Putting the FORMAT procedure at the wheel: advanced applications of table lookup

Peter Kretzman, BR Associates, Inc.

Abstract

Past and present SUGI conferences have offered a wealth of papers covering the use of SAS user-defined formats for table lookup. Mastery of this simple but powerful programming technique is a crucial step towards achieving fluency in SAS programming and systems design. Yet no previous SUGI papers and no SAS manuals have gone beyond presenting basic techniques and truly plumbed the depths of the immense potential of the FORMAT procedure.

This tutorial bridges that gap. It shows how you can extend your use of the basic PROC FORMAT table lookup technique into virtually any larger SAS System application, in ways possibly not dreamed of even by the original designers of PROC FORMAT. Specifically, PROC FORMAT-based table lookup techniques are discussed that do the following:

- facilitate easy maintenance for SAS-based systems by making them table-driven instead of purdy code-driven;
- permit recoding, categorizing, and list checking, all in fundamentally the same manner;
- provide fast and transparent data base-like information retrieval, even for lookups based on multiple variables.

Aside from just exhibiting technological razzle-dazzle, however, these table lookup techniques allow the SAS programmer to increase the overall maintainability of a SAS-based system by allowing a sharp separation between program data and program code. This tutorial shows you ways to get the data out of your program, making coding, debugging, and maintenance considerably easier.

I. Introduction

Almost all treatment of SAS formats in SUGI presentations or in SAS manuals has focused on the role of formats in report writing, either in custom reports or in SAS procedures such as PROC PRINT and PROC FREQ. This paper will briefly summarize basic format techniques, but will mostly concentrate on the rarely discussed (and quite powerful) more advanced techniques that use formats within ordinary DATA steps to control program flow and influence data selection. With respect to such techniques, the very word "format" thus is not especially descriptive, since that word evokes associations of some kind of output rather than of internal processing. When using advanced format techniques, it makes more sense to talk of "recoding" or "categorizing" values, by means of a tool that the SAS System just happens to call formats.

The recoding and categorizing capabilities that are available through the use of advanced format techniques allow the SAS programmer to address larger, systems-related concerns: how to increase the overall maintainability of a SAS-based system by concentrating on separating program data from program code. The tenet to follow (which is not unique to SAS programming) can be simply expressed:

GET THE DATA OUT OF YOUR PROGRAM

Most people respond to that slogan with confusion: they don't think that there is any data in their program. All the data that they know about is in SAS data sets. However, when they look carefully at their program, they finally come to see a different kind of "data," but data nonetheless: long blocks of IF/THEN/ELSE logic, checking for particular hard-coded values which influence program flow or which affect the creation of new variables. I call this kind of data value correspondences. For example,

IF X=2 THEN Y=A

is a hard-coded piece of program logic showing the correspondence of the value 2 to the label "A".

Having that kind of data in the program causes all sorts of ripples through the software lifecycle. Whenever those hard-coded values change, or when a value needs to be added to an existing IF/THEN/ELSE chain, the program itself needs to be modified, usually trivially but sometimes drastically. It's fairly easy, even for experienced programmers, to make small, often unseen changes when entering even "trivial" changes to long-standing code. Sometimes these value correspondences can be found in several portions of a large system in virtually identical incarnations; the hard-coding approach requires that changes be made consistently to each version of the data. In short, large systems that still have this kind of data inside them are set-ups for failure, houses of cards, ticking time bombs.

The SAS format facility, especially in its advanced recoding and categorizing capabilities, provides the SAS programmer an immensely powerful tool to defuse the time bomb by making the SAS code table-driven. By coding value correspondences within formats rather than within hard SAS code, you isolate the data from the main program logic itself. The program itself thus becomes much shorter and simpler, since it can concentrate on what it has to do rather than on the involved lists that are needed for recoding or program logic. With shorter and simpler code, code development and code debugging are both faster and easier.

