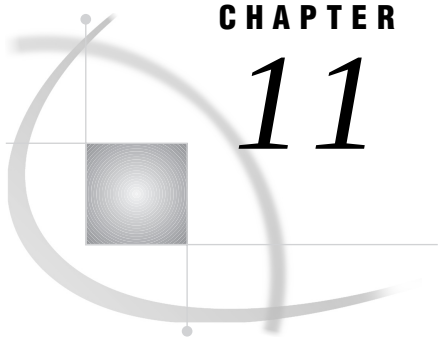

CHAPTER

11

Writing Efficient and Portable Macros

Introduction

The macro facility is a powerful tool for making your SAS code development more efficient. But macros are only as efficient as you make them–there are several techniques and considerations for writing efficient macros. If you intend to extend the power of the macro facility by creating macros that can be used on more than one host environment, there are additional considerations for writing portable macros. This chapter gives you information on both these topics.

Keeping Efficiency in Perspective

Efficiency is an elusive thing, hard to quantify and harder still to define. What works with one application may not work with another, and what is efficient on one host environment may be inefficient on a different system. It is the well-known “your

mileage may vary" situation. However, there are some generalities that you should keep in mind.

Usually, efficiency issues are discussed in terms of CPU cycles, elapsed time, I/O hits, memory usage, disk storage, and so on. This chapter does not give benchmarks in these terms because of all the variables involved. A program that runs only once needs different tuning than a program that runs hundreds of times. An application running on a mainframe has different hardware parameters than an application developed on a desktop PC. You must keep efficiency in perspective with your environment.

There are different approaches to efficiency, depending on what resources you want to conserve. Are CPU cycles more critical than I/O hits? Do you have lots of memory but no disk space? Taking stock of your situation before deciding how to tune your programs is a good idea.

The area of efficiency most affected by the SAS macro facility is human efficiency–how much time is required to both develop and maintain a program. Autocall macros are particularly important in this area because the autocall facility provides code reusability. Once you develop a macro that performs a task, you can save it and use it not only in the application you developed it for but also in future applications–without any further work. A library of reusable, immediately callable macros is a boon to any application development team.

The Stored Compiled Macro Facility (described in Chapter 9, "Storing and Reusing Macros") may reduce execution time by allowing previously compiled macros to be accessed during different SAS jobs and sessions. But it is a tool that is efficient only for production applications, not during application development. So the efficiency techniques you choose depend not only on your hardware and personnel situation but also on what stage you are at in the application development process.

Also, remember that incorporating macro code into a SAS application does not automatically make the application more efficient. When designing a SAS application, concentrate on making the basic SAS code that macros generate more efficient. There are many sources for information on efficient SAS code, including *SAS Programming Tips: A Guide to Efficient SAS Processing*.

Writing Efficient Macros

Just as SAS code can benefit from being written with efficiency in mind, so can macro code. By saving time, CPU cycles, and effort, efficient macros make sense.

Use Macros Wisely

An application that uses a macro to generate only constant text is inefficient. In general, for these situations consider using a %INCLUDE statement. Because the %INCLUDE statement does not have to compile the code first (it is executed immediately), it may be more efficient than using a macro–especially if the code is executed only once. If you use the same code repeatedly, it may be more efficient to use a macro because a macro is compiled only once during a SAS job, no matter how many times it is called.

However, using %INCLUDE means you have to know exactly where the physical file is stored and specify this name in the program itself. Because with the autocall facility all you have to remember is the name of the macro (not a full pathname), the gain in human efficiency may more than offset the time gained by not compiling the macro. Also, macros provide additional programming features, such as parameters, conditional sections, and loops, as well as the ability to view macro variable resolution in the SAS log.

So, the first efficiency tip is: Use a macro only when necessary. And, balance the various efficiency factors and gains (how many times you use the code, CPU time versus ease-of-use) to reach a solution that is best for your application.

