SESUG Paper 156-2019

Hexadecimal Encoding Can Mitigate Some SAS® Macro Quoting Issues

Thomas E. Billings, MUFG Union Bank, N.A., San Francisco, California

ABSTRACT

Passing special characters in a SAS® macro variable can be challenging; see Rosenbloom and Carpenter (2013). As messy as the situation is for fixed (constant) strings, the possibility that a macro variable is dynamically loaded from an input data source is even more challenging, as the values that come in via inputs may throw an error in production. Here we describe methods that mitigate some of the macro quoting issues surrounding special characters. We begin by clarifying what it means for a macro variable to contain nearly anything, i.e., the issue of trailing blanks and how they are handled by select input methods. Then we give a simple method (no encoding required) that works if the macro variable is used in the current session. Next we show how hex encoding can mitigate macro quoting issues when the objective is to use the target macro variable string(s) in a compiled macro to be used in other, downstream sessions. We end by discussing the constraints of this method.

THE PROBLEM

You need to pass a string in a SAS® macro variable that may contain multiple special characters that are problematic for the macro language, i.e., require quoting (masking). The string may originate from an external data source, be dynamically read from an input file or database, or perhaps be user-generated via a stored process or other program. You want to use the string downstream in a compiled macro (for later use in other sessions) where the string would, if it did not contain special characters, be the operand of a simple %LET statement.

An example where this might be needed is to create a compiled, encrypted macro to hide passwords and/or encryption keys, especially when the password and/or keys are generated by a random process that may include special characters. For more information on this application and metadata-based alternatives, see Billings (2017A, 2017B, 2018).

The SAS macro language provides a set of quoting functions to handle special characters. The functions are complex and there are constraints:

1. some functions are applied at compile time,
2. other functions are applied at run time, and
3. unary (unmatched) quotes and parentheses may need to be pre-processed for the string to be compiled correctly.

Choosing the optimal quoting functions can be complex, as illustrated in Rosenbloom and Carpenter (2013). The possibility that the target strings may be random, defined during a program run, and may contain multiple special characters makes it more challenging.

In the next section, we begin our exploration of methods to mitigate some macro quoting issues by clarifying what “nearly anything” (in a string) means, and constraints imposed by existing methods to input strings into macro variables.

CONSTRAINTS ON INPUT STRINGS

Considering that the input string may be randomly generated and may contain multiple special characters, then such a string might contain leading and/or trailing blanks that are significant, i.e., “anything”. However, trailing blanks are generally ignored/not significant in the SAS 9.4 system; see:

http://documentation.sas.com/?docsetId=lrcon&docsetTarget=p00iah2thp63bmn1lt20esag14lh.htm&docsetVersion=9.4&locale=en

We shall see that the choice of input method can impact the handling of leading and trailing blanks. Let’s check how CALL SYMPUT (no X) and the SYMGET function handle leading/trailing blanks:

```
%global T1 T2 T3;

Data _null_;
       length x $20.;
       x = '12345678901234567890';
       call symput('T1', x);
       x = '  345678 ';
       call symput('T2', x);
       x = ' %f &gh ';
       call symput('T3', x);
       stop;
run;
```

T1 is entered to serve as a “ruler” of sorts to help detect/count blanks in output. Next:

```
40         %put &t1.;
12345678901234567890
41         %put &t2.;
345678
42         %put &t3.;
WARNING: Apparent invocation of macro F not resolved.
WARNING: Apparent symbolic reference GH not resolved.
%f &gh
```

Now &T2 is defined with leading and trailing blanks; the leading blanks are missing in the output of the %PUT – is that due to the %PUT or SYMPUT? &T3 has leading blanks that are gone and the value is unmasked/unquoted hence the warning messages. The latter issue can be corrected – for the most part – by use of %SUPERQ:

```
46         %let t3=%superq(t3);
48         %put &t3.;
 %f &gh
49
```

What about its usage in macro variable concatenation?:

```
52         %put AAA&t1.BBB;
AAA12345678901234567890BBB
53         %put CCC&t2.DDD;
CCC  345678            DDD
54         %put RRR&t3.TTT;
RRR %f &gh             TTT
```

The results above show that CALL SYMPUT does preserve leading blanks, but the macro variable has a length defined by the input variable (or expression used to create the variable) which means that significant trailing blanks are generally not preserved.

