ABSTRACT

One of the most exciting features of Release 6.06 of the SAS System under VMS* is the full macro facility. As VMS users take full advantage of the macro facility, it is necessary for them to know how to efficiently debug their macro applications. This paper uses a hands-on approach to present debugging techniques and tools that are useful in macro applications running under VMS. The audience for this paper is programmers who are familiar with the SAS language and have a basic understanding of the SAS macro language.

INTRODUCTION

Debugging macro applications is often a frustrating, "guess and by golly" affair, full of trial and error. But it doesn't have to be that way.

One of the ways to decrease the time spent debugging a macro application is to develop it in such a way that problems are minimized. You can accomplish this by modular development and by adding problem-catching code. Then, when problems do arise, you can use several features of the SAS System to pinpoint the cause and location of the problem. This paper presents these and other techniques for debugging SAS macro applications to help you eliminate the guesswork when debugging your macro applications.

A macro application is used throughout the following sections to illustrate the debugging techniques. This example is called GETINFO, an interactive macro application that allows you to display a list of directories or files, and the attributes, such as memory usage, file protection, and so on, for these files. You indicate which file you want information about by using a directory name (including the default, '['), a filename, or a VMS symbol. You can use this macro application, and the debugging techniques you learn as the application is developed, to debug your own SAS macro applications.

Although the debugging techniques this paper presents are geared toward macro applications developed under VMS, many of the techniques can be adjusted for debugging macro applications developed under other host operating systems.

THE DEBUGGING PHILOSOPHY

Debugging macro applications is often difficult because of the flow of control, resolution of variables, and so on are invariably more complex in a macro application. There are several basic debugging techniques that you can apply to your macro applications:

• Develop your macro application one module at a time.

• While working on each module, use SAS system options to monitor the macro's execution.

• Add your own debugging code to each module, using automatic macro variables, macro program statements, and macro functions to pinpoint or prevent problems.

Each of these techniques is discussed in this section.

Modularize Your Macro Application

One of the most common problems in debugging macro applications is caused by developing and executing the entire application all at once, then being overwhelmed by the number of error messages and incorrect results. You don't know where to begin to fix the problems. You can avoid a lot of frustration by developing and debugging each module of your application separately.

Another good idea is to develop the open code first, then add the macro layer around it. (Open code is SAS programming statements that are outside a macro definition.) As you add the macro layer, add the simple macro code first, then the more complex macro code. This last technique is especially important for interactive macro applications such as GETINFO because debugging the use of the %WINDOW and %DISPLAY statements and macro variable resolution can be particularly complex.

This section provides examples of how we applied the techniques of developing each module separately and layering the code as we completed the GETINFO application.

Developing and debugging each module separately

There are 11 modules in the GETINFO application. Some are short, such as the CHK_SPEC macro shown below. This macro checks to see if the user invoked the GETINFO application using the default, a symbol, directory, or filename.

```sas
%macro chk_spec(filecheck);
%global type;

/* If the user specified the default, (); goto BCKEV;
   $if $quote($filecheck)=$str(1) or $quote($filecheck)=$str(2) or
   $quote($filecheck)=$then $do;
   $let type=dir;
   $goto BCKEV;
$end;

%chk_sym;
%* First, see if the user specified a symbol;
   $let spec=symget($filecheck);
   $if $spec="" $then $do;
   $put The value of the symbol is: $spec;
   $let filecheck=$spec;
   $end;
$end_sym;

%chk_sym;
/* Now, determine if the user specified a file or a directory name;
   $let len=length($filecheck);
   $if $len=$str(1) $then $do;
   $let dir=brkt;
   $if $quote($brkt)=$str(1) or
   $quote($brkt)=$str(2) $then $let type=dir;
   $else $let type=file;
$end_sym: $put type=type;
@end chk_spec;
```

When we first ran this module, we discovered two bugs: the %SUBSTR macro function was misspelled as %DINDYT; also, we didn't use the %QUOTE and %STR macro functions when checking the value of BRKT. This second error caused the SAS System to
treat the character > as the greater than operator instead of just a character.

When we ran the whole application, including the erroneous CHK_SPEC macro, the log messages were obscure:

WARNING: Apparent invocation of macro GETINFO not resolved.
ERROR: Required operator not found in expression: (tbrln ^ II
ERROR: The macro will stop executing.