Use Name Style Macros

Macros come in three invocation types: name style, command style, and statement style. Of the three, name style is the most efficient. This is because name style macros always begin with a `%`, which immediately tells the word scanner to pass the token to the macro processor. With the other two types, the word scanner does not know immediately whether the token should be sent to the macro processor or not. Therefore, time is wasted while the word scanner determines this.

Avoid Nested Macro Definitions

Nesting macro definitions inside other macros is usually unnecessary and inefficient. When you call a macro that contains a nested macro definition, the macro processor generates the nested macro definition as text and places it on the input stack. The word scanner then scans the definition and the macro processor compiles it. If you nest the definition of a macro that does not change, you cause the macro processor to compile the same macro each time that section of the outer macro is executed.

As a rule, you should define macros separately. If you want to nest a macro's scope, simply nest the macro call, not the macro definition.

As an example, the macro STATS1 contains a nested macro definition for the macro TITLE:

```
/* Nesting a Macro Definition--INEFFICIENT */
%macro stats1(product,year);
   %macro title;
      title "Statistics for &product in &year";
      %if &year>1929 and &year<1935 %then
         %do;
            title2 "Some Data May Be Missing";
         %end;
   %mend title;

   proc means data=products;
      where product="&product" and year=&year;
      %title
   run;
%mend stats1;

%stats1(steel,1991)
%stats1(beef,1997)
%stats1(fiberglass,1996)
```

Each time the macro STATS1 is called, the macro processor generates the definition of the macro TITLE as text, recognizes a macro definition, and compiles the macro TITLE. In this case, STATS1 was called three times, which means the TITLE macro was compiled three times. With only a few statements, this takes only micro-seconds; but in large macros with hundreds of statements, the wasted time could be significant.

The values of PRODUCT and YEAR are available to TITLE because its call is within the definition of STATS1; therefore, it is unnecessary to nest the definition of TITLE to make values available to TITLE's scope. Nesting definitions is also unnecessary

because no values in the definition of the TITLE statement are dependent on values that change during the execution of STATS1. (Even if the definition of the TITLE statement depended on such values, you could use a global macro variable to effect the changes, rather than nest the definition.)

The following program shows the macros defined separately:

```
/* Separating Macro Definitions--EFFICIENT */
%macro stats2(product,year);
   proc means data=products;
      where product="&product" and year=&year;
      %title
   run;
%mend stats2;

%macro title;
   title "Statistics for &product in &year";
   %if &year>1929 and &year<1935 %then
      %do;
         title2 "Some Data May Be Missing";
      %end;
%mend title;

%stats2(cotton,1990)
%stats2(brick,1994)
%stats2(lamb,1995)
```

Here, because the definition of the macro TITLE is outside the definition of the macro STATS2, TITLE is compiled only once, even though STATS2 is called three times. Again, the values of PRODUCT and YEAR are available to TITLE because its call is within the definition of STATS2.

Note: Another reason to define macros separately is because it makes them easier to maintain, each in a separate file. △

Assign Function Results to Macro Variables

It is more efficient to resolve a variable reference than it is to evaluate a function. Therefore, assign the results of frequently used functions to macro variables.

For example, the following macro is inefficient because the length of the macro variable THETEXT must be evaluated at every iteration of the %DO %WHILE statement:

```
/* INEFFICIENT MACRO */
%macro test(thetext);
   %let x=1;
      %do %while (&x > %length(&thetext));
         .
         .
         .
      %end;
%mend test;

%test(Four Score and Seven Years Ago)
```

A more efficient method would be to evaluate the length of THETEXT once and assign that value to another macro variable. Then, use that variable in the %DO %WHILE statement, as in the following program:

```
/* MORE EFFICIENT MACRO */
%macro test2(thetext);
   %let x=1;
   %let length=%length(&thetext);
   %do %while (&x > &length);
      .
      .
      .
   %end;
%mend test2;

%test(Four Score and Seven Years Ago)
```

As another example, suppose you want to use the %SUBSTR function to pull the year out of the value of SYSDATE. Instead of using %SUBSTR repeatedly in your code, assign the value of the %SUBSTR(&SYSDATE, 6) to a macro variable, then use that variable whenever you need the year.

