# PROGRAMMING FOR JOB SECURITY REVISITED:

# EVEN MORE TIPS AND TECHNIQUES TO MAXIMIZE YOUR INDISPENSABILITY

Arthur L. Carpenter
California Occidental Consultants

Tony Payne
Software Product Services, Ltd.

## KEY WORDS

STYLE, TECHNIQUE, SECURITY, NAMING CONVENTIONS

## ABSTRACT

A great deal has been said about programming techniques for efficiency and maintainability of SAS® programs. We are taught to write code that minimizes machine and programmer resources. Unfortunately, easily maintained code requires fewer programmers, and fewer programmers means pink slips and less job security. In these troubled times, programmers need to be able to maximize their indispensability.

Programmers can increase their job security and thereby protect themselves and their families by applying the tips and techniques discussed within this paper. The programmer will be advised when to apply the techniques, and whether the use of the techniques should be subtle or gross.

Techniques will cover programming style, editing style, statements to use and avoid, and naming conventions. You will learn to blur data steps, make non-assigning assignment statements, and in general write code that not even you will be able to figure out how or why it works.

All the techniques discussed in this paper have been field tested by the authors and other SAS programming professionals. Portions of this paper are based on a similar "Best Contributed Paper" presented at SUGI 18 (Carpenter, 1993), SUGI 21 (Carpenter, 1996), and SEUGI 15. It has also been presented at WUSS, SUGISA (South Africa) and Views (UK).

## INTRODUCTION

*"Don't be irreplaceable, if you can't be replaced, you can't be promoted."* Dilbert's Laws of Work

General rules for writing SAS programs have been suggested and promoted for a number of years. When followed, the suggested rules create efficient programs that are easily maintained by either the author or by other programmers. Obviously, the primary beneficiaries of well written code are the stockholders of the company that runs and maintains the programs. In a well ordered and polite society the programmer also receives benefits such as an hourly wage, self satisfaction and an occasional pat on the back.

Since well written code is easy to maintain, flaws in the logic or bugs in the code can be easily ferreted out using the documentation and by understanding the appropriate section of the program. Unfortunately this means that the original programmer may not even be necessary to the process; indeed the original programmer can be freed to work on other tasks that require the special talents of a programmer. However as you carefully code your programs, keep in mind that in general, easily maintained programs require fewer programmers and one of the fewer programmers could be you.

Fortunately it is possible to write programs that no one else could possibly maintain. Programs can be written that produce results that cannot be predicted from either a quick or even fairly careful inspection of the code. Once you know these techniques and have learned to properly apply them, your job will be secure for as long as your programs are in use.

Although it is possible to use the described techniques to write a totally indecipherable program that works, often the subtle application of only one or two techniques in an otherwise ordinary program, will achieve the same results. The selection of a technique and its application to the program is an art form that can be achieved with practice and perseverance.

## PROGRAMMING STYLE

*"Eat one live toad first thing in the morning and nothing worse will happen to you the rest of the day."* Dilbert's Laws of Work

Programming style refers to the general approach that a programmer takes when designing and coding programs. Although most programmers develop their own unique programming style, specific guidelines are often imposed by the employer or the client. In the absence of specific guidelines, consider incorporating some of the following.

✱ DESIGNING THE CODING DESIGN

✔ Avoid the logical separation of tasks, e.g. separate tasks only if it is logical to keep them together.

✔ Avoid structured programming approaches.

✔ Nest function calls whenever possible. Exceeding a depth of three provides added complexity. The following statement only nests three deep and therefore lacks necessary complexity.

```
depth2 = input(substr(station,
       index(station,'-')+1),3.);
```

✔ Use functions when they are not required.

```
* DATE is a SAS date;
date=
mdy(month(date), day(date),
year(date));

* NAME never contains a comma;
name=
substr(name,index(name,',')+1,
       length(name));
```

✔ Write code to replace functions entirely. The ABS function becomes:

```
if a < 0 then b = a*-1;
else b = a;
```

✔ Use non-standard statement structures. A statement to average two positive numbers could be written as:

```
ab = (a*(a>0) +
b*(b>0))/((a>0)+(b>0));
```

✔ Nest macro calls and definitions. The addition of macros that call macros that define macros that call macros will always add a nice touch.

