Paper 1163-2017

Îf ÿøù have to process difficult characters: UTF-8 encoding and SAS®

Frank Poppe, PW Consulting

ABSTRACT

Many SAS® environments are set up for single-byte character sets (SBCS). But many organizations now have to process names of people and companies with characters outside that set, like in “Îf ÿøù have”. You can solve this problem by changing the configuration to the UTF-8 encoding, which is a multi-byte character set (MBCS). But the commonly used text manipulating functions like SUBSTR, INDEX, FIND, and so on, act on bytes, and should not be used anymore. SAS has provided new functions to replace these (K-functions). Also, character fields have to be enlarged to make room for multi-byte characters. This paper describes the problems and gives guidelines for a strategy to change. It also presents code to analyze existing code for functions that might cause problems. For those interested, a short historic background and a description of UTF-8 encoding is also provided. Conclusions focus on the positioning of SAS environments configured with UTF-8 versus single-byte encodings, the strategy of organizations faced with a necessary change, and the documentation.

INTRODUCTION

Many SAS environments are set up for ‘single byte character sets’ (SBCS), most often some form of extended ASCII. Because of several trends, which can loosely be described as ‘globalization’, organizations are now being confronted with names of people and companies that contain characters that do not fit in those character sets. They have uncommon accents and other diacritical signs, like the characters *Îÿøù* in the title.

Organizations working in a Japanese or Chinese environment have, for some time, already the opportunity to use SAS with a ‘double byte character set’ (DBCS). But since then a more general solution has become available. The Unicode system has been developed, and is now maintained by the internationally recognized Unicode consortium. All known characters and symbols for any languages are given a ‘code point’. At the same time systems have been developed to encode these ‘code points’ in bytes. These all need more than one byte, and thus are ‘multi byte character sets’ (MBCS). The most widely used of those is the UTF-8 encoding, which is a using one to four bytes. Most web pages nowadays use UTF-8, and it is also the default encoding for XML.

Switching an existing SAS installation from any SBCS encoding to UTF-8 is no trivial task, alas. Commonly used text manipulating functions in SAS like SUBSTR, INDEX, FIND, etc., act on bytes, and do not take into account that a character now may need more than one byte.
And character variables need more bytes to contain the same number of characters, if the text contains characters that take more than one byte.

The problems with these old functions, acting on bytes instead of on character, are described, and syntactic differences with the new K-functions that replace them are indicated (there is KSUBSTR and KINDEX, but no KFIND e.g.). Some guidelines are given for determining the extent to which character fields need to be enlarged to contain multi byte characters.

To assess the changes needed in existing code an approach is described to search code for occurrences of character functions that should not be used for multi byte characters. The code tries to find the actual occurrences of function calls in the code, ignoring function names in comments or variable names that are equal to a function name.
The full SAS code is appended.

Pointers are given to relevant documentation on the web.

Also some historical background to encoding is given, to put everything in context. This is accompanied by a description of the UTF-8 encoding method itself.

Conclusions will focus on the positioning of UTF-8 configured SAS environments versus single byte encodings, on the strategy of organizations faced with a necessary change, and on the documentation.

THE PROBLEM

In this paragraph first some background is being given, and then how you might notice them in practice.

THE BACKGROUND

There are three areas to discuss here:

- The character functions in the Data Step and in Macro code.
- The width of the variables to store character values.
- The Input statement.

Character functions

The often used character functions in SAS like SUBSTR, INDEX, etc., act on bytes, not on characters. And they will continue to do so. This is understandable because some applications expect them to do that, also when working with multiple byte characters. E.g., when encoding a long character string towards BASE64 it has to be cut in smaller strings with a specific number of bytes, regardless of character boundaries.

But in most case one wants to process characters, not bytes. The following Data Step code snippet illustrates this:

```
name = 'Bálasz' ;
name2 = substr ( name , 1 , 2 ) ;
put name2 = ;
```

In a single byte configured SAS environment one see:

```
name2=Bá
```

And, usually, one will *also* want to see that in a UTF-8 configured environment. But the result will be like this:

```
name2=B�
```

The character “á” takes two bytes in UTF-8, the hex values 'C3'x and 'A1'x. The SUBSTR function selects two bytes: the “B” and the hex value 'C3'x, and if that hex value is shown in itself it has no meaning (this will be explained in the section on the UTF-8 encoding method), leading to the questionmark-in-a-black-diamond.