Turn Off System Options When Appropriate

While the debugging system options, such as MPRINT and MLOGIC, are very helpful at times, it is inefficient to run production (debugged) macros with this type of system option set to on. For production macros, run your job with the following settings: NOMLOGIC, NOMPRINT, NOMRECALL, and NOSYMBOLGEN.

Even if your job has no errors, if you run it with these options turned on you incur the overhead that the options require. By turning them off, your program runs more efficiently.

Note: Another approach to deciding when to use MPRINT versus NOMPRINT is to match this option's setting with the setting of the SOURCE option. That is, if your program uses the SOURCE option, it should also use MPRINT. If your program uses NOSOURCE, then run it with NOMPRINT as well. △

Note: If you do not use autocall macros, use the NOMAUTOSOURCE system option. If you do not use stored compiled macros, use the NOMSTORED system option. △

Use the Stored Compiled Macro Facility

The Stored Compiled Macro Facility reduces execution time by allowing macros compiled in a previous SAS job or session to be accessed during subsequent SAS jobs and sessions. Therefore, these macros do not need to be recompiled. Use the Stored Compiled Macro Facility only for production (debugged) macros. It is not efficient to use this facility when developing a macro application.

CAUTION:

Save the source code. Because you cannot re-create the source code for a macro from the compiled code, you should keep a copy of the source code in a safe place, in case the compiled code becomes corrupted for some reason. Having a copy of the source is also necessary if you intend to modify the macro at a later time. △

See Chapter 9 for more information on the Stored Compiled Macro Facility.

Note: The compiled code generated by the Stored Compiled Macro Facility is not portable. If you need to transfer macros to another host environment, you must move the source code and recompile and store it on the new host. △

Centrally Store Autocall Macros

When using the autocall facility, it is most efficient in terms of I/O to store all your autocall macros in one library and append that library name to the beginning of the SASAUTOS system option specification. Of course, you could store the autocall macros in as many libraries as you wish–but each time you call a macro, each library is searched sequentially until the macro is found. Opening and searching only one library reduces the time SAS spends looking for macros.

However, it may make more sense, if you have hundreds of autocall macros, to have them separated into logical divisions according to purpose, levels of production, who supports them, and so on. As usual, you must balance reduced I/O against ease-of-use and ease-of-maintenance.

Although all autocall libraries in the concatenated list are opened and left open during a SAS job or session the first time you call an autocall macro, any library that did not open the first time is tested again each time an autocall macro is used. Therefore, it is extremely inefficient to have invalid pathnames in your SASAUTOS system option specification. (You see no warnings about this wasted effort on the part of SAS, unless no libraries at all will open.)

Other efficiency tips involving the autocall facility include the following:

- Do not store nonmacro code in autocall library files.
- Do not store more than one macro in each autocall library file.

Although these practices are allowed by the SAS System and do work, they contribute significantly to code-maintenance effort and therefore are less efficient.

Other Useful Tips

Some other efficiency techniques you can try include the following:

- Reset macro variables to null if the variables are no longer going to be referenced.
- Use triple ampersands to force an additional scan of macro variables with long values, when appropriate. See “Storing Only One Copy of a Long Macro Variable Value” below for more information.
- Adjust the values of the MSYMTABMAX and MVARSIZE system options to fit your situation. In general, increase the values if disk space is in short supply; decrease the values if memory is in short supply. MSYMTABMAX affects the space available for storing macro variable symbol tables; MVARSIZE affects the space available for storing values of individual macro variables. See Chapter 13, “Macro Language Dictionary,” for a description of these system options.

Storing Only One Copy of a Long Macro Variable Value

Because macro variables can have very long values, the way you store macro variables can affect the efficiency of a program. Indirect references using three ampersands enable you to store fewer copies of a long value.

