

A Modular Approach to Portable Programming

Michael A. Litzsinger, Quintiles Inc.

Lisa Kaye Brooks, Quintiles Inc.

*There is a balance between programming in a flexible **modular** manner **versus** a direct **standalone** manner. Segmenting programs into components aids flexibility but also introduces complexity. Creating standalone programs from redundant and repetitive components makes applying widespread changes tedious and time consuming. Consistency and version issues are raised as a program library grows.*

*This paper presents SAS® programming techniques that allow operating system and project assignments to be made centrally. **Components** generally include anything that can be best defined in a single location, such as LIBNAMEs, TITLEs, FOOTNOTEs, or system options. More advanced uses include single-source definition of macro variables in a conditional manner, allowing numbering or key elements to be controlled from one place.*

*This paper also details the use of **MFILE**, in conjunction with **MPRINT**, to direct executed program statements to a standalone output program. **MFILE** is a relatively unknown system option that was known as **RESERVEDB1** in SAS 6.12. It is a simple way to port code using the output directing capabilities of the **MPRINT** system option. The resulting SAS program is stripped of all traces of modularity and underlying macros, and can be executed as a standalone program. The combination of these two techniques results in truly portable SAS programs while retaining the flexibility.*

Modular versus Standalone:

Modular programming is the breaking down of a program into parts that are of more manageable size and have a well-defined purpose. Modules should fit intuitively together into a system of components. In any development environment, including SAS, modular programming is how larger, more complicated programs are constructed. The first step is to break the task into its basic parts, which leads to defining intermediate steps, and ultimately devising a comprehensive and efficient solution. Modules can be developed individually, validated, and

then used throughout an organization, promoting teamwork, efficiency, and innovation.

The advantages of modularity are obvious. Code only needs to be written once, which allows quick modifications. It provides a framework that dictates how subsequent programming should be incorporated. Modularity fits an environment where several programmers share work. The drawbacks include a slightly higher learning curve, a moderate level of complexity (even for simple tasks), and some required management of the macro libraries of SAS programs. An important issue is how to deliver simple portable programs to achieve contractual client obligations without providing proprietary SAS programs. Lastly, programs must adhere to corporate or departmental standards to maximize work sharing.

Standalone programs however are inherently portable. Consultants and service industry programmers create code that may be used across platforms or systems, and are often asked to provide executable SAS programs to satisfy documentation or audit trail requirements. Standalone programs are also the obvious method for starting a task from scratch. There is no reason to build in any unneeded complexity as the development unfolds. This works best for a single programmer, or in an environment where specifications do not change. De-macroized standalone programs are the way to provide code without disclosing proprietary work.

The main drawback of standalone programming can be quite problematic. Without the planning intrinsically provided by modular programming, it can be difficult to quickly apply unexpected changes. In fact, the redundant nature of standalone programs can make applying modifications very tedious.

Presented on the next page is an example of a modular program and a standalone program side-by-side for easy comparison followed by a table that contrasts the advantages of each method:

Modular Program (demo itt.sas):

```

*System options;
%include "settings.sas";

*Define project libraries;
%include "libnames.sas";

*Define output settings;
%include "titles.sas";
%include "tnum.sas";
%include "foots.sas";

*Define macro library;
%include "age.sas";
%include "freq.sas";
%include "npct.sas";
%include "maketbl.sas";

*Main program;
data agedata;
  set datalib.demo;
  age=%age(birthdt,visitdt);
run;
%freq(indsn=agedata,colvar=pop,total=Y,
      npct="000 (000%)",out=freqout1);
%freq(indsn=agedata,depvar=age,
      colvar=pop,rowvar=sex,total=Y,
      npct="000 (000%)",out=freqout2);
data tbldata;
  set freqout1
      freqout2;
run;
%maketbl(indsn=tbldata,
         method=ProcReport,
         style=Standard3,source=Y)

