Paper SY15

Rules for Tools – The SAS Utility Primer

Frank DiIorio, CodeCrafters, Inc., Chapel Hill NC

INTRODUCTION

Let's start with the premise that good programmers are lazy by nature. They want to use tools such as formats and ODS for execution-time efficiency or to pretty-up our output, functions to perform calculations, and so on. Another hallmark of a good programmer is a keen eye for pattern recognition. Rather than rewrite basically the same program over and over, they identify similarities and parameterize the program, making it into a general-purpose program, a "utility."

This paper steps through the life cycle of a simple utility. It starts with "naïve" code that doesn't exploit program similarities, then illustrates how a general-purpose utility may be developed. It ends with the initial program becoming a call to a simple, powerful routine in a macro library. The transition from simple, brute-force programming into a compact, general-purpose utility isn't a random event. The last sections of the paper present a set of design principles for utilities.

Although we focus on Base SAS in Version 9.0, the principles and techniques are readily extended across SAS versions and products. The reader will come away from this paper with an appreciation of both the process and the tool set required to build generalized programs.

BASICS

Let's start by defining what, exactly, a utility program is. We'll look at two definitions. First, we have the more stringent:

> *A utility is a generalized program used only by other programs.*

This is the classic definition. It encompasses programs that perform low-level tasks never directly seen by an end-user and invoked only by the programmer. These utilities verify the existence of a file, parse a string and count the number of words in it, and the like.

A more encompassing definition follows:

> *A utility is a generalized "helper" program that assists in the production of a final product.*

This is a broader definition. It includes items from the first definition, and extends its reach to applications that may be used by both programmers and end-users. Thus in addition to the low-level functionality of the first definition, it includes programs that peform diagnostic and reference tasks such as printing records from data sets, creating HTML summarizing library contents, and identifying data inconsistencies. It is important to note that it is likely and desirable that these higher-level utilities would, in turn, invoke lower-level utilities to get the job done. This second, broader definition of a utility is the one that will be used throughout this paper.

Another basic feature of a utility is the breadth of tools that it employs. In order to be generalized and functional, all but the simplest of utilities will require some or all of the following Base SAS tools:

- **Dictionary Tables.** These are tables of metadata about the SAS environment. They make access to information about system options, tables, titles and footnotes, macro variables, and other features either easier or possible. Knowledge of their contents and quirks is essential for many applications.
- **SQL.** The ability of PROC SQL to perform a vast amount of data handling in a compact manner makes it an essential tool. It also provides the most efficient way to access the contents of the dictionary tables.
- **The Macro Language.** Life in the world of generalized code is never simple. Error conditions may arise, unpredictable numbers of task repetitions may be required, and so on. Programming for such uncertainty is impossible without the functionality of the macro language. Its conditional execution of some, all, or multiple program statements makes it a prerequisite for building utilities. Note that due to its pervasive nature in utility programming, we use "macro" and "utility" nearly interchangeably in this paper.

The finer points of these topics are beyond the scope of this paper. Many SAS publications and excellent papers in SUGI and regional user group publications that address them at an appropriate length and detail.

AN EXAMPLE

With the broader definition and tool set in mind, let's look at a simple task and see how it might evolve from a simplistic, hard-to-manage standalone program to a generalized utility that exercises a variety of Base SAS tools. The final product is a "helper" program consistent with the second definition described above. As we go through the iterations of the example, we identify each version's shortcomings and introduce new tools in the tool set to make life easier (or, at minimum, more of a technical challenge!) Keep in mind that this progression is here for pedagogical purposes, following a particularly torturous path to a good solution. An experienced programmer will often produce what we see in "Version 5" (or better) without any intermediate steps.

The example is deceptively simple: print 10 records from each data set in a library. Let's watch it progress from clunky to elegant.

VERSION 1: BRUTE FORCE

In the first go-round, we write the PROC PRINT statements for the first data set, then duplicate and modify them for the other data sets in the library.

```
libname in "C:\Project\Data\Archive";
proc print data=in.air(obs=10);
     title "First 10 obs from AIR";
run;
proc print data=in.citiday(obs=10);
     title "First 10 obs from CITIDAY";
run;
proc print data=in.class(obs=10);
     title "First 10 obs from CLASS";
run;
proc print data=in.gnp(obs=10);
     title "First 10 obs from GNP";
run;
proc print data=in.shoes(obs=10);
     title "First 10 obs from SHOES";
run;
proc print data=in.steel(obs=10);
     title "First 10 obs from STEEL";
run;
proc print data=in.workers(obs=10);
     title "First 10 obs from WORKERS";
run;
```

The program brings to mind the Volkswagen slogan of the 1960's: "it's ugly, but it gets you there." It's tedious and error-prone. Consider the possibility of changing the name of the DATA option but forgetting to do so in the title. Consider, too, the dubious prospect of coding for a library with dozens of data sets. There must be a better way.

VERSION 2: IDENTIFY THE REPETITION

The program below begins to hint at good design. We identify and take advantage of the repetition in the original program, but can do so only by using one of the basic tools, the SAS macro language. We still have to invoke the macro for each data set in the library, but at least we have the comfort of knowing that changes are centralized. If we have to make a change to the program, it will be made only once, in the macro, rather than "n" times, once for each data set.

```
libname in "C:\Project\Data\Archive";
%macro PrintIt(data=);
       proc print data=in.&data.(obs=10);
            title "First 10 obs from %upcase(&data.)";
       run;
%mend;
%PrintIt(data=air)
%PrintIt(data=citiday)
```

```
%PrintIt(data=class)
%PrintIt(data=gnp)
%PrintIt(data=shoes)
%PrintIt(data=steel)
%PrintIt(data=workers)
```

VERSION 3: USE THE MACRO LANGUAGE

The solution in Version 2 was a bare-bones macro, containing no control structures or function calls. We can improve on the previous program by using a simple DO loop and some simple assignment statements. The result is only one call to the macro. This comes at the minor expense of some extra coding within the macro. The DO loop is executed for each data set. Once the SCAN function returns a null value, execution terminates.

```
libname in "C:\Project\Data\Archive";
%macro PrintIt;
       %do i = 1 %to 100;
           %let data = %scan(air citiday class gnp shoes steel workers, &i.);
           %if &data. = %then %goto %bottom;
           proc print data=in.&data.(obs=10);
                title "First 10 obs from %upcase(&data.)";
           run;
       %end;
       %bottom: ;
%mend;
%PrintIt
```

While cleaner than the earlier examples, this version still burdens the user with the need for *a priori* knowledge of the data set names. What happens when a new data set appears in the library? What happens when an existing data set is deleted? The result is an incomplete listing and an error, respectively. Clearly, it would be helpful to let the macro figure out what entities are in the library.