The SAS documentation advises that once a macro variable is masked/quoted, it does not need to be done again. However there are limits and constraints on this as this code shows:

```
58         %let newvar=MMM&t3.NNN;
59         %put &newvar.;
MMM %f &gh             NNN  ← quoted status preserved in concatenation
60
61         %put %trim(&t3.);  ← quoted status removed in %TRIM
WARNING: Apparent invocation of macro F not resolved.
WARNING: Apparent symbolic reference GH not resolved.
%f &gh
62         %let T3A=%trim(&t3.);
WARNING: Apparent invocation of macro F not resolved.
WARNING: Apparent symbolic reference GH not resolved.
63         %let T3B=%trim(%superq(t3));   ← quoted status removed in %TRIM
WARNING: Apparent invocation of macro F not resolved.
WARNING: Apparent symbolic reference GH not resolved.
64
65         %let T3C=%qtrim(&t3.);  ← quoted status preserved with %QTRIM
66         %put &T3C.;
 %f &gh
```

The SAS macro language documentation advises that some macro functions unmask the results. (Also confirmed by the code above and via a conversation with Art Carpenter.) This constraint should be kept in mind if your macro variable may contain (nearly) anything. Next, let's verify if the SYMGET function preserves leading/trailing blanks.

```
70         data _null_;
71           length x2 y $20. x $12. rbc $20.;
72           x = '  345678  ';
73           x2 = symget('T2');
74           y = '12345678901234567890';
75           rbc = cat(x,"ABCD");
76
77           if (x ne x2) then
78                  put @1 "Left blanks NOT preserved by symget";
79           else put @1 "Left blanks preserved by symget";
80           put @1 y $20.;
81           put @1 x $char10.  @20 x $hex20.;
82           put @1 x2 $char10.  @20 x2 $hex20.;
83           put @1 rbc $20.;  * right blanks not preserved;
84           stop;
85         run;
```

```
Left blanks preserved by symget
12345678901234567890
  345678           20203334353637382020
  345678           20203334353637382020
  345678    ABCD
```

The above shows that leading blanks are preserved by SYMGET but not trailing blanks. What about CALL SYMPUTX?

```
%global z1 z2 z3;

Data _null_;
       length x $10.;
       x = '1234567890';
       call symputx('z1', x);
       x = '  345678 ';
       call symputx('z2', x);
       x = ' %f &gh ';
       call symputx('z3', x);
       stop;
run;
```

%PUT yields similar results as before:

```
40         %put &z1.;
1234567890
41         %put &z2.;
345678
42         %put &z3.;
WARNING: Apparent invocation of macro F not resolved.
WARNING: Apparent symbolic reference GH not resolved.
%f &gh
```

And %SUPERQ masks the special characters as expected. Let’s check if blanks are preserved in concatenation:

```
52         %put AAA&z1.BBB;
AAA1234567890BBB
53         %put CCC&z2.DDD;
CCC345678DDD
54         %put RRR&z3.TTT;
RRR%f &ghTTT
```

The above shows that CALL SYMPUTX does not preserve leading or trailing blanks. Results for SYMGET are similar to the previous case. CALL SYMPUTX is preferred by many programmers as it provides greater control of the target macro symbol table. However a workaround for this is to issue a %GLOBAL for the target macro variable before issuing the CALL SYMPUT. That will, barring the same macro variable name being present in both local and global tables, force the CALL SYMPUT to write into a (target) global variable.