These messages do not point to the problem; furthermore, they hide the source of the error.

In contrast to the obscure warning and error messages generated when we ran all of the application, the messages generated when we ran this module by itself made the errors easy to fix. Below is an excerpt from the log messages generated when we executed the CHK_SPEC macro (with errors):

WARNING: Apparent invocation of macro DIR/FULL not resolved.
ERROR: Required operator not found in expressions: (forkt = 1), or (forkt <>)
ERROR: The macro will stop executing.

The warning messages point directly to the misspelled macro function, and the error message about operators gives a clue to the other bug. It would have taken us much longer to find and fix the errors if we had not debugged the CHK_SPEC macro separately.

Developing each layer
Another way to modularize the development of your macro application is to add the macro layer at the very end. For example, while developing the GETINFO application, we concentrated first on simply getting the information generated by the DIR/FULL command into a SAS data set. When we could do that without errors, we then placed the code into a macro. Finally, we wrote another macro to create macro variables from the SAS data set variables. If we had tried to create a macro right away that performed all these tasks, the debugging time would have increased greatly.

Another advantage of creating the SAS data set first and then the macro variables is that the SAS data set variables provide material for many macro variables. For example, although the SAS data set has just one variable for file size, we created two macro variables from this one data set variable (number of blocks allocated and number of blocks used). This approach gives you more flexibility than creating only the macro variables.

Even adding the macro layer of code can be done in stages. For the GETINFO application, adding the %WINDOW and %DISPLAY code was the last thing we did. We wanted to be sure each task of the application worked correctly (tasks such as reading in the DIR/FULL command results, creating the SAS data set, and creating the macro variables) before we began to add the interactive portions of code. This cut down on our debugging time because debugging window macro application code can be time consuming; you have to consider window layout, the complexity of the macro variable resolution, and so on.

If you are writing a macro that has several logical divisions, code each piece separately, then combine them into the larger macro when each piece works by itself. For example, when we developed the CHK_SEL macro, which validates the user’s selection of a file, we tested our use of the autocall macros %DATATYP and %VERIFY by creating a small macro named TEST. When the TEST macro ran without errors, we added that code to the original CHK_SEL macro.

The concept of modularization during development can be reversed and applied to debugging as well. If you have a fairly complex macro that contains an error, break that macro into several smaller pieces, and test each one. If you are creating a SAS data set with a macro and encounter errors, pare down the data set to just a few variables and observations. This approach helps you isolate problems and makes the debugging phase go much faster.

Each of these modularization techniques saved us a lot of time as we developed and debugged the GETINFO application. However, we occasionally ran across an error that was a mystery. In those cases, we used SAS system options to help us locate the problem.

Some of these system options are discussed in the next section.

Use Debugging System Options to Your Advantage
The SAS language provides several system options to help you debug your SAS macro applications:

- MACRO controls whether the SAS macro facility is available.
- MERROR controls whether the SAS macro language compiler issues a warning message if the macro processor cannot match a macro-like name to an appropriate macro keyword.
- SERROR controls whether warning messages are issued when the SAS System encounters a macro variable reference that cannot be matched with an appropriate macro variable.
- MLOGIC displays information about the execution of a macro.
- MPRINT displays statements the SAS compiler receives.
- SYMBOLGEN displays text resulting from macro variable resolution.

MACRO, MERROR, and SERROR should always be set. If they are not set, use the OPTIONS statement or OPTIONS window to turn on these options. The last three options, MLOGIC, MPRINT, and SYMBOLGEN, are useful tools during the debugging phase of a macro application. Knowing when and how to use each of these options can help you debug your macro application much faster.

MLOGIC, MPRINT, and SYMBOLGEN provide different results. Below are examples of using these system options with an error we found as we developed the GETINFO application. The macro GETINFO calls the macro CRT_DS, which creates a SAS data set from the text file containing the file attribute information returned from the DIR/FULL command. Initially, the CRT_DS macro contained some erroneous code:

```
options ls=80 ps=60;
filename fileinf 'Hfileinfo';
filename filecvt 'Hfilecvt';
data convert;
  infile fileinf;
  file filecvt recfm=fs recpos=f lrecl=80;
  input;
  put _inf ile_;
run;
```

The concept of modularization during development can be reversed and applied to debugging as well. If you have a fairly complex macro that contains an error, break that macro into several smaller pieces, and test each one. If you are creating a SAS data set with a macro and encounter errors, pare down the data set to just a few variables and observations. This approach helps you isolate problems and makes the debugging phase go much faster.