✔ When using macro variables the use of %LOCAL and %GLOBAL definitions allows you create symbolic variables that have more than one definition at any given time. The following macros print two different values for &AA. Resetting &AA in the macro %INSIDE does not change its value in the macro %OUTSIDE.

```
%macro inside(aa);
   %put inside &aa;
%mend inside;

%macro outside;
   %let aa = 5;
   %inside(3)
   %put outside &aa;
%mend outside;
%outside
```

This prints:

```
inside 3
outside 5
```

✔ Maximize the number of steps and lines of code. The following PROC SQL is too compact and should be rewritten.

```
proc sql;
create table report as
select * from sales
having saleprce gt mean(saleprce)
group by region;
```

Fortunately the single step PROC SQL can be replaced with:

```
proc sort data=sales;
by region;

proc summary data=sales nway;
by region;
var saleprce;
output out=stats mean=meansale;

data report;
merge stats sales;
by region;
if saleprce gt meansale;
```

✱ ASSUME THE ASSUMPTIONS OF OTHERS

Most programmers make assumptions about the code they are reading; the objective is to key in on those assumptions.

✔ Use LENGTH to assign the same variable different lengths in different data sets. This is especially useful if that variable is used in the BY statement during a MERGE.

More subtle is to define the length of the variables without using the LENGTH statement.

✔ Variables with the same names could be numeric in one data set and character in another. This is most useful when the variables are flags that take on the values of numbers.

✱ ELIMINATE, HIDE (LOSE) SOURCE CODE

Code that is compiled no longer depends on the original source code (except when modifications are required) and this give us an opportunity to hide the source code.

✔ After compilation eliminate, rename, or change:

the SCL source code used with AF or FSP applications

compiled DATA steps

compiled stored macros

DATA step views

SQL views (although the source can still be recovered by using the DESCRIBE option).

✔ When editing SCL for SAS/AF® or SAS/FSP® applications change the color of the code to match the background color of the editor.

✱ USE THE DATA STEP EFFECTIVELY

The DATA step offers a number of opportunities to make our programs more effective.

✔ Use multiple steps when one would suffice.

The data set TWO could have been created in a single step.

```
data one;
set master;
actual=saleprce+tax;
data two;
set one;
profit = actual-budget;
```

The following PROC SORT should be rewritten using a completely unnecessary DATA step.

```
proc sort data=old out=new;
by date;
```

With the DATA step this becomes:

```
data new;
set old;
proc sort data=new;
by date;
```

✔ Create data sets that are never used or are used unnecessarily.

✔ Use implicit naming rules for data sets. This type of code is also very susceptible to minor problems that can cause major crashes.

```
data;set master;
actual=saleprce+tax;
proc print;
data;set;
profit=actual-budget;
proc means noprint;
output;
proc print;
run;
```

✔ Data set names can be reused thus preventing another programmer from getting overly familiar with the variables that they contain. The previous example becomes:

```
data temp;set master;
actual=saleprce+tax;
proc print data=temp;
data temp;set temp;
profit=actual-budget;
proc means data=temp noprint;
output out=temp;
proc print data=temp;
run;
```

✱ USING SYSTEM OPTIONS

Several of the system options provide special programming opportunities.

✔ Debugging aids can be turned off by using NOSOURCE, NOSOURCE2, NONOTES, and NOLABEL.

✔ When turning off debugging aids, prevent full discovery by turning them off in several places.

✔ When using macros, the use of the macro debugging options MPRINT, MLOGIC, SYMBOLGEN should be avoided.

✔ Some options remove your ability to do things that we take for granted *e.g.* NOMACRO.

✔ You can control the both the number of observations in a data set using the OBS= option. The FIRSTOBS= and LASTOBS=options tell SAS when to start and stop processing SAS data sets. Change these options and don't tell.

✔ Reroute the WORK or temporary data sets to another location by using the USER= option. In the following DATA step NEW is actually SASUSER.NEW.

```
options user=sasuser;
data new;
set project.master;
```

As an added bonus NEW will not be erased at the end of the SAS job. This can cause disk and clutter problems.

Changing the USER= option several times within a program, makes "WORK" files a bit tough to find.

✔ The ERRORABEND option is designed for batch programming and when an error occurs the job immediately will abend. When used in an interactive session, the slightest error causes the end of the session and the LOG will not even be available to help determine why the session ended.