VERSION 4: USE DICTIONARY TABLES

In Version 4, we extend the usefulness of the macro by letting it build the list of data set names in the library. This requires use of SQL and the dictionary tables. The SQL statements create two macro variables, N and NAMES. N is the count of the number of observations in the TABLES table that had a LIBNAME value of IN. NAMES is a blank-delimited list of these data set names. Collectively, the variables tell us that NAMES has N items, or tokens, each with the name of a data set. Our knowledge of the characteristics of the TABLES table also tells us that the names are upper-cased and that they are accessed in ascending alphabetical order.

```
libname in "C:\Project\Data\Archive";
%macro PrintIt;
       proc sql noprint;
            select count(*), memname
            into   :n,       :names separated by ' '
            from dictionary.tables
            where libname = 'IN'
            ;
       quit;
       %do i = 1 %to &n.;
           %let data = %scan(&names., &i.);
           proc print data=in.&data.(obs=10);
                title "First 10 obs from %upcase(&data.)";
           run;
       %end;
       %bottom: ;
%mend;
%PrintIt
```

This is almost enough to make us content. We simply have to run the macro in order to print from each data set. If data set names change, or if data sets are added or deleted the SQL code ensures the list is complete. The

macro is so useful that we think it would be nice to share it, to let any one point to a library and let PRINTIT go to work. The final version of the program addresses this need.

VERSION 5: USE A MACRO LIBRARY

To be truly generalized and accessible, we need to do two things. First, hard-coded references must be either removed or turned into parameters. The rewritten macro adds a parameter for the LIBNAME and, anticipating end-user input, adds a parameter to control the number of observations that will be printed. Both parameter values are reflected in the title of the output.

```
%macro PrintIt(libname=, obs=10);
       proc sql noprint;
            select count(*), memname
            into   :n,       :names separated by ' '
            from dictionary.tables
            where libname = "%upcase(&libname.)"
           ;
       quit;
       %do i = 1 %to &n.;
           %let data = %scan(&names., &i.);
           proc print data=in.&data.(obs=&obs.);
                title "First &obs. obs from LIBNAME &libname, data set 
%upcase(&data.)";
           run;
       %end;
%mend;
```

The second task is to store the macro in an autocall library. Assume the program shown above is stored in a file named `c:\programs\macros\PrintIt.sas`. The OPTIONS statement in the following program lets the macro be used by any program with access to drive H:.

```
options mrecall mautosource sasautos=('H:\Programs\Macros\', sasautos);
libname in "H:\Project\Data\Archive";
%PrintIt(libname=in, obs=5)
```

VERSION "NEXT": TWEAKS AND MORE TWEAKS …

Thus we see that even something as prosaic as printing from a data set exercises a variety of Base SAS tools. By progressing from a boring series of PRINT statements to a parameterized autocall macro, we have accomplished the following: identified items that were repeated and therefore eligible for parameterizing; used SQL and dictionary tables to dynamically generate lists and counts; and employed the macro language to remove hard-coded values and provide control structures. Even though Version 5 is powerful and generalized, it could still benefit from a few modifications:

- **Error checks.** We accept on faith that there are, in fact, data sets in the library. It would be prudent to check for an empty library. If this is the case, the macro should write a message to the output destination. Even more basic, the macro should verify that the LIBNAME actually exists.
- **Inclusion / exclusion.** The user may not want to print all data sets. Adding INCLUDE and/or EXCLUDE parameters would be helpful. If a data set identified in this list is not in the library, the macro should print a message either to the SAS Log or the output destination.
- **Empty data sets.** If a data set exists but has no observations, the utility could write a message to the output destination. This requires knowledge of PRINT's default behavior, which is simply to say "sorry, nothing to print," in the Log. It also requires a bit of extra coding effort, but the reward is that the user is kept informed.

THE BIG PICTURE

Several general points can be gleaned from the development sequence shown above:

- **Identify the need**. This utility, like all others, is motivated by the impulse to make someone's (the programmer's and/or end user's) life easier.
- **Detect the pattern.** Patterns come in several flavors. The need for the utility could be driven by a series of basically identical statements, as in the example we just saw. The repetition may not always be this overt,

however. Consider the need to count the number of variables in a data set and store them in quoted and unquoted lists. The pattern and repetition may arise not from the program but from the programmer saying "I'm doing this SQL coding yet again! Maybe I should write a macro."

- **Use the tools.** Successful completion of even a simple utility requires knowledge of a variety of tools. SQL, dictionary tables, and the macro language are indispensible features of Base SAS. The utility builder should know their capabilities and be able to implement them efficiently.
- **Assume growth.** We got a hint of the potential for the growth of the macro in Version 5. The tool builder need not be reactive and simply respond to the user's requests. By thinking about how the utility may be used, the programmer can suggest extensions that the end-user may not have considered. In the example, the OBS parameter was an obvious addition. Thinking in "user mode" makes development a more collegial process.

DEVELOPMENT GUIDELINES

The tool development process is characterized by the need for a broad knowledge of SAS's capabilities and is facilitated by the programmer's creative spark. If the process is left solely to the unfettered musings of the programmer, however, chaos usually results. This section addresses the core subject matter of this paper. It presents guidelines and other considerations for tool development. Some of the ideas are inappropriate for small, independent utilities. Regardless of program size and complexity, *all* the ideas here are worthy of review, because the more background you have when you start writing an application of *any* size, the better the final product will be.

Let's reiterate what a utility does:

- performs a task that is repeated, possibly verbatim, in multiple locations in one or more programs
- handles a single, clearly-identified set of tasks
- uses other utilties in order to reduce code volume and improve reliability and functionality
- does not leave unexpected artifacts in the SAS or operating system environment once it terminates

To these worthy ends, the guidelines are grouped around a set of features that characterize good utility programming style:

- Robust structure
- Thorough error handling
- Appropriate sizing
- Consistent entity representation
- Effective user communication

The next sections look at these characteristics. As we discuss them, remember that they are usually not independent of each other.

FEATURE 1: CONSISTENT REPRESENTATION ("EMERSON WAS NOT A PROGRAMMER")

Ralph Waldo Emerson wrote in the essay *Self-Reliance* that "a foolish consistency is the hobgoblin of little minds." Fair enough, but in programming, particularly when writing utilities that will be used by many programs, consistency is seldom foolish, but rather, is both valuable and desirable. Consistent naming and predictable program structure makes the program easier to develop, more usable for both the programmer and end-user, and makes maintenance and debugging "non-onerous."