Each of these modularization techniques saved us a lot of time as we developed and debugged the GETINFO application. However, we occasionally ran across an error that was a mystery. In those cases, we used SAS system options to help us locate the problem.

Some of these system options are discussed in the next section.

Use Debugging System Options to Your Advantage
The SAS language provides several system options to help you debug your SAS macro applications:

- MACRO controls whether the SAS macro facility is available.
- MERROR controls whether the SAS macro language compiler issues a warning message if the macro processor cannot match a macro-like name to an appropriate macro keyword.
- SERROR controls whether warning messages are issued when the SAS System encounters a macro variable reference that cannot be matched with an appropriate macro variable.
- MLOGIC displays information about the execution of a macro.
- MPRINT displays statements the SAS compiler receives.
- SYMBOLGEN displays text resulting from macro variable resolution.

MACRO, MERROR, and SERROR should always be set. If they are not set, use the OPTIONS statement or OPTIONS window to turn on these options. The last three options, MLOGIC, MPRINT, and SYMBOLGEN, are useful tools during the debugging phase of a macro application. Knowing when and how to use each of these options can help you debug your macro application much faster.

MLOGIC, MPRINT, and SYMBOLGEN provide different results. Below are examples of using these system options with an error we found as we developed the GETINFO application. The macro GETINFO calls the macro CRT_DS, which creates a SAS data set from the text file containing the file attribute information returned from the DIR/FULL command. Initially, the CRT_DS macro contained some erroneous code:

```
options ls=80 ps=60;
filename fileinf 'Hfileinfo';
filename filecvt 'Hfilecvt';
data convert;
  infile fileinf;
  file filecvt recfm=fs recpos=f lrecl=80;
  input;
  put _inf ile_;
run;
```
Using no options
When we invoked GETINFO with all the macro debugging system options set to off, the log messages were as follows:

```
131 options nomlogic nomprint nosymbolgen;
133 startinfo();
ERROR: Invalid physical name.
ERROR: Error in the LIBNAME or FILENAME statement.
ERROR: Invalid physical name.
ERROR: Error in the LIBNAME or FILENAME statement.
ERROR: No logical assign for filename FILEINF.
NOTE: The file FILECVT is:
      file=SUGI$[SASDEMO.SAS]FILECVT.DAT
```

The log messages in this case are not all that helpful. You cannot even tell what macro was executing when the errors occurred.

Using the MLOGIC option
The MLOGIC option monitors macro execution, including information such as when the macro begins and ends, all results of conditional branching, whether a macro variable is local or global, and the values of parameters passed to a macro. When we invoked the GETINFO macro with the MLOGIC option, the log messages were as follows:

```
407 options mlogic nomprint nosymbolgen;
MLOGIC(GETINFO): IF condition &sysrc ne 1 is FALSE
MLOGIC(GETINFO): Beginning execution.
ERROR: Invalid physical name.
ERROR: Error in the LIBNAME or FILENAME statement.
ERROR: Invalid physical name.
ERROR: Error in the LIBNAME or FILENAME statement.
ERROR: No logical assign for filename FILEINF.
NOTE: The file FILECVT is:
      file=SUGI$[SASDEMO.SAS]FILECVT.DAT
```

From these log messages we could see that the problem occurred in the CRT_DS macro. But we still couldn’t see which statements were causing the error. Because our error was not in the area of program logic, the MLOGIC option was not the best choice for discovering the problem.

