Creating a Developer’s Toolkit: Facilitating the Development of SAS/AF® Applications
Dan Gronell, American Cyanamid Co.

ABSTRACT
This paper analyzes several types of SAS/AF application development tools. Its intent is to provide some insight into tool kit development and use, and to provide readers with some useful tools for their own tool kit. It is suited to developers having moderate experience with the SAS/AF product, Screen Control Language (SCL), and the SAS® software macro facility.

INTRODUCTION
In any user interface there are many tasks that are performed repeatedly. Requesting user confirmation, parsing a string, processing return codes — each of these actions is frequently required. In some applications the code that performs these tasks is simply replicated, the developer copy-and-pasting it throughout the application, each time modifying it slightly to suit the current need. However, it is far more efficient to determine which tasks are performed repeatedly and to create flexible, reusable tools to perform these tasks. This approach to application development provides several benefits:

• reduces development time
• simplifies system modification and maintenance
• helps ensure interface consistency
• reduces mundane coding and the chance for developer error

WHAT IS A TOOL?
A tool is a reusable code module that performs a common task. For SAS/AF developers, it is useful to think of tools as user-created SCL functions. This is because tools have many of the traits of standard SAS SCL functions (e.g., the OPEN function). As with standard SAS SCL functions, you invoke a tool to perform some task, and, as with SCL functions, tools frequently return a value or conditionally execute some action. Also, as with standard SCL functions, tools are generally passed 1 or more parameters which are used to tailor their execution to the current need. For example, the OPEN function is invoked to open a dataset (the task), accepting the name of a dataset to open (a parameter), and returning a numeric value which indicates the success or failure of the operation. The tools discussed here operate in a similar manner.

TYPES OF TOOLS
For SAS/AF developers, a tool generally consists of one or more of the following: a SAS/AF program entry, a SAS macro, or a SCL entry method. In this paper, tools are divided into three broad categories: Screen Tools, which are flexible SAS/AF program entries invoked through the CALL DISPLAY command; Macro Tools, which are simply macros which generate SCL code; and Utility Tools, which include both program entries which have no screen definition, and the new SAS 6.07 SCL catalog entry methods. Very simple tools of each type are analyzed here.

SCREEN TOOLS
The POPMSG screen tool
The simplest tool in any toolkit is one that displays several lines of text to the user (e.g., error messages). Since it is highly inefficient to create many program entries for this, each screen hard-coded with text appropriate to a single situation, it would be useful to have a flexible, reusable tool for this task. Note that the message line will not meet our needs here, as we want to be able to display many lines of text. Nor will the Legend window, which is intended to be called in one labelled section (generally INIT) and cleared in another (generally TERM), meet our needs. Our tool should suspend execution of the calling program, display some text to the user, and then, when the user has finished reading the text, return control to the calling program. A very simple tool which fills this function is the POPMSG tool. The code for POPMSG appears in figure 2, with an invocation example shown in figure 1. The POPMSG screen, not shown here, consists of 11 protected character fields (named msg1 to msg11) of length 70.

```sas
rc = WHERE(dsid,'quantity GT 25') ;
if rc GT 0 THEN CALL DISPLAY('popmsg','WARNING: WHERE condition was not applied successfully',
'screen PRINTER. Function return code was ', rc,
'This program will continue executing without the WHERE,
'condition being applied. Please notify support staff',
'sysmsgo1') ;
```

Figure 1 - POPMSG Invocation
The POPMSG tool displays up to nine lines of text, sizing itself to the number of lines passed. Note that the tool reads the context in which it is called and modifies its execution accordingly (e.g., setting the border color to red for all messages beginning with 'ERROR:'), thus minimizing the information we must pass it. Figure 1 shows one way POPMSG can be used. Later, when we look at macro tools which process SCL return codes, we will see a more generic solution to return code processing.

```
NAME: POPMSG
DESCR: Displays up to 9 centered messages. Window size adjusts to # of msgs passed. Window clears when user presses ENTER.
ASSUMES: error messages start w/ERROR; warnings w/WARNING:
----------------------------------------
ENTRY msg1 $10 ;
LENGTH msg1type $10 ;
INIT:
CONTROL ENTER ;
  * determine the number of lines passed *
  DO i=9 TO 2 BY -1 ;
  IF msgij = BLANK. THEN LEAVE ;
END ;
msg(i) = '-' Press ENTER to continue '-' ;
  * resize the window to our needs. *;
  CALL WIDE(2,2,6+i,WINFO('numcals')-2) ;
msg1type = UPCASE(SCAN(msg1,i));
  IF msg1type = 'ERROR:' THEN DO ;
    ALARM ;
    CALL WNAME ('SYSTEM ERROR') ;
    CALL EXECCMDI ('COLOR BORDER RED REVERSE') ;
  END ;
  ELSE IF msg1type = 'WARNING:' THEN DO ;
    CALL WNAME ('SYSTEM WARNING') ;
    CALL EXECCMDI ('COLOR BORDER YELLOW REVERSE') ;
  END ;
  ELSE CALL WNAME ('SYSTEM MESSAGE') ;
RETURN ;
MAIN:
  _STATUS_ = 'H' ;
RETURN ;
```

```
NAME: CONFIRM
DESCR: Returns a 1 if user confirms, or 0 if not confirmed
USAGE: CALL DISPLAY('canfirm',canfirmd:Delele all reparls?)
  If canfirmd THEN
  DO ........
```