Other advantages of fully stretching the SAS format technique: since formats are basically just simple value correspondences, they can be taken in at a glance more easily than their equivalents that are hard-coded within a
program. Changes or additions tend to be easy and relatively risk-free to make, and most program maintenance can usually be taken away from the programmer and given to clerical staff. Redundant code can be largely eliminated, since formats can be maintained in libraries (both in original text form and in object form after processing by PROC FORMAT) and can be reused for several code modules within a system or for completely separate systems.

This tutorial attempts to cover most of the relevant issues regarding the use of formats to make code table-driven. Examples are provided of various table lookup techniques using formats within SAS DATA steps. In general, the paper focuses on the use of formats as a program logic tool, not as an output tool. Formats are a fairly large topic in the SAS system, and this tutorial will not cover all aspects of them (e.g., picture formats, VALUES, FZUZ). The reader is encouraged to refer to the bibliography for more information on these and other format-related topics.

II. Basics of SAS Formats

Our discussion here will focus on how to create a format table (a value correspondence chart) and how to make use of that table within a SAS program. We will first show the traditional, most often discussed method of using formats for output control, and then move quickly on to using formats for logic control.

Let's first show a small fragment of SAS code to illustrate the points we need to make. This fragment consists of two things: a PROC FORMAT, which creates a value correspondence table, and a DATA step, which makes use of the table.

```
PROC FORMAT LIBRARY=LIBRARY;
  VALUE A B C D
  1 -> 'X';
  2 -> 'Y';
  3 -> 'Z';
RUN;
DATA WALK;
  VAR = 1;
  FORMAT VAR A B C D;
  PUT 'Variable value is ' VAR; PUT 'Variable value is ' VAR;
RUN;
```

A frequent source of confusion for SAS newcomers is the difference between PROC FORMAT and the FORMAT statement itself. Ignoring for the moment the special case of internal (SAS-supplied) formats, we can show the different characteristics of the two statements:

- **FORMAT**
  - Runs outside of the DATA step, completely separately from data set processing
  - Neither uses nor references any data set variables
  - Uses a table of value correspondences that was created earlier by PROC FORMAT
  - Contained within a DATA step
  - Associates a particular DATA step variable with an already existing format table. As a result of this association, when that variable is printed (via PROC PRINT, for example), the "corresponding values," rather than the actual values, will be printed as defined in the format table. In the second part of the code above, for example, the variable VAR is associated with the format ABCD by means of the FORMAT statement within the DATA step. When the DATA step runs, the value printed will be X, not 1, even though the variable itself actually contains a 1.

Many things are worth noting here that are not always immediately obvious to those who are new at SAS formats. First, the two portions of our code fragment in FIGURE I are completely separate. The first portion, the PROC FORMAT, merely creates the value correspondence table. That table can then be associated, via the FORMAT statement within a DATA step, with any variable the programmer wishes. The same format can be (and often is) associated with more than one variable. Finally, the name of the format and the variable need not be at all similar.

The most important thing to note is that it is not necessary (nor is it usually desirable) to run the PROC FORMAT code each and every time the DATA step is run. Creation of the table need only occur once, assuming that the table is stored appropriately in a permanent SAS format library (see the SAS manual appropriate to your operating system). In and of itself, the PROC FORMAT does absolutely nothing for your program; only when you use the format or formats which are thereby created do you reap the benefits.

That table can then be associated, via the FORMAT statement within a DATA step, with any variable the atmosphere is provided. The same format can be (and often is) associated with more than one variable. Finally, the name of the format and the variable need not be at all similar.

Most importantly, it is not necessary (nor is it usually desirable) to run the PROC FORMAT code each and every time the DATA step is run. Creation of the table need only occur once, assuming that the table is stored appropriately in a permanent SAS format library (see the SAS manual appropriate to your operating system). In and of itself, the PROC FORMAT does absolutely nothing for your program; only when you use the format or formats which are thereby created do you reap the benefits.

