The Standalone Program Grows Up: Strategies for System Design

Frank DiIorio, Trilogy Consulting, Durham, NC

INTRODUCTION

The scene is familiar. A program which used to perform one or two simple tasks needs to be changed. The time period covered by a report has changed, new data sources need to be added to the analysis dataset, and new reports are defined. The naive or harried programmer's first response is, usually, to make the program larger. After all, another DATA step or procedure usually won't affect program readability too much, and it's easier to keep everything in one place.

Revisit the program a few weeks and iterations later. Additional user requests have resulted in modifications rendering the program nearly unreadable. It has become long, is filled with obsolete report code which is commented out, and contains calculations and other data manipulations used for some reports but not others. It is, in a word, confusing.

In retrospect, the user should have considered dividing the program into multiple programs and/or program fragments. The decision to abandon the one-program format is often beneficial, usually not obvious, and the subject of this paper. It identifies situations where the transition to multiple programs is advisable, describes the tools and techniques for implementing systems of programs, and puts the tools to work in a small case study.

BACKGROUND

Programs and systems of even modest complexity are typically influenced by at least two constituencies, each with different objectives. At the risk of over-generalizing, the *programmer's* objective is design of code which is well-structured, easy to maintain, and uses system resources efficiently. The consumer's, or *user's,* objective is ease of use of the code, while at the same time not sacrificing access to its functionality. The programs should present themselves to the end-user in a "friendly, manner.

With these considerations in mind, let's return to the scenario described in the Introduction and see what went wrong. The program became hard to read due to its length. Identifying portions needing change became difficult. Execution slowed when DATA steps and procedures for the different reports began to contain redundant code. Errors were introduced into the code. Running some reports and not others required changing titles and commenting out unnecessary portions of the program. This was not always done correctly: respecification of a DATA= option in a procedure was not always reflected in the PROC's title. Maintenance became difficult due to the increasing size of the program. *The program did not react well to change.*

What were the sources of the change? Program growth and change is driven by a number of forces, some of which were hinted at in the earlier discussion. The first source of program change is *internal* to the program and its usage. Reports and other analyses are often underspecified. Questions may be raised during development about the program's objectives. The scope of the inquiry may be widened and thus generate demand for more data and output.

A second source of program change is *external.* The data from which the report was generated may increase in volume to the point where a complete listing becomes impractical. Summary statistics and/or partial listings may be required. External sources may extend beyond the organization itself: governmental reporting standards may change, for example.

Another source of the need for change is *programmatic.* Internal or external changes may force the programmer to look for a new approach to the implementation of the code. How, for example, will subsetting of the report dataset be implemented? WHERE statements? WHERE dataset options? IF-THEN statements creating new datasets? Each approach has efficiency and logic considerations which must be addressed.

Programmatic change may also be motivated by the software. The tools offered in a new release or version may hold enough promise to justify a rewrite of the program. The introduction of SQL in Version 6.06 is an example of such a tool. In many instances it was worth the effort to replace long sequences of DATA steps, sorts, DATA steps, etc. with a single, compact invocation of the SQL procedure.

TIME TO REPROGRAM?

The programmer's typical reaction to the changes noted above is to simply increase the size of the program. Frequently, however, the code should be broken up into multiple programs and files. How could the programmer recognize the need for this fragmentation? This section describes some of the conditions.

Excessive program length. "Excessive," of course, varies greatly by both context and programmer disposition. For this paper's purposes it means a program long enough to get lost in; a length where it becomes hard, even with good commenting and formatting, to quickly locate DATA steps, macros, and internal documentation.

Increased breadth of program function. The earliest version of a program may simply read a raw dataset and print the contents. If the program's function grows to reading, printing, performing a variety of statistical analyses, producing graphics, and so on, it may be appropriate to split it up. This point is continued below.

Continual commenting. As complexity grows, so does the likelihood that not all of the program's capabilities will be exercised in a given execution of the program. For example, if a single program creates a SAS dataset and four different reports, it is likely that the

first time the program is run the dataset will be created. It is also likely that subsequent runs will not need to re-create the dataset, focusing instead on creating one or more reports. The brute force method to conditionally execute parts of a program is use of /* and */ comments. They "turn off" code contained between them. This is awkward, error-prone, and not feasible if a naive user needs to run the program.

Modification of items. Constants or names in the program may require periodic modification. The beginning and ending months of the reporting period may need to change, and items used in calculations such as interest rates may also change. Once the program is completed it is usually undesirable for the person using the program to edit it to make these modifications. The program may be inadvertently (and incorrectly) altered, or the items in question may not all be changed. An interest rate used *five* times in the program may be changed only *three* times, for example. Ideally, these values should be *passed* to the program, which in turn should use them as needed.

End-user involvement. As hinted at above, if the program will be run by a "naive" user not familiar with SAS syntax or programming, there should be some mechanism in place for insulating the program from the user. The response to this need should benefit all audiences: the programmer will preserve program integrity, and the end-user will be given a simpler way to provide values to the program.

THE SYSTEM TOOLSET

The scenarios described above are forbidding. Circumstances require program growth, constants and entity names are constantly changing, and parts of the program must be turned on an off depending on the needs of the user. Fortunately, the SAS programmer has a variety of tools to make order out of what could be a chaotic state of affairs. This section describes these tools and gives brief examples of their use.