Next we consider PROC SQL SELECT INTO. However, we need to define a test dataset; code and data follow. Note that the $CHARw. informat is used below because it does not trim leading or trailing blanks. Also, while the trailing blanks are highlighted in SAS® Enterprise Guide®, they are not highlighted in the copy/paste operation used to show the text/data here:

```
data test_cases;
       infile datalines;
       length tstring $32. qstring $40.;
       input @1 tstring $char32.;
       format tstring $char32.;
       qstring = quote(tstring);
       len_tstring = length(tstring);
       len_qstring = length(qstring);
       LC_tstring = lengthc(tstring);
datalines4;;;;
123456789012345678901234567890
  ' %nmac ) &junk "(
abc '''" ))(( %f &j2 '"
)=(|¬+^-~*%, /&'>#"<
 a LT  (B ;;;
 X GE y
Y LE %X;;
&V LE Z
Z GE V;
J EQ m
(S OR W)
 (a NE b)   AND (C NE D)
    z; IN (1,2,3)
```

```
not MISSING(h)
0 = 1'
0 ne 1
0 NE "1
  no spec chars ex 1
;;;;
run;
```

The rows above provide good test cases for the methods discussed herein. Readers are cautioned that the default formats used to display data in SAS Enterprise Guide (and presumably other SAS interfaces) may trim trailing/leading blanks, a complication. (The code above includes some additional tests that we will discuss later; can ignore for now.)

Next we define a macro to select 1 row from the test data set and input it into a macro variable via SELECT INTO:

```
%let ruler=1234567890123456789012345678901234567890;

%macro getline(num=2);
%global sq&num.;
       data sql_extract;
             set test_cases;

             if (_n_ = &num) then
                    output;
       run;

       proc sql;
             select tstring into :sq&num. from sql_extract;
       quit;
       run;

       %put &ruler.;
%mend;
```

Using the above macro on row 2 in the test data set yields the result:

```
1234567890123456789012345678901234567890
46         %let sq2=%superq(sq2);
47         %put &sq2;
  ' %nmac ) &junk "(
48         %put AAA&sq2.BBB;
AAA  ' %nmac ) &junk "(            BBB   ← BBB starts in col. 35 – this shows macro
variable length is same as input character variable length
```

The result above – and running the macro for other rows in the test file – show that SELECT INTO:

- Preserves leading blanks, but
- The length of the macro variable created is determined by the length of the input variable (or expression), and the right side is padded with blanks hence trailing blanks are generally not preserved.

Savvy readers might ask whether using the function: QUOTE(target_string_variable) with SELECT INTO will preserve trailing blanks. [Note: this is the DATA step function QUOTE, not the macro function %QUOTE.] The answer to this is NO; the test data set code above verified this. If the resultant variable length permits, QUOTE(target_string_variable) will yield a variable of length: fixed length of target_string_variable + 2 (for begin/end quotes) + 1 for each double quote in the string (" instances in the input string are replaced by "" in the output).

The results above can be summarized in the table:

How Select SAS Macro-Related Features Handle Leading/Trailing Blanks

Feature	Leading blanks preserved?	Trailing blanks preserved?
CALL SYMPUT	Yes	No
SYMGET Function	Yes	No
CALL SYMPUTX	No	No
PROC SQL SELECT INTO (simple)	Yes	No; macro variable right-padded with blanks to match length of input SQL variable

Notes:
#1. This table is specific to standard fixed-length character variables; VARCHAR may be handled differently.

#2. Simple means a single variable SELECT INTO; expressions may be more complex.

Note the constraint above – the results are re: fixed-length SAS character variables; the new VARCHAR variables were not tested for this paper.

Finally, although trailing blanks are not preserved by the methods above, if the string to be input into a macro variable has fixed length, then you can force the macro variable to be the target length by:

1. Use %QTRIM to remove trailing blanks
2. Determine length of the result of #1
3. Add blanks via concatenation to result of #1 (use %STR).

A SIMPLE SOLUTION (NO ENCODING)

If you need to create a macro variable from inputs that can contain (nearly) anything AND you will use it in the same session ONLY [this is the most common case] then you can handle this as follows:

1. Input the target string into a %GLOBAL macro variable, possibly using one of the methods above.
2. Mask/quote the macro variable using %SUPERQ
3. Adjust the macro variable length if a fixed-length variable is required (using approach described above)
4. Caution: if you must operate on the global variable with macro functions, be sure to use %Q functions.