Using the MPRINT option
Our next step was to try the MPRINT option. The MPRINT option displays the statements that the SAS compiler receives. When we invoked GETINFO again, we got the following log messages:

```
411 options mlogic mprint nosymbolgen;
414 startinfo();
MPRINT(CRT...DS): OPTIONS IS=80 PS=60;
MPRINT(CRT...DS): FILENAME FILEINF 'fileinfo';
ERROR: Invalid physical name.
ERROR: Error in the LIBNAME or FILENAME statement.
MPRINT(CRT...DS): FILENAME FILECVT 'filecvt';
ERROR: Invalid physical name.
ERROR: Error in the LIBNAME or FILENAME statement.
MPRINT(CRT...DS): DATA CONVERT;
MPRINT(CRT...DS): FILE FILECVT RECFM=F LRECL=80;
MPRINT(CRT...DS): INPUT;
ERROR: Invalid physical name.
ERROR: Error in the LIBNAME or FILENAME statement.
ERROR: No logical assign for filename FILEINF.
NOTE: The file FILECVT is:
      file=SUGI$[SASDEMO.SAS]FILECVT.DAT
```

These messages point directly to the source of the problem: the physical file specifications in the FILENAME statements are incorrect. Macro variables won’t resolve when they are enclosed in single quotes. With this information, we could fix the code. Here are the corrected statements:

```
filename fileinfo "fileinfo";
filename filecvt "filecvt";
```

Another potential problem with the original FILENAME statements was our use of a double ampersand. Although in this case && was equivalent to &, confusion about how many ampersands are necessary is common. The general rule is that && resolves to 8. You can avoid potential errors in macro variable resolution by familiarizing yourself with macro variable resolution rules before starting to develop your macro application.

Using the SYMBOLGEN option
The SYMBOLGEN option shows you what macro variables resolve to. This option is especially useful to check the value of a macro variable during macro execution. The log messages tell you what each macro variable’s value is, and you can readily see if the value is not what you expected. When we invoked the GETINFO macro with the SYMBOLGEN option, the log messages were as follows:

```
250 options nomlogic nomprint symbolgen;
251 startinfo();
SYMBOLGEN: Macro variable FILECHK resolves to |;
SYMBOLGEN: Macro variable FILEINFO resolves to |;
SYMBOLGEN: Macro variable SYRC resolves to 1;
SYMBOLGEN: Macro variable TYPE resolves to dir
type=dir
SYMBOLGEN: Macro variable TYPE resolves to dir
type=dir
SYMBOLGEN: Macro variable FILECVT resolves to dir
type=dir
SYMBOLGEN: Macro variable FILECVT resolves to dir
type=dir
SYMBOLGEN: Macro variable SYSRC resolves to 1
SYMBOLGEN: Macro variable FILEINFO resolves to
sugiS: [SASDEMO.DAT]file.data
SYMBOLGEN: Macro variable FILEINFO resolves to
sugiS: [SASDEMO.SAS]FILECVT.DAT
ERROR: Invalid physical name.
ERROR: Error in the LIBNAME or FILENAME statement.
ERROR: Invalid physical name.
ERROR: Error in the LIBNAME or FILENAME statement.
ERROR: No logical assign for filename FILEINF.
NOTE: The file FILECVT is:
      file=SUGI$[SASDEMO.SAS]FILECVT.DAT
NOTE: 0 records were written to the file FILECVT.
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.CONVERT may be incomplete. When this step was stopped there were 0 observations and 0 variables.
NOTE: The infile FILECVT is:
      file=SUGI$[SASDEMO.SAS]FILECVT.DAT
NOTE: 0 records were read from the infile FILECVT.
NOTE: The data set SUGI$LIB.TEMP has 0 observations and 18 variables.
```

Because our error (using single quotes instead of double quotes) was not caused by an incorrect macro variable value, the SYMBOLGEN option did not provide too much more useful information. However, it does trace the resolution of all the variables in the macro, which could be valuable information at a later date in the debugging cycle.

Deciding which option to use
You should not use more options than you need; too much information may actually obscure the solution to the problem. Each option is useful for a particular type of error, at a particular place in macro execution, as shown in Figure 1.
I

SAS Program (Input Stack)

SAS Wordscancer

Yes

Macro

MLLOGIC

Yes

Symbolic

Substitution

SAS Compiler

Figure 1 The Role of System Options during Macro Execution

Use MLOGIC to trace macro execution. The MLOGIC option is especially useful for monitoring macro execution. The information generated by this option includes when a macro begins and ends and full tracing of branch execution (for example, %IF-%THEN-%ELSE code). It also tells you which macro variables are local and global, and displays the values of parameters passed to a macro. If you suspect your errors are caused by faulty logic or a section of code executing at the wrong time, use the MLOGIC option to trace what is really happening.