✔ In preparation for the millennium set YEARCUTOFF=1800. The variable YEAR in the following step will have a value of 1897.

```
* Hide this option statement;
options yearcutoff=1800;
data a;
date = '23mar98'd;
year = year(date);
```

✔ The S= option limits the number of columns read from the program source. Only the first 10 columns are used in the following DATA step.

```
options s=10;
data new;
set olddata
    master
    adj;
profit =
   sales + tax;
cnt+1;
```

The LOG will show (unless you use NOSOURCE) that the data set OLDDAT was used instead of OLDDATA and that the variable TAX is never used. Also the SUM statement (CNT+1) becomes part of the assignment statement that is used to create PROFIT.

```
options s=10;
data new;
set olddat
    master
    adj;
profit =
   sales +
cnt+1;
```

✔ The CAPS option can be used to change how literals are compared. The data set NEW will have zero observations.

```
options nocaps;
data old;
  x='a';
  run;
options caps;
data new;
  set old;
  if x='a';
  run;
```

## EDITING STYLE

*"If you can't get your work done in the first 24 hours, work nights."* Dilbert's Laws of Work

Most programmers will develop a style that they use to code their programs. Although there is merit to the argument that no style is the best style, there are also specific techniques that you should consider.

✱ CODE LAYOUT

The layout of the program as seen by the programmer is determined by the programmer's editing style. Once again, in the absence of specified guidelines, consider these to improve the unreadableness of your code. The following DATA step can be fixed using several different techniques.

```
data sasclass.biomass;
infile rawdat missover;
input  @1 STATION $
       @12 DATE DATE7.
       @20 BMPOLY
       @25 BMCRUS
       @31 BMMOL
       @36 BMOTHR
       @41 BMTOTL ;
format date date7.;
label BMCRUS = 'CRUSTACEAN BIOMASS'
      BMMOL  = 'MOLLUSC BIOMASS'
      BMOTHR = 'OTHER BIOMASS'
      BMPOLY = 'POLYCHAETE BIOMASS'
      BMTOTL = 'TOTAL BIOMASS'
      DATE   = 'DATE'
      STATION  = 'STATION ID';
run;
```

✔ Never indent.

```
data sasclass.biomass;
infile rawdat missover;
input  @1 STATION $
@12 DATE DATE7.
@20 BMPOLY
@25 BMCRUS
@31 BMMOL
@36 BMOTHR
@41 BMTOTL ;
format date date7.;
label BMCRUS = 'CRUSTACEAN BIOMASS'
BMMOL  = 'MOLLUSC BIOMASS'
BMOTHR = 'OTHER BIOMASS'
BMPOLY = 'POLYCHAETE BIOMASS'
BMTOTL = 'TOTAL BIOMASS'
DATE   = 'DATE'
STATION  = 'STATION ID';
run;
```

✔ Use multiple statements per logical line.

✔ Break statements in the middle.

```
data sasclass.biomass;infile rawdat
missover;
input  @1 STATION $ @12 DATE DATE7.
@20 BMPOLY @25 BMCRUS @31 BMMOL @36
BMOTHR @41 BMTOTL ; format date
date7.;label
BMCRUS = 'CRUSTACEAN BIOMASS'
BMMOL  = 'MOLLUSC BIOMASS'
BMOTHR = 'OTHER BIOMASS'
BMPOLY = 'POLYCHAETE BIOMASS'
BMTOTL = 'TOTAL BIOMASS'
DATE   = 'DATE'
STATION  = 'STATION ID';
run;
```

As an added bonus notice that the LABEL assignments look a bit like assignment statements.

✔ Reform your code using the TextFlow program editor prefix option.

```
data sasclass.biomass;infile rawdat
missover; input @1 STATION $ @12
DATE DATE7.  @20 BMPOLY @25 BMCRUS
@31 BMMOL @36 BMOTHR @41 BMTOTL
; format date date7.;label BMCRUS =
 'CRUSTACEAN BIOMASS' BMMOL =
'MOLLUSC BIOMASS' BMOTHR =
'OTHER BIOMASS' BMPOLY = 'POLYCHAETE
BIOMASS' BMTOTL = 'TOTAL BIOMASS'
DATE = 'DATE' STATION = 'STATION ID'
; run;
```