However this simple approach will not work if you want to inject the global macro variable (that contains special characters) into a compiled macro. Code like this will compile correctly:

```
%macro utility_mc;
%local secret_value;
%let secret_value=&my_global_variable;

%* various code that uses &secret_value;

%mend;
```

However: the code above will fail in execution because while &my_global_variable was input and created in the program where the macro is compiled; it is not defined in downstream programs that merely call the compiled macro. Instead, what we would like is the functional equivalent of:

%let secret_value=constant_text_string;

in our code, and the presence of special characters in the string is problematic. Next, we describe a solution for this problem.

HEXADECIMAL ENCODING CAN MITIGATE SOME MACRO QUOTING ISSUES

It would be nice if we could issue a simple %LET statement and not worry about special characters. In fact we can, if the target string is represented in hexadecimal notation, which is limited to the characters 0-9 and A-F inclusive, i.e. no special characters. The solution proposed here is as follows:

1. The target string is contained in a SAS character variable – input from an RDBMS, spreadsheet, flat file, XML, etc., or created in a program.
2. Create a hexadecimal version of the string; will need the length of the string in bytes to accomplish this
3. Issue a %LET statement in the compiled macro that specifies the hexadecimal string.
4. Convert the hex string to a masked/quoted macro variable using %QSYSFUNC with INPUTC and $hexw. informat.

The approach above is straight forward and works with any/all special characters including line-end and carriage return characters. However there is a challenge here: how to get the target hexadecimal string(s) into downstream SAS macro code? There are 3 main approaches:

1. CALL EXECUTE (DATA step to generate the code)
2. PROC LUA – some constraints apply here
3. Write the code using a DATA step and then execute it later using %INCLUDE.

Here we focus on approaches #1 & 2. Approach #3 is similar to #1 so to avoid redundancy, will not be covered here. To begin, for both approaches we need a macro that will take a single row from the test data set and:

- Determine the length in bytes and hexadecimal encoding of the target string
- Create another string that contains the $hexw. format/informat
- Use PUTC and $hexw. format with CALL SYMPUTX to copy the hexadecimal representation of the target string into a (global) macro variable
- Similarly, output the $hexw. specification into a global macro variable for use later as an informat.

Then later/downstream, the hex string and hex format can be used with %QSYSFUNC and INPUTC function to ingest and mask/quote the resultant macro variable.

The code below includes debug prints; the most important lines are highlighted.

```
%macro gethex(num=2);

       data extract;
              set test_cases;
              length hexfmt $12.;

              if (_n_ = &num) then
                     do;
                            call symput("v_temp",tstring);
                            numbytes = 0;
```

```
                           nbhfw = 0;

                           if (not missing(strip(tstring))) then
                                 do;
                                        numbytes = length(tstring);
                                        nbhfw = 2*numbytes; * <- w for hexw format;
                                        hexfmt=cats("$hex",put(nbhfw,12.),".");
                                 end;

                           call symputx("numbytes",strip(put(numbytes,6.)),"G");
                           call symputx("nbhfw",strip(put(nbhfw,6.)),"G");
                           call symputx("hexstr",putc(tstring,strip(hexfmt)),"G");
                           call symputx("hexfmt",strip(hexfmt),"G");
                           output;
                    end;
       run;

       %let v_temp=%superq(v_temp);
       %put &ruler.;
       %put &numbytes.;
       %put &nbhfw.;
       %put &v_temp.;
       %put &hexstr.;
       %put &hexfmt.;

%mend;
```

Note that the LENGTH function provides a single-character-equivalent count for single, double, and multiple byte characters (check the SAS documentation for internationalization details). Next we illustrate sample **CALL EXECUTE** code to create the desired SAS macro code.

```
%macro ck_m3;
       %do i=2 %to 18;
             %gethex(num=&i);

             data _null_;
                    length mycode $1000.;
                    mycode = cats('%macro spec_char_3;
                           %local spec_char;
                           %let spec_char=%qsysfunc(inputc(',
                           symget('hexstr'),',',symget('hexfmt'),
                           '));
                           %put Test of hex conversion 3, row=&i.;
                           %put &spec_char.;
                            %mend;  run;');

         put mycode=;   * check code prior to call execute;
         call execute(mycode);
         put "Check macro compile after this step";
stop;
run;

%spec_char_3;
       %end;
%mend;

%ck_m3;
```