Using the alternative KSUBSTR does give the desired result:

```
name2 = Ksubstr ( naam , 1 , 2 ) ;
```

will give again:

```
name2=Bá
```

Note that these K-functions can be used in all environments, also when operating in a single byte environment. So it would sensible to stop using the ‘old’ functions, and start using the K-functions exclusively.

Variable length

But note also that the string "Bá" stored in the variable 'name2' now is *three* bytes long. If the variable would have been defined with a length of $2 it will *still* produce "B�". The KSUBSTR function produces the right *three* bytes, but the last one is still lost when storing it into the variable.

So the length of all character fields will have to be considered. In general fields that contain codes and such are not in danger. Also codes that are being maintained externally usually use only the simple ASCII characters. E.g., the codes defined by the ISO organization for country, currencies, etc., all only use ordinary (capital) characters.

But any field that comes from 'the real world' and may contain names of people, organizations, etc., may contain some, or many, multi byte characters.

The INPUT statement

When reading external files in a Data Step one also has to take into account the possibility that they nowadays often will be in UTF-8 format. Reading files using an input statement that relies on 'separators' between the fields will not be problem. So if the fields are being separated by spaces or tabs, or by semi-colons or comma's, there will not be a problem, provided of course that the variables receiving the values that are being read have a length large enough to receive any multiple byte characters.

But input statements that rely on character positions, or *think* they are, can cause problems. Take the following two statements.

```
INPUT @3 Naam $. ;
INPUT naam $ 3-23 ;
```

If the first two columns contain characters that take up more than one byte, the statements probably do not produce the intended results.

There does ***not*** seem to be a reliable method to use a column oriented INPUT statement when dealing with multi byte formatted input files.

THE PRACTICE

How do you notice that you will have to convert from a SBCS encoding to a MBCS? And how do you notice after converting that some consequences seem to have been overlooked?

Running in a SBCS environment

If external files that were formatted using a MBCS encoding are being processed in a SAS environment that uses the Latin1 encoding, or a similar SBCS encoding, a problem can surface in several ways.

- It can remain hidden for a while when the multi byte characters can be mapped to the extended ASCII set used. Characters like á, ü, etc., take two bytes in a MBCS encoding, but do exist in most SBCS encoding.
  In some instances (particularly when using EG to display input or output) some characters are silently changed to simpler forms, by omitting the diacritical signs (the accents).
- Empty spaces appear where characters are being expected.
- A warning in the log appears that some characters could not be 'transcoded', meaning that they could not be mapped to a codepoint in the SBCS encoding.
- When using the XMLV2 engine the same situation generates an error (which is a rather surprising difference with the previous point).

Running in a MBCS environment

When your environment is already configured with a MBCS encoding, there still might be problems. These all are variants of the two situations already described:

- A SAS string function not suitable for a MBCS environment is being used.
- A character field is not wide enough (in bytes) to hold all characters.

Neither of these produce in general warnings or errors in the SAS log. So the only way these problems can surface is that somebody notices unexpected results. Of course, it can then happen that that situation has been existing for quite some time.

Therefor it is a good idea to do a thorough analysis before using existing code in a MBCS environment. In a next paragraph some advice is being given how to transfer from a SBCS to a MBCS environment.

HISTORY: UNICODE AND UTF-8

A SHORT DEVELOPMENT OF ENCODING

In a file each character is stored as a code. At the start of the computer era the code could always be stored in one byte. Immediately different code tables emerged. One of the oldest and most widely used system is the ASCII encoding, using the codes 0 through 127 (using 7 bits). (Another well used coding standard is EBCDIC.)

The remaining bit was used as a ‘control character’ to minimize the chance of transmit errors.

The first 32 of those (0 through 31) are ‘control characters’, the best known being number 10 (linefeed, or LF), 13 (carriage return, CR) and 27 (escape, ESC).
Number 32 is the space, and then follow the ‘visible’ characters.
The last one in the series, 127, is delete (DEL).