Program code for the CONFIRM tool appears in figure 4, with sample usage included in the program header. A screen layout appears in figure 3. The screen variables CONFIRM and CANCEL belong to a choice group CCHOICE, with ATTR window LIST attributes of CONFIRMED and CANCELLED, respectively. Note that suitable defaults are supplied for several parameters; as with POPMSG, we want a tool that is flexible enough for many situations, yet requires a minimum of information when it is invoked. Note also that the unconditional _STATUS_ = 'R' in TERM forces the user to make an appropriate selection; only when the user makes a selection will the _STATUS_ = 'H' return control to the calling program.

```
ENTRY confirmed OPTIMAL=mgstx1-mgstd3 $48 title $32 :
INIT:
  confirmed = 0 ;
  IF mgstx2 = '' AND mgstd3 = ''
    THEN mgstd4 = 'Please confirm your request' ;
  IF title = BLANK, THEN title = "CONFIRM CHOICE" ;
  CALL WNAME (title) ;
RETURN ;
MAIN:
  _STATUS_ = 'H' ;
RETURN ;
TERM:
  ALARM ;
  _MSG_ = 'Please CONFIRM or CANCEL' ;
  _STATUS_ = 'R' ;
RETURN ;
```

Figure 2 - POPMSG

The CONFIRM screen tool

Developers must frequently request confirmation for an action (for example, when the user requests deletion of a dataset). For this task, it is useful to have a tool which displays the context to the user (e.g., "You have requested deletion of dataset XYZ. Please confirm or cancel your request") and then offers the user 2 pushbuttons, CONFIRM and CANCEL. As with POPMSG, this tool can be passed 1 or more lines of text. Unlike POPMSG, this tool also returns information to us: a numeric value of 1 (confirmation) or 0 (cancellation).
MACRO TOOLS

Macros are frequently used to reduce redundant code. A single macro can generate many lines of code, and the code that is generated can be tailored to a specific need through the passing of parameters. By creating SAS/AF tools which generate SCL code in this fashion, we can realize one of the prime benefits of tools. That is, our tools can reduce the amount of repetitive and mundane coding we must do, thus freeing us to concentrate on enhancing the interface itself.

Macro tools are easily used: you simply invoke the macro wherever you want its code imbedded. Since the code is used in many places, but is stored in one place, this approach can make system maintenance and modification easier; if you subsequently modify the code stored in your tool, these modifications are incorporated into any entry that invokes the tool. While this sounds useful, and is, the process is not quite as simple as it sounds. This is because the SAS macro facility acts differently in SCL programs than it does in DATA step code. Normally, macros resolve at run time. However, since SCL is compiled code, in a sense, macros imbedded in SCL code are resolved at compile time, not at run time. Consequently, if you change a macro, you must remember to recompile any program entry that references that macro. Only if you recompile these program entries will they reflect your modifications. The following are some rules to remember when working with macros in SCL:

- Macros invoked in an SCL program resolve when the SCL program is compiled. If you use macro variables in your SCL, your SCL program will use their values as they existed at compile time, not at run time.

- Any macro used by an SCL program must be available whenever that SCL program is compiled (i.e., the macro must already be compiled, or it must reside in an autocall library).

- If you create an SCL variable in your tool, be careful when naming that variable. If you create a variable with a name that is already used in the invoking program, and your tool alters the value of its own like-named variable, you have probably introduced a bug into your system.

- To view the code that is generated by your macro, use the options SYMBOLGEN and MPRINT. When you compile your SCL program, the resolved code will appear in the MSG window.

- While developing macros, keep them in a program entry so you can access and recompile them quickly (compiling the entry will compile any macros it contains). When your macros are stable, consider moving them into an autocall library.

In the previous section, we looked at screen tools, which were invoked through the CALL DISPLAY command. In this case, the SCL code in the tool executed outside the calling program. In other words, when program A invoked program B (our screen tool) with CALL DISPLAY, program A briefly relinquished control to program B, and statements executed in B were 'insulated' from those executing in A. Our tool communicated with the calling program only through parameters. With macro tools, however, the SCL code in the tool is actually incorporated into the invoking program. Clearly, this is what we want here: to imbed our reusable code. However, this also means that we need to use caution in defining our macros. If we create SCL variables in our macros, we must be careful: if we create a variable with a name already in use in the calling program, and we change the value of the like-named variable within our macro, we have probably introduced a bug into the calling program. To clarify this, let's look at a macro tool, SCLTITLE.