```

Standalone Program (demo itt.sas):

```

*System options;
options nocenter ls=150 ps=55 yearcutoff=1910;

*Define project libraries;
libname datalib "c:\datalib";
libname library "c:\datalib";

*Define output settings;
title1 "My Project"
title2 "Table 10.1.1 (Draft)";
title3 "Demographics Summary";
title4 "Intent-to-Treat Population";
footnote1 "Note: Only randomized subjects
are presented.";
footnote2 "Source: DEMO_ITT.SAS,
&sysdate9 &systemtime";

*Main program;
data agedata;
  set datalib.demo;
  age= (year(visitdt)-year(birthdt))-
(month(visitdt)<=month(birthdt))+
(month(visitdt)=month(birthdt)&
day(visitdt)>=day (birthdt)));
run;
[.some code has been skipped .].
proc freq data=agedata;
  tables pop/out=freqout1;
run;
proc freq data=agedata;
  tables pop*sex/out=freqout2;
run;
[.some code has been skipped .].
data tbldata;
  if _n_=1 then set pct;
  set freqout1
      freqout2;
  array n n1-n3;
  array pct pct1-pct3;
  array val $ val1-val3;
  do val=1 to pop;
    val=put(n,3.)||" ("||put(pct,3.)||"%)";
  end;
run;
proc report data=tbldata;
[.some code has been skipped .].
run;

```

Modular:

1. Best for medium and large solutions, whether static or dynamic.
2. Facilitates solving complex problems by breaking them down.
3. Global modification is made easy by only having to update one location.
4. Flexible structure encourages innovation.
5. Eliminates/reduces re-inventing the wheel.
6. Creates environment conducive to sharing workload and farming out programming tasks.

Standalone:

1. Best for small static solutions.
2. Large problems are not clearly defined and thus can be overwhelming.
3. Widespread changes are likely repetitive and tedious.
4. Reluctance to try new ideas because changes are difficult to implement.
5. Code is generated as many times as needed.
6. Only one person can work on a program at a given time.

Primary Components:

For this presentation, the modules have been grouped together into five primary functions (these are the first five listed below). A sixth module type, really a technique to automate elements of the other five modules, will be discussed briefly.

Modules discussed in this presentation:

1. System options, draft stamps
2. Library names and formats
3. Titles, numbering of tables, footnotes
4. Macros for data step manipulations and customized procedures
5. Main program body
6. (Advanced) Conditional execution of code based on macro variable (automatic or explicit)

System Options

Placing system options in their own programs facilitates making project-wide system changes. For example, during SAS Y2K remediation each project had to incorporate the appropriate cutoff year. It was only necessary to add the YEARCUTOFF option to the existing system option module.

Library Names and Formats

A single location for LIBNAMEs and FORMATs is a natural solution for portability. Projects can easily share programs with colleagues on different platforms or directory structures. Referencing unique LIBNAME modules makes it unnecessary to update individual programs.

Titles, Tables Numbers, and Footnotes

Placing project specific details in one location facilitates making quick changes. If an additional table is inserted, this technique makes

it easy to renumber subsequent tables. A central location for table components allows code to be easily reused across similar projects. Differences can be clearly specified, aiding understanding as well.

Macros

The macro language can do some slick tricks in controlling program execution. But macros do not have to be complicated to be extremely useful. The primary use of macro language at all programming levels is the ability to use code repeatedly. Modularizing macros makes them more reusable with availability to all programs within a project or even an organization.

Main Program Body

This is the primary module. It contains code that is unique to the purpose of the program. It usually brings in data and manipulates and/or presents it. The main program brings together all of the sub-modules by calling or passing parameters to them.

Conditional Execution of Code

This applies mainly to the assignment of titles, table numbers and footnotes. Efficiencies can be gained by uniquely setting a macro variable within each of the main program bodies (for example: %LET PGMNAME=DEMO_ITT). Code can then be conditionally executed based on which program is accessing the module. This is a time saver when a global change has a domino effect, like the change of table numbers. A detailed example can be found later in this paper. The SAS system can automatically set program specific macro variables (SYSPARM), but this method is outside the scope of this paper.

Generating Code Using MFILE:

In SAS 6.12, “MFILE” debuted as system option RESERVED1. It became MFILE in SAS 7.

The MPRINT option is traditionally used as a technique in debugging or to display all executed statements to the log. MPRINT sends macro generated code to the log with the prefix MPRINT beginning each line:

```
MPRINT(PGMCODE): libname datalib "c:\datalib";
MPRINT(PGMCODE): data newdemo;
MPRINT(PGMCODE): set datalib.demo;
MPRINT(PGMCODE): dsvar=12;
MPRINT(PGMCODE): run;
```

This code has all macro variables and references resolved and can be thought of as source or “compiled” code reduced to its simplest form. A great feature of MPRINT is that this code can be directed to an external file.

SAS program code is redirected to an output destination when:

- FILENAME MPRINT *path* is defined
- MPRINT and MFILE are in effect
- Code is executed through a macro

The required parts are bolded below. Using this technique, the macro facility saves every executed statement to a file for you.