Throughout the Exhibits and Figures, text in *italics* indicates "meta code," high-level descriptions of a programming activity rather than working code. An ellipsis (...) indicates a portion of the program not displayed because it was not germane to the example.

%INCLUDE. The %INCLUDE statement identifies an external file containing SAS statements which will be inserted, or "included," in the current program. The program actually executed by SAS is the original, or "calling," program plus the contents of the included file. The program is the same as if you manually copied over the contents of the external file with an editor rather than referenced it with %INCLUDE.

The %INCLUDE statement gives the programmer a great deal of flexibility in designing programs. It encourages the development of a program hierarchy: a main, or driver, program includes external code as necessary. If code for "weekly report, format 1" is needed, the driver can include file WKFMT1. Likewise, "weekly report, format 2" can be inserted as needed. The only code which actually executes is the code which is needed. %INCLUDEs for unwanted reports or program tasks can be commented out.

The price of this flexibility is ease of development and maintenance. Code can be scattered across many files. Tracing program execution can become difficult if included code is nested (included within an included file). Changing item names can become problematic. In the single-program solution a dataset could be renamed with a single "change" command to the program editor. With multiple programs, however, the change command must be executed for each affected file.

MACROS. The SAS macro language makes large standalone programs viable. It is also a useful tool for developing systems of programs. Its use in the standalone program has the benefit of being both extremely powerful and simple to implement. Consider the program described in abstracted form in **Exhibit 1**.

Exhibit 1: Meta-code

```
Create dataset PERM.PASS1 with a DATA step

Sort PERM.PASS1 by region, create PASS2

*Report 1:* PROC REPORT on dataset PASS2, order
by region

PROC SQL, to create macro variables for Report 2

*Report 2:* DATA step with PUT statements.   Run
against PASS1
```

There are a few distinct tasks being carried out here: permanent dataset PASS1 is created, and two reports using two datasets are written. If we only wanted to create the dataset we could comment out the remainder of the program. If we had already created PASS1 and wanted to create Report 2, we would comment out the DATA step, SORT, and REPORT procedures. A cleaner way to do this is by using a macro, shown in **Exhibit 2**.

Exhibit 2: Macro Implementation

```
%macro driver(data=, rept1=, rept2=);
%if &data. ^= %str() %then %do;
   Create dataset PERM.PASS1: DATA step: read
        raw data, create new variables
   %end;
%if &rept1. ^= %str() %then %do;
   Sort PERM.PASS1 by region, create PASS2
   Report 1: PROC REPORT on dataset PASS2, order
        by region
   %end;
%if &rept2. ^= %str() %then %do;
   PROC SQL, to create macro variables for Re-
        port 2
   Report 2: DATA step with PUT statements. Run
        against PASS1
   %end;
%mend;

%driver(data=yes, rept2=yes);
```

This is an improvement over the first version of the program. Code which doesn't need to be executed does not need to be commented out; the macro simply avoids passing it to the SAS Supervisor for execution. Note that the macro call (the last line in the program) is still part of the program. The program must still be al-

tered each time a macro parameter changes. Notice also that the macro contains all the program statements. A different approach is to use %INCLUDE statements in the macro to bring in external files. This is illustrated later, in Exhibit 12.

Later Exhibits illustrate another benefit of developing macro-oriented, modular programs. Macros can be treated as generalized tools usable by multiple programs. Development time can be spent on crafting a well-documented, thoroughly debugged program whose benefits can be shared many times by different programs. Contrast this with continual development and rewriting of essentially the same code in many locations. Modular code is quicker to develop, easier to debug, and ensures consistency of usage across all programs in a system. These advantages are not unlike those put forth for subroutines and objects in other programming languages.

MACRO VARIABLES. One of the problems identified in the previous section was continual adjustment of constants and item names in a program. Although this capability is vital for robust, useful code, the practice is error-prone for all who use it, naive end-user and weary programmer alike. Macro variables are a partial solution to the problem. They can be employed as part of a macro, as shown above, or implemented independent of the macro language. Let's look at a sample program (**Exhibit 3**) before the macro variable "makeover."

Exhibit 3: Without Macro Variables

```
data select;
set master(where=(region = "1A"));
adjust = base * 1.03;
run;

proc print;
title "Region 1A, adjustment of 1.03";
run;
```

Changing REGION or the operand in the ADJUST assignment statement is trivial in this case. But even here, one must remember that as region or rate is changed in the DATA step it must also change in the TITLE statement. Consider how easy it would be to forget to make changes if the program extended over hundreds of lines - a not unrealistic scenario. The revised program might look like **Exhibit 4.**

Exhibit 4: Using Macro Variables

```
%let region = 1A;
%let rate_chg = 1.03;

data select;
set master(where=(region = "&region."));
adjust = base * &rate_chg.;
run;

proc print;
title "Region &region., adjustment of
       &rate_chg.";
run;
```

The new program is more reliable. Changes made to macro variables REGION and RATE_CHG are reflected throughout the program. The variables are defined at the start of the program, so there is relatively little impact on the code - the user gets into the program, makes changes, and leaves. A tool to isolate these changes from the program is discussed next.