A format is nothing but a "reencoding engine," a means of getting from one system of values (in our example 1, 2 or 3) to another system of values (X, Y, and Z), in a predictably mapped way. This tutorial is primarily concerned with ways of exploiting the power implied by the presence of this engine.
To touch briefly on permanent (SAS-supplied) formats, these are formats that are internal to the SAS System and that are not generated via PROC FORMAT at all. They can also be associated with DATA step variables and operate in the same "recoding" manner, however, doing such things as enabling the programmer to print an internal SAS date value such as 10/06/85 in a more legible form such as 07/23/87.

The reasons that formats are such powerful tools in the SAS System will become clearer in the next section, as we cover the various kinds of table lookup that can occur using formats. For now, we should also note that much of the rudimentary power of formats stems from the fact that many SAS procedures, such as PROC FREQ, allow the programmer to use the FORMATTED value of the variable, rather than the "pure" value of the variable, when categorizing and printing.

### DATA step recoding of variables (the PUT function)

Before moving on to table lookup, however, one key concept remains to discuss before we can tap the full power of the format tools. If formats were only available to be associated with variables in order to affect how those variables appeared upon output, we would have little cause to consider them as anything special. But SAS provides an important function available in the DATA step, a function that performs the recoding of any variable's value in accordance with any existing format. This function, the PUT function, allows you to have access, within your program logic, to the recorded corresponding value to whatever value is actually contained in your variable. Do not confuse the PUT function with the PUT statement; the PUT function performs recoding only and has nothing at all to do with output.

An elementary example of the PUT function is provided by modifying the DATA step portion of the code fragment in FIGURE 1 as follows (note that this next code still makes use of the format table created in the first portion of FIGURE 1):

```
DATA_NULL;  
VAR = 3;  
LENGTH NEWVALUE $1;  
NEWVALUE = PUT(VAR, ABCD.);  
IF NEWVALUE = "1" THEN  
  PUT 'Hello hello hello';  
ELSE  
  PUT 'goodbye goodbye goodbye';  
RUN;  
```

As exhibited in FIGURE 2, the PUT function always results in a character string, since the right hand side of the equivalences in a format table are always character strings. Here, based on whether the variable VAR is 1, 2, or 3, the character variable NEWVALUE will be assigned the value 'X', 'Y', or 'Z'. That recorded value is then used further in the same "recoding" process as outlined above. Note that no output occurs as a result of the PUT function itself; the PUT statements that occur in the IF/THEN/ELSE chain do the actual outputting.

In most of the subsequent examples in this tutorial, we will focus on the creative use of formats and the PUT function, rather than on how formats affect output.

### III. Making your code table-driven

The basic FORMAT tools can easily be put to work in SAS programs to substitute for structures that normally contain a great deal of hard-coded data. Again, the purpose behind using formats rather than hard-coded data is to separate out the parts of a system (i.e., the constants, the codes, etc.) that are most likely to change. In most systems, the overall system logic needs changing considerably less often than the particular values that a program operates on or that are embedded within the program. If those values (data) are completely separate from the program itself, and especially if they are in easily assimilable list form, changes and additions are much easier.

The SAS format facility provides a simple, bottled way to use lists (tables) of values or value correspondences within the logic of your program, without having to incorporate those lists themselves into the working body of the program. The key concept here is that of separation of the lists (as created by PROC FORMAT) from the program (the DATA steps and PROCs) itself. Again, the PROC FORMAT can run at any time prior to the running of the program, as long as the results of that PROC FORMAT are stored in a permanent format library. Aside from when the table is created, though, the important issue is that the information in the table is no longer embedded in the program itself.

Three kinds of simple table lookup using formats will be discussed: simple recoding of values, categorizing of values, and list checking. All three kinds of table lookup use basically the same code; only the information in the table being used is structured somewhat differently for the different purposes.

#### Recoding

Recoding using SAS formats is simply a way of tapping into the value correspondence list created when the PROC FORMAT was run that defined the correspondences. You need to recode when you have a value in one form (e.g., numeric code number) and you wish to "get" (into your program, perhaps into another variable) a different representation of that same value. In short, you are simply converting between one form and another, on a one-to-one basis.