✔ Form your code to make pictures. The company logo would be a logical choice.

```
                    data
             sasclass.biomass;
       infile     cards   missover;
  input @1                 STATION $
  @12 DATE                    DATE7.
 @20                           BMPOLY
@25        BMCRUS    @31         BMMOL
@36        BMOTHR    @41        BMTOTL
;                               format
 date                           date7.
   ;label                   BMCRUS=
         'CRUSTACEAN BIOMASS'
                BMMOL=
'MOLLUSC BIOMASS' BMOTHR='OTHER BIOMASS'
BMPOLY=  'POLYCHAETE BIOMASS'   BMTOTL=
            'TOTAL BIOMASS'
               DATE='DATE'
                STATION
                   =
              'STATION ID';
                 run;
```

✱ USE EDITOR COLUMNS > 80

Occasionally use columns > 80 in the editor (these columns are not usually visible while editing e.g. PROGRAM WINDOW in the DISPLAY MANAGER).

✔ Place key variables out of sight.

```
                                80
data newdata;
set olddata (drop=name      | fname
      address city state); |
```

✔ Use the asterisk to comment out key formulas or statements.

```
data new; set old;          | *
  wt = wt/2.2;              |
```

✔ Place special equations entirely out of sight.

```
data new; set old;          | x=x+5;
```

✔ Place special equations partially out of sight.

```
data new; set old;          |
degc = (degf                | +5
          -32)*5/9;         |
```

✱ JUDICIOUS USE OF COMMENTS

Comments are only useful when they are used correctly and with discretion.

✔ The following comments contain an executable PROC MEANS.

```
* The comments in this section do more ;
* than it seems                       ;
*                                     ;
* modify data to prep for; proc means  ;
* after adjusting the data using      ;
* the; var for weight                 ;
```

✔ The /* ... */ style comments can be used to mask or unmask code by taking advantage of the fact that this style of comment cannot be nested.

The first comment is accidentally not completed, thus commenting out the DATA step.

```
/* ****************
* Apply the
* ***very ***
* important adjustment;
data yearly;
set yearly;
income = income*adjust;
run;

/* Plot the adjusted income */
proc gplot data=yearly......
..... code not shown .....
```

An embedded comment can be used to cause a portion of the "removed" code to be executed.

```
/* ****************
REMOVE FOR PRODUCTION
proc print data=big obs=25;
title1 'Test print of BIG';
var company dept mgr /*clerk*/;
data big;
set big;
if name='me' then salary=salary+5;
*END OF REMOVED SECTION;
***************** */
* Next production step;
..... code not shown .....
```

## CHOICE OF STATEMENTS

*"To err is human, to forgive is not our policy."* Dilbert's Laws of Work

Some statements when used properly have the capacity to add several layers of complexity to a SAS program. Others should be avoided as they tend to reduce confusion.

### ✱ STATEMENTS TO AVOID

Avoid statements that tend to allow the programmer to retain less mental information.

✔ Always maintain extra variables in the Program Data Vector (PDV). Avoid the use of the KEEP and DROP statements.

✔ When variables must be eliminated from the PDV use the DROP statement. The KEEP statement lets the programmer know what variables remain in the PDV, while the DROP statement only reveals what was eliminated.

✔ Comments are to be avoided unless used in the ways mentioned above.

✔ RUN and QUIT statements are rarely required, and usually only serve to reduce program complexity by separating steps.

### ✱ STATEMENTS TO USE

Several statements and combinations of statements can be used to advantage.

✔ Statement style macros can be used to redefine variable relationships.

```
* hide this definition;
%macro sat (name) / stmt;
  set &name;
  wt = wt + 5;
%mend sat;

* Why is the value of WT always 6 in
* the data set NEW?;
data old;wt=1;output;
data new;sat old;
```

✔ Implicit arrays are preferred to explicit arrays.

✔ Define arrays using the names of other arrays. This technique was used on purpose prior to the advent of multi-dimensioned explicit arrays. Consider the following three implicit arrays.

```
array a (I) wt ht;
array b (I) xwt xht;
array c (j) a b;
x = c;
```

For I=1 and J=2 the variable X will be assigned the value contained in XWT.

✔ The GOTO statement can be used to breakup program structure; use it liberally.

✔ The label statement can be disguised.