PASSING PARAMETERS. One of the primary reasons for splitting the standalone program into a system of programs is to prevent the program's users from modifying the code. Macro variables, discussed earlier, are a step in this direction. Still, they require end-users to alter files written by programmers. Several SAS features can completely remove the user from the program. In both interactive and batch environments we have the %INCLUDE statement and AUTOEXEC.SAS files. Developers in interactive environments can also use SAS/AF and macro and DATA step windows.

%INCLUDE REVISITED. We have already seen %INCLUDE as a means to bring in code from an external file. That is, it inserted DATA steps and PROCs into the program stream. Its use may be safely generalized to inserting *any* statement or even portion of a statement. Its flexibility becomes an ally when building systems, since it can also be used to insert parameters into the program. The previous section's example is rewritten in **Exhibit 5** and **Figure 1.**

Exhibit 5: %INCLUDE a Parameter File

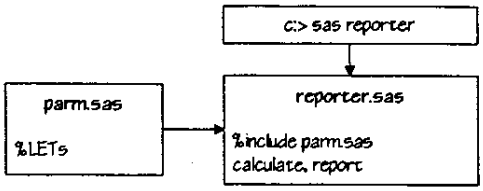
PARM.SAS

```
* Used by REPORTER.SAS, REPORT2.SAS;
%let region = 1A;
%let rate_chg = 1.03;
```

REPORTER.SAS

```
* Insert macro variable definitions ;
%include 'parm.sas';

data select;
set master(where=(region = "&region."));
adjust = base * &rate_chg.;
run;

proc print;
title "Region &region., adjustment of
      &rate_chg.";
run;
```

Figure 1: Program Flow, Exhibit 5



This small system of programs demonstrates principles used in the most complex applications. A parameter file, PARM.SAS, holds macro variable definitions. The production program, REPORTER.SAS, immediately reads and executes the parameter file. Thus even though we don't see the definition of macro variables REGION and RATE_CHG in the program they do, in fact, have values. This design means that the parameter file could be changed and rerun with no change to REPORTER. The insulation

of code from user comes at the price of having to look in two files to fully understand what the program is doing. The comments at the beginning of the files suggest their function and program linkage.

AUTOEXEC FILES. Writing systems of programs often results in the programmer "discovering" parts of the SAS system that went unnoticed in simpler times. The AUTOEXEC file falls in this category for many SAS programmers. This file contains statements to execute during startup of the SAS system. In interactive SAS, the statements are executed before the user is able to submit statements within Display Manager. In batch SAS, the statements are executed prior to the program file(s) specified in the command line invoking SAS. AUTOEXEC files typically set options and define frequently-used FILENAMEs and LIBNAMEs. If the programmer writes a macro and checks the value of the SYSENV system macro variable some activities may be carried out only in interactive settings, others only in batch.

Their use is not unlike an %INCLUDE statement. Any valid SAS statement may be coded. The only difference is that AUTOEXEC files are executed at a specific time and usually perform activities which are appropriate for *all* of the user's SAS sessions. **Exhibit 6** and **Figure 2** extend the example in the previous Exhibit.

Exhibit 6: AUTOEXEC Implementation

AUTO.SAS

```
* Used by REPORTER.SAS, REPORT2.SAS;
%let region = 1A;
%let rate_chg = 1.03;
```

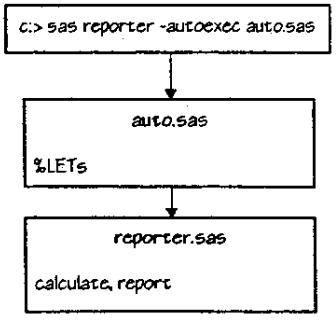
Invocation

```
C:> sas reporter -autoexec auto.sas
```

REPORTER.SAS

```
* Macro vars defined in AUTOEXEC file. ;
data select;
set master(where=(region = "&region."));
adjust = base * &rate_chg.;
run;

proc print;
title "Region &region., adjustment of
    &rate_chg.";
run;
```

Figure 2: Program Flow, Exhibit 6



Program REPORTER does not need to bring in, or %INCLUDE, the parameter file. This is done during startup, via the AUTOEXEC option in the command line. Note also that *any* program using the AUTOEXEC file will use the same macro definitions. This provides a measure of consistency across all programs within the project.

BASE SAS WINDOWS AND SAS/AF. Earlier examples have demonstrated how AUTOEXEC and %INCLUDE files can be used to pass parameters (macro variables) to production programs. The advantage to the programmer is that the programs are run without being repeatedly "touched" by non-programmers. The end-user, however, continues to be encumbered by needing to open a file for editing, remembering macro variable syntax, and so on.

