Introduction to Pull-Down Menus
Roger Staum, SAS Institute Inc., New York, NY
Jeremy Staum, Paul Waldron Consulting Inc., West Nyack, NY

ABSTRACT
Pull-down menus, or pmenus as they are known in the SAS System, are a key component in creating modern, interactive applications. This tutorial will help you become a more effective user or developer by explaining:

- turn pmenus on and off
- pmenu components
- binding
- how to create custom pmenus
- how to assign pmenus to windows
- pseudocommands

INTRODUCTION
Methods of issuing commands
In the SAS Display Manager System, there are three ways to issue commands:

- command prompt
- pmenu
- function key

Function keys are always available during a display manager session. Pmenus, however, are an alternative to the command prompt, allowing you to issue commands without typing. Popularized by PC software, pmenus have become accepted as an integral part of a "user-friendly" computer environment.

Compared to the two other methods, pmenus:

- require less memorization.
- reduce the number of errors you are likely to make.
- facilitate your interaction with the software, especially if you have a pointing device such as a mouse.

Types of pmenu
There are two types of pmenu. Default pmenus are incorporated in SAS system software. Developers and users build custom pmenus using PROC PMENU.

Storing pmenus
Compiled pmenus are stored in catalog entries of type PMENU. The default pmenus reside in the SASHELP library, in catalogs appropriate to the window and product with which they are associated (CORE, FSP, GRAPH, etc.). You may store custom pmenus in any library and catalog to which you have write access.

TURNING PMENUS ON AND OFF
There are two commands which control whether pmenus are available. The global command, affecting all windows, is:

PMENU <ON|OFF>

The local command, affecting the active window only, is:

COMMAND <ON|OFF>

If you anticipate switching often between pmenus and the command prompt, assign these commands to function keys. Note that pmenu status (ON|OFF) is one of the window attributes stored by the WSAVE command.

PMENU COMPONENTS
Pmenus are built from several components. We will use the familiar PROGRAM EDITOR pmenu to show examples of these components.

Action bar
The action bar is the top level of a pmenu, and consists of a list of items displayed on the command line. In well-designed action bars, these items are typically command categories. For example:

<table>
<thead>
<tr>
<th>File</th>
<th>Edit</th>
<th>View</th>
<th>Locals</th>
<th>Globals</th>
<th>Help</th>
</tr>
</thead>
</table>

Items
An item is an element of an action bar or menu list which, when selected, invokes one of the following:

- menu
- dialog box
- command

For example, File is the first item on most default pmenu action bars.

Menu
A menu is a list of items which can be invoked from an action bar or another menu. Menus usually pull down from action bars if there is enough room in the Application Workspace, but pull up otherwise. Menus appear next to parent menus.

Menu items indicate, with a special symbol following the text, what type of component will be invoked if the item is selected:

<table>
<thead>
<tr>
<th>Next component Invoked</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Another menu</td>
<td></td>
</tr>
<tr>
<td>Dialog box</td>
<td></td>
</tr>
<tr>
<td>Command</td>
<td>[none]</td>
</tr>
</tbody>
</table>

For example:

<table>
<thead>
<tr>
<th>File</th>
<th>Edit</th>
<th>View</th>
<th>Locals</th>
<th>Globals</th>
<th>Help</th>
</tr>
</thead>
</table>

Dialog box
A dialog box, which may be invoked from either an action bar or menu, is a pop-up window which prompts the user to provide additional information needed to construct a command. Dialog boxes may contain any of the following:

- Text
- Input field
- Checkbox (a binary-state object)
- Radiobox (a multiple-state object consisting of mutually exclusive radio buttons)

OK and Cancel buttons automatically appear at the bottom of dialog boxes.
The selected dialog box contains text, one input field, and three checkboxes:

- **Read object**
- **Enter object name to copy:**

Select options:
- **NOTABS**
- **NOATTR**
- **REPLACE**

**OK**

**Cancel**

### BINDING

**Binding** refers to the process by which components and information necessary to software execution are brought together. For custom menus, binding may occur either at compile time or run time.

**Pmenu compile time** refers to the interval when PROC PMENU code is submitted and compiled, creating an executable PMENU catalog entry. Typically, pmenu structure and command associations are bound at compile time.