Use MPRINT to see generated statements. The MPRINT option is especially useful for seeing the statements the SAS compiler receives. This option allows you to see in the SAS log the actual generated statements. If you suspect your errors are caused by incorrect SAS programming statements, use the MPRINT option.

Use SYMBOLGEN to see resolved values of macro variables. The SYMBOLGEN option is especially useful for seeing the resolved values of macro variables. If you suspect your errors are caused by a macro variable resolving to the wrong value, use the SYMBOLGEN option to display the values of the macro variables as your macro executes.

Add Your Own Error Diagnostics

There are many components of the macro facility you can use to add error diagnostic code to your macro application. This paper discusses only the most common components. Each macro application is different, and you may find other techniques that are useful as you develop your macro applications.

Some of the most common techniques for adding diagnostic code to a macro application include:

- using automatic macro variables, such as SYSRC, with the %SYSEXEC and X statements
- using macro program statements, such as %PUT
- using macro functions, such as %SYSGET and the quoting functions
- using autocalc macros supplied by SAS Institute, such as DATATYPE and VERIFY.

Using automatic macro variables

Automatic macro variables are very useful in developing macro applications that perform error checking. The automatic macro variable we found most useful is SYSRC. This automatic macro variable contains the operating system return code for statements submitted to the operating system (with a %SYSEXEC or X statement or the X command, for example). We used this variable extensively throughout the GETINFO application. Here is an example of how we used the SYSRC automatic macro variable:

```sas
%sysexec dir:z:disp2-display=60/quit=+filesinfo igetfile;
@if systerm eq 1 then
  $do;
  put igetfile saysrc;
  put lstx();
  put igetfile does not exist....;
  put lstx();
  $end;
```

In this code, we check the value of SYSRC. A successful DCL command sets SYSRC to 1. If SYSRC is not 1, then the command failed, and we print a message to that effect.

There are many automatic macro variables; some of the ones you may find most useful under VMS include:

- SYSEDEVIC gives the current graphics device. This macro variable is often used in macro applications requiring a specific graphics device.
- SYSSCP gives the current operating system. This macro variable is often used in macro applications running under several operating systems.

Using macro program statements

You may want to use macro program statements in your diagnostic code. One very useful statement is the %PUT statement. You can use this statement in combination with automatic macro variables and other macro variables to send diagnostic messages to the log. While we were developing the CHK_SPEC macro, we used the %PUT statement extensively to check the values of macro variables such as SPEC (the macro variable containing the resolved VMS symbol), FILECHK (to see what filename, symbol, or directory the user supplied), and BRKT (to see what the last character of &FILECHK was).

We also used the %PUT statement when we began to add the interactive layer to our macro application, as in this example:

```sas
$do i=1sto 10obs;
  $if %eval(i)<1 then
    $do;
    put dir"="diri+"diri";
    $end;
    put file="filei";
    put outst="outst1";
    put style="stylei";
    put warn="warni";
    put file="filei";
    $end;
```

Using macro functions

Macro functions can also be useful tools. One such function, %SYSGET, is especially useful under VMS. This function returns the value of a VMS symbol. So, for example, if a user supplied the...
VMS symbol MYSYM when invoking the GETINFO application, we could use the following code to catch undefined symbols:

```sas
%let symbol=%sysget(MYSYM);
%if %sysget(symbol) %then
  %do;
  %put there is no such symbol;
  %end;
%else
  %do;
  %put The value of symbol is %sysget(symbol);
  %end;
```

Another set of macro functions that are useful when you are debugging a macro application are the quoting functions, such as %QUOTE and %STR. (%QUOTE quotes the result of a macro variable resolution, and %STR removes the significance of special characters in a text string.) Errors or oversights in quoting strings and macro variable values are a common source of macro bugs. For example, the first time we tried our CHL_SPEC macro, discussed earlier in Developing and debugging each module separately, we forgot to quote the character `, so the SAS System treated it as a mathematical operator instead of plain text. We had to use the %QUOTE function on the macro variable value and the %STR function for the ` character to get the text comparison to work correctly.

Another problem we had with quotes was using a single quote in a macro comment statement. This caused a string greater than 200 characters error. It is never a good idea to use contractions such as didn’t in comment statements.