The power of the SAS format table lookup technique becomes most clear when we look at the equivalent "hard-coded" approach to accomplish the same task. Our example involves a set of branch offices for a national corporation. Each branch is represented via a four-digit code number within the various corporate data bases. In our SAS program, we wish to obtain the branch name for the branch number contained in each observation. Of course, we could sort our primary data set (by branch number) and perform a MERGE with a SAS data set that contains the branch number and the branch name. However, this solution is not always practical, especially when the primary data set is large and these sorts of simple table lookups are plentiful. Faced with this situation, many SAS programmers would thus code something like the code in FIGURE 3:
Depending on the number of branches, this approach will use many (sometimes even hundreds of) lines of code to accomplish the table lookup, turning what is basically one logical operation (i.e., find the name of this branch from the branch number) into a major portion of the program. When the programmer (or his forlorn successor) reads the program in order to maintain or enhance it, these lines become a major distraction and impediment to understanding, due to their size alone. Moreover, the odds of making an error when typing or modifying boring, repetitive code are quite high. And who is to say that the list constructed by one programmer is identical to the list constructed by her colleague?

Recoding using formats

The solution obtained using SAS format techniques (see FIGURE 4) may at first glance look no shorter than that of the hard-coded approach. However, in this case the list of values is separated from the rest of the program logic. In fact, the list may be in a separate "job" or computer run altogether; for production lists such as branch numbers, it definitely should be separated, so you would not see the PROC FORMAT in the same listing as the DATA steps that use the format.

In this latter example, the recoding from one form to another (a single logical operation) takes up a single line of code, no matter how many branches we have to deal with. If branches should be added, deleted, or renamed, the program logic does not have to be changed at all. Only the table (the PROC FORMAT code) needs to be modified, and that kind of maintenance typically is less error-prone and does not require a high-level programmer.

All of the subsequent examples will show quite analogous code, since the basic technique of combining SAS formats with the use of the PUT function underlies all of them. And all of them replace equivalently contorted IF/THEN/ELSE chains that accomplish the same goal at the cost of embedding data into the body of the program itself.

Categorizing using formats

Categorizing is one of the most powerful ways of using advanced SAS format techniques, and it is nothing more than a variation on simple recoding. Using our same example with branch numbers: imagine that these branches are grouped by the company into different regions. Some branches are in the Eastern region, some in the Southern region, and so on. Obviously, an IF/THEN/ELSE chain could easily accomplish the necessary categorizing, but then there will be data in the program. GET THE DATA OUT. The format solution eliminates the data from the program itself:

There is only one real difference between FIGURE 5 and our simple recoding example: the format table in FIGURE 5 contains lines that have the same value (e.g., "WEST") on the right hand side. In other words, formats can map a "many to one" relationship as easily as they can map a simple "one to one" relationship. Note that we are still using the variable BRANCH as the value that is being recoded. Depending on the format being used ($BRNAME vs. $B_REGION), the program can obtain the branch name or the region for the branch.

Slipping back for a moment into the realm of everyday (i.e., output-oriented) format use, we can make use of this same kind of categorizing facility provided by formats combined with certain SAS procedures such as PROC FREQ. For example, if we wanted to see how many branches were in each region, we could use the same $B_REGION format associated with the BRANCH variable within the PROC FREQ.
in either case, the format itself is doing all the categorization work, leaving the programmer to contemplate more important things.

**Categorization with numeric results**

Normally, as stated before, the result of any PROC FORMAT table lookup is a character string, since the right hand side of any PROC FORMAT table must be a character string. However, it is possible to do categorization with numeric results, using one more trick (the INPUT function) provided by the SAS Function library, or by relying on automatic conversion performed by the SAS system.