**Pmenu run time** refers to the interval during which a user is working with a SAS application and selects a pmenu item. Some aspects of pmenu behavior can be bound at run time, and thus designed to depend on data values, user preferences, etc.

As we discuss various techniques, we will point out when they are bound. Although run-time binding adds flexibility and power to applications, you should not disregard the simplicity and predictability which come from compile-time binding.

### CREATING CUSTOM PMENUS

**Motivation**

- **to satisfy user preference**
  - The default menus are not universally appealing. The words LOCALS and GLOBALS, for example, baffle some people.
- **to conceal native commands**
  - You may want to conceal certain commands from your users.
  - For example, you might want to discourage access to display manager primary windows from within a SAS/AFS application.
- **to add functionality**
  - Your application may require special functionality, which you would like users to access via pmenus.

**Using PROC PMENU**

PROC PMENU is a base product procedure. Like several other procedures, PROC PMENU supports RUN groups. You submit a RUN statement to create each pmenu, and a QUIT statement to terminate execution.

Custom menus are often revised after they have been used for a while, so remember to save your PROC PMENU source code. A good place to store it is in one or more SOURCE entries in the same catalog in which the PMENU entries reside.

This tutorial covers only the main syntax features of PROC PMENU, and is not intended to be comprehensive. As an example, we will build part of a pmenu modeled after the default PROGRAM EDITOR pmenu.

The PROC PMENU statement

The syntax is:

```
PROC PMENU CATALOG=<libref>catalog <DESC 'text'>;
```

The `CATALOG=` option identifies the catalog in which pmenus will be stored. The `DESC `text` option allows you to override the default description for the catalog.

The **MENU statement**

The syntax is:

```
MENU name;
```

If this is the first statement in a RUN group, `name` is the catalog entryname, and represents an action bar. Otherwise, `name` is simply the logical name of a menu, linking it to an item at a higher level. In either case, the MENU statement must be followed by ITEM statements, in the order you wish the items to be displayed.

For example, the following code defines an action bar in a pmenu to be stored as SASUSER.SUGI.PGMED.PMENU:

```
PROC PMENU CATALOG=SASUSER.SUGI;
MENU PGMED;
RUN;
```

The **ITEM statement**

The syntax is:

```
ITEM 'text'<action option><other options>;
```

This statement defines an item on either an action bar or a menu.

Text without quotes must be a valid SAS name, and is displayed to the user in upper case. Text within quotes can be any string, and is case-sensitive.

**The action option** identifies the type of object to be invoked when the item is selected:

- **MENU=name**
- **SELECTION=name**
- **DIALOG=name**
  - where `name` is the logical name of an object defined later in the RUN group. Omit action option when the item is a native command.

Some other options include:

- **GRAY** controls the pmenu at compile time, changing the appearance of the item and preventing it from being selected.
- **ID=** allows you to assign a unique id number to an item. This option allows you to bind graying at run time, which can help you minimize the number of pmenus needed by your application.

For example, the following code defines six menu items on the action bar:

```
MENU PGMED;
ITEM 'File' MENU=FILEMENU;
ITEM 'Edit' MENU=EDTMENU;
ITEM 'View' MENU=VIEWMENU;
ITEM 'Locals' MENU=LOCMENU;
ITEM 'Globals' MENU=GLBMENU;
ITEM 'Help' MENU=HELPMENU;
```

### File Edit View Locals Globals Help
By adding another menu statement with its items, we can define the contents of the first menu:

```
MENU FILEMENU;
ITEM "Open" MENU=OPENMENU;
ITEM "Save";
ITEM "Save as" MENU=SAVEMENU;
ITEM "Print" MENU=PRINTMENU;
ITEM "Exit" DIALOG=EXITBOX;
menu and dialog statements;
```

The SELECTION statement
The syntax is:
```
SELECTION name command\"command-string\";
```
A selection statement defines an alias, which we will discuss in greater detail in the section about pseudocommands. Multiple-word commands must be quoted.

For example:
```
MENU VIEWMENU;
ITEM "Colors" MENU=COLUMNMENU;
ITEM "Save attributes" SELECTION=WSAVESEL;
MENU COLUMNMENU;
item statements:
SELECTION WSAVESEL \"WSAVE\";
```

The DIALOG statement
The syntax is:
```
DIALOG name \"command-string\" <field numbers>;
```
This statement provides the logical name of a dialog box, connecting it to a higher-level item. Any TEXT, CHECKBOX, or RADIOBOX statements needed to define the dialog box must immediately follow the DIALOG statement.