The SCLTITLE macro tool

Applications in the pharmaceutical industry generally process data for a single drug at a time. Thus, a company standard might require that the 'active' drug ID appear right-justified in the titles of all screens. To comply with such a standard, a SAS/AF developer could simplify coding through use of a macro tool such as SCLTITLE (fig. 5).

The SCLTITLE macro determines the active drug and then uses the WNAME function to left justify the title text (the settit value) and right justify the drug ID. Note that the active drug is available as a global macro variable. To collect this value at run time the SYMGET function is used. If a macro variable reference were used instead (i.e., &drug), the generated screen title would always display the value of &drug as it existed when the SCL program was compiled. Note also that the names of SCL variables used by the macro all begin with an odd prefix, "x_". As mentioned above, this is done to
avoid modifying the values of SCL variables in the invoking program (this presumes that no one else is using such a highly meaningful prefix as _x_ for their SCL variables).

%MACRO scltitle(sclttl)
%-------------------------------
NAME: scltitle
DESCR: Generate screen title with Drug ID right justified.
USAGE: %scltitle(My window title here)
ASSUMES: window width of 80 columns
Note: drug ID is placed in global macro pool when user enters application.
%-------------------------------;

..xJen_ = LENGTH("&sclttl") ;
..xJep_ = 65 - ..xJen_ ;
CALL WNAME( "&sclttl"l1 repeat('-',-'CJep_) II 'DRUG; ' II SYMGET('drug') ) ;
%MEND scltitle;

Figure 5 - SCLTITLE

The YES macro tool

It is possible to create macro tools which use no SCL variables, as illustrated by the macro YES (fig. 6). The key here is the way the macro is invoked: the macro call is imbedded in an SCL program statement, and the macro definition contains no semi-colon. This type of macro tool has limited utility, however. First, it is useful only for boolean, or 2-state conditions (i.e., yes/no, true/false, etc.). Second, because of the manner in which it is invoked, you cannot include in your tool definition a WREGION call to affect the window size and placement. Despite this, this type of invocation and usage does make for readable code, and it is certainly simple to use.

%MACRO yes(msgtext)
%-------------------------------
NAME: YES
DESCR: Collect a YES/NO response from the user.
USAGE: IF %yes(Send output to printer?) THEN DO...
INPUTN(TRANSLATE(SHOWUST( 'YES', 'NO',' &msg text'), '1', 'YES'), '2, ')
%MEND yes;

Figure 6 - YES

Macros which process return codes

Most SCL functions return special values called return codes. These return codes indicate the success or failure of an operation, and it is the programmer's job to check these codes and perform appropriate error routines when a function fails. Macros are extremely useful for this task. The macro RCWHERE (fig. 7) takes code similar to that used earlier in our POPMSG example (fig. 1) and dramatically simplifies its usage (fig. 8). RCWHERE simply warns the user when the WHERE condition has not been applied successfully. The variable _SCRID_ is a screen ID assigned in the INIT section of all program entries (another company standard), and thus is available to all tools. In companies with many large AF applications, this type of tool can be especially helpful when users call with problems, as they can tell the developer the name of the screen in which the error occurred, the generated SYSMSG(), and any other diagnostic information you include. Also, if you do not want to include these return code tools in your production version, you may find them useful as diagnostic tools during development.

%MACRO rcwhere(rc)
IF &rc GT 0 THEN CALL DISPLAY('popmsg',
'WARNING; WHERE condition was not applied successfully;
'in screen 'lI...scrid_ II 'This program will continue;
'Please notify support staff (x3917 or x4202);
'System error message (if any) follows:');
%MEND rcwhere;

Figure 7 - RCWHERE

%rcwhere(rc)

Figure 8 - RCWHERE Invocation

UTILITY TOOLS

While SCL macros allow reuse of SCL code, we have seen that they have drawbacks, in particular the need to recompile all program entries which reference a macro whenever that macro is modified. Consequently, to create tailorable, reusable blocks of SCL code, it is often better to use non-display program entries or SCL entry methods. Methods, introduced in version 6.07 for the Family 3 operating systems (MVS,CMS,VMS), are specifically designed for this purpose.

The major benefit of utility tools is that if the reusable module is modified, the calling programs do not need to be recompiled. Their major drawback is that they cannot contain any window-oriented functions (e.g., WINFO) or EXECCMDI calls, and are of limited use when reusing array processing code.
Non-Display Program entries

Non-display program entries are simply program entries which have no screen definition. When such an entry is CALL DISPLAYed, all standard labelled sections (INIT, MAIN, or TERM) are executed and then control is automatically returned to the calling program (the user does not need to END or CANCEL, since no screen is displayed). As with 'normal' program entries, parameters can be passed to the non-display program entry through the ENTRY statement.