The code generated by CALL EXECUTE can run immediately OR after the next step boundary; check the SAS documentation for details. The example above uses a character variable to contain the desired code, and the hex string and format are constants in the code/text (rather than macro variables that need to be resolved at run time) before the code is parsed by the SAS system via CALL EXECUTE. The timing of when the code will run is important; it may mean the difference between macro code that runs correctly every time vs. code that compiles now but might not run correctly later.

PROC LUA is another approach that can inject the hex string and format into SAS code. This is illustrated by the code below.

```
%gethex(num=2);

proc lua;
       submit;  ← begin block of LUA code
       local hexstr=sas.symget("hexstr");  ← calls SAS SYMGET function to ingest
                                              SAS macro variable into LUA variable
       local hexfmt=sas.symget("hexfmt");
       sas.submit_ ([[  ← begin block of SAS code
       libname maclib "/cfs/var/userdata/bfs/risk/cdm/users/ub43880/mclib2";
       options mstored sasmstore=maclib;  ← stored, compiled macro
       %macro spec_char_test4 / store;
       %local spec_char;
       %let spec_char=%qsysfunc(inputc(@hexstr@,@hexfmt@)); ← LUA variable values
                                                              Injected into SAS code
       %put Test of hex conversion 4;
       %put &spec_char.;
    %mend;
       ]]) ← end block of SAS code
       sas.submit("run;");  ← runs a line of SAS code
    endsubmit;  ← end block of LUA code
run;
```

The code above:

1. Uses SYMGET – from Lua – to import the value of SAS macro variables (hex string and format) into Lua variables
2. Then injects the values of the Lua variables (hex string and format) into SAS macro code
3. Issues a “run;” command and the macro is compiled and stored per the SAS code

Note that in test runs for this paper, using SAS Enterprise Guide:

- Setting the macro as non-stored and adding an invocation of the macro <u>inside</u> the Lua code block worked, however:
- Setting the macro as non-stored and adding an invocation of the macro <u>outside</u>/after the Lua code block failed – SAS could not find the macro.

The SAS system might not find the macro compiled via PROC LUA as it, like RUN_MACRO and a few other SAS features, may execute code in a separate/side SAS session, hence not fully integrated with the session where PROC LUA is called (?) Compiling and storing the macro overcomes this limitation.

Note: test runs were done with SAS 9.4M**3**; this issue might be fixed in later releases.

The compiled macro can then be called in a separate session:

```
libname maclib "/cfs/var/userdata/bfs/risk/cdm/users/ub43880/mclib2";

options mprint;
options nocenter  mstored sasmstore=maclib;

%spec_char_test4;
```

and it works. Calls to the compiled macro may work in the same session if you are running in batch; however SAS Enterprise Guide has issues with closing files (i.e., your read access to the macro may be blocked) and it is often easier to access the compiled macro in a separate SAS Enterprise Guide session. See Hemedinger (2016) for tips on closing files in SAS Enterprise Guide.

Savvy readers will note that the process of injecting the hex string and hex format into downstream SAS macro code is cumbersome. This is acknowledged herein as a significant constraint on applications of this approach. Finally, although this method was developed for dynamic strings that are read in-program, the method can be used for fixed strings as well, especially those that are throwing errors in your programs.

ACKNOWLEDGEMENT OF PRIOR, SIMILAR USE OF HEX ENCODING

The only prior use of hex encoding to handle special characters that was found while researching this paper was a (2002) SAS-L post by Paul Dorfman:

Subject: Re: assign macro var a string with embedded nonprintable characte r?
Date: 28 Mar 2002
URL: https://listserv.uga.edu/cgi-bin/wa?A2=SAS-L;d4a7ec1b.0203d

Note that you will need to be a SAS-L website user and signed into the website* for the URL above to work. In the post above, hex encoding is suggested to handle embedded, non-printable, characters in a macro string; it does not address using the approach to handle other, printable special characters/macro triggers.