Dialog boxes are most often used to allow the user to select command options at run time.

The command-string is the part of the command not under the user's control. The three types of field numbers may be intermixed with the text of the command string, and represent symbolics to be replaced by text:

<table>
<thead>
<tr>
<th>Symbolics</th>
<th>Replaced by text</th>
</tr>
</thead>
<tbody>
<tr>
<td>@1...@n</td>
<td>entered by the user in the input fields defined by the 1st to nth TEXT statements which contain such fields.</td>
</tr>
<tr>
<td>&amp;1...&amp;n</td>
<td>defined in CHECKBOX statements 1 to n, corresponding to checkboxes turned ON by the user.</td>
</tr>
<tr>
<td>%1...%n</td>
<td>defined in RBUTTON statements 1 to n, corresponding to radio buttons turned ON by the user.</td>
</tr>
</tbody>
</table>

For example:
```
DIALOG COPYBOX \"COPY\" @1 &1:
```

The entire command, consisting of command-string combined with field-numbers replaced by text, is issued when the user exits the dialog box by selecting the OK button.

The TEXT statement
The syntax is:
```
TEXT #line @col field-description <ATTR=attribute> <COLOR=color>;
```
where field-description is either:
- "text" - descriptive text
- LEN - the length of an input field, whose value will replace the corresponding @symbolic in the DIALOG statement.

For example:
```
DIALOG COPYBOX \"COPY\" @1:
TEXT #1 @20 \"Enter object name to copy\";
TEXT #4 @10 LEN=5;
```

The CHECKBOX statement
The syntax is:
```
CHECKBOX <ON> #line @col \"display text\" <COLOR=color> <SUBSTITUTE=cmd text>;
```
By default, display text appears in the window, and replaces the & symbolic in the DIALOG statement if the checkbox is ON. However, if the SUBSTITUTE option is coded, cmd text replaces the & symbolic if the checkbox is ON. This allows you to display text which is easy for the user to understand but does not correspond to the text used in command syntax.

The initial value of the checkbox is OFF unless the ON option is used.

For example:
```
DIALOG COPYBOX \"COPY\" @1 &1 &2 &3:
TEXT #1 @20 \"Read object\";
TEXT #3 @6 \"Enter object name to copy\";
TEXT #4 @10 LEN=16;
TEXT #5 @8 \"Select options\";
CHECKBOX #7 @10 \"NOTABS\";
CHECKBOX #7 @30 \"NOATTR\";
CHECKBOX #7 @32 \"REPLACE\";
```

The RADIOBOX statement
The syntax is:
```
RADIOBOX DEFAULT=text-number;
```
This statement must be followed immediately by RBUTTON statements. The text-number identifies which RBUTTON statement is ON by default.

The RBUTTON statement
The syntax is:
```
RBUTTON <NONE> #line @column \"display text\" <COLOR=color> <SUBSTITUTE=cmd text>;
```
By default, display text appears in the window, and replaces the % symbolic if the button is ON. However, if the SUBSTITUTE= option is coded, cmd replaces the % symbolic if the button is ON. This allows you to display text which is easy for the user to understand but does not correspond to the text used in command syntax.

NONE causes a null string, rather than text, to replace the % symbolic.

For example:

```
DIALOG NUMBOX 'COLOR NUMBERS %1';
TEXT #1 @34 'Numbers';
TEXT #3 @ 6 'Select color';
RADIOBOX DEFAULT='1';
  RBUTTON @ 6 'BLUE';
  RBUTTON @16 'RED';
  RBUTTON @25 'GREEN';
  RBUTTON @35 'CYAN';
```

### ASSIGNING CUSTOM PMENUS TO WINDOWS

Assigning custom pmenus to particular windows is not as straightforward as you might think. The available techniques depend on factors such as window type, how the window is invoked, and the number of catalogs used in the application.

#### Base product windows

For standard display manager windows such as PROGRAM EDITOR, the SAS system will not search for pmenus outside the SASHHELP library. There is thus no practical way to replace these default pmenus with custom ones.