SAS statement key words make excellent labels.

```
do: I=1;
```

Labels can be hidden in comments.

```
* did you notice that, this comment
* contains a; label:
```

✔ Data steps with multiple SET statements or better yet SET and MERGE statements add complexity quickly, but not quietly.

✔ Macro quoting functions can be used to prevent the resolution of macro variables resulting in FALSE comparisons that are obviously TRUE. The macro %DOIT quotes &CITY so that it can not be resolved.

```
%macro doit(city);
  %put &city;
  %* Hide the following statement;
  %let city=%nrstr(&city);
  %put &city;
  %if &city = LA %then
        %put CITY is LOS ANGELES;
  %else %put city is not LA;
%mend doit;
```

When the macro is called with LA as the parameter:

```
%doit(LA)
```

The LOG will show:

```
LA
&city
city is not LA
```

&CITY is not resolved so it will NEVER be equal to LA.

## NAMING CONVENTIONS

*"You can go anywhere you want if you look serious and carry a clipboard."* Dilbert's Laws of Work

Naming conventions are perhaps the most useful tool in the arsenal of the Job Security specialist. A novice may think that no naming conventions or the random selection of names will create the most secure program. In actuality there are a number of subtle and not so subtle techniques which can be applied by the sophisticated programmer. These rules can be applied to both variables and data sets, and come from the schools of confusion, misdirection, and inconsistency.

### ✱ CONFUSION

Create names that have no mental references or that make memorization difficult.

✔ Use meaningful names during the debugging process, then when the program is working use the editor CHANGE command to convert variable names:

```
===> c 'age' 'qwrtxzqr' all
```

✔ Use 8 digit names when possible.

✔ Avoid the use of vowels, include subtle variations.

```
QWRTXZQR, QWRTZXQR, QWRZTXQR
```

✔ Some letters go well together.

```
H & I   HHHIIHIH HHIHIHIH

V & W   WVWVWVVW WVWWVWVW

l & 1   testnum1 testnuml
(number 1 and lower case L)

0 & O   test0001 testOOO1
(number zero and letter O)

Z & 2   QWRTZXQR QWRT2XQR
```

✔ SAS statement keywords are not reserved. The following DATA step does NOT contain a DO loop.

```
DATA SET; SET DATA;
DO = 5+ TO -15;
```

✔ SAS Version 7 will lift the 8 character restriction on names - this will give the Job Security expert additional opportunities to utilize confusion.

### ✱MISDIRECTION

When the name or use of a variable or data set has an obvious (implied) meaning and is then used for something else entirely the reader may be caught unawares.

✔ Obvious names work well for other purposes.

```
SEX    ===> number of fish caught

WEIGHT ===> height of patient

INCHES ===> height in centimeters
```

✔ Placement of observations and data set names.

```
IF SEX = 'MALES' THEN OUTPUT FEMALES;
```

✔ The LABEL statement can be used to provide information to the user. For instance consider the variable SEX which contains the number of fish caught, then:

```
LABEL sex = 'Sex of the Patient';
```

✔ Users of WINDOWS may wish to start SAS sessions using a WORD icon.

### ✱ INCONSISTENCY

Subtle inconsistencies are very difficult to detect and can be very useful, especially in a program that has what seem to be clearly defined parameters.

✔ YES/NO variables should take on the values of YES=0 and NO=1 (of course never Y & N), except somewhere for some variable that has YES=1 and NO=0.

A useful variation has `ANSWER='N'` when the response is YES and `ANSWER='Y'` when the response is NO.

✔ Variable and data set names should not have unique definitions throughout the program.

✔ Variables should on occasion disappear and reappear later with different definitions.

## BLURRING THE DATA STEP

*"If you are good, you will be assigned all the work. If you are really good, you will get out of it."* Dilbert's Laws of Work

The SAS supervisor recognizes the end of steps by detecting a RUN, QUIT, or the start of the next step. We already know not to use the RUN or QUIT, so all we need to do to blur two steps into one is to hide the start of the second step.

### ✱ USING COMMENTS

Once in a great while the use of comments can be forgiven.