For example, suppose your program contains long macro variable values that represent sections of SAS programs, as shown here:

```
%let pgm=%str(data flights;
   set schedule;
```

```
   totmiles=sum(of miles1-miles20);
   proc print;
   var flightid totmiles;);
```

Because you want the SAS program to end with a RUN statement, you write the macro CHECK:

```
%macro check(val);
         /* first version */val
    %if %index(&val,%str(run;))=0 %then %str(run;);
    %mend check;
```

First, the macro CHECK generates the program statements contained in the parameter VAL (a macro variable that is defined in the %MACRO statement and passed in from the macro call). Then, the %INDEX function searches the value of VAL for the characters **run;**. (The %STR function causes the semicolon to be treated as text.) If the characters are not present, the %INDEX function returns 0. The %IF condition becomes true, and the macro processor generates a RUN statement.

To use the macro CHECK with the variable PGM, assign the parameter VAL the value of PGM in the macro call:

```
%check(&pgm)
```

As a result, the SAS System sees these statements:

```
data flights;
   set schedule;
   totmiles=sum(of miles1-miles20);

proc print;
   var flightid totmiles;
run;
```

The macro CHECK works properly. However, the macro processor assigns the value of PGM as the value of VAL during the execution of CHECK. Thus, the macro processor must store two long values (the value of PGM and the value of VAL) while CHECK is executing.

To make the program more efficient, write the macro so that it uses the value of PGM rather than copying the value into VAL, as shown here:

```
%macro check2(val);  /* more efficient macro */&&val
   %if %index(&&&val,%str(run;))=0 %then %str(run;);
%mend check2;

%check2(pgm)
```

The macro CHECK2 produces the same result as the macro CHECK:

```
data flights;
   set schedule;
   totmiles=sum(of miles1-miles20);

proc print;
   var flightid totmiles;
run;
```

However, in the macro CHECK2, the value assigned to VAL is simply the name **PGM**, not the value of PGM. The macro processor resolves &&&VAL into &PGM and then into the SAS statements contained in the macro variable PGM. Thus, the long value is stored only once.

Writing Portable Macros

If your code runs in two different environments, you have essentially doubled the worth of your development effort. But portable applications require some planning ahead of time. For more details on any host-specific feature of the SAS System, refer to the SAS documentation for your host environment.

Using Portable SAS Language Functions with %SYSFUNC

You can use the %SYSFUNC macro function to access SAS language functions to perform most host-specific operations, such as opening or deleting a file. You can find more information on these and other functions in the description of %SYSFUNC in Chapter 13.

Using %SYSFUNC to access portable SAS language functions can save you a lot of macro coding (and is therefore not only portable but also more efficient). Table 11.1 on page 136 lists some common host-specific tasks and the functions that perform those tasks.

Table 11.1 Portable SAS Language Functions and Their Uses

Task	SAS Language Function(s)
assign and verify existence of fileref and physical file	FILENAME, FILEREF, PATHNAME
open a file	FOPEN, MOPEN
verify existence of a file	FEXIST, FILEEXIST
list available files	FILEDIALOG
get information about afile	FINFO, FOPTNAME, FOPTNUM
write data to a file	FAPPEND, FWRITE
read from a file	FPOINT, FREAD, FREWIND, FRLEN
close a file	FCLOSE
delete a file	FDELETE
open a directory	DOPEN
return information about a directory	DINFO, DNUM, DOPTNAME, DOPTNUM, DREAD
close a directory	DCLOSE
read a host-specfic option	GETOPTION
interact with the File Data Buffer (FDB)	FCOL, FGET, FNOTE, FPOS, FPUT, FSEP

Task	SAS Language Function(s)
assign and verify librefs	LIBNAME, LIBREF, PATHNAME
get information about executed host environment commands	SYSRC