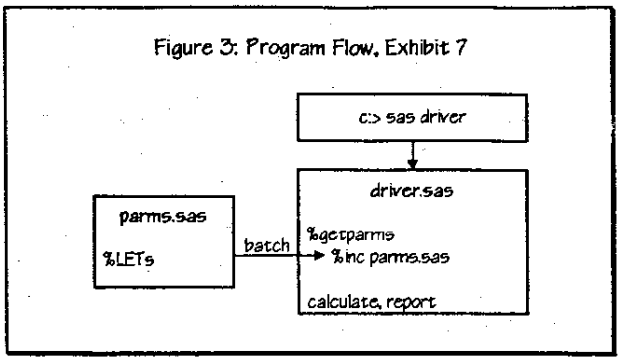
Programs which run in an interactive setting can take advantage of the windowing features of Base SAS. The DATA step and macro language allow definition and display of windows. These screens can display information about settings, reports, and other features controllable by the user. The window definition statements can also identify fields to be filled in by the user. They have a limited ability to react to the user's entries. This allows the program to validate user entries and recover if invalid or questionable values are entered. SAS/AF programs written in Screen Control Language (SCL) extend these capabilities. Validation capabilities are more powerful, and screens are easier to design and are visually appealing.

Once user entries are validated, the DATA step or macro managing the screens can write macro variables as needed. The effect is the same as the %INCLUDE, AUTOEXEC, and in-program macro variable definitions discussed earlier. The program statements using the macro variables are *exactly* the same. The only difference is that the program environment is friendlier, allowing the user to make mistakes and immediately recover from them. The program in **Exhibit 7** and **Figure 3** uses macro windows.

Exhibit 7: Invoking Programs via a Macro Window

DRIVER.SAS

```
%macro getparms;
%if &sysenv. = INTERACTIVE %then %do;
    %* Prompt for macro variables REGION and
          RATE_CHG ;
    %window parmwind macro window definition;
    %do %while (stop condition);
        %display parmwind;
        validate fields entered in window
    %end;
    %end;
    %else %do;
          %inc '[userid.project1]parms.sas';
          %end;
%mend;

%getparms;

data select;
set master(where=(region = "&region."));
adjust = base * &rate_chg.;
run;

proc print;
title "Region &region., adjustment of
          &rate_chg.";
```



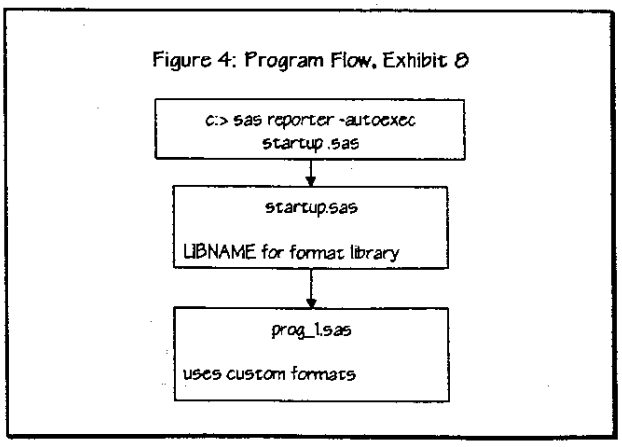
Notice we are, in the interactive case, back to a standalone program. If the program is running in batch mode (determined by automatic macro variable SYSENV) it attempts to insert the contents of an external file. In both environments, the program which executes below the macro is identical and the user does *not* need to modify production code. The desired developer-user separation is maintained.

FORMAT LIBRARIES. User-written formats are powerful additions to even the simplest programs. Their utility is even more evident when systems of programs are involved, since they can ensure consistent mapping and recoding across programs. One strategy for their use is to %INCLUDE a file with PROC FORMAT definitions.

This approach is inefficient. The inefficiency arises when SAS has to create the formats each time a program executes. Even if the program needs only a few of the library's dozens of formats *all* the formats must be created. A cleaner solution is to store the formats in a library and place a LIBNAME in the AUTOEXEC file. This is illustrated in **Exhibit 8** and **Figure 4**.

Exhibit 8: Using a Format Library

```
AUTOEXEC file STARTUP.SAS

   libname library '[userid.project.formats]';

Program invocation (VMS)

   $ sas /autoexec=startup.sas prog_1

PROG_1.SAS (partial):

   proc print data=section1;
   format branch branch.;
```



The appeal of this approach is apparent even in the case of PROG_1. The code used to create the format library in the FORMATS subdirectory does not need to be read in and executed. Nor does LIBRARY need to be defined, since this is handled by the AUTOEXEC file. (In this VMS example, keep in mind that the call to SAS could be made even simpler for the user by placing the invocation in a COM file or defining it as a symbol.)

The real power of the format library is revealed when another program needs one or more of the formats. If it uses the same AUTOEXEC file it will, by definition, have the same library and the same formats available for use. We are assured of data consistency across all applications, and have it efficiently implemented to boot.

DESIGN GUIDELINES

The discussion so far has been limited to a rather detailed discussion of some of the SAS system's tools. This section presents some general comments about the use of the tools and identifies features of the host operating system which should be used. The successful transition from standalone code to a system of programs is intimately tied to the programmer's ability to organize and manage program files and libraries. This ability depends, in turn, on being able to use non-SAS tools effectively.

This section describes design and usage guidelines for both SAS and non-SAS tools. It emphasizes that good systems are the result of exercising a number of skills in both SAS and the operating system in which the SAS code functions.

SAS PROGRAMMING CONSIDERATIONS. Use of macros, parameter files, and %INCLUDEd code increases the need for program order and stylistic consistency among the programs. Good style and logical consistency are *always* important, even in trivial, single-program applications. If left unheeded in *systems* of programs, however, one can be guaranteed chaos and instability. Consider the guidelines and suggestions presented below.