This example will also illustrate the use of ranges in the PROC FORMAT table, which are in themselves ways of categorizing. The example PROC FORMAT table arises from a city tax application in which a percentage (of some permeability, if it matters) is available in each observation. Based on the range in which the particular percentage falls, a particular dollar amount needs to be assigned to that observation. That dollar amount happens to represent the tax rate to be applied for that customer.

Again, this requirement would ordinarily be coded as an IF/THEN/ELSE chain, with the dollar values hard-coded into the program. But tax rates aren't code, they're data! GET THE DATA OUT.

The PROC FORMAT table created to handle the current list of rates appears in FIGURE 7, followed by the single line of SAS code which is required to obtain the dollar amount.

```sas
CATEGORIZING WITH NUMERIC RESULTS
proc format library=Library;
  value rates
    0.000 - 0.150 = 'Low' 
    0.150 - 0.350 = 'Medium' 
    0.350 - 0.650 = 'High' 
    HIGH = 'High' 
  run;
In your DATA steps:
  RATE = input(PCT, RATES., 6.2);
FIGURE 7
```

Here, the program performs the table lookup on the percentage for the current observation, available in the variable PCT. The result of this lookup is a character string, as always. The INPUT function, however, converts that character string back to a number, which can then be stored in the numeric variable RATE. Of course, you could instead rely on the SAS system to convert automatically between character and numeric in this instance. In general, it is recommended that explicit conversions be performed as shown above, rather than relying on defaults.

**List checking**

It may not be obvious from the previous examples that format table lookup can be exploited in ways that transcend recording. Most notably, a format table can be used in a program just as a list of entries that have a specific attribute. Rather than having to hard-code this list in the program, or perform arduous sorts and merges to check values against a list, using a format table combined with a single use of the PUT function determines whether the value is present in the list.

For example, imagine that a record collector has created a SAS-based system to organize her music collection. In one of the associated programs, she wishes to determine whether the artist for a particular disc is contained in a list of music certified as "OK". To do this, a format called $OKMUSIC can be created that contains only acceptable artists.

```sas
LIST CHECKING USING FORMATS
proc format library=Library;
  value SOMMATIC
    'Beatles' = 'Y'
    'Beethoven' = 'Y'
    'Michael Jackson' = 'Y'
    'Elvis Presley' = 'Y'
    'The Weeknd' = 'Y'
    (etc.) = 'N'
  run;
In your DATA steps:
  IF ARTIST='Julie Andrews' THEN PCTL="OK";
FIGURE 8
```

In this example, the right hand side of the format table is not really used at all, although it certainly could be if more information (e.g., music category) were required for each artist in the list. Note: if you attempt a PUT function on a value that is not in the format that is referenced, you get the value itself. For example, if we did not have the OTHER = 'N' in the format in FIGURE 8, and we attempted to perform the PUT function on ARTIST='Julie Andrews', the result would be 'Julie Andrews'. The OTHERWISE clause nicely prevents confusion by forcing all cases to be included in the recoding scheme in a consistent way.

**Comparative CPU Performance**

A natural question that arises when these techniques are presented is about their relative performance versus standard (i.e., hard-coded) methods. The answer: in general, execution time alone, these methods are not quite as quick as the hard-coded equivalents. The reader is referred to the bibliography for more information on specific benchmarks (Lutomski and Mullally, 1988).

The obvious rejoinder to quibbles about relative performance focuses on issues of readability and maintainability. It does not help to have a program run in 10 minutes instead of 13 minutes if you need to spend eight extra hours chasing a bug that has emerged because the program relies on hard-coded data and is thus larger and less-readable. Overall, the debugging and maintenance phases represent a huge part of the software lifecycle, and it is often (although most assuredly not always) worth sacrificing a small amount of machine performance in order to facilitate them.
IV. Multi-variable Table Lookup

The power of SAS formats and the general table lookup technique which they allow can be taken to even greater extents than in the examples shown above. In the prior examples, all of the value correspondences are fairly simple ones, where a single variable has a single corresponding value. It is an easy extension of the technique to incorporate multiple variables corresponding to a single value, or single variables corresponding to multiple values.