Naming. Here in utilties, as *anywhere* in SAS coding, entity names should convey meaning. Macro variables, LIBNAMEs, data set names, file names, and labels should give the reader a clear understanding of their purpose. Keep humor to a minimum. The statement:

```
%goto aDarkQuietPlaceToEnd;
```

may be humorous in some cultures, but:

```
%goto Bottom;
```

has more meaning, since it implies that the %GOTO will jump to the bottom of the macro. It should go without saying that the label BOTTOM really *is* at the bottom of the macro, and not near the top!

Naming consistency extends *across* utilities, not just *within* them. Consider the following calls to two macros:

```
%tblGen(data=, prt=no);
%dbgAE(dset=, print=n);
```

Clearly, both macros accept the same general types of parameters – a data set name and a printing option. Notice the inconsistency, though. DATA and DSET refer to the same underlying idea but require different names. PRT and PRINT produce a double whammy – not only do the parameter *names* differ, but the *values* they take are different. Even if the macros function exactly as documented, the developer is doing the users a disservice by requiring them to remember the subtleties in the name and value-pairings. The following would be more appropriate:

```
%tblGen(data=, print=no);
%dbgAE(data=, print=no);
```

This form has the added advantage of using parameter names similar to those found elsewhere in the SAS language (in this example, the DATA parameter is easily recognized by the user as the name of an input data set).

Notice the use of keyword parameters. These are always preferable to positional parameters, and have the added advantage of conveying meaning. For example,

```
%tblGen(,,10);
```

is harder to code than:

```
%tblGen(maxRec=10);
```

The parameters should connote meaning. They should *not* contain actual code. If a utility has an option for subdirectory recursion, it might use the DOS DIR command switch '/s'. Do *not* place the burden of syntax on the user by requiring syntax such as:

```
%ReadDir(subs=/s)
```

Instead, make it easy for the user:

```
%ReadDir(subs=yes)
```

Rather than pass the SUBS value directly to the DOS command as in the first example, we do a little extra work in the program:

```
%local subOpt;
%if %upcase(&subs.) = YES %then %let subOpt = /s;
    %else                       %let subOpt = ;
```

Notice the use of %LOCAL. Every variable's scope should be clearly identified. There are, to be sure, many rules that address how scoping is done by default. It's easier to be explicit about scope rather than having to remember the rules. Writing the %LOCAL and %GLOBAL statements is a small but effective means of documenting the program, providing explanatory power similar to keyword parameters in the %MACRO statement.

Structure. Utilities and macros come in many sizes, but do have some structural similarities. The program should begin with a comment block describing input and output, rules for use, error-handling, revision history, and the like (examples are found in "Maintenance Notation," below). Execution should begin by standardizing parameters, checking for parameter errors, and other housekeeping matters. Once these are handled, "core" processing – the functionality the macro was intended to do – takes place. The last general section is clean up of data sets and variables that are not needed, followed by termination messages.

Some programs require this structure. Those that are more straightforward and require less logic and processing may be able to omit or scale back some of the sections. This topic is addressed below, in "Feature 4: Structure."

Input and Output. We have already seen the benefits of consistent parameter naming. Consistent naming of *output* entities is equally important. Consider this naming convention for return codes:

_rc_*macroName*_

Following this strategy, we know without even needing to refer to documentation that the global macro variable _RC_LIBCHECK_ will be produced once the macro LibCheck terminates. If we extend the consistency to the *value* of the variable, we would also know that a value of 0, by default and convention, indicates normal completion.

One point that can't be overemphasized is to be consistent in producing only what the utility's documentation promises. If the header comment describes output as macro variables N, LIST, and LISTQ, the *only* difference in the SAS environment between when the macro began execution and when it ended should be the addition or replacement of N, LIST, and LISTQ. The user should *not* get "bonus" items such as data sets TEMP and TEMP1, nor should the CENTER option become NOCENTER and page numbers turned off. It is the programmer's responsibility to leave behind only what was promised, and that promise is made in the program's documentation.

FEATURE 2: SIZING

Judging whether a utility is sized correctly is similar to assessing pornography: you'll know an improperly-sized utility when you see one. If the program is supposed to produce a simple set of macro variables, yet uses a thousand lines of code to do so, it may be too large. Likewise, if many outputs are promised and the program uses no external utilities and is very short, it may be too small. This is of concern for several reasons. The "too large" program may be overlooking an external utility or SAS feature that would simplify (i.e., reduce) the number of statements. The "too small" program may not be performing required error checking and may make too many assumptions during processing. In other words, it may not be "bulletproofed." What we want is something that's small, but not too small. How do we achieve this?

Scope. The program should not be all things to all people. A utility, by any definition, is single-purpose. If multiple purposes begin to appear, it may be time to create other utilities. The divided functionality would also serve other utility developers, since the generalized activities performed by the would-be single large program are now, in effect, surfaced and visible as standalone programs.

Parameters. It may be possible to control program size without compromising its usefulness. Consider the following program. It has four parameters: LIB, which identifies the library to process; and COUNT, LIST, and QLIST, which instruct the macro to create macro variables containing the count, names, and quoted names of members in a library. Notice that there is not error-trapping. If COUNT, LIST, and QLIST are NO, the SQL call will fail. Notice, too, that much of the code focuses on inclusion of program statements based on the values of COUNT, LIST, and QLIST.

```
%macro DSnames(count=yes, list=yes, qlist=yes, lib=work);
       %let count = %upcase(&count.);
       %let list  = %upcase(&list.);
       %let qlist = %upcase(&qlist.);
       proc sql noprint;
            select %if &count. = YES %then count(*), ;
                   %if &list.  = YES %then trim(memname), ;
                   %if &qlist. = YES %then quote(trim(memname)) ;
            into   %if &count. = YES %then :dsn, ;
                   %if &list.  = YES %then :dslist separated by ' ', ;
                   %if &qlist. = YES %then :dsqlist separated by ' ' ;
            from dictionary.tables
            where libname="%upcase(&lib.)";
       quit;
%mend;
```

Once we examine what the macro *really* does – counting and listing data sets – we can see that it would be easier to write it, maintain it, and enhance it if we produced the count, list, and quoted list variables by default. This relieves us of the need for all the %IF-%THEN coding; we take away some functionality in order to gain simplicity. Even if the user doesn't need &DSQLIST, she knows it was created because it was documented in the macro header (not shown here). The slimmed-down and easier-to-read program follows:

```
%macro DSnames(lib=work);
       proc sql noprint;
            select count(*), trim(memname),            quote(trim(memname))
            into   :dsn,     :dslist separated by ' ', :dsqlist separated by ' '
            from dictionary.tables
            where libname="%upcase(&lib.)";
       quit;
%mend;
```

External Utilities. It's hard to overemphasize the importance of using other utilities. Rather than parse a data set name to see if it is one or two levels, we pass it to a utility that creates LIBNAME and MEMNAME parameters.