To assign custom pmenus to the REPORT and COMPUTE windows of PROC REPORT, open the PROFILE window and specify libref, catalog, and pmenu entry names. Save the profile and point it to the PROFILE= option in subsequent invocations of PROC REPORT.

To assign custom pmenus to DATA step and macro windows, use the MENU= option on the WINDOW/ or %WINDOW statements.

#### SAS/AF windows: General attributes

There is a single, 6-character field entitled Command menu on the GATTR window of PROGRAM, MENU, HELP, and CBT entries. This allows enough room to type the entry name of a pmenu, but not its libref or catalog name. If you use this approach, therefore, the pmenu must be stored in the same catalog as the display window entry. Therefore, although this approach is straightforward, it is limited to SAS/AF entries in single-catalog applications.

Note that this technique binds the pmenu to the window at the time the application is built.

#### SAS/AF and SAS/FSP® windows: PMENU function

The best solution for PROGRAM entries in multiple-catalog applications is to use the PMENU function in a Screen Control Language (SCL) program. This function will accept a full 3-level name, allowing you to store the pmenus wherever you want:

```
r = PMENU('libref.catname.entryname');
```

There is no Command menu field on either the FSEEDIT GENERAL PARAMETERS window or on the FSVIEW PARMS window. Regardless of the number of catalogs in your application, therefore, you should use the PMENU function in the FSE/INIT block of an FSEEDIT SCL program, or in an FSVIEW formula.

Notice that this technique allows you to bind the pmenu to the window at run time.

#### Stacking the SETPMENU command

How do you assign a custom pmenu to a window which does not fall into any of these categories? For example, an FSLIST window has no GATTR window, nor can it contain an SCL program. Many windows accept a command called SETPMENU, whose syntax is:

```
SETPMENU libref.catname.entryname
```

where entryname refers to a pmenu entry. This assigns the specified pmenu to the active window.

Needless to say, a user should not have to issue such a command just to get a custom pmenu. How can a developer issue this command for the user? The strategy is to stack the SETPMENU command in the previous window, so that it waits for execution until the target window is displayed.

If the previous window contains an SCL program, the appropriate tool for stacking a command is the ExeCmd routine:

```
CALL ExeCmd ('SETPMENU ...');
```

This subroutine call should be followed by a call to the target SAS/AF, SAS/FSP, or native selection list window, for example:

```
• CALL FSLIST(...);
• CALL FSVIEW(...);  
• list = DRLIST(...);
```

If, however, you are submitting code from the previous window, stack the SETPMENU command using the DM statement. For example:

```
DM 'FSLIST libref; SETPMENU libref.catname.entryname' ;
FSLIST;
```

Note that these techniques also bind the pmenu to the window at run time.

#### PSEUDOCOMMANDS

Pseudocommands are not official SAS software terms. We use it to refer to four techniques which are ideal for pmenus and have capabilities beyond those of native commands:

- aliases
- custom commands
- CDISPLAY
- macro commands

### Aliases

An alias is simply text on an action bar or menu item which differs from the native command which will actually be issued. The SELECTION statement in PROC PMENU associates the alias with the command. For example, the following code:

```
PROC PMENU CAT='libref.catname';
MENU menucause;
  ITEM 'RUN' SELECTION=RUN;
other item statements;
SELECTION RUN 'END';
RUN;
```

associates the action bar item RUN with the command END.

An alias is helpful when a native command functions acceptably, but the command name may be not intuitive to some users. On the other hand, remember that aliases cannot be issued from the command prompt. If some of your users prefer to type commands at the command prompt, you should use custom commands.

### Custom commands

You can direct an SCL program to intercept commands before they are passed to the window's command processor. The SCL program then processes these custom commands according to your directions.
Custom commands have a number of advantages:
- They can provide functionality which native commands do not support.
- They can be issued from a PMENU, function key, or command prompt.
- It is possible to intercept, and either redefine or reject, some (but not all) native commands.12
- Since custom commands are processed by SCL, they are bound at run time.

But you need to be aware of the following:
- Custom commands require SCL, so you cannot allow the user to issue a custom command from a window which does not support SCL, such as a HELP entry display window.
- Every window which needs to accept a custom command must contain the code to process it. If you support custom commands in many entries, you could write a lot of repetitive code. To avoid this and minimize maintenance problems, place this code in SCL macros13 or method blocks14.
- Custom command processing requires you to use at least the ENTER option of the CONTROL statement. This means that the MAIN block of your SCL program will execute whenever the user presses ENTER, regardless of whether any fields were modified. You will have to adjust your SCL programming logic to account for this.