What you are about to see is another way of significantly slashing the amount that data are embedded in programs. However, this expanded use of advanced formatting, although based on the same simple techniques shown earlier, can be a bit confusing, especially to less experienced SAS programmers, and should thus be used in moderation.

Again, the technique is most discernible through an example. An electric utility needs to classify as 'summer' or 'winter' the electricity consumed on a particular day by a particular customer. However, different definitions of what constitutes 'summer' and 'winter' apply to different groups of customers, based upon their 'serial' or when their meter is typically read each month. Customers read early in the month, for example (serial='W'), might have winter (1984-85) classified as December 3rd through May 30th. A customer read somewhat later in the month (serial='C') might have winter as December 4th through May 30th. In short, whether a particular day of electricity consumption is summer or winter depends both on the date and on the serial of the particular customer.

This example assumes, of course, that sorting the primary data set by individual date is not possible or not practical. Format techniques are used to substitute for more conventional methods when needed.

The hard-coded alternative for this quite sticky scenario is especially ugly and unwieldy. A fragment of it is shown in FIGURE 9. Please look at it imagining that you need to maintain it, adding new years as appropriate, correcting old information, and so on.

This code isn't code at all -- it's data, right there in your program. GET THE DATA OUT.

Multi-variable table lookup using formats

Once again, the use of formats with the PUT function can come to the rescue, this time in slightly more involved coding:

```
proc format library=library;
   value SEASON=
   'B941203' = 'S'
   'B941203' = 'W'
   'B951203' = 'W'
   'B951203' = 'W'
   'C841203' = 'S'
   'C861203' = 'S'
   'C851254' = 'S'
   'C851254' = 'S'
   'C861206' = 'S'
   'C861206' = 'S'
   'C871210' = 'S'
   'C871210' = 'S'
   end;

data your step;
   length string $7;
   string = serial || put(date, yymmd6.);
   season = put(string, SEASON=);
   run;
```

FIGURE 10

This code takes a bit more explanation. Basically, each format range (to the left of the equal sign in the PROC FORMAT) is composed by splicing two things together: the serial ('B', 'C', etc.) and the end dates of the appropriate season for that serial. (Since entries in the format table cannot overlap, the dates within the values must be in sortable order).

Moving on to the DATA step, then, if we know a particular date and a particular serial, it is easy to build, via character string concatenation, a similar string, composed of the serial in the first character and the date, in MMDDYY6 form. Then, a simple PUT function call, invoking the format established in the PROC FORMAT call, will obtain whether that date and that serial are summer or winter. The format itself, therefore, contains ranges for which the corresponding value ('S' or 'W') applies; however, the format is applied to one date and serial combination at a time.

Clearly, you are not limited just to two values spliced together in the format ranges; you can have as many characters as SAS will allow on the left side of the equal sign (currently 16). Of course, the table lookup in this more advanced case also involves concatenating the value to be looked up, through concatenating the different values.

A suggestion for SAS Institute

The suggestion that could further increase the power of the SAS System in applications that use these techniques would involve extending the capabilities of the PUT function, so that it accepts the name of a character string variable in place of the format name. The contents of that character string variable would then contain the format, rather than the format needing to be hard-coded within the program. This capability would be most useful if you need to apply different formats entirely based upon information contained within the current observation. For example, you might have totally different formats for different years, and you would want to control programmatically which format is applied.
V. Building FORMAT library entries

When format tables get extremely long, or when they involve more than one variable in the ranges as in the example in FIGURE 10, it may be desirable to maintain the list as a SAS data set, rather than in PROC FORMAT text format. You then need to construct the SAS format from the contents of that data set, in some kind of automated way.

Techniques for doing this have been presented at other SUGI conferences (see Bibliography). These techniques involve various work-arounds that were required prior to SAS Version 6.03, where the process of creating and manipulating formats has been vastly eased. Prior to 6.03, in order to create PROC FORMAT code from the values contained in a DATA step, you needed to use a SAS DATA step to write the SAS code, either via the macro facility or out to a flat file which was then included as source code to execute the PROC FORMAT.