Running this code:

```
filename mprint "c:\pgmlib\newdemo.pgm";
options mprint mfile;

%macro pgmcode;
  libname datalib "c:\datalib";
  %let var1=12;
  data newdemo;
    set datalib.demo;
    dsvar=&var1;
  run;
%mend pgmcode;
%pgmcode;
```

creates this executable code (newdemo.pgm):

```
libname datalib "c:\datalib";
data newdemo;
  set datalib.demo;
  dsvar=12;
run;
```

By now you may have realized an obstacle: SAS will automatically resolve *all* macro elements in the code it redirects to your saved file. You can’t

have it selectively pass code that contains desired macro elements, such as %include. Because our goal here is to create standalone code containing *some* modular elements (the ability to change system options and LIBNAMEs/FILENAMEs location), we have to be creative in how and when MFILE is used. Generally, we want to use it to create a standalone file of the proprietary elements, and then tweak this file so it can run on a new system. There are several ways to do this, and most methods have a “burden” of knowing the new operating system and location.

The best way to end the redirection of statements is by ending the macro. Other methods write an unwanted remnant to the redirected program:

- OPTIONS NOMFILE; or OPTIONS NOMPRINT; will become the last statement written to the output program.
- FILENAME MPRINT *path*; becomes the last statement written to the first output program. Then the second output program picks up where the previous one left off.

An important decision that must be made upfront is which type of comment text should be used in your source programs. Not all types of comments are passed to the standalone program:

Comment type	Result
*Comment;	Passed through as is: *Comment;
*Comment1 *Comment2;	Text will flow to long lines: *Comment1 *Comment2;
/*Comment*/	Ignored
%*Comment;	Ignored

Thus, only “*” comments are passed unchanged through to the MPRINT file. To avoid long comments on a single line, always end each line with a semicolon. Documentation blocks need to use “*,” comments to neatly pass them through to the standalone program. Comments in macros should not use “*” comments as these are not desired in the standalone program (proprietary elements should be kept transparent).

Outlined in the following table are solutions that make standalone programs created by MFILE more portable. They range from the simple “Text Replacement” method to the most portable “Append” method. The method of choice depends on the knowledge of the target operating system, the degree of desired portability, and the client’s ability to alter and run SAS programs.

MFILE Methods to Create Portable Standalone Programs:

Method	Issues for provider	Issues for client	True Standalone?
Text replacement	Create upfront if OS/location known	Client must replace token text in each program (Operating system may have a text replacement utility to make this easier)	Yes
Portable Media	Define on portable media (Zip disk or diskette)	Client must leave work on media	Yes, restricted – must run directly on media (rules out CDs)
Comprehensive Program		Client must modify a single program in a directory	No – runs all programs in directory
Interactive Program		Client must set LIBNAMEs/ FILENAMEs once per session	No – requires interactive mode
Append	Create upfront if OS/location known	Client must run macro to create	Yes

Text Replacement Method:

The most straightforward method for creating portable standalone programs is to place a token string at the beginning of each source program. This token can then be replaced or swapped out prior to running the program on the new system once it is determined where datasets and included programs will reside. However, for this token text to be benignly passed through, it must be defined as a “*comment;”. Remember, you cannot pass “%include ...” through because it is resolved by the MPRINT facility. Other types of comments are not passed through at all.

Running this code:

```
filename mprint "c:\pgmlib\newdemo.pgm";
options mprint mfile;

%macro pgmcode;
  **** new libnames go here ****;
  %let var1=12;

  data newdemo;
    set saslib.demo;
    dsvar=&var1;
  run;
%mend pgmcode;
%pgmcode;
```

creates this code (newdemo.pgm):

```
**** new libnames go here ****;
data newdemo;
  set datalib.demo;
  dsvar=12;
run;
```

A swap or replacement in each program will need to be made.

Replace the text:

```
**** new libnames go here ****;
```

with the text:

```
%include c:\pgmlib\libnames.sas";
```

and define LIBNAMEs centrally in libnames.sas.

This process can be automated using operating system specific utilities, or each program can be modified individually. For example, a DOS com file or Visual Basic macro can make such changes automatically. And on an OpenVMS system, a SWAP com file can easily make large-scale replacements from one command.

Thus, if the target operating system and file locations can be determined in advance, the standalone programs can be readily created prior to shipment. Alternately, if the client is able to make the changes, they could determine the locations.