CDISPLAY
There is a command called CDISPLAY which can invoke a PROGRAM, SCL, HELP, MENU, or CBT entry from any SAS/FSP or SAS/FSP display window. This command, abbreviated as CDI, was developed for internal SAS Institute use, and is undocumented.

The syntax of this command is:

```
CDI libref:catname.entryname.entrytype
```

When the user exits the CDISPLAYed entry (or its execution stream), control returns to the window from which the CDI command was issued.

This technique has great flexibility:
- Like custom commands, CDI can provide functionality which native commands do not support, and is ideal for a PMENU or function key. CDI can be issued from a command prompt. This is not practical, however, since the user would have too much to remember and type.
- Since CDI is a native command, you need no SCL in the calling program. This makes your work easier, and makes this kind of pseudocommand available from windows which have no SCL program.
- Since you determine the name of the CDISPLAYed entry when the pmenu is created, this functionality would seem to be bound at compile time. However, the CDISPLAYed entry may itself contain SCL, and thus be able to function differently based on run-time conditions.
- For example, suppose a user selects an item entitled SESSION INFO from the pmenu on the display window of entry XHELP.
  The information to be shown depends on whether the user is a beginner or advanced. The corresponding PROC PMENU code is:

```
PROC PMENU CAT=libref.catalog;
  MENU menuitem;
    ITEM 'SESSION INFO' SELECTION=SI;
    other item statements;
    SELECTION SI 'CDI MYLIB.MYCAT.SIPROGRAM';
  RUN;
```

SI.PROGRAM then determines the user's status by table lookup, displays appropriate information, and returns control to XHELP.

Macro commands
Macros will execute and global macro variables will resolve when issued as commands.15 Therefore a pmenu item can contain macro code if the generated text is either a native command or pseudocommand. You will always want to use an alias in such items.

Ordinarily, macro variables and macros are bound at compile time, but if you single-quote them, they will be bound at run time. For example:

```
PROC PMENU statements Blending
  ITEM 'SESSION INFO' SELECTION=SI; compile time
     SELECTION SI ' escalationi';
  ITEM 'SESSION INFO' SELECTION=SI; run time
     SELECTION SI ' escalationi';
```

This can be a convenience:
- when you want to change details of a command without modifying all the PROC PMENU code which refers to that command.
- when the behavior of a command needs to be conditional on global macro variables, whose values may change during a session.

**SUMMARY**
Pmenus make issuing commands very easy for users of SAS software. This tutorial has provided you with the knowledge, at both basic and advanced levels, to use default pmenus effectively, as well as create custom pmenus which make your applications more flexible and powerful.

To contact the first author, write to:

Roger Stann
SAS Institute Inc.
820 Madison Avenue
New York, NY 10022

or:

sasres@mvs.sas.com
REFERENCES


ENDNOTES

1 Depending on your host/window manager, you may also be able to control menus globally from the Application Workspace action bar, a resource file, etc.

2 Depending on your window manager and window type, an action bar may be implemented as a popup menu or as buttons at the bottom of the window.

3 For more information, see Reference 2.

4 This feature was inadvertently omitted from the Technical Reports. It is currently documented only in a usage note.

5 This can be done using new features of the SCL WINFO function (see Reference 9).

6 This feature is documented in Reference 8.

7 See Reference 3.

8 FRAME entries have a wider field in which to enter a menu name on their GATTR windows.

9 You can find a description of SETPMENU in Reference 5. In early print runs of this manual, as well as in Reference 2, this command was incorrectly called PMENUNAME. For some reason, SETPMENU is not mentioned in Reference 4.

10 If a formula is defined, the menu assignment will be stored permanently. See Reference 10, pp. 14-15.

11 The technique is explained in Reference 6.

12 See the ALLCMDS option of the CONTROL statement in Reference 7, ch. 4 and 20.

13 SCL macros are macros which contain SCL statements. They can be placed in autocall libraries, and will execute when the SCL program is compiled. Compile time and memory usage will increase slightly, but SCL execution time will not be affected.

14 See Reference 7, chapter 16.

15 Reference 1 covers this subject very well.