Version 6.03 brought several new options to PROC FORMAT that greatly facilitate the creation of formats directly from values contained in SAS data sets, as well as the obtaining of SAS data sets from existing SAS formats. Using the CNTLIN option on PROC FORMAT and referencing a SAS data set with appropriately named variables, building a SAS format is simple and automatic.

VI. Accessing FORMAT library entries

Just as you may wish to create format library entries from a SAS data set, you may also wish to obtain a SAS data set from the formats contained in an already created format library. You can then use the resulting data set in all ordinary SAS processing.

Prior to version 6.03, the SUGI supplemental library procedure named PROC FMTLIB provided the capability to obtain a SAS data set from a format library. For example,

DIRECT CREATION OF A FORMAT TABLE (V. 6.03)

**FIGURE 11**

In FIGURE 11, the referenced data set VALUES needs to contain one observation per format table entry, with the variable FMTNAME containing the name of the format to be created. START and END are the end points of the ranges on the left hand side of the format table, and LABEL is the corresponding value that is usually on the right hand side of the format table. For a more complete list of options, consult your SAS 6.03 manual.

VI. Miscellaneous other applications

The implications of the techniques presented in this paper are quite far-reaching. Countless applications can be found for getting the data out of your program using format table lookup.

One example that can be pursued (Weiler, 1989) is how to use SAS formats, combined with macro variables and Screen Control Language, to dynamically define screens in SAS/AF. Whatever can be placed in a table can be used to drive a system, ranging from data manipulation to actual system definition. One large applications system actually used format tables to define system flow, where table lookup methods were used to determine the next module to execute.

VII. Conclusion

SAS formats, particularly with respect to their flexibility in table lookup, have too long been ignored as a sophisticated tool for SAS programmers. In my experience, advanced SAS programmers tend to gravitate to macros (and rightly so) as a great source of coding power. Systems emerge where everything is macroized, sometimes to the detriment of maintainability (Kretzman, 1986). When those same programmers then see the true power of formats, particularly in making systems table-driven, formats become their favorite aspect of the SAS language, and countless instances are then found where the use of formats create tighter, more concise, more maintainable systems.
Questions and comments are very welcome. Please contact
the author at:

Peter Kretzman
BR Associates, Inc.
2323 Eastlake Ave. E.
Seattle, WA 98102
Voice: (206) 328-0100
FAX: (206) 324-8886

The author may also be contacted at the Sea-SAS electronic
bulletin board service (BBS), by setting your modem to
N/8/I and dialing:
Sea-SAS: (206) 324-9116
Hours 10 p.m. to 4 p.m. Pacific time

VIII. Bibliography

1. German, Hallett, "Demystifying PROC
   1262-1267.
2. Horwitz, Lisa, and Thompson, Tim, "Efficient
   Programming with the SAS System for Personal
   Computers: Save Time, Save Space", SUGI 13
   Proceedings, 1988, pp. 625-630.
3. Kretzman, Peter, "Maintainable SAS-based
   systems and the SAS Macro Language: a Catch-
4. Lutomski, Leonard S., and Mullally, Henry A.,
   "The Comparative Efficiency of Table Lookup
   645-653.
5. Ray, Craig, "A Comparison of Table Lookup
   44-50.
6. SAS Institute, Inc., SUGI Supplementary
7. SAS Institute, Inc., SAS Applications Guide,
8. Weiler, Louise, "Dynamic Screen Generation
   Using Formats: the Case Against Hard-

IX. Acknowledgements

I wish to acknowledge the valuable contributions and
insights of those who participated in the evolution,
implementation, and testing (in real world code) of the
techniques discussed in this paper. Thanks to Judy
Alicante, Don BeLow, Steve Marino, Stan Sibley, and
Louise Weiler.

NOTE:
SAS and SAS/AF are registered trademarks of SAS
Institute Inc., Cary, NC, USA.