Review the macro quoting functions before you begin to develop your macro application, and watch out for the following:

- unbalanced quotes and characters that are also used as mathematical operators
- mnemonic operators such as NE, OR, and AND
- delimiters
- special macro characters such as % and &.

Errors in quoting can cause strange, misleading error messages, so avoiding such errors in the first place saves you time.

Using autocall macros supplied by SAS Institute

SAS Institute supplies several macros in the default autocall library that can help you write diagnostic code. Below are some of the most useful ones:

- **CMPRES** returns the unquoted argument after compressing multiple blanks to single blanks and deleting any leading blanks and trailing blanks.
- **DATATYP** tells you if the argument passed is numeric or character.
- **LEFT** returns the argument without any leading blanks.
- **QMPRES** is the same as CMPRES, only it returns a quoted argument.
- **QLEFT** is the same as LEFT, only it returns a quoted argument.
- **QTRIM** is the same as TRIM (see below), only it returns a quoted argument.
- **TRIM** returns the unquoted argument without trailing blanks.
- **VERIFY** returns the position of the first character in the argument that is not in the target value.

These functions are available if the MAUTOSOURCE system option is set. Also, the SASAUTOS= system option must point to the location of the default autocall library. You can determine the default value of the SASAUTOS= option with the OPTIONS procedure. See your SAS Software Representative for more information on the location of the default autocall library.

We used the DATATYP and VERIFY autocall functions in the interactive layer of the GETINFO application to check the selection the user typed next to a file. Here is a portion of the code:

```sas
%do i=1 %to n:
  %if (%qleft(%sysget(i))) %then
    %do;
    %put %sysget(i);
    %end;
  %else
    %do;
    %put The value of %sysget(i) is %sysget(i);
    %end;
%end;
```

This code uses the DATATYP autocall macro to check the user's selection to be sure it is a character. The VERIFY macro compares the entered selection to the character string X. If the entered selection passes these tests, the file attributes information is then output to the screen.

Under VMS, these autocall macros are stored in the autocall library referenced by the VMS logical name SASAUTOS. To see how an autocall macro is coded, you can type the following command at the VMS prompt:

```sas
S ASAUTOS:macro-name.SAS
```

For more information on using these and other autocall macros supplied by SAS Institute, see the SAS Guide to Macro Processing, Version 6, Second Edition.

**EFFICIENT MACRO APPLICATION DEVELOPMENT**

Much of the frustration commonly involved in debugging macro applications can be avoided through an efficient development plan. Although everyone has his or her own way of organizing files and developing macro applications, there are several rules of thumb that you should follow to make your macro application development more efficient:

Use an appropriate compilation method. Use the %INCLUDE statement to compile macros during the development and debugging phase of a macro application, not the autocall facility. The autocall facility (discussed in the next section) is not efficient in the debugging phase of macro application development. There are several reasons for this. One reason is that when you use the autocall facility with a "buggy" macro application, you constantly need to reset the MAUTOSOURCE and MRECALL system options. This can be tedious.

Also, the development phase focuses on separate modules of a macro application rather than having all utility macros available at once. If you use the %INCLUDE statement, you can selectively access and compile just those macros you need. If you use the autocall facility and have a long list of concatenated macro libraries, accessing all these libraries can be slower than accessing only a few macros from each library with the %INCLUDE statement.

The following suggestions are for the %INCLUDE method of compiling macros, not for the autocall facility.

Store source code separately from data files. You should store all your source code in one place (or in a set number of places) and...
keep it separate from your data files. For example, for the GETINFO application, our source code is stored in SUGI$[SASDEMO.SAS], while our data files are stored in SUGI$[SASDEMO.DAT]. The advantage of this technique is that you always know where to find a file, and if you have several files of the same filename but different file types (such as FILEINFO.DAT and FILEINFO.SAS), you are less likely to delete, rename, or move the wrong file.

Store macro code separately from other SAS code. You should store all your macro definitions in a separate place from your other SAS code. This helps reduce confusion over filenames, as both macro files and other SAS code files have a file type of .SAS. Also, when you migrate to using the autocall facility, there will be fewer files for the facility to keep track of.