Change critical datasets sparingly. Changes to important datasets should take place in relatively few locations. If the task at hand is basically reading data into a SAS dataset and having multiple programs report or analyze the data, there should be as few programs as possible modifying the permanent datasets.

Develop utility macros. The macro language is a useful tool to make SAS code modular. A set of generalized macros available to all system programs is a good base on which to craft complex systems. These utilities could perform tasks such as testing for the existence of a dataset or variable, perform calculations and assignments performed in multiple locations, or set SAS/Graph symbols and device driver parameters. Their function could be more prosaic: simply standardizing the presentation of titles or footnotes is a valuable activity. In general, any task that is performed more than once, especially if it is performed in more than one *program* or file, is a candidate for inclusion in the utility library.

Avoid unnecessary nesting. If you catch yourself writing macros which call macros which %INCLUDE programs you are probably missing a simpler way to write the code. Excessively nested code is hard to maintain and devilish to debug. If you have to nest macros or %INCLUDEd code, turn on the tracing options and use the debugging techniques described below.

Do not alter the SAS environment in subordinate code. Most systems are written as a hierarchy of programs. A top-level, or driver, program accepts parameters, calls subordinate code as required, and adjusts the SAS environment. These tasks include aesthetic features such as line and page size, as well as more critical issues such as how SAS should react to invalid data, missing datasets, invalid formats, and so on. Only the driver should establish these settings. Consider the confusion which would result if the MISSING option was reset by %INCLUDEd code to "?" and a procedure in a later %INCLUDE displayed a host of ?s. If the resetting is necessary, document it with comments.

Do not exit SAS or deallocate resources in subordinate code. If we invoke a macro or include an external file, we should expect to return from it. Likewise, if a LIBNAME was defined when a subordinate program was called, we should have it available when control returns to the calling program. Don't exit SAS or free/deallocate resources in subordinate code.

Leave "footprints." Numerous options (described below) are available to display macro and included code. It is still helpful, though, to insert your own messages into the SAS Log. %PUT statements are easy to code and provide an extra level of control over the monitoring the program's flow. The message should include the name of its macro or %INCLUDE file and should be easily spotted in the SAS Log (use a row of #'s or *'s). At a minimum, use %PUTs to indicate when the program or macro began and ended execution. Documentation standards should include at least identification of datasets read and written, as well as macros and macro variables used. This is illustrated in the next section's Exhibits.

Use SAS system options. Several options can be used to control the amount of output resulting from use of macros and included code. Macro options implemented as toggles include: MPRINT (display the macro code generated at compile time), SYMBOLGEN (display resolved values of parameters passed to a macro), and MAUTOSOURCE (display macro code generated by calls to the autocall macro library). Another macro option, SASAUTOS, specifies the search path for macro autocall libraries. An %INCLUDE-related option, implemented as a toggle, is SOURCE2 (print the external program in the SAS Log).

OPERATING SYSTEM CONSIDERATIONS. The SAS programs should not exist in a vacuum. Indeed, if a system of programs is developed without investigation of how to use the host system's utilities and file structures, it is likely that development will be unduly complex or inefficient . Some of these features are described below.

File structures. Even simple systems using relatively few programs and datasets can be difficult to organize. Program files, documentation, SAS datasets, raw datasets, and format catalogs should be separated into different datasets or directories. In an MVS environment, for example, programs could be stored in a partitioned dataset. In directory-oriented systems such as DOS and VMS, raw data, programs, and SAS datasets could be placed in separate directories with names suggesting their content. This practice does not incur any performance penalties and makes the system's organization a bit more apparent to both developers and maintenance programmers.

Editors and utilities. The SAS Display Manager editor is adequate for most SAS programming tasks. An editor with macro or script capabilities, or which can horizontally shift blocks of text is possibly not needed for everyday work. However, these capabilities are sometimes required, and the default SAS editor falls short. You should have your programming "toolbox" filled with editors and utilities which are suited for these and other specialized tasks.

Beyond program editing, consider another routine task in system development and maintenance: locating a string (variable or dataset name) in more than one program. Utilities such as Norton's TS, UNIX GREP and its variants, and VMS SEARCH greatly simplify this task. The appropriate response for these programming needs should not be limited to the SAS toolset. Other tools such as change control programs and programming shells such as IBM's ISPF are programming "environments" which help maintain order and consistency across program files.

Command procedures. Most operating systems provide the system developer with a set of tools and a scripting language which allow a dialog with the user. Screens may be "painted" requesting and displaying information. The user's responses can be captured and validated before being acted upon.

The developer can use the system's command language to build a parameter file to pass along to SAS for execution. This can be a palatable alternative to developing with similar tools (macro windows, SAS/AF) in the SAS environment. Note that the capability of these tools varies *greatly* by operating system . Use of command procedures is demonstrated, albeit in an abstract manner, in the next section.

CASE STUDY

The SAS features described in the previous sections are perhaps best illustrated by a small case study. Again, we will see that the features are most effective when used together: macros which %INCLUDE files result in better code than a macro-only or %INCLUDE-only design. The case describes a small project with a single dataset and, at first, a single report. As it progresses, the specifications change and the program evolves into a small system.