After some time also the eighth bit was taken into use, because transmission became reliable enough to make a control bit superfluous. This made the codes 128 through 255 available. That made room to include accented characters (á, é, ü, etc.), ligatures (joins of characters like æ, œ and ij), and symbols (§, €, etc.). This gave rise to yet another array of different variants for assigning characters to code, reflecting the needs from different languages for particular characters (Scandinavians want the ø, French want the ç, Germans the ß, the Dutch want the capital IJ ligature, etc.).

The Latin1 encoding became the most widely used, trying to incorporate at least the most used characters from the different languages, as long as there was room.

SAS is still often installed by default with the Latin1 encoding. Most characters that are in used in Western Europe can be show with this encoding, and thus it is still widely used.

UNICODE AND UTF-8

The growing use of internet and globalization in general meant that these different encodings created more and more friction and problems. Gradually a system evolved to incorporate all possible systems into one uniform system: Unicode. In Unicode all characters from all scripts have been assigned a ‘code point’.

At the same time various systems were being developed to store those code points into bytes, such as UTF-8, UTF-16 and UTF-32. The first one is the most efficient, and the most widely used.
The most used characters (in the roman script) take up one byte in UTF-8. These are the first 127 ASCII codes, so for those characters there is no difference between ASCII and UTF-8.

Less used characters take up two or three bytes. Two bytes are needed to code the modern ‘Western’ languages. This is the Roman script with all possible diacritical signs, and also includes Cyrillic, Greek, Hebrew and Arabic). Some symbols however that are commonly used in Western languages take three bytes, like the Euro (€).

Three bytes are needed for all other ‘living’ languages. The four byte codes are used for scripts of languages that are not spoken anymore (e.g., Egyptian hieroglyphs) and for ‘symbols’ like Mahjong tiles and emoticons or emoji.

UTF-16 uses either two or four bytes to encode the Unicode points, and UTF-32 always uses four bytes. Because of its greater efficiency UTF-8 has been the most successful, although there are some other arguments for and against the different systems. They deal e.g. with the ability to recover from corrupted bytes, but that is beyond the scope of this paper.
There is also UCS-2, which is predecessor of UTF-16.

‘READING’ UTF-8

As a programmer it is sometimes necessary to understand how text is being parsed. How does the parser know if a byte should be interpreted as being a single byte character code, or as part of a multi byte character?

As noted already the first 127 codes are identical to ASCII. This is because a byte with the first bit set to 0 in UTF-8 indicates a single byte character code. Bytes with the first bit set are part of a multi byte sequence. If the two first bits are set, it is a starting byte, bytes starting with ‘10’ are continuation bytes. How many continuation bytes follow a starting byte, is indicated by the number of set bits after the first two. If the starting byte only has the two first bits set, followed by a ‘0’, there will be only one continuation byte. If it starts with three set bits and a ‘0’, there will be two, etcetera.

The remaining bits are used to specify the character code. This has been summarized in Table 1 (below), based on the Wikipedia lemma on UTF-8.

Number of bytes	Byte 1	Byte 2	Byte 3	Byte 4	Available bits
1	0xxxxxxx				7
2	110xxxxx	10xxxxxx			11
3	1110xxxx	10xxxxxx	10xxxxxx		16
4	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx	21

Table 1. Interpreting UTF-8 bytes

So, with one byte there are 7 bits available, which gives 2^7 -1 = 127 different codes, as we already have seen.
With two bytes, there are 11 bits, which give an additional 2^{11} = 2,047 code points. Three bytes add 65,535 possibilities, and four bytes another 2,097,151. Together this creates a total of 2,164,860, although for various reasons some codes will never be used.

THE DOCUMENTATION

DATA STEP FUNCTIONS

The documentation on the SAS web site on *‘Internationalization Compatibility for SAS String Functions’* has an overview of all functions, indicating whether they can be used without problems on text with multi byte characters.

However, one should be careful using the documentation. The situation is improving, but particularly the 9.3 documentation still contains many confusing and sometimes incorrect entries. The 9.4 documentation has been improved lately.

For instance, at the time of finalizing this paper, the table in the 9.3 documentation that lists all the character functions and their suitability for use with MBCS data, still has an entry for the function ‘SUBSTR (left of =)’ stating that it *“extracts a substr”*.

This is not true; this should be in the entry on ‘SUBSTR (right of =)’. But there is no such entry (there should be).