Rather than write the code (yet again!) to see if a library exists and identify the path allocated to it, we call a utility to do the work. It's easy to see that if the utility requires processing that is off topic, we'd like much of this work to be handled by macros in a utility library.

Let's look at an example. Macro PRINTIT does what you'd expect, printing from each data set in a list contained in parameter DATA. To do this, the macro must parse the data set list, verify that other parameters are acceptable, and read data set names from a library. Then, the core functionality of the macro, the actual printing, can take place. This is represented in general terms below:

```
%macro PrintIt(data=, all=);
       <parse data set parameter>
       <verify other parameter is in an acceptable list>
       <gather data set names from a library>
       <print from each data set>
%mend;
```

Much of the macro's code will be spent performing non-core (non-printing) activities. This could require hundreds of line of code. These are lines that will likely be repeated for similar utilities, and are candidates for being standalone utilities. Using a library, we could rewrite the program as shown below:

```
%macro PrintIt(data=, all=);
      %ParseDSN(data=&data.);
               %if &_rc_ParseDSN_. ^= 0 %then <terminate>;
      %InList(var=&all., values=y n dk);
             %if &_rc_InList_. ^= 0 %then <terminate>;
      %GetDSN(libname=&level1.);
      %if &dsnCount. > 0 %then %do;
          <print from each data set>
          %end;
%mend;
```

The rewritten program begins to reveal the power and elegance of the utility library concept. We call PARSEDSN. If the return code from it was not zero, we know there was a problem and we terminate. We then call INLIST, and make a similar continue/terminate decision based on its return code. Finally, we call GETDSN, passing it the library name created by PARSEDSN. If the counter returned by GETDSN is greater than zero, we peform the core, utility-specific processing (in bold face). The potentially hundreds of lines of code from the first example is replaced by five actual lines. Not only do we have a smaller program, we likely have one that is more robust than the original, and one that is based on validated tools.

This only becomes possible if we have documentation for the macro library. We need to know what utilities are available, how they raise error conditions, and what entities they produce. Documentation is discussed later, in "Advanced Topics."

FEATURE 3: ERROR TRAPPING

A key feature of any good utility is its ability to detect problems and handle them appropriately. There are three aspects to error-trapping: timing, reporting, and reacting.

Timing. At the risk of over-generalizing, errors can occur at two distinct phases of program execution: startup and later processing. Startup errors include: a null value for a required parameter; an inappropriate value for a parameter; a valid value for a parameter becoming invalid because of another parameter's value. In other words, these are conditions that can be tested at the start of the utility. It is up to the programmer to decide whether a condition really is an error. This is obvious in, say, the case of a null value for a required parameter. Other cases live in a grayer area. For example, if a parameter has "yes" or "no" values should the program terminate if it received a value of "n"? Most likely it shoudn't, but the program *should* write a message to the SAS Log describing the correct syntax.

Later errors are not as predictable. The program may run out of disk space, not find an expected data set, find only numeric variables rather than the expected mix of numerics and characters, and so on. Such events cannot be tested at startup. The distinction between problems identified at startup and those detected later is not simply academic, and has a direct impact on how the errors are reported.

Reporting. One of the best ways to illustrate the need for a coherent error-reporting strategy is to play the role of the end user. Here is where the timing issues noted above come into play. Suppose a macro tests parameters for two conditions and will fail if either is true. One approach is the following:

```
%if <condition1> %then %do;
    %put <Error message for condition 1>;
    %goto bottom;
    %end;
%if <condition2> %then %do;
    %put <Error message for condition 2>;
    %goto bottom;
    %end;
```

If the first error condition is true, we print a message and branch to the section labeled BOTTOM, presumably a cleanup and termination section. Similar logic is applied to the second error condition.

Consider, though, the sequence if *both* tests are true. In this scenario, the user runs the macro and raises, say, error condition 2. The macro displays the error message, then terminates. The user makes the correction, then reruns the macro, feeding it parameters that raise error condition 1. Two executions, two errors. It would be cleaner to write the macro so that *all* possible parameter-checking (startup) conditions were checked and reported, thus saving the user the frustration of making multiple erroneous calls. Part of the revised program is shown below:

```
%local _errFlag;
%global _rc;
%let _errFlag = f;
%let _rc      = 0;
%if condition1 %then %do;
    %put Error message for condition 1;
    %let _errFlag = t;
    %let _rc      = 1;
    %end;
%if condition2 %then %do;
    %put Error message for condition 2;
    %let _errFlag = t;
    %let _rc      = 1;
    %end;
%if &_errFlag = t %then %do;
    %put Execution terminating due to error(s) noted above;
    %goto bottom;
    %end;
```

Here, a bit more coding effort is required than the first formulation. The payoff is in usability. We set an error flag (_ERRFLAG) to false, then reset it if either error condition is raised. Note that if condition 1 is true, we print a message, set the error flag and return code, and continue to execute, checking for other errors. If at the end of this code fragment the error flag is true, we print a termination message and branch to the bottom of the program. Note, too, that whenever an error condition is raised, _RC, the global macro variable containing the return code is set to 1. This leads us to the third aspect of error reporting.

Reacting. The question that logically follows from the above is, essentially, “Now that I’ve found a problematic condition, what do I do with it?” The problem has been reported via a %PUT statement, but is it critical (a null value for a required parameter with no default) or just annoying (“Y” instead of “YES”)? This is a judgment call that can be implemented via careful setting of error flags. Let’s modify the previous code fragment:

```
%global _rc;
%let _rc = 0;
%if &data. = %then %do;
    %put Required parameter DATA was missing. ;
    %let _rc = 2;
    %end;
%if %substr(&prt., 1, 1) = Y %then %do;
    %put PRT value Y assumed to be YES. Execution continues.;
    %let prt = YES;
    %if &_rc. < 1 %then %let _rc = 1;
    %end;
```

```
%if &_rc. > 1 %then %do;
    %put Execution terminating due to error(s) noted above;
    %goto bottom;
    %end;
```

The strategy employed here is simple: the lower the value of the error flag, the better the chance of not stopping execution. Macro variable _RC is set to 0, indicating a clean execution of the macro ("innocent until proven guilty").