The project starts out simply enough. In fact, at this stage it is simply a small, probably one-time request for a report. The program resembles **Exhibit 9**. The program hardly merits discussion. It *is* instructive to include it as a reference point, however, since we'll see in later Exhibits how simple requests can realistically turn into complex applications.

Exhibit 9: Starting Point - A Small, "Innocent" Program

```
libname ...;
proc print ...;
```

Based on information in the report, and possibly impressed by the speed with which it was generated by the programmer, the end-user requests new reports from the same dataset. Calculations are required for one of the reports. Another report requires the display of a district code as a more understandable text string. This is accomplished by using the FORMAT procedure. The reports are all run on the current master dataset and will need to be run more than once, on an *ad hoc* basis. The program is displayed in **Exhibit 10**.

Exhibit 10: Program Growth in Response to Changing Specs

```
libname ...;
data perm.ver2;
set ... ;
calc's ;

sort master dataset, creating temp2
transpose temp2, creating temptran
report1, using temptran

report2, using perm.ver2

proc format;
report3, using custom format and master dataset
```

The program may be correct, but it is far from ideal. Running reports selectively requires commenting out not only the actual report-writing procedure code but also procedures and DATA steps before it. One way to avoid the potential errors which could be introduced by this commenting is to use a macro. This solution is a reasonably effective way to keep the project code contained in a single program (**Exhibit 11**).

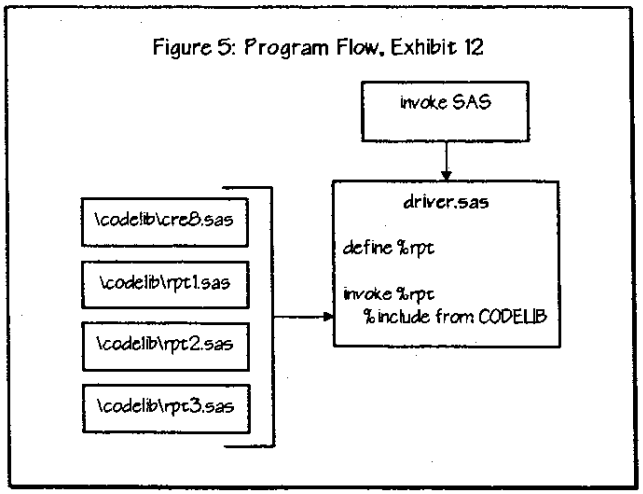
Exhibit 11: Macro Implementation

```
libname ...;
%macro rpt(cre8=, rpt1=, rpt2=, rpt3=);
%if &cre8. ^= %str() | &rpt2. ^= %str()
    %then %do;
    data perm.ver2;
    set ... ;
    calc's ;
    %end;
%if &rpt1. ^= %str() %then %do
    sort master dataset, creating temp2
    transpose temp2, creating temptran
    report1, using temptran
    %end;
%if &rpt2. ^= %str() %then %do;
    report2, using ver2
    %end;
%if &rpt3. ^= %str() %then %do;
    proc format;
    report3, using format and master dataset
    %end;
%mend;

%rpt(cre8=Y, RPT3=y);
```

The code in Exhibit 11 is appealing, and has many of the same characteristics of the code in Exhibit 2. It effectively separates the different tasks in the program and results in efficient code, since only those DATA steps and procedures needed for a report will actually be run. The macro implementation eliminates errors arising from incorrectly commenting out code. Fine, but what if the program was unacceptably long? Rather than have all program statements in-line, we could implement the macro with %INCLUDEd code, as shown in **Exhibit 12** and **Figure 5**.

Exhibit 12: Reduce Macro Size via %INCLUDE

```
\PROGRAMS\DRIVER.SAS
   libname ...;
   %macro rpt(cre8=, rpt1=, rpt2=, rpt3=);
   %if &cre8. ^= %str() | &rpt2. ^= %str()
        %then %do;
        %include '\codelib\cre8.sas';
        %end;
   %if &rpt1. ^= %str() %then %do;
        %include '\codelib\rpt1.sas';
        %end;
   %if &rpt2. ^= %str() %then %do;
        %include '\codelib\rpt2.sas';
        %end;
   %if &rpt3. ^= %str() %then %do;
        %include '\codelib\rpt3.sas';
        %end;
   %mend;

   %rpt(cre8=Y, RPT3=y);

\CODELIB\CRE8.SAS
   data perm.ver2;
   set ... ;
   calc's ;

\CODELIB\RPT1.SAS
   report2, using ver2

\CODELIB\RPT2.SAS
   sort master dataset, creating temp2
   transpose temp2, creating temptran
   report1, using temptran

\CODELIB\RPT3.SAS
   proc format;
   report3, using format and master dataset
```



Unless the reports are thoroughly spec'ed out, revisions are likely. **Exhibit 13** introduces an expanded use of macro variables. It allows for selection of specific districts in the report. Note the use of the global macro variable SUBSET in the %INCLUDEd, external files. We see that as the system matures and requirements

expand, so does our use of the tools described in the previous section.

