at a command line, although I think the toolbar or widget approach is more intuitive.

1.6.3 What If No Help is Available?

When HELPMODE is used on a widget that has no object help, an error is returned by SAS/AF. This violates one of my basic development principles: don't mention the word error to the user if they haven't done anything wrong. In situations where it is impractical to have any help available, avoid the error message by overriding the _HELP_ method. This is the method that causes the object help to be displayed.

The code in the FRAME entry SCL consists of a temporary override to the FRAME entry's _POSTINIT_ method. _POSTINIT_ is an internal method that runs after the code in your INIT section, but before the FRAME actually displays. This is needed because I want to get a list of all widgets and, at the time

POSTINIT runs, the display is created and all widgets have been created. To override _POSTINIT_, enter the following statement in your INIT section. This is an example of a per instance method.

```
call notify('.','_set_instance_method_','_postinit_',
             'sasuser.book24.methods.scl','postinit') ;
```

Note that if you have a FRAME subclass in use, the above is not needed in the FRAME SCL because the subclass could specify the _POSTINIT_ override.

In the above example, the override code is in SASUSER.BOOK24.METHODS.SCL. Change the SASUSER.BOOK24 to the catalog you use to store your method overrides.

The actual override code in SASUSER.BOOK24.METHODS.SCL gets a list of all the widgets in the FRAME entry and checks each for the presence of the Object Help attribute. If the attribute does not exist, a per instance override is created for the widget's _HELP_ method to prevent the error message being issued. The override does nothing, not even a CALL SUPER as the CALL SUPER would generate the message. This is one of those occasions where the bypassing of CALL SUPER is desirable.

The code and explanation follow.

FRAME POSTINIT.	`postinit:`

Start by getting a list of all the widgets in the FRAME. Note that the _GET_WIDGETS_ method is being sent to the calling FRAME for execution. That occurs because _SELF_ is the FRAME entry object ID because this override is at the FRAME level.	`method ;`
	`widget_list = makelist() ;`
	`call send(_self_,'_get_widgets_',widget_list) ;`

Loop through the widgets in the FRAME, looking for any without the HELP attribute set.	`do i=1 to listlen(widget_list) ;`

Get the current widget ID from the list.	`curr_widget = getiteml(widget_list,i) ;`
Determine if HELP is set.	`if nameditem(curr_widget,'HELP') eq 0 then`
No HELP set, so override the _HELP_ method to avoid error messages.	`call send(curr_widget,'_set_instance_method_', '_help_','sasuser.book24.methods.scl', 'help') ;`
There may be a HELP, but it could be blank. Don't continue in that case.	`else if getnitemc(curr_widget,'HELP') eq ' ' then call send(curr_widget,'_set_instance_method_', '_help_', 'sasuser.book24.methods.scl','help') ; end ;`
Always override the _HELP_ method at the FRAME level. Also, use the override if there is no HELP or a blank HELP.	`if nameditem(_self_,'HELP') eq 0 then call send(_self_,'_set_instance_method_', '_help_','sasuser.book24.methods.scl','help'); else if getnitemc(_self_,'HELP') eq _blank_ then call send(_self_,'_set_instance_method_', '_help_', 'sasuser.book24.methods.scl','help') ;`
Delete the list of widgets.	`rc = dellist(widget_list) ;`
Don't forget to SUPER the _POSTINIT_ method!	`call super(_self_,'_postinit_') ; endmethod ;`

Note that a blank help entry is possible and must be accounted for above. You could get a blank entry if you had entered an object help in the Object Help attribute field, then backspaced over it to delete.

The _HELP_ override in METHODS.SCL is as follows:

```
help:
  method ;
  endmethod ;
```

If you wanted to display a message rather than do nothing when there is no help defined, assign your message to _MSG_ using the _SET_MSG_ method.

```
help:
  method ;
    call send(_frame_,'_set_msg_,
             'No HELP Available For This Object') ;
 endmethod ;
```

You need to be wary with per-instance methods, as you risk losing an existing override. You cannot tell if the method is already overridden because the class editor is where you see that information.

1.7 Summary

What I have tried to do here is provide a few ideas about how to approach the whole GUI development paradigm from a screen design perspective. The concept of GUI is quite new to many SAS programmers, and we need to modify our thinking from just resolving the issues that the application addresses to considering how to let the user interact with that application.

If you want to see hundreds of different ideas on GUI screen design, check out some World Wide Web home pages on the Internet. You will rapidly come to appreciate consistency in other applications. WWW is like anarchy at the best of times, and the home pages show many forms of human interaction with applications.

There are two excellent texts that I recommend if you wish to explore some of the subtleties of screen design. They are quite different, one preaching the Microsoft line, the other taking a wider view of screen design. Both go into much deeper discussion of GUI screen design than it is possible to do here.

Cooper, A. (1995), *About Face: The Essentials of User Interface Design,* Foster City, CA: IDG Books Worldwide Inc.

Microsoft Corporation (1994), *The Windows Interface — An Application Design Guide,* Redmond, WA: Microsoft Press.