Note: Of course, you can also use other functions, such as ABS, MAX, and TRANWRD, with %SYSFUNC. A few SAS language functions are not available with %SYSFUNC; see Chapter 13 for more details. △

Example Using %SYSFUNC

The following program deletes the file identified by the fileref MYFILE:

```
%macro testfile(filrf);
   %let rc=%sysfunc(filename(filrf,physical-filename));
   %if &rc = 0 and %sysfunc(fexist(&filrf)) %then
      %let rc=%sysfunc(fdelete(&filrf));
   %let rc=%sysfunc(filename(filrf));
%mend testfile;

%testfile(myfile)
```

Using Portable Automatic Variables with Host-specific Values

The portable automatic macro variables are available under all host environments, but the values are determined by each host. Table 11.2 on page 137 lists the portable macro variables by task. The “Type” column tells you if the variable can be changed (Read/Write) or can only be inspected (Read Only).

Table 11.2 Portable Automatic Macro Variables with Host-specific Results

Task	Automatic Macro Varible	Type
list the name of the current graphics device on DEVICE=.	SYSDEVIC	Read/Write
list of the mode of execution (values are FORE or BACK). Some host environments allow only one mode, FORE.	SYSENV	Read Only
list the name of the currently executing batch job, userid, or process. For example, on UNIX, SYSJOBID is the PID.	SYSJOBID	Read Only
list the last return code generated by your host environment, based on commands executed using the X statement in open code, the X command in display manager, or the %SYSEXEC (or %TSO or %CMS) macaro statements. The default value is 0.	SYSRC	Read/Write

Task	Automatic Macro Varible	Type
list the abbreviation of the host environment you are using.	SYSSCP	Read Only
list a more detailed abbreviation of the host environment you are using.	SYSSCPL	Read Only
retrieve a character string that was passed to the SAS System by the SYSPARM= system option.	SYSPARM	Read/Write

Examples Using SYSSCP and SYSSCPL

The macro DELFILE uses the value of SYSSCP to determine the platform that is running SAS and deletes a file with the TMP file extension (or file type). FILEREF is a macro parameter that contains a filename. Because the filename is host-specific, making it a macro parameter enables the macro to use whatever filename syntax is necessary for the host environment.

```
%macro delfile(fileref);
   %if  /* Unix */sysscp=HP 800 or &sysscp=HP 300 %then
         %do;
            %sysexec rm &fileref..TMP;
         %end;

   %else %if            /* VMS */sysscp=VMS %then
         %do;
            %sysexec %str(DELETE &fileref..TMP;*);
         %end;
   %else %if  /* PC platforms */sysscp=OS2 or &sysscp=WIN %then
         %do;
            %sysexec DEL &fileref..TMP;
         %end;

   %else %if  /* CMS */sysscp=CMS %then
         %do;
            %sysexec ERASE &fileref  TMP A;
         %end;
%mend delfile;
```

Here is a call to the macro DELFILE in a PC environment that deletes a file named C:\SAS\SASUSER\DOC1.TMP:

```
%delfile(c:\sas\sasuser\doc1)
```

In this program, note the use of the portable %SYSEXEC statement to carry out the host-specific operating system commands.

Now, suppose you know your macro application is going to run on some flavor of Microsoft Windows. It could be Windows NT, Windows 95, or Windows 3.1. Although these host environments use similar host environment command syntax, some terminology differs between them, different system options are available, and so on. The SYSSCPL automatic macro variable provides information about the name of the host environment, similar to the SYSSCP automatic macro variable. However, SYSSCPL provides more information and enables you to further tailor your macro code. Here is an example using SYSSCPL.

```
%macro whichwin;  /* Discover which OS is running. */
   %if &sysscpl=WIN_32S %then
      %do;
         %let flavor=32-bit version of Windows;
         %let term=directory;
      %end;

   %else %if &sysscpl=WIN_95 %then
      %do;
         %let flavor=Windows 95;
         %let term=folder;
      %end;

   %else %if &sysscpl=WNT_NT %then
      %do;
         %let flavor=Windows NT;
         %let term=folder;
      %end;

   %else %put *** You must be running a 16-bit version of Windows. ***;
%mend whichwin;
%macro direct;  /* Issue directions to the user. */
   %whichwin
   %put This program is running under &flavor;
   %put Please enter the &term your SAS files are stored in:;
   .
   .   more macro code
   .
%mend direct;
```