Exhibit 13: Add Subsetting Code, Macro Variables in %INCLUDEd Code

```
\PROGRAMS\DRIVER.SAS
   libname ...;
   %global subset;
   %macro rpt(cre8=, rpt1=, rpt2=,
              rpt3=, select=);
   %if &select = %str() %then %do;
       %let subset = %str();
       %end;
       %else %do;
             parse &SELECT, write macro variable
               SUBSET, which contains a WHERE
               statement
             %end;
   rest of macro same as Exhibit 12
   %mend;

   %rpt(cre8=Y, RPT3=y, select=1 1a 120);

\CODELIB\CRE8.SAS
   data perm.ver2;
   set ... ;
   calc's ;
   &subset.;
   run;

\CODELIB\RPT1.SAS
   report2, using ver2 and &subset.

\CODELIB\RPT2.SAS
   sort master dataset, creating temp2 and and
        using &subset.
   transpose temp2, creating temptran
   report1, using temptran

\CODELIB\RPT3.SAS
   proc format;
   report3, using format, master dataset, and
        &subset.
```

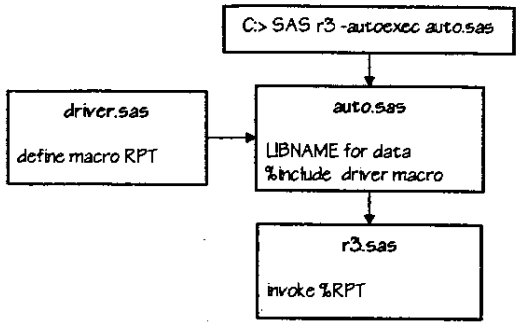
At this point we have a small system of programs. The driver creates the macro variable used for subsetting and inserts code for execution via the macro language and %INCLUDE statements. New reports are easily added. The remaining task is to insulate the program from the end-user, thus preventing inadvertent alteration of the code. The first, non-interactive, attempt is shown in **Exhibit 14** and **Figure 6**.

Exhibit 14: Create Separate End-User Program

```
"Driver" macro RPT in file \CODELIB\DRIVER.SAS

AUTOEXEC file (AUTO.SAS):
   options line, page sizes, macro printing;
   libname ...;
   %include '\CODELIB\DRIVER.SAS';

SAS invocation:
   c:> sas r3 -autoexec auto.sas

User program R3.SAS:
   %rpt(cre8=y,rpt3=y);
```

The macro definition is stored in an external file and defined during startup, as part of the AUTOEXEC sequence. Any user program using the AUTOEXEC file can use macro RPT. The problem, of course, is that the user needs to know how to use the macro and deal with its syntax.

Figure 6: Program Flow, Exhibit 14



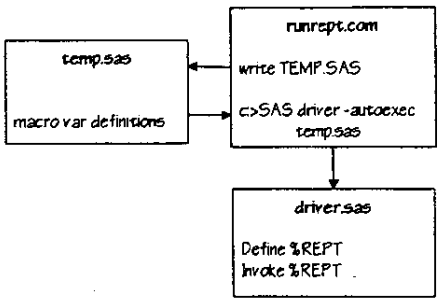
An alternative solution to Exhbit 14's file organization is making DRIVER.SAS part of an Autocall macro library. Yet another variation is concatenation of the AUTOEXEC file and R3.SAS. Multiple program files are not supported in all systems, however, so program portability across platforms would be sacrificed if this technique were used.

Any of the solutions noted above will work, but all lack "friendliness." A solution to this problem is shown below. The "system command file" could be a DOS BAT file, VMS COM file, TSO CLIST, or any operating system facility which can query the user and write to an external file. This approach, of course, is feasible only in interactive environments. A VMS implementation is shown in **Exhibit 15** and **Figure 7**.

Exhibit 15: Interactive Invocation via System Command File

```
System command file:
   Prompt for, validate parms. build temporary
   AUTOEXEC file TEMPAUTO.SAS, which looks like:
   %let subset = %str(WHERE dist in ('1',
                      '1a'));
   %let cre8 = Y;
   %let rpt1 = ;
   %let rpt2 = ;
   %let rpt3 = Y;

   C:>sas \codelib\driver -autoexec tempauto.sas


DRIVER.SAS
   * no parameters for RPT, since these are de-
       fined by the %LETs in the temp. AUTOEXEC
       file;

   %macro RPT;
   macro definition, as in Exh. 12, 13
   %mend;

   %RPT;
```

Figure 7: Program Flow, Exhibit 15



Notice that the small system described in the Exhibit does not contain *any* user-written programs. The end-user simply responds to prompts and visual cues and fills in the blanks. Once the responses are validated, the code is executed. The user never has to code a macro or use a program editor.

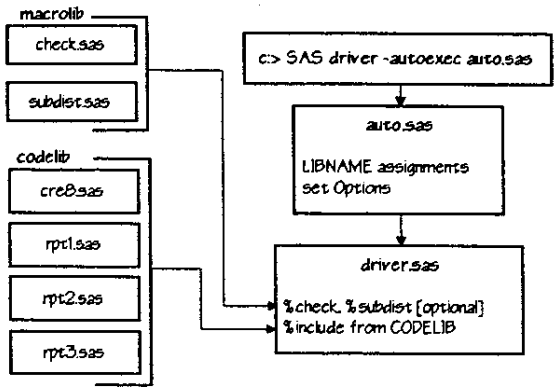
The last example (**Exhibit 16** and **Figure 8**) builds on the code in Exhibits 12, 13, and 15. The code and program flow diagrams are presented first, followed by an extended discussion of their features and usage.