Links also can be confusing or incorrect. E.g., the page on the KSUBSTR function in the 9.3 documentation refers also to the “SUBSTR (left of=)” entry. But, contrary to the SUBSTR function, the KSUBTR function cannot even be used left of the equals sign! The KUPDATE function should be used instead.

The 9.4 documentation has correct entries for ‘SUBSTR (left of=)’ and ‘SUBSTR (right of =)’. It now also has a short introduction on the behavior of the old SBCS functions when using MBCS data, similar to the example in the problem description in this paper.

But the page on ‘SUBSTR (left of =)’ still incorrectly refers to the KSUBSTR function instead of to KUPDATE.

At the same time the documentation on the KUPDATE function does not refer to its older SBCS variant ‘SUBSTR (left of =)’, as the other K-functions do.

The fact that there are also KSUBSTRB and KUPDATEB functions that act on *bytes*, like the SUBSTR function in its two forms (left and right of the =) is confusing as well. What exactly then is the difference between these new functions and the old functions?

Also the Data Step functions KLOWCASE and KUPCASE do exist, but why? The old LOWCASE and UPCASE functions are being described as suitable for use with MBCS data.

MACRO FUNCTIONS

The documentation on the macro functions also leaves some question marks. The %KUPCASE and %QKUPCASE macros, are being described, as replacements for %UPCASE and %QUPCASE.

But nothing is being said there on %LOWCASE and %QLOWCASE. This is even more strange since %QLOWCASE and %QLOWCAS *do* exist.
Also there is a function %KCMPRES to replace %CMPRES, but there is no %QKCMPRES to replace %QCMPRES. Does that mean that %QCMPRES is Level2 but %CMPRES is not?

GOING FROM A SBCS TO MBCS

It would be best to avoid the transition from a SBCS environment to a MBCS environment at all.

So when installing and configuring a completely new SAS installation (or requesting one), ask for it to be configured for UTF-8. This is just a small ‘switch’ somewhere in the configuration process, and will be prevent a lot of analysis later.
If this new installation will be used to run existing code it will of course not be as simple as that. In that case ask for *two* ‘application servers’ to be installed, one configured in Latin1 (or whatever is used locally), and one in UTF-8. You are then already set for the transition process.

(An application server is a concept in the SAS metadata; it is the context in which servers like the Workspace server, the Stored Process server, etc. operate. There can be several application servers in the metadata. The default one usually is called ‘SASApp’, a next one might be called ‘SASApp-UTF8’.)

GENERAL STRATEGY

As will be clear from the previous paragraphs there can be quite some analysis and work involved to make a successful transition. For many existing installations it will be difficult to do that in one giant step. A gradual transition is easier to accomplish.
To make that possible one can configure a new application server, to be used to run the code that already has made the transition.

It is important to analyze the different processes in your environment. The risk that SAS tables that have been produced by code running in UTF-8 are being used by code that is still running in the old application server should be minimized. This analysis will have to determine in which order code is being transferred.

When developing new code always use the K-functions, even if a transition to a UTF-8 encoding is not imminent.

ANALYSIS

The items that have to be analyzed follow logically from the problems already described.

Make certain that no string functions are being used that cannot reliably process multi byte characters (i.e., in the SAS documentation are not labelled 'Level 2'). The code in the appendix can be used as a starting point to analyze existing code for the occurrence of such functions. It can be executed on any file with SAS code. That can be code exported from EG, or deployed from within DI Studio.
Some code generated by standard transformations in DI Studio can be signalized as well. One will have to consider if that may lead to problems.

Check the length of your string variables.
No action is needed if they only will contain codes and such. If they may contain names one will have to consider if disk space is an issue. If not, when simply set the length of a character field to three times the number of characters they should be able to contain.
If you have to be careful about space, the following can help:

- For European names with occasional 'strange characters': 10 to 20% longer will be enough;
- If only 'Western' languages will have to be processed (including Greek, Cyrillic, Hebrew) the original length times two should suffice;
- Otherwise, take the length times 3; this is also the default used by the engine in the Access to SQL Server module (possibly also by the engine used in other Access products).

Finally, check any input statements for the use of character pointers etc. (which in effect act as byte pointers.