✔ The data set NEW in this example will have the variable x=zzstuff as read from the data set GUDSTUFF. Notice that the second comment has no semicolon, hence the data set SECOND is never replaced. In the single DATA step below the value of Y in OLD is effectively never used.

```
* start of the step;
data new ; set old;
x = 5* y;

* this starts the second step
data second; set gudstuff;
x= zzstuff;
```

This data step has two SET statements which is usually good for a few laughs all by itself.

Adding a colon instead of a semicolon to close the comment is even harder to find. The second comment in the above example becomes:

```
* this starts the second step:
data second; set gudstuff;
```

### ✱ USING UNBALANCED QUOTES LEGALLY

Usually unbalanced quotes cause syntax errors, but when two strings close together both have missing quotes they can cancel each other out. This can leave interesting and syntactically correct code.

✔ The unbalanced quote for the variable NAME completely masks the creation of the data set SECOND and the use of the data set GUDSTUFF.

```
* start of the step;
data new;y=5;frankwt=0;x=5*y;
length name $6;
name='weight;
data second;set gudstuff;
*for weight use Franks';
x=frankwt;
proc print;run;
```

In this example the variable X in NEW will always be zero (not 25). Since GUDSTUFF is never read the value of FRANKWT in GUDSTUFF makes no difference.

✔ Unbalanced quotes can also be used in LABEL statements and to mask the end and beginning of macro definitions.

## ON THE ROUGH SIDE

*"It doesn't matter what you do, it only matters what you say you've done and what you say you're going to do."* Dilbert's Laws of Work

This section contains an eclectic set of what may be rather extreme measures and should not be used by those with gentle dispositions or a sense of professional integrity.

### ✱ LEARNING MORE (than the next guy)

Keeping current in the Job Security industry is difficult. Consider these additional sources of informtion.

✔ Virgile (1996) discusses the behavior of SAS under interesting conditions. All you need is some imagination.

✔ You may want to use the SASNOTES to find system 'features' that can be exploited. An example might be the SORT options NODUPKEY and NODUPLICATES which under some circumstances do not create the data sets with the anticipated subset of the observations.

### ✱ IN THE DATA STEP

✔ Assignment statements that change the values of BY variables in a merge can be used to promote interesting combinations of the resulting data.

✔ When merging data sets, variables that are in both incoming data sets (but not on the BY statement) can have interesting properties especially if the number of

observations within the BY group is different for each data set.

✔ The POINT and NOBS options on the SET statement do not perform as you might expect when the incoming data set contains deleted observations.

```
data a;
do i = 1 to 5;
output;
end;
run;

* Use FSEDIT to delete ;
* the second obs;
* (I=2);
proc fsedit data=a;
run;

data b;
do point=1 to nobs;
  set a point=point nobs=nobs;
  output;
end;
stop;
run;

proc print data=b;
title 'Obs = 2 was deleted';
run;
```

The OUTPUT window shows:

```
Obs = 2 was deleted

OBS    I
 1     1
 2     1
 3     3
 4     4
 5     5
```

Notice that OBS 2 was still in the data set A and is read into the data set B. When this happens the variable I is incorrectly assigned a value of 1.

✱ USING AUTOEXEC.SAS AND CONFIG.SAS

Since many programmers do not use (or even know about) the AUTOEXEC.SAS and CONFIG.SAS programs, any options and code fragments that they contain may remain undetected for some time.

These may range in severity from what are merely irritants, such as the use of the ERRORABEND option which was mentioned above, to extreme measures that will halt further processing. A couple of the latter are shown below. Remember subtle is often most powerful. Folks must by necessity look harder when things are completely broken.

✔ The most hidden place to set the ERRORABEND option is in CONFIG.SAS.

✔ If the ENDSAS or ABORT statements appear in AUTOEXEC.SAS, SAS execution will terminate before the user's program even starts.

✔ In the section on using comments we showed how to use the /* to inadvertently hide code. If the last line of the AUTOEXEC.SAS is a /* all of the submitted code will be commented out until the end of the first /* ...*/ style comment.

A similar result can be achieved by placing the statement `%MACRO DUMMY;` in the AUTOEXEC.SAS. This will exclude all code until either a `%MEND`; or `%MEND DUMMY;` is encountered.

✔ The AUTOEXEC.SAS can also be used to house %INCLUDE statements that bring in and execute code that twists things a bit. Remember to use the NOSOURCE and NOSOURCE2 system options.