Exhibit 16: "Final" System

```
AUTO.SAS
   libname perm ...;       * data ;
   libname library ...;    * formats ;
   libname sascats ... ;   * catalogs (graphics,
                             AF apps) ;
   options mautosource sasautos='\macrolib';

<driver: operating system, interactive SAS, or
user's batch program containing:>
   %check(perm.master, stop);  * optional ;
   %subdist(1 1a 120);         * optional ;
   %include '\codelib\cre8.sas';
   %include '\codelib\rpt2.sas';

<invocation>
   c:>SAS driver -autoexec auto.sas

\MACROLIB\CHECK.SAS
   %macro check(dsn, action);
   %put Macro CHECK called with parm &DSN. ***;
   %if dataset exists %then %do;
       %let status = OK;
       %put Dataset was located. ;
       %end;
       %else %do;
             %let status = NOTFOUND;
             abort if &ACTION is STOP;
             %put Dataset could not be located;
             %end;
    %put Macro CHECK terminating *****;
    %mend;

\MACROLIB\SUBDIST.SAS
    %macro subdist(list);
    %put Macro SUBDIST called with selection
         parms '&list.' ****;
    create macro var SUBSET
    %put Macro SUBDIST terminating. Macro var
         SUBSET is '&subset.';
    %mend;

\CODELIB\CRE8.SAS
    see Exhibit 12

\CODELIB\RPT1.SAS
    see Exhibit 12
```

```
\CODELIB\RPT2.SAS
    * \CODELIB\RPT2.SAS
      Writes report style 2: straight listing
         using PROC REPORT.
      Uses dataset PERM.VER2, created by macro
         CRE8.
      ;
    %put Begin execution of \CODELIB\RPT2. ;
    %put Subsetting criteria: '&subset.';

    proc report data=perm.ver2 ...;
    ...
    run;

    %put End execution of \CODELIB\RPT2. Return
         code from REPORT (macro var SYSERR):
         &syserr. ;

\CODELIB\RPT3.SAS
    see Exhibit 12
```

Figure 8: Program Flow, Exhibit 16



Many of Exhibit's features were touched upon earlier. Their integration in a single example is illustrated here, however, and warrants extended discussion.

- To give a feel for how the external files should be documented and %PUTs used, a sample comment block is displayed for RPT2.SAS and the macros.
- The program actually executed by SAS (*driver*) is only four lines long. The techniques employed here rely heavily on %INCLUDEd files and macros to bring in code as needed.
- This external code takes advantage of the host operating system's file organization. It stores programs referenced by %INCLUDEs in directory CODELIB and macros referred to in the AUTOCALL library in directory MACROLIB. The separation *per se* does not make the system run better. It is, instead, a mechanism to help developers and maintenance programmers be able to more readily locate portions of the system.
- The AUTOEXEC file allocates resources (in this case, LIBNAMEs) needed by *all* programs in the project, not just the current program. This may result in unused LIBNAMES but may, in the long run, be more efficient: we know that everyone will have access to *all* project resources. The AUTOEXEC file makes another global feature available to all programs: a collection of macros is allocated via the SASAUTOS option.
- The SAS invocation can be bundled into a system command file. This would save the user the trouble of entering the

AUTOEXEC reference. In a DOS environment, for example, the following BAT file, RPT, could be used:

```
c:\sas\sas c:\proglib\rpt_driver
   -autoexec \proglib\auto.sas
```

The user would have to enter RPT at the "C" prompt to execute the statement.

- The macros and included code contain diagnostic %PUT statements. They indicate the beginning of the code and the end, along with other relevant information such as parameters passed, return codes, and so on. Some of this information is available using SAS options (macro variables at macro invocation can be displayed via the SYMBOLGEN option, for example). The advantage of using %PUTs is that you can limit the amount of information displayed and give it descriptive text. This gives more context to the message and assists debugging. The advantage of this technique is minimal in the case of Exhibit 16, but becomes much more pronounced in larger, more complex systems.
- A subtle but important feature of the system is its ability to change. New macros can be developed, programs rerun with different subsetting criteria, etc. without affecting existing code. Changes are isolated to the new programs (%INCLUDEs and macros) and possibly to the driver program (if a new program has to exist). The modular nature of the code minimizes the effort to add new code.

CONCLUSION

No single document can be an exhaustive treatment of the issues underlying system design. What this paper has attempted is to give a feel for when the need for a *system* of programs (rather than a single program) arises. It also presented some of the tools available for use in the SAS System and some approaches to their implementation. It is up to the client and programmer to determine the suitability of the tools and approaches to their needs. Suitability and choice of tools will be determined by degree of interactivity desired, the type of reports and data needed, and to a large degree the working habits and preferences of the programmer.

CONTACT INFORMATION

Your comments and suggestions are valued and encouraged. Contact the author at:

Trilogy Consulting Corporation
102 Westbury Drive
Chapel Hill, NC 27516-9154

work: 919-990-5993
message: 919-942-2028

Internet: fcd24712@glaxo.com