Portable Media Method:

Packaging all programs and data onto read/write media (zip drive, diskette) allows programs to be run in place. The programming environment is dependent on the media. There must be ample space available to write back to the disk when required (creating files, table output, SAS system writes, etc.). If LIBNAME and operating system definitions are known prior to delivery it is possible to prepare the programs ahead of time.

MFILE Methods, continued:

The resulting code is “ready to run” and would need no manipulation at run time. The programmer could develop the code directly onto the delivery media. Or it can be created in a testing area and then the text replacement or append method can be used to prepare the code for porting to the new media. This method is really only an option for smaller projects unless an external hard drive is delivered, or higher-density portable read/write storage becomes available.

Prologue to the Comprehensive Program and Append Methods:

The comprehensive and append methods require the program created by MFILE to be stripped of all operating system specific code (such as LIBNAMEs). A program is then run to attach code necessary for running on the new system.

The following example shows how you can modularize specific elements of the source program to make it more portable. Specifically, libnames.sas resides outside of the macro that redirects code to the standalone program. The *librefs* are available for running on the original system but are not included in the output standalone program.

Running this code (newdemo.sas):

```
*Code not directed to the output program;
%let pgmname=newdemo;
%include "c:\pgmlib\libnames.sas";

*set up program for mfile;
options mprint mfile ls=230 ps=80;
filename mprint "c:\pgmlib\&pgmname..pgm";

*Code directed to the output program;
%macro pgmcode;
%let var1=12;
  data newdemo;
    set datalib.demo;
    dsvar=&var1;
  run;
%mend pgmcode;
%pgmcode;
```

creates this standalone executable code (newdemo.pgm):

```
data newdemo;
  set datalib.demo;
  dsvar=12;
run;
```

Comprehensive Program Method:

For this method, a comprehensive program is built in addition to each individual program provided. The logical arrangement is to link similar tasks or group all the programs from a subdirectory. So if you had 20 summary table programs in a subdirectory, you would create another program that called the other 20 programs sequentially. If the operation system and location were known in advance, the comprehensive program could be built upfront. If not, the client user would have to fill in information for file locations (LIBNAMEs/ FILENAMEs) and the location of the included program library.

This method is not a true standalone because all programs are launched from one program. But the simplicity of this approach means that the LIBNAMEs only have to be defined once at the top of the comprehensive program. This approach may be clear enough to the client that they are willing to fill in the location information themselves.

For individual program execution, just comment out unwanted macro calls. For the same result, submit only a highlighted selection of code in an interactive session.

To run this code below, either the program and data locations need to be defined upfront or the client must provide this information in each comprehensive program:

```
%let srcpath=c:\pgmlib\;
%let datpath=c:\datalib\;

options nocenter ls=150 ps=55 yearcutoff=1910;

libname datalib "&datpath";

%macro runall(&pgmname=);
  proc printto file=&srcpath.&pgmname..tbl";
  run;
  %include "&srcpath.&pgmname..pgm"
%mend runall;

%runall(newdemo);
%runall(neweff);
[... et cetera ...]
```

MFILE Methods, continued:

An alternate to the comprehensive program solution is the **Interactive Program** solution, where LIBNAMEs define temporary work files, and individual programs merely call work files and are run subsequently. LIBNAMEs must define work files at the start of every session. The Interactive Program method isn't really a viable solution in a validated program environment, so an example is not provided, but it could be an option in some situations.

Append Method:

In the append method, a library name module is appended to each standalone program generated by MFILE. The following code (append.sas) attaches libnames.sas to newdemo.pgm to create the new program newdemo.sas.

The append.sas program makes use of put, input, and file statements to build the portable program that contains both the libnames.sas component and the standalone program created by MFILE. It is only necessary to update the macro variables DATPATH and PGMPATH in append.sas. The libnames.sas program uses these macro variables to set the *librefs*.