The first %IF sequence identifies a critical error (missing required parameter). It sets _RC to 2. The second %IF sequence identifies a more benign condition, but sets _RC to the greater of _RC or 1. This ensures that an _RC value of 2 will not be replaced by a less critical value of 1. Notice that we need to both identify the problem *and* reset PRT to its assumed value of YES.

Once problem identification is complete, we evaluate the error flag. If it exceeds the threshold value, we terminate. Otherwise, we continue, knowing that the user is alerted in several ways to a minor problem: a message is written to the SAS Log, and the return code value passed to the program calling the macro will be 1, indicating a non-fatal error.

FEATURE 4: STRUCTURE

The fourth aspect of utility design is both vital and situation-dependent. Here, as elsewhere in SAS programming, the flexibility of the language can lead to program-to-program variation at best or sheer chaos at worst if no guidelines are in place. This section presents a set of guidelines that are applicable for developing a wide range of activities. Bear in mind that a full implementation of what is laid out here is not always required, and may lead to an over-architected solution.

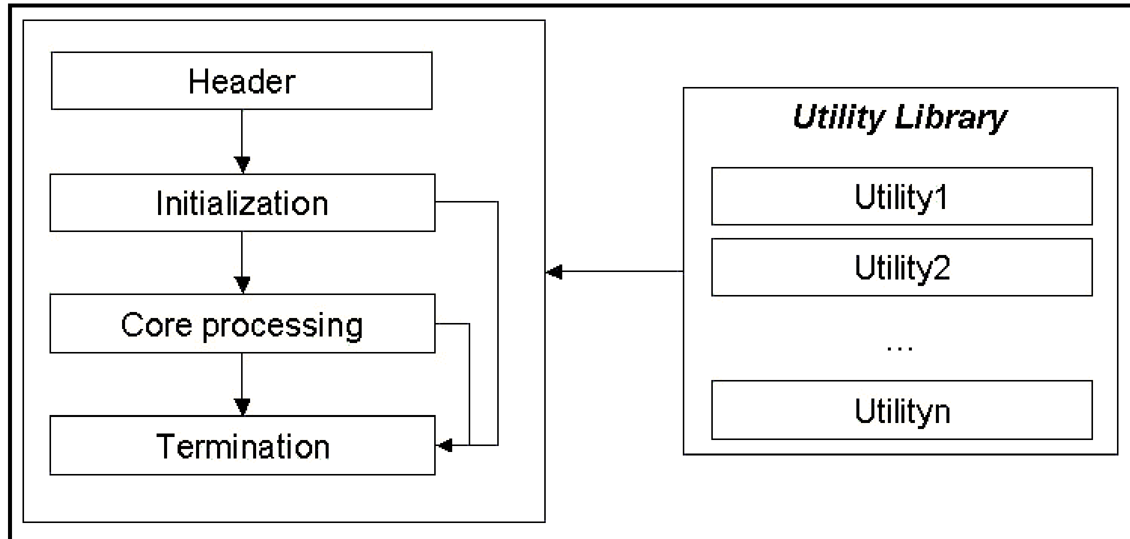
A utility typically consists of four sections:

- **Header Documentation.** This comment block is at the beginning of the file containing the macro, preceding even the %MACRO statement. The header may contain information about the utility's function, input and output, usage, references to other macros, and its revision history. A key item here is the output produced by the utility. The header comment should identify *every* artifact (macro variable, data set, file, option settings, etc.) that the macro may produce, mindful that if processing terminates prematurely not all items may be created. The output item list should be the *only* items in the SAS environment that differ between when the utility began execution and when it ended. See "Termination," below, for more on this.
- **Initialization.** This section establishes the utility's working environment. It does some or all of the following:
    - Writes a message to the SAS Log indicating program start up, optionally echoing macro parameters.
    - Defines local and global macro variables (via %LOCAL and %GLOBAL statements).
    - Saves settings of system variables that the macro may modify. These will be reset in the Termination section.
    - Standardize macro parameters. This includes conversion to upper case, assigning variable values based on input parameters, and the like
    - Verifies macro parameters. The utility should handle problematic conditions created by the macro parameters.
- **Core Processing.** Once initialization is complete, the macro begins a section, possibly quite long, that essentially says "if I'm executing here, then I know I was given parameters that make sense. Now I can begin the processing described in my header comment [i.e., the core processing]." Ideally, the macro will use calls to other macros in a common utility library to accomplish some of the more routine tasks. There is no guarantee, of course, that execution will continue uninterrupted. If the macro detects an error condition either in its own processing or from an invoked utility macro, the flow of program control may head directly to the fourth and final section.
- **Termination.** This section is the site of activity that takes place once initialization and, possibly, core processing is complete. It handles some or all of the following:
    - Delete temporary global macro variables that for reasons of logic and design could not be identified as local.
    - Reset system options that were adjusted by the program to their original values.
    - Purge files (SAS data sets, text files, *et al.*) that were created by the program and were not identified as an output item by the header documentation.
    - Write a message to the SAS Log indicating program termination, optionally including the return code and any other key variables not already reported.

> Notice the verbs: delete, reset, purge. *It is vital to design any utility, regardless of size and stature, to leave the SAS environment just as it found it prior to initialization. The only changes should be artifacts – macro variables, data sets, etc. – that were documented in the output section of the header documentation. If the header says the macro will produce macro variables N, LIST, and LISTQ, then the* only *difference in the SAS environment between the beginning and end of the macro's execution should be the addition or replacement of these three macro variables!*

Figure 1, below, graphically represents the points just covered.

Figure 1: Generalized Program Structure



Two of the key points in the figure are that members of the utility library can be used during *any* part of the macro's execution and the termination phase can be reached in three ways (at the end of normal completion of core processing, or via branching from initialization or during core processing).

FEATURE 5: COMMUNICATION

The fifth and last aspect of utility design that we'll discuss in this paper is, arguably, the most important. All utilities should communicate results, and not simply go about their business and produce expected outputs. The messages are important during all phases of a utility's life cycle. During initial development, messages are for the benefit of the developer, and may contain content that will be irrelevant for the end user. Once the utility is debugged and put into production, messages for the user indicate completion status and names and values of output items. All messages are helpful later in the utility's life, when it is being debugged or enhanced.