Example Using SYSPARM

Suppose the SYSPARM= system option is set to the name of a city. That means the SYSPARM automatic variable is set to the name of that city. You can use that value to subset a data set and generate code specific to that value. Simply by making a small change to the command that invokes SAS (or to the configuration SAS file), your SAS job will perform different tasks.

```
/* Create a data set, based on the value of the */
/* SYSPARM automatic variable. */
/* An example data set name could be MYLIB.BOSTON. */
data mylib.&sysparm;
   set mylib.alltowns;
      /* Use the SYSPARM SAS language function to */
      /* compare the value (city name) */
      /* of SYSPARM to a data set variable. */
   if town=sysparm();
run;
```

When this program executes, you end up with a data set that contains data for only the town you are interested in–and you can change what data set is generated *before* you start your SAS job.

Now suppose you want to further use the value of SYSPARM to control what procedures your job uses. The following macro does just that:

```
%macro select;
   %if %upcase(&sysparm) eq BOSTON %then
      %do;
         proc report ... more SAS code;
            title "Report on &sysparm";
         run;
      %end;

   %if %upcase(&sysparm) eq CHICAGO %then
      %do;
         proc chart ... more SAS code;
            title "Growth Values for &sysparm";
         run;
      %end;
   .
   .  /* more macro code */
   .
%mend select;
```

SYSPARM Details

The value of the SYSPARM automatic macro variable is the same as the value of the SYSPARM= system option, which is equivalent to the return value of the SAS language function SYSPARM. The default value is null. Because you can use the SYSPARM= system option at SAS invocation, you can set the value of the SYSPARM automatic macro variable before your SAS session begins.

SYSRC Details

The value of the SYSRC automatic macro variable contains the last return code generated by your host environment. The code returned is based on commands you execute using the X statement in open code, the X command in display manager, or the %SYSEXEC macro statement (as well as the nonportable %TSO and %CMS macro statements). Use the SYSRC automatic macro variable to test the success or failure of a host environment command.

Note: While supported on all host environments in that it does not generate an error message in the SAS log, the SYSRC automatic macro variable is not useful under all host environments. For example, under some host environments, the value of this variable is always 99, regardless of the success or failure of the host environment command. Check the SAS companion for your host environment to see if the SYSRC automatic macro variable is useful for your host environment. △

Macro Language Elements With System Dependencies

Several macro language elements are host-specific, including the following:

any language element that relies on the sort sequence
: An expression that compares character values uses the sort sequence of the host environment. Examples of such expressions include %DO, %DO %UNTIL, %DO %WHILE, %IF-%THEN, and %EVAL.

  For example, consider the following program:

```
%macro testsort(var);
   %if &var < a %then %put *** &var is less than a ***;
   %else %put *** &var is greater than a ***;
```

```
%mend testsort;

%testsort(1)
     /* Invoke the macro with the number 1 as the parameter. */
```

On EBCDIC systems, such as MVS, CMS, and VSE, this program causes the following to be written to the SAS log:

```
*** 1 is greater than a ***
```

But on ASCII systems (such as OS/2, OpenVMS, UNIX, or Windows), the following is written to the SAS log:

```
*** 1 is less than a ***
```

%KEYDEF
The names and number of function keys are different on each operating system.

MSYMTABMAX=
The MSYMTABMAX system option specifies the maximum amount of memory available to the macro variable symbol tables. If this value is exceeded, the symbol tables are stored in a WORK file on disk.