*Note: you can subscribe to/access the SAS-L website portal at: https://listserv.uga.edu/cgi-bin/wa?A0=sas-l

SUMMARY

The need to include special characters in a SAS macro variable drives substantial complexity in programming; the SAS system provides macro functions to handle these characters at compile and run time. Programs that ingest strings that contain special characters and promote those into macro variables are at risk of failure if the code is not written to handle the special characters encountered.

In this paper we have:

- Reviewed the most common methods of inputting strings to macro variables and summarized how they treat leading and trailing blanks; this is important in applications where these blanks are significant
- Described a simple method: import string to macro variable and then quote it with %SUPERQ, in a global macro variable
- Noted that some macro functions unmask special characters hence %Q macro functions should be used instead
- Described how to use hex encoding to avoid some quoting issues:
    - ➢ Input target string to SAS DATA step variable
    - ➢ Create hex representation of the string; also $hexw. format for I/O
    - ➢ Inject hex string into SAS macro code (usually for compiled macros) using CALL EXECUTE or PROC LUA
    - ➢ In the macro code, use %QSYSFUNC with INPUTC and $hexw. format to input (and mask/quote) the string in a SAS macro variable
    - ➢ When using CALL EXECUTE method, the timing re: when the generated code will execute can be important – pay attention to this detail

- Acknowledged that the need to inject the generated hex string into macro code limits the applicability of this approach.

APPENDIX 1:
BSD 2-CLAUSE COPYRIGHT LICENSE (OPEN SOURCE)

```
* All program code in this paper is released under a Berkeley Systems
  Distribution BSD-2-Clause license, an open-source license that permits
  free reuse and republication under conditions;

/*
Copyright (c) 2019, MUFG Union Bank, N.A.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimer in the documentation
and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE
LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
POSSIBILITY OF SUCH DAMAGE.
*/
```

REFERENCES

Note: all URLs quoted or cited herein were accessed in March 2019.

Billings T (2017A). Secure Macro-Based Method to Assign LIBNAMEs for Databases. Paper to be presented at 2017 *Western Users of SAS Software Conference Proceedings.* URL: https://www.lexjansen.com/wuss/2017/22_Final_Paper_PDF.pdf

Billings T (2017B). Keeping Passwords, AES Encryption Keys, and Other Sensitive Parameters Out of Source Code and Logs. *Western Users of SAS Software Conference Proceedings.* URL: https://www.lexjansen.com/wuss/2017/21_Final_Paper_PDF.pdf

Billings T (2018). Non-metadata Methods to Keep Passwords and Sensitive Strings out of SAS® Source Code and Logs. *SAS Global Forum Conference Proceedings.* URL: https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/1736-2018.pdf

Hemedinger C (2016). Tip: How to close all data sets in SAS Enterprise Guide. *SAS Blog: The SAS Dummy.* URL: http://blogs.sas.com/content/sasdummy/2016/10/18/close-data-sas-eg/

Rosenbloom M, Carpenter A (2013). Macro Quoting to the Rescue: Passing Special Characters. *Western Users of SAS Software Conference Proceedings.* URL: https://support.sas.com/resources/papers/proceedings13/005-2013.pdf

ACKNOWLEDGEMENTS

Thanks to Mary Rosenbloom and Art Carpenter for their informative 2013 paper. Note that any errors herein are solely the responsibility of the author.

CONTACT INFORMATION

A list of the author's SAS-related papers, including URLs for free access, is available at the URL (hosted by Google Drive): https://goo.gl/uCUHoa

Note: Your enterprise web filter might prevent access to this URL from work, in which case you will need to access via a personal device.

Thomas E. Billings
MUFG Union Bank, N.A.
San Francisco, CA 94104

Remote from:
Merritt Island, FL 32952
Email: tebillings@gmail.com

Disclaimer: The contents of the paper herein are solely the author's thoughts and opinions, which do not represent those of MUFG Union Bank N.A. The bank does not endorse, recommend, or promote any of the computing architectures, platforms, software, programming techniques or styles referenced in this paper.