Types and Locations. Messages come in various flavors. Some are produced at significant checkpoints. The beginning and end of the utility are obvious choices, as are the starting points of major or otherwise significant sections of the program. You could, for example, insert messages at the top of an iterative %DO loop, at the start of sections that process and report on SAS metadata, and just before sections of the program that were problematic during development. This is generalized below:

```
%macro demo(lib=, dataOut=);
       %put Demo-> Begin execution. Input libname [&lib.] output data set
[&dataOut.].;
       %put Demo-> Begin parameter checks.;
       initialization, parameter checking statements
       %put Demo-> Read library from dictionary tables.;
       invoke macro MEMLIST, which creates macro variables NMEM and MEMLIST
       %put Demo-> We found NMEM [&nMem.] data sets.;
       %do nData = 1 %to &nMem.;
           %let name = %scan(&memList., &nData.);
           %put Demo-> Process data set &nData. of &nMem.: &name.;
           process data set &name.
       %end;
```

```
        %bottom: %put Demo-> Done printing, now clean up.;
                 termination activities
                 %put Demo-> End.;
%mend;
```

Content. Notice two key features of the %PUT text in the example. First, every message begins with the macro name, DEMO. Thus when macros call macros, we have a clear sense of which macro is executing at any given time. This knowledge may seem like overkill to the end user, but it is vital to the programmer who is tasked with developing or debugging. Selected lines written to the SAS Log from DEMO may look like:

```
Demo-> Begin parameter checks.
Demo-> Read library from dictionary tables.
MemList-> Begin execution. Library name [CLINICAL].
MemList-> End.
Demo-> We found NMEM [9] data sets.
Demo-> Process data set 1 of 9: AE
Demo-> Process data set 2 of 9: DEMOG
```

We can see the relevance for debugging purposes, in particular. Regardless of macro system option settings (MPRINT, SYMBOLGEN, *et al.*), we will know which utility was executing when the program failed.

The other feature of the messages is that they are informative, displaying not only a macro variable, but explanatory text as well. Imagine the head scratching created by unlabeled output that looks like this:

```
9
1 AE
2 DEMOG
```

Don't snicker – people do this! Such tossing of lonesome scraps of data into the Log creates a situation where the program communicates *dis*information. It's frustrating, unprofessional, and makes the user uncomfortable with results even if he/she knows the program has been validated. Take the time to describe output. If anything, err of the side of being too chatty.

Configuration. The degree of the macro's chattiness can be controlled by a macro parameter. Just *which* messages should be printed and *how many* levels of control are appropriate are a judgment call, based on the needs of both the end-user and programmer. As an example, let's rework the DEMO macro (new code is in bold face):

```
%macro demo(lib=, dataOut=, printLevel=1);
       %if &printLevel. >= 1 %then %put Demo-> Begin execution. Input libname
[&lib.]
           output data set [&dataOut.].;
       %if &printLevel. >= 2 %then %put Demo-> Begin parameter checks.;
       initialization, parameter checking statements
       %if &printLevel. >= 2 %then %put Demo-> Read library from dictionary
tables.;
       invoke macro MEMLIST, which creates macro variables NMEM and MEMLIST
       %if &printLevel. >= 1 %then %put Demo-> We found NMEM [&nMem.] data sets.;
       %do nData = 1 %to &nMem.;
           %let name = %scan(&memList., &nData.);
           %if &printLevel. >= 2 %then %put Demo-> Process data set &nData. of
               &nMem.: &name.;
           process data set &name.
       %end;
       %bottom: %if &printLevel. >= 2 %then %put Demo-> Done printing, now clean
up.;
                termination activities
                %if &printLevel. >= 1 %then %put Demo-> End.;
%mend;
```

Invoking with a PRINTLEVEL of 2 ("fully loaded") would yield output like this:

```
Demo-> Begin execution. Input libname [CLINICAL]
Demo-> Begin parameter checks.
Demo-> Read library from dictionary tables.
MemList-> Begin execution. Library name [CLINICAL].
```

```
MemList-> End.
Demo-> We found NMEM [2] data sets.
Demo-> Process data set 1 of 9: AE
Demo-> Process data set 2 of 9: DEMOG
Demo-> Done printing, now clean up.
Demo-> End.
```

Invoking with a PRINTLEVEL of 1 (the default) would yield output like this::

```
Demo-> Begin execution. Input libname [CLINICAL]
MemList-> Begin execution. Library name [CLINICAL].
MemList-> End.
Demo-> We found NMEM [2] data sets.
Demo-> End.
```

PRINTLEVEL=0 produces no output from DEMO, but will still show output from MEMLIST. This points out the interconnected nature of these and all utilities. If properly designed, *all* macros in the utility library would honor the same three-tiered output strategy. This would allow DEMO to pass its value of PRINTLEVEL to MEMLIST. MEMLIST would say, in effect, "A message setting of *n* means I should restrict my output in the following way …"

ADVANCED TOPICS

The preceding discussion has only touched on the toolset and mindset needed for developing solid, generalized applications. Among the topics not discussed are: methods for standardizing program start up, thereby ensuring consistency of library allocations, macro autocall paths, and the like; version control and strategies for moving a utility from test to production environments; dealing with macro growth, whether to modify an existing program or split it into multiple, interdependent ones. There are two topics that warrant attention in this final section. We discuss maintenance notation, then focus on a documentation strategy for a system of macros.

MAINTENANCE NOTATION

This topic is as important as the name is obscure. This is an elaboration of the "Header Documentation" discussion in "Feature 4: Structure," above. In that section, we emphasized the importance of clearly specifying output items such as macro variables and data sets. The program's maintenance history was given only passing mention. For programmers charged with upgrading or debugging, however, the method employed for recording changes is critical. Let's look at part of a program that has been revised:

```
/*
          Name:  PrintIt
   Description:  Print first n observations from all data sets in a library
    … statements omitted …
   History:   Date        Init  Comments
              2004/05/18  FCD   Initial release to AutoCall library
              2004/05/22  FCD   Add test for 0 members in library
*/
%macro PrintItV2(lib=work, obs=10);
       %put PrintIt-> Begin. LIB [&lib.]  OBS [&obs.];
       %optVals(action=save, options=mprint notes);
        … statements omitted …
       %if &_rc_MemList_. ^= 0 %then %do;
           %emph(PrintIt-> Execution terminating prematurely.);
           %goto Bottom;      /* <<<< <<< << < <<<< <<< << < <<<< <<< << < <<<<
<<< << < */
           %end;
       %if &_nMemList_. = 0 %then %do;
           %put PrintIt-> No members to process!;
           data _null_;
                file print print;
                put 80 * '*' // "No data sets in library &_path_.!" // 80 * '*' ;
           run;
```

```
            %goto Bottom;      /* <<<< <<< << < <<<< <<< << < <<<< <<< << < <<<<
<<< << < */
            %end;
        %do i = 1 %to &_nMemList_.;
            %let data = %scan(&_Memlist_., &i.);
            %put;
        … statements omitted …
%mend;
```