✱ OPERATING SYSTEM SPECIFICS

Each operation system has a few commands and options that are specific to that OS. These are found in the Companion for that system and are usually less well known.

✔ GENERAL

Write your own procedures using SAS/TOOLKIT®. Remember no documentation!

Use operating specific printer control characters to reset fonts or other printer characteristics.

✔ WINDOWS

The icons used to start SAS have associated properties that can be used to assign system options and to select specific AUTOEXEC.SAS and CONFIG.SAS files. In addition, any SAS options may be overridden in icon parameters where they are well hidden from casual observers.

The option -ARCH=BIT16 forces WIN3.1 to work in 16 bit mode (for improved performance or rival programmer's programs).

Access and use external DLLs.

✔ VMS

The CONCUR engine allows shared access to data sets

✱ SPECIAL OPTIONS

As a final thought for this section, the Job Security expert might like to consider using some other less well known system options - we'll leave it as an exercise for the reader to decide how best to apply them.

✔ DKRICOND=NOWARN
Suppresses error message when variables on DROP, KEEP, and RENAME statements are missing from an input data set.

✔ NOREPLACE
Prevents any permanent data set from being replaced.

✔ NOWORKINIT/NOWORKTERM
Applied at invocation, these options can be used to suppress cleanup of the WORK library on the initialization/termination of SAS session.

## SUMMARY

*"People who go to conferences are the ones that shouldn't."* Dilbert's Laws of Work

In reality there is a constant demand for SAS programmers. Just knowing how to write good, clean, and tight SAS code provides a high level of job security. Obviously, the techniques discussed in this paper are to be avoided; however, they have all been encountered in actual programs. Usually they were the result of accidents and ignorance, but no matter the source, they still caused problems. Being aware of the potential of these types of problems is a major step in the direction of writing better code.

## ACKNOWLEDGMENTS

Since this paper was first presented in 1993, a number of SAS users have made suggestions and contributions to this topic. Many of these tips have been included in this extended paper. For some reason many of these contributors have asked to remain anonymous. Those who were prepared to be named include Peter Crawford, John Nicholls, and Dave Smith. The authors welcome further contributions on Job Security concepts.

## REFERENCES

Carpenter, Arthur L., 1993, "Programming For Job Security: Tips and techniques to Maximize Your Indispensability", presented at the Eighteenth Annual SAS Users Group International Conference and published in the conference proceedings.

Carpenter, Arthur L., 1996, "Programming For Job Security: Tips and techniques to Maximize Your Indispensability", presented at the Twenty-first Annual SAS Users Group International Conference and published in the conference proceedings.

Virgile, Robert, 1996, *An Array of Challenges - Test Your SAS® Skills*, Cary, NC:,SAS Institute Inc., 174 pp.

## ABOUT THE AUTHORS

ARTHUR L. CARPENTER

Art Carpenter's publications list includes two chapters in *Reporting from the Field*, two books *Quick Results with SAS/GRAPH® Software* and *Carpenter's Complete Guide to the SAS® Macro Language*, and over two dozen papers and posters presented at SUGI, WUSS, and PharmaSUG. Art has been using SAS since 1976 and has served as a steering committee chairperson of both the Southern California SAS User's Group, SoCalSUG, and the San Diego SAS Users Group, SANDS; a conference cochair of the Western Users of SAS Software regional conference, WUSS; and Section Chair at the SAS User's Group International conference, SUGI.

Art is a SAS Quality Partner™ and through CALOXY he teaches SAS courses and provides contract SAS programming support nationwide.

TONY PAYNE

Tony Payne has worked as a SAS developer, project manager and course instructor for Software Product Services since 1986. His main area of specialization is in applications development. Tony has written six papers presented at SEUGI and other conferences. He is yet to write a book or chair a conference. SPS is a Quality Partner of SAS Institute, UK.

## AUTHOR CONTACT

Art Carpenter
California Occidental Consultants
PO Box 6199
Oceanside, CA 92058-6199

(760) 945-0613

art@caloxy.com
http://www.caloxy.com

Tony Payne
Software Products Services Ltd.
19-20 The Broadway
Woking, Surrey, GU21 5AP
United Kingdom

+44 1483 730771

tpayne@sps-uk.co.uk
http://www.sps-uk.co.uk

## TRADEMARK INFORMATION