CONCLUSIONS

Many organizations may in the future face a necessary transition to a multi byte character set encoding. There are several actions that can be taken to simplify that transition, and that are easy to execute. Configure new environments by default in UTF-8 (although the default in the SAS scripts is still Latin1).

Also start using K-functions for all text manipulation.
The current SAS documentation might advocate this more explicitly (e.g. by mentioning this more clearly in the documentation on the old functions). Current confusing and sometimes incorrect information in the documentation should be improved.

The availability of the UTF-8 encoding in the SAS system makes it possible to create applications that can process the different languages of the world without problems, making them future-proof.

REFERENCES

SAS Documentation for SAS 9.3. "SAS(R) 9.3 National Language Support (NLS): Reference Guide". Visited 2 March 2017. Available at http://support.sas.com/documentation/cdl/en/nlsref/63072/HTML/default/viewer.htm#titlepage.htm.

SAS Documentation for SAS 9.4. "SAS® 9.4 National Language Support (NLS): Reference Guide, Fifth Edition". Visited 2 March 2017. Available at http://support.sas.com/documentation/cdl/en/nlsref/69741/HTML/default/viewer.htm#n1d2i07lune0dxn1bsah301axr4v.htm.

Wikipedia. "UTF-8". Visited 2 March 2017. Available at https://en.wikipedia.org/wiki/UTF-8.

ACKNOWLEDGMENTS

The author is grateful to his former colleagues at De Nederlandsche Bank for the discussions on the topic, and the people from Technical Support at the Dutch SAS office.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Frank Poppe
PW Consulting
Telephone: +31 6 2264 0854
Frank.Poppe@PWconsulting.nl

APPENDIX: CODE TO ANALYZE CODE

This code was written to be used in a Stored Process, to be called from a web page. The code was developed using DI Studio. It can be used in other ways, but some details in the version described here give away that the DI Studio origin, and the Stored Process destination.

The code examines SAS source code files, and tries to determine calls to a list of functions, given in an input table.

The input table contains a list of all functions that are not ‘cleared’ to be used in a double byte environment (i.e. listed in the SAS documentation as ‘Level 0’ or ‘Level 1’).
It also contains comments (mostly giving the alternative K-function to be used), that will be later be joined into the report that is produced.

Below some comments are inserted that each time precede the following block of code.

COMMENTS AND CODE

The function names are concatenated in one long macro variable, in order to build a Regular Expression later.

```
proc sql noPrint ;
select ktrim( function ) into :functionList separated by '|' from
&_input ;
%put &=functionList ;
quit ;
```

The number of functions in the input table is also stored in a macro variable.

```
proc SQL noPrint ;
select count ( * ) into :nLines from &_input ;
quit ;
%put &=nLines ;
```

The three macro variables in the following FILENAME statement determine the files with SAS code to be examined. They may contain wild cards.

```
filename SAScode "&filepath\&filename..&filetype" ;
```

To prevent a warning when building the Regular Expression (because the string because the string will extend 262 chars

```
options NOQUOTELENMAX ;
```

Now read all the lines and check them. The possible problems are written to the output table.

```
data &_output (keep = sequence currfile linenum thisline function )  ;
length
            expression currfile thisline remains  compareline $ 1200
            thisfile $ 400
;
retain
            RegExpr
            linenum sequence
            0
;
```

The array will contain the lines of code that should be ignored.

```
      array ignore [&nLines] $ 200 _TEMPORARY_ ;
      array reason [&nLines] $ 200 _TEMPORARY_ ;
      nIgnore = dim ( ignore ) ;

      infile SAScode filename=thisfile eov = newfile  end = lastrecord  ;
      if _N_ = 1 then do ;
```
We also use an input table with a number of code lines would have been flagged, but can be ignored because they contain known calls to ‘Level 0’ functions that can do no harm. This is can be because the function will never operate on anything that can contain double byte characters.

Also lines that are generated by standard SAS transformation in DI Studio that call such functions can be added here, also not all these cases are innocent ones. But a developer cannot do anything about it.