MVARSIZE=
The MVARSIZE system option specifies the maximum number of bytes allowed for any macro variable stored in memory. If this value is exceeded, the macro variable is stored in a WORK file on disk.

%SCAN and %QSCAN
The default delimiters that the %SCAN and %QSCAN functions use to search for words in a string are different on ASCII and EBCDIC systems. The default delimiters are

ASCII systems	blank . < (+ & ! $ *); ^ – / , % |
EBCDIC systems	blank . < (+ | & ! $ *); ¬ – / , % | ¢

%SYSEXEC, %TSO, and %CMS
The %SYSEXEC, %TSO, and %CMS macro statements enable you to issue an host environment command.

%SYSGET
On some host environments, the %SYSGET function returns the value of host environment variables and symbols.

SYSPARM=
The SYSPARM= system option can supply a value for the SYSPARM automatic macro variable at SAS invocation. It is useful in customizing a production job. For example, to create a title based on a city as part of noninteractive execution, the production program might contain the SYSPARM= system option in the SAS configuration file or the command that invokes SAS. See “SYSPARM Details” on page 140 for an example using the SYSPARM= system option in conjunction with the SYSPARM automatic macro variable.

SASMSTORE=
The SASMSTORE= system option specifies the location of stored compiled macros.

SASAUTOS=
The SASAUTOS= system option specifies the location of autocall macros.

Host-Specific Macro Variables

Some host environments create unique macro variables. Table 11.3 on page 142, Table 11.4 on page 142, and Table 11.5 on page 142 list some commonly used host-specific macro variables. Additional host-specific macro variables may be available in future releases. See your SAS companion for more details.

Table 11.3 Host-specific Macro Variables for MVS

Variable Name	Description
SYS99ERR	SVC99 error reason code
SYS99INF	SVC99 info reason code
SYS99MSG	YSC99 text message corresponding to the SVC error or info reason code
SYS99R15	SVC99 return code
SYSJCTID	value of the JCTUSER field in the JCT control block
SYSJMRID	value of the JMRUSEID field in the JCT control block
SYSUID	the TSO userid associated with the SAS session

Table 11.4 Host-specific Macro Variables for OpenVMS

Variable Name	Description
VMSSASIN	value of the SYSIN= system option or the name of the noninteractive file that was submitted

Table 11.5 Host-specific Macro Variables for the Macintosh

Variable Name	Description
SYSSTNAM	step name
SYSETIME	elapsed time
SYSFRMEM	free memory available
SYSBFTRY	number of attempts to access data in a host buffer used for SAS files
SYSBFHIT	number of successful attempts to access data in a host buffer
SYSXRDS	number of reads to external files
SYSXWRS	number of writes to external files
SYSXBRD	number of bytes read from external files
SYSBWR	number of bytes written to external files
SYSXOPNS	number of opens for external files
SYSSRDS	number of reads to SAS files

Variable Name	Description
SYSSWRS	number of writes to SAS files
SYSSBRD	number of bytes read from SAS files
SYSSBWR	number of bytes written to SAS files
SYSSOPNS	number ofopens for SAS files

Naming Macros and External Files for Use with the Autocall Facility

When naming macros that will be stored in an autocall library, you should consider the following:

- Every host environment has file naming conventions. If the host environment uses file extensions, use .SAS as the extension of your macro files.
- Although SAS names can contain underscores, some host environments do not allow them in the names of external files. Some host environments that disallow underscores do allow the pound sign (#) and may automatically replace the # with _ when the macro is used.
- Some host environments have reserved words, such as CON and NULL. Do not use reserved words when naming autocall macros or external files.
- Some hosts have host-specific autocall macros. Do not define a macro with the same name as these autocall macros.
- Macro catalogs are not portable. Remember to always save your macro source code in a safe place.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc., *SAS Macro Language: Reference, Version 8*, Cary, NC: SAS Institute Inc., 1999. 310 pages.

SAS Macro Language: Reference, Version 8

1-58025-522-1

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

1st printing, October 1999