Well, fine. Programmer FCD documented the change made on May 22, but the real question is *"Where was the program changed?"* We have a hint – the header comment implies there should be a test for something equaling zero, maybe with a message containing an error or warning message. Identifying the location(s) touched by the change should not require sleuthing. Adopting a simple documentation strategy (hence this section's title) solves the problem. Let's implement it using the previous example:

```
/*
          Name:  PrintIt
   Description:  Print first n observations from all data sets in a library
    … statements omitted …
   History:   Date        Init  Comments
              2004/05/18  FCD   Initial release to AutoCall library
              2004/05/22  FCD   [U01] Add test for 0 members in library
*/
%macro PrintItV2(lib=work, obs=10);
       %put PrintIt-> Begin. LIB [&lib.]  OBS [&obs.];
       %optVals(action=save, options=mprint notes);
        … statements omitted …
       %if &_rc_MemList_. ^= 0 %then %do;
           %emph(PrintIt-> Execution terminating prematurely.);
           %goto Bottom;      /* <<<< <<< << < <<<< <<< << < <<<< <<< << < <<<<
<<< << < */
           %end;
       %if &_nMemList_. = 0 %then %do;   /* this DO group is [U01] */
           %put PrintIt-> No members to process!;
           data _null_;
                file print print;
                put 80 * '*' // "No data sets in library &_path_.!" // 80 * '*' ;
           run;
           %goto Bottom;      /* <<<< <<< << < <<<< <<< << < <<<< <<< << < <<<<
<<< << < */
           %end;
       %do i = 1 %to &_nMemList_.;
           %let data = %scan(&_Memlist_., &i.);
           %put;
        … statements omitted …
%mend;
```

The executable portion of the macro is *exactly* the same as before. The critical addition is found in two places. First, the program history line now has a change code – [U01]. The format of the code is arbitrary, but *does* need to be followed consistently. The second location of the change code points us to where the change was actually made. Notice that the entire %DO group was part of the change. Labeling every statement in the group with the [U01] code is overkill and tedious. The comment at the beginning of the group is sufficient to identify the change.

The benefit of this technique becomes apparent when you need to review a set of changes. Using a text editor Find command, you can search for [U01], find it in the header comment, search again and find it in the %IF &_NMEMLIST statement. There can, of course, be many statements that had to be added or modified for a given change. If the changes were noted correctly during program modification, all the updates will be easy to locate.

This technique is invaluable in large programs, and can be extended *across* programs as well. Imagine an enhancement that requires modification of three programs. This really isn't *three* separate changes, but *one* change affecting three programs. If we treat it as one change and use the same change code in each of the affected programs, we can use a generic "Search in Files" text editor command to quickly locate the changes. As with the single program example, this is simple in concept and requires just a little discipline to implement. The trick here, of course, is to ensure that the new change code isn't already in use in any program. One way to approach this is code notation of the form [Mnn] (M for "Multiple"). Assign it, e.g. [M23] , search for it in *all* programs in the program library (not just the ones you'll be touching), and use it if it isn't found. This technique clearly identifies a program change as being standalone – Uxx – or one of several programs – Mxx.

DOCUMENTATION STRATEGY

All discussion of documentation thus far has focused on internal commenting. If we are to encourage one of the cornerstones of good design – liberal use of a library of macros – we need to describe the capabilities of the macro library members. As always seems to be the case with SAS coding, there are as many styles of doing this as there are programmers writing the documentation. One particularly coherent and reasonable approach is presented in Sy Truong's paper from the PharmaSUG 2004 *Proceedings.* "How to Develop User Friendly Macros" outlines the merits of style of document delivery (web or PDF). It also goes beyond the macro files, literally, and describes the nature and content of a range of documents that accompany the macro library and enhance its effectiveness. These include:

- Reference manual
- Usage manual
- Administration manual
- Glossary of ERROR and WARNING messages
- Training materials
- Quick reference card
- FAQ
- Next release Wish List

It's quite a list, and may be a bit daunting to a programmer who inserts, perhaps grudgingly, some comments at the start of the program and calls it a day. Perhaps anticipating these squirmers and whiners, Truong notes

> It is not required that all of these documentation methods be used, especially for small macros. However, for larger sets of macros that form a system, it is recommended that most of these methods be implemented. Some of the content may be redundant but different users absorb information in different ways. It is therefore more effective to have many alternatives.

It would be nice to be able to automate the creation of system documents. This would be preferable to updating a macro and then, hopefully, manually updating one or more documents. Such tools are not packaged with the SAS System, but they can be made with Base SAS tools. Here is an overview of two programs written and frequently used by the author.

MacroDoc. This is where consistent coding style pays off. Programs in the course-length treatment of this paper use a standard program header. The following is the full text of the header used in the previous example. Pay attention to the items in bold face.

```
/*
         Name:  PrintIt
  Description:  Print first n observations from all data sets in a library
        Input:  All parameters are case-insensitive
                LIB  Libname of library to print
                     Default: work
                OBS  Number of observations to print from each data set
                     Default: 10
       Output:  PRINT procedure output and/or notes to default output location
  Usage Notes:  If the data set is empty, the macro prints a message
```

```
                Indirectly uses dictionary tables MACROS (via OPTVALS),
                OPTIONS (via OPTVALS), TABLES (via COUNTOBS and MEMLIST),
                MEMBERS (via MEMLIST call to LIBCHECK)
   References:   Memlist CountObs OptVals Emph
  History:   Date        Init  Comments
             2004/05/18  FCD   Initial release to AutoCall library
             2004/05/22  FCD   [U01] Add test for 0 members in library
*/
```

The entire header is a single comment of the form /* … */. A content identifier (Name, Description, etc.) is followed by a colon (:). The "References" identifier lists names of autocall macros used by the macro (MemList, CountObs, OptVals, and Emph). Finally, as we saw in the previous section, we use the change notation [U01] to identify statements in the program that were made while adding a test for an empty library.

Armed with this knowledge of header layout and change notation format, MacroDoc reads each member of the autocall library and produces output similar to Figure 2, below. The program builds an HTML frameset that allows the user to view an HTML version of each macro or open the macro with the application associated with file type "SAS". The "Webified" version of the macro builds hyperlinks to macros identified in "References" (testing first to see if the file exists) and displays update codes in red to make their location easier when scrolling through the file. Finally, there is a link to a file containing plain text of all the macro headers.