The lines are stored in an array, together with an array of reasons for ignoring them, for reporting purposes later.

```
                        do i = 1 to &nLines ;
                             set &_input1 ;
                             ignore [i] = ignoreLine ;
                             reason [i] = ignoreReason ;
                 end ;
```

Now build the Regular Expression that can be used to check each line. It looks for occurrences of any of the function names, but *only* if they are followed by a left parenthesis. There may be ‘white space’ (space, tab) after the function name. Also function names followed by only white space and an end-of-line are selected.
This will prevent most function names in comments to be marked, because they are seldom followed by the left parenthesis. A function name that is followed by a left parenthesis on the next line will be marked wrongly, however. Looking ahead to the next line was determined to complicated for this purpose.

```
                             expression = cat ( '/\b(' , "&functionList" ,
                             ')\s*(\(|$)/i' ) ;
                             RegExpr = prxparse ( expression ) ;

                             if not RegExpr then stop ;
```

The code reports file by file (if the wildcard Filename selects more then one file). For this we need to initialize the flag *newfile* to 1.

```
                             newfile = 1 ;
                 end ;

                 input ;

                 thisline = _INFILE_ ;
                 currfile = thisfile ;
```

Also initialize the *linenum* variable to 1, if we have a newfile.

```
                 if newfile = 1 then do ;
                             linenum = 1 ;
                             newfile = 0 ;
                 end ;
                 else linenum + 1 ;

                 compareline = kleft ( ktrim ( thisline ) ) ;
```

Initialize the variable *check* to 1, and then see first if this is a line to be ignored. If so, set the variable *function* to the reason for ignoring (with some style attribute).

```
                check = 1 ;
                do i = 1 to nIgnore ;

                        if compareLine = ignore [i] then do ;

                                check = 0 ;
                                function = kstrcat ( '~{STYLE
    [fontstyle=italic]' , ktrim(reason [i]) , '}' ) ;

                        end ;
                end ;

                if check then do ;
```

Now initialize the flag *found* to 1, to start the DO WHILE loop. It is quite possible that there are several function names in one line, so if something has been found, the loop continues with the remainder of the line.

```
                        found = 1 ;
                        paren = 1 ;
                        remains = _INFILE_ ;
                        do while (found) ;
                            found = prxmatch ( RegExpr , remains ) ;
                                if found then do ;
                                    percent = ksubstr ( remains ,
                                          found - 1 , 1 ) ;
```

Remember the character *before* the found function name, to see later if it is a percent sign (then this is a macro function).

```
                                remains = ksubstr ( remains , found ) ;
                                paren = kindex ( remains , '(' ) ;
                                if paren then do ;
                                     function = ksubstr ( remains ,
                                          1 , ( paren -1 ) ) ;
                                     remains =  ksubstr ( remains ,
                                          paren ) ;
                                end ; else do ;
```

Generate a message

```
                                    function = kstrcat ( ktrim (
                                      remains ) ,
                                      ' ~{STYLE [fontstyle=italic]
                                      (if paren on next line)}' ) ;
```

And stop the loop.

```
                                    found = 0 ;
                                end ;
```

Have make a difference between the Data Step functions UPCASE or LOWCASE, and the macro functions, because only macro functions seem to be Level0

```
                                if kupcase ( function )
                                    IN ('UPCASE', 'LOWCASE')
```

```
                                     and percent ^= '%' then do ;
```
Do nothing
```
                                  end ; else do ;
                                     sequence + 1 ;
                                     OUTPUT ;
                                   end ;
                              end ;
                         end ; /*do while */
             end ;
             else do ;
                 OUTPUT ;
              end ;


      run ;
```
Now join the output table with file names, line numbers, functions and messages to the table with function names and comments.
```
      proc sql;
         create table tec_sas.UTF_Lines_to_be_checked as
         select
            W3DVUH9.currfile length = 1200,
            W3DVUH9.thisline length = 1200,
            W3DVUH9.linenum length = 8,
            W3DVUH9.sequence length = 8,
            W3DVUH9.function length = 200,
            UTF_I18N_level0_Functions.comment length = 160
         from
            work.W3DVUH9 as W3DVUH9 left join
            tec_sas.UTF_I18N_level0_Functions as UTF_I18N_level0_Functions
               on
               (
                  klowcase ( kscan ( W3DVUH9.function  , 1 ) ) = klowcase (
      UTF_I18N_level0_Functions.function  )
               )
         order by
            currfile,
            linenum,
            sequence
         ;
      quit;
   enddata;
   run;
   quit;
```