Run this program (append.sas):

```
%let pgmpath=c:\pgmlib;**target programs;
%let datpath=c:\datalib; **data directory;

%macro append(pgm=,srcpath=);
filename pgmout "&srcpath.&pgm..sas";
data _null_;
  file pgmout;
  put @1 "%" "let pgmname=&pgm;";
  put @1 "%" "let srcpath=&srcpath;";
  put @1 "%" "let datpath=&datpath;";
run;
data _null_;
  infile "&srcpath.libnames.sas" pad;
  input @1 line $256.;
  file pgmout mod;
  put line;
run;
data _null_;
  infile "&srcpath.&pgm..pgm" pad;
  input @1 line $256.;
  file pgmout mod;
  put line;
run;
%mend append;
%append(pgm=newdemo,srcpath=&pgmpath);
```

along with this code (c:\pgmlib\libnames.sas):

```
libname datalib "&datpath";
libname library "&datpath";
```

to create this executable code (newdemo.sas):

```
%let pgmname=newdemo;
%let srcpath=c:\datalib;
%let datpath=c:\pgmlib;
libname datalib "&datpath";
libname library "&datpath";
data newdemo;
  set datalib.demo;
  dsvar=12;
run;
```

Note that a single append.sas program can be constructed to process multiple programs. The append method is probably the best all-around solution. It can be set up to prompt for dataset and program locations in an intuitive manner.

Advanced Components:

One of the most helpful ways to automate modular programs is the use of single-source macro variables. Passing the program name in a macro variable can be used to conditionally select code to be executed in other modules. In the following example, table attributes are assigned based on the name of the program (on the next page):

Section of code (tnum.sas):

```
%macro tnum;

%if pgmname=DEMO_ITT %then %do;
  %let tnum=10.1.1;
  %let tbltitle=Demographics Summary;
%end;
%else if pgmname=DEMO_SAF %then %do;
  %let tnum=10.1.2;
  %let tbltitle=Demographics Summary;
%end;
[... more definitions have been skipped ...]

%if %index(&pgmname,_ITT)>0 %then %let
  pop=Intent-to-Treat Population;
%else %if %index(&pgmname,_SAF)>0
  %then %let pop=Safety Population;

%mend tnum;

title2 "&tnum";
title3 "&tbltitle";
title4 "&pop";
```

Advanced Components, continued:

Running this section of code (demo_itt.sas):

```
%let pgmname=DEMO_ITT;  
%include tnum.sas;  
%tnum;
```

The resulting macro variables assigned:

```
%let tnum=10.1.1;  
%let tbltitle=Demographics Summary;  
%let pop=Intent-to-Treat Population;
```

The resolved code:

```
title2 "Table 10.1.1";  
title3 "Demographics Summary";  
title4 "Intent-to-Treat Population";
```

Whenever you want to plug in the values 10.1.1, Demographics Summary, or Intent-to-Treat Population, you need only use the tokens &tnum, &tbltitle, or &pop respectively.

Lastly, such conventions make it much easier to rename the delivered programs to number-based names. During development, it is preferred to have function-based program names (program demo_itt produces the Demographic Summary for the ITT population). However, the client may prefer to receive number-based program names (program t10_1_1 produces table 10.1.1) or even both (program t10_1_1_demo_itt produces Table 10.1.1 - Demographic Summary for the ITT population).

Conclusion:

The optimal solution combines the flexibility of modular programming and the straightforwardness of standalone programming. This solution must provide comprehensive code that still allows proprietary work to be preserved. Providing system options and LIBNAME project declarations in a modular format allows true portability to another platform/system. Other components are then delivered as “compiled” non-proprietary code, void of all macros, so that the entire set of programs serves as documentation. This permits easy replication of work. This approach is the most intuitive and is fairly straightforward.

An alternative is to also provide the project specific details (table numbers, titles, and footnotes) to the client in a modular format. This

would allow renumbering and other cosmetic modifications to be made by the client, but this depends on the purpose of the ported code and the roles of those involved. This risks overwhelming the client. The system option and LIBNAME components also can be largely pre-built should that information be determined as a project is being set up. Changing the degree of modularity provided in the portable programs “after the fact” is significantly more time-consuming.

Version Note:

“MFILE” first appeared as a system option named RESERVEDB1 in SAS version 6.12. Beginning with SAS version 7 this system option was renamed MFILE. The functionality remained the same. Companies running both 6.12 as well as a more recent version will need to remember this and to change the system option name when moving programs across versions.

Contact Information:

Michael A. Litzsinger
Lisa Kaye Brooks
Quintiles, Inc.
Statistical Programming
P.O. Box 13979
Research Triangle Park, NC 27709-3979
(919) 998-2888
mike.litzsinger@quintiles.com
lisa.brooks@quintiles.com

SAS and all other SAS Institute, Inc. product and service names are registered trademarks or trademarks of SAS Institute, Inc. in the USA and other countries.

Other brand and product names are registered trademarks or trademarks of their respective companies.