The result is an application that is easy to use (thanks to the now-intuitive Web interfaces) and provides ready access to key macro features. Bear in mind that no matter how slick and helpful the interface, if the headers are poorly written or incomplete, the entire application is rendered useless.

MacroXref. A question arises when macros rely on other macros to perform low-level processing: "If macro X is changed, what macros will have to be tested to ensure they still function correctly?" Up to a point, this information can be carried in the programmer's head, or at least on Post-It notes in his cube. However, any reasonably complex macro and utility library will have a large degree of dependencies. At this level, clearly, automation is called for.

The MacroXref macro identifies macros in the SAS autocall library and identifies references to them in a program list supplied by the user. The list is one or more directories, with optional subdirectory processing. All SAS programs (file type "SAS") are reviewed for references to the autocall macros. The output is two listings: "program 'x' uses which autocall macros?" and "autocall macro 'y' is used in which programs?" Hyperlinks provide ready access to all programs (autocall members and those in user-specifed directories). Example output is shown in Figure 3, below.

CONCLUSION

SAS software is a vast electronic playground, allowing the programmer to exercise a great degree of ingenuity and creativity. Think back to your childhood, though, and recall that even on the playground there were rules. There is a need for rules on the SAS playground, too. They *are not* about stifling the fun of designing and implementing a clever and helpful tool. They *are* about fostering an environment where new utilities can be developed quickly, and with confidence that they can be readily maintained and expanded because they are all based on the same solid architecture. To that end, here are the key development guidelines reviewed in this paper:

- **Focus on core functionality.** If the program uses hundreds of lines of code to build lists or count items, chances are there is already a utility to do this. The bulk of the code should be the utilty's unique contribution to the programming tool set.
- **Use the macro library.** All but the simplest utilities use external macros to reduce code volume and improve reliability and functionality.
- **Code consistently.** Macro parameter names and values, return codes, and all entities under your control should be created consistently. Seeing a new macro in the library and being able to guess its purpose and parameters before seeing its documentation is a sign of elegance, not mindless repetition!
- **Communicate clearly with the user.** Write messages that are informative and clearly labeled. It is better to be too "chatty" than to keep mum and leave the user in the dark. Remember that you, the programmer, become a user when you're debugging or enhancing the program.

- **Deliver only what you promised.** The only footprints that the program should leave in the SAS environment are the output items identified in the program header.

CONTACT AND ACKNOWLEDGEMENTS

Thanks to Jeff Abolafia and John Ingersoll, my colleagues at Rho, Inc. in Chapel Hill NC, for providing valuable comments while reviewing early drafts of this paper. Thanks as well to April "The English Major" Sansom for being both my wife and my proofreader.

Thanks, too, to the folks at SAS for making their fine products. And lest we forget:

"SAS and all other SAS Institute Inc. product or service names are registered trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration."

As always, your comments are welcomed and encouraged. For related papers and complete contact information, visit the author's Web site, www.CodeCraftersInc.com, or contact him at Frank@CodeCraftersInc.com.

Figure 2: MacroDoc Output


```
/*
         Name:  PrintItV2

  Description:  Print first n observations from all data sets in a library

        Input:  All parameters are case-insensitive
                LIB  Libname of library to print
                     Default: work
                OBS  Number of observations to pr...                      set
                     Default: 10

       Output:  PRINT procedure output and/or n...                t location

  Usage Notes:  If the data set is empty, the ma...                  e

                Indirectly uses dictionary tables MA...              TVALS),
                OPTIONS (via OPTVALS), TABLES (via COUNTOBS and MEMLIST),
                MEMBERS (via MEMLIST call to LIBCHECK)

   References:  Memlist CountObs OptVals Emph

   History:   Date        Init  Comments
              2004/05/18  FCD   Initial release to AutoCall library
              2004/05/22  FCD   [U01] Add test for 0 members in library

*/

%macro PrintItV2(lib=work, obs=10);
      %put PrintIt-> Begin. LIB [&lib.]  OBS [&obs.];

      %optVals(action=save, options=nprint notes);
      options noprint nonotes;

      %let lib = %upcase(&lib.);
      %local i data;

      %MemList(lib=&lib.);
      %if &_rc_MemList_. ^= 0 %then %do;
         %emph(PrintIt-> Execution terminating prematurely.);
         %goto Bottom;      /* <<<< <<< << < <<<< <<< << < <<<< <<< << < <<<< <<< << < */
         %end;

      %if &_nMemList_. = 0 %then %do;    /* [U01] */
         %put PrintIt-> No members to process!;
         data _null_;
            file print print;
            put 80 * '*' // "No data sets in library &_path_.!" // 80 * '*' ;
         run;
         %goto Bottom;      /* <<<< <<< << < <<<< <<< << < <<<< <<< << < <<<< <<< << < */
         %end;

      %do i = 1 %to &_nMemList_.;
         %let data = %scan(&_Memlist_., &i.);
         %put;
         %put PrintIt-> Data set &i. of &_nMemList_.: &data. ;
         %CountObs(dataset=&lib..&data.);

         title "Up to first &obs. obs from LIBNAME &lib., data set %upcase(&data.)";
         title2 "Data reside in this directory: &_path_." ;
```

Figure 3: MacroXref Output

Run time: 10:14, 08JUL04

Directories
Report

By Referencing Program
Report

By macro
Report

Macro 'x' Is Used By Which Programs?

Current as of 10:14 on 08JUL04, 08JUL04

Autocall Macro	Program References
C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\AllMacVars.sas	C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\SetUp.sas C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Misc\Run_WriteFiles.sas
C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\CharCheck.sas	C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Misc\Run_CharCheck.sas
C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\CountObs.sas	C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\CharCheck.sas C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\PrintIt.sas C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\PrintItV2.sas C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\VarList.sas
C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\CurrFile.sas	C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\WriteFiles.sas C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\write_files.sas C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Misc\WriteMacroDoc.sas
C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\Emph.sas	C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\PrintIt.sas C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\PrintItV2.sas C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\QuickDSList.sas C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\ReadDir.sas C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\WriteFiles.sas C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Misc\Run_Emph.sas
C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\HREF.sas	C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\MacroXref.sas
C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\IsMacVar.sas	C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\NItems.sas C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\QuoteList.sas C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Misc\Run_IsMacVar.sas
C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\LibCheck.sas	C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\CharCheck.sas C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\MemList.sas
C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\MacroXref.sas	C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Misc\Run_MacroXref.sas
C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\MemList.sas	C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\PrintIt.sas C:\-- Common Files --\Publications\Books\DictionaryTables\Programs\Macros\PrintItV2.sas