SCL Entry Methods

SAS 6.07 for Family 3 has introduced a new SAS/AF entry type which serves as a utility tool librarian. These entries, called SCL entries, are used to store 'modular code routines'. As with non-display program entries, they have no screen definition, but are used solely to store SCL code. If SCL entries are CALL DISPLAYed, code stored in INIT, MAIN, and TERM will execute. More importantly, you can store multiple code routines in an SCL entry, defining each routine with the new METHOD and ENDMETHOD statements, and invoking them with the CALL METHOD routine. Parameters are passed to methods in a format analogous to the ENTRY statement. Figure 9 shows the MAKELGND method.

```
ARRAY ml41 $70 :
MAKELGND:
METHOD ml $70 OPTIONAL = ml2-m4 $70 title $60
tcolor brdcolor bokcolor tB
CALL SETFLD("Selection List Instructions",title) :
CALL SETFLD("green",tcolor) :
CALL SETFLD("grey",brdcolor) :
DO i = 1 to 4 :
  IF ml(i) = "" THEN CALL PUTLEGEND(i,ml(i),txtcolor,m): :
  ELSE LEAVE :
END :
* size window to # of lines passed * :
CALL WREGION(1,1,1+i,80,' ') :
CALL PUTLEGEND(title,bokcolor,brdcolor,"r") :
RETURN :
ENDMETHOD :
```

Figure 9 - MAKELGND

Painting a LEGEND window normally requires several function calls. If your company standards require you to use non-default colors or attributes, these calls can add up to a lot of typing. Using a tool such as MAKELGND, which effectively sets new default values for the PUTLEGEND and LEGEND functions, can save work, and help enforce company standards (although this version still allows the developer to override those standard values). Note that the sample usage (fig. 10) is also more readable than the standard manner of painting a legend window. As a point of interest, the developers of the SAS/ASSIST® product (which is 'just' a large SAS/AF application) used a macro tool, ALEGEND, for this purpose (if you have SAS/ASSIST software you can view this and other ASSIST macro tools by entering COPY SASHELP.ASSIST.MACROS.SOURCE on the program editor command line).

```
* generate LEGEND window using company standards * :
CALL METHOD('/lib/tools.lib.scl.makelgnd'.
'You may choose 1-10 protocols. Protocols displayed are valid for'
'the currently active drug. To select a value, place cursor on choice',
'and press ENTER. Press F3 or enter END command when finished.') :
* set winsize for the DATAWIN window * :
CALL WREGION(9,40,16,35) :
* present selection list of protocols * :
ml = DATALISTN(dsid,'protocol7 ','N',10) :
```

Figure 10 - MAKELGND Invocation

DATA-DRIVEN APPLICATIONS

Tools achieve much of their utility from the fact that they are parameter-driven. In a sense, their execution is dynamic, tailored to the current need. Data-driven applications extend this concept to your entire application. In this case, system master data, generally called metadata, is stored in one or more SAS datasets (or, with 6.07 Family 3, in SCL lists). Data-driven applications are written so that, like tools, their execution is affected by parameters (in this case, the metadata). Through metadata, your system execution is itself made dynamic. The SAS/ASSIST product is a good example of this. SAS/ASSIST is to some degree data-driven, with metadata stored in a dataset named SASUSER.SASPARMS. When you invoke SAS/ASSIST, this dataset is scanned for information saved during your previous session (last dataset referenced, color preferences, printer ID, etc.). In this manner, the application can 'remember' individual preferences between sessions, and thus tailor itself to the individual user. Metadata can also be used to create dynamic menus, to provide very granular system access control (on a subsystem or screen level), etc. Applying this data-driven concept to your application design in combination with employing a toolkit methodology can dramatically enhance the flexibility and reduce the maintenance costs of almost any large application.
CONCLUSION

By creating a toolkit you can increase the power of the already powerful SAS/AF development environment. Essentially, you are creating a library of custom SAS functions. In doing this, remember the following:

» When creating tools, be modular. Remember that tools can call other tools.

» Make your tools 'smart' by supplying intelligent defaults for your parameters (and even try to eliminate parameters by having the tool read the context in which it is called).

» Tools should be thoroughly debugged before being placed in production. Because of the nature of tools, one bug in one tool can result in this bug being proliferated throughout your applications.

CONTACTS

For further information, feel free to contact the author at:

American Cyanamid
Middletown Rd 60/202
Pearl River, NY 10965
Phone: (914)-732-3917

REFERENCES


SAS, SAS/AF, and SAS/ASSIST are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. © indicates USA registration.