Use the macro name as the filename. This is not a requirement of the SAS language; you can name your files and macros anything you want. But your macro application should be self-documenting as possible. You will spend far less time thinking “Now what did I name that macro (or file)?” when you are finding and fixing bugs if you give the macros and files the same name. It also helps to give the macros and files informational names that remind you of their purpose.

If you later decide to use the autocall facility, the macro name must be the same as the filename, so it saves time if you plan for this at the start of your application development.

Store only one macro definition in each file. As far as the SAS System is concerned, you can store all your macro definitions in one file. However, if you want to use one of the macros stored in the file, you must compile the entire file with the %INCLUDE statement. This compilation step can take a long time. It is more efficient to store only one macro definition in a file because you only have to compile the macros you want to use, and recompilation is faster when you correct a bug.

Autocall Facility
A final technique to modularize your macro application is to adjust how you tell the SAS System where to find the macros you are using. When you are first developing the application, it is best to use a %INCLUDE statement for each macro you want to compile. You could put all these statements in a start-up file. Below is an example of our initial start-up file:

```sas
libname sugi 'sugi$:[sasdemo.dat]';
libname sugi 'sugi$:[sasdemo.sas]';
%include sugi([windows]);
%include sugi([welcome]);
%include sugi([ct..spec]);
%include sugi([getinfo]);
%include sugi([ch..solv]);
%include sugi([ct..fs]);
%include sugi([ct..user]);
%include sugi([dir..cla]);
%include sugi([ch..sel]);
%include sugi([ct..fvars]);
```

As you can see, we used quite a few %INCLUDE statements, and the compilation step takes considerable time. However, this technique ensures each macro is compiled separately and makes error tracing much easier.

When your macro application is completely bug-free, however, it is more efficient to use the autocall facility to tell the SAS System where to find your macros. For example, when we were sure all our macros worked correctly, we simply invoked the SAS System with the following command:

```
$ SAS/SASAUTOS = 'sugi$:[sasdemo.sas]', SAsAUTOS
```

This tells the SAS System that we have macros stored in SUGI$[SASDEMO.SAS] and that we want to concatenate this library with the macro library supplied by SAS Institute, which is pointed to by the logical name SAsAUTOS. Then, during our SAS session, we could simply invoke the GETINFO application; the autocall facility takes care of all compilation.

The techniques for efficient macro application development discussed previously also apply when you move into the production phase of the application and begin to use the autocall facility. Below are some other suggested efficiency considerations for using the autocall facility:

- Use filerefs to specify autocall libraries when possible. Because specifying physical file names as autocall libraries with the SASAUTOS= system option can result in repeated attempts at opening unavailable files with no error messages.
- Be aware of the efficiency trade-off between using many autocall libraries with fewer macros per library versus using fewer autocall libraries with more macros per library. Having fewer libraries means less time is spent searching libraries, but you have less flexibility in storing logical groups of macros.

**SUMMARY**

One of the rewards of debugging is the great feeling of satisfaction when your macro applications run correctly — IT WORKS!!! We hope the techniques this paper presents help you reach that satisfaction more quickly as you develop your macro applications under VMS. These techniques include:

- modularizing your macro application and development process
- using SAS system options efficiently
- adding your own debugging code to pinpoint problems
- using the autocall facility to improve macro application efficiency only after the debugging phase of development is finished.

It may also be helpful to familiarize yourself with macro variable resolution rules and macro quoting functions before you begin to develop and debug a new macro application.

As you develop and debug your own macro applications, you can adjust and add to the debugging techniques presented in this paper, quickly and efficiently developing bug-free macros.

**SOURCES OF MORE INFORMATION**

- SAS Macro Language Course Notes (for Version 6, available July 1, 1990)
- SAS Companion to the VMS Environment, Version 6, First Edition

830
ACKNOWLEDGEMENTS

We would like to acknowledge Ginny Dineley and Yvonne Selby for their help in developing the GETINFO application. We would also like to thank Tom Cole, Anne Corrigan, Ginny Dineley, Darylene Hecht, Wayne Hester, Susan O'Connor, Dee Stribling, Jana Van Wyk, Keith Wagner, and Harriet Watts for their help in reviewing this paper.

SAS is a registered trademark of SAS Institute Inc., Cary, NC.

VMS is a trademark of Digital Equipment Corporation.