PharmaSUG 2018 - Paper BB-08

UTF What? A Guide for Handling SAS Transcoding Errors with UTF-8 Encoded Data

Michael Stackhouse, Lavanya Pogula, Covance, Inc.

ABSTRACT

The acronyms SBCS, DBCS, or MBCS (i.e. single, double, and multi-byte character sets) mean nothing to most statistical programmers. Many do not concern themselves with the encoding of their data, but what happens when data encoding causes SAS to error? The errors produced by SAS and some common workarounds for the issue may not answer important questions about what the issue was or how it was handled.

Though encoding issues can apply to any set of data, this presentation will be geared towards an SDTM submission. Additionally, a common origin of the transcoding error is UTF-8 encoded source data, whose use is rising in popularity by database providers, making it likely that this error will continue to appear with greater frequency. Therefore, the ultimate goal of this paper will be to provide guidance on how to obtain fully compliant SDTM data with the intent of submission to the FDA from source datasets provided natively in UTF-8 encoding.

Among other topics, in this paper we first explore UTF-8 encoding, explaining what it is and why it is. Furthermore, we demonstrate how to identify the issues not explained by SAS, and recommend best practices dependent on the situation at hand. Lastly, we review some preventative checks that may be added into SAS code to identify downstream impacts early on. By the end of this paper, the audience should have a clear vision of how to proceed when their clinical database is using separate encoding from their native system.

INTRODUCTION

Character data is stored as a series of bytes. Bytes are made up of bits (binary digits), which are the smallest unit of data in a computer, containing a value of 1 or 0. Typically, bytes consist of eight bits. Bytes can be organized into different sequences to represent a number between 0 and 255. Encoding is how a computer interprets and represents the values within data. Data encoding will consist of a character set, which is the list of characters capable of being represented by the encoding. To connect the byte level data to the desired characters, a coded character set maps the number represented by a byte to its corresponding character.

One distinguishing factor of data encoding types is the number of bytes used to store the characters. Encoding may be a Single Byte Character Set (SBCS), Double Byte Character Set (DBCS) or Multi Byte Character Set (MBCS). A standard SBCS would be Latin1 Encoding. This encoding consists of the standard ASCII character set, which includes upper and lower case English characters, the digits 0 through 9, and some special and control characters (e.g. $, *, carriage returns, etc.). Latin1 also includes the extended ASCII character set which includes additional characters used in most Western European languages (e.g. ñ, ø) and additional special characters (e.g. ©, ¿). But the characters within the extended ASCII character set are not all inclusive. The focus is on Western European languages. What about Eastern languages? What about Asian languages, Cyrillic, etc.? These written languages consist of thousands of characters, and one byte is not enough. This is where a DBCS or a MBCS come into use.

Universal character set Transformation Format-8-bit (UTF-8) encoding attempts to represent all characters in all languages. It is capable of representing more than 120,000 characters covering 129 scripts and multiple symbol sets. While the first 128 characters of the ASCII code range may be represented by one byte, other characters within UTF-8 may require up to 4 bytes to be represented (Dutton, 2015).

WHERE YOU ARE AND WHERE YOU NEED TO GO

The SAS session encoding is the encoding that is used by SAS while it works with data. The encoding of a permanent dataset being read into SAS may differ. In these cases, the dataset being read in must be transcoded into the session encoding for SAS to work with it. In many cases, SAS can do this itself by using Cross-Environment Data Access (CEDA).

Adjusting your session encoding differs between programming environments, but in order to determine your session encoding, you can simply use PROC OPTIONS:

```
proc options option=encoding;
run;
```

This will output a message to your log, telling you what encoding your SAS session is using.

```
ENCODING=UTF-8    Specifies the default character-set encoding for the SAS
session.
```

Output 1. Log output generated by PROC OPTIONS.

As for checking the encoding of a permanent dataset, this information is available when using PROC CONTENTS on the permanent dataset:

```
proc contents data=utf8.myDS;
run;
```

The encoding of the dataset is displayed within the PROC CONTENTS output.

```
                                The CONTENTS Procedure

Data Set Name          RAWDATA.MYDS                                 Observations
0
Member Type            DATA                                         Variables
0
Engine                 V9                                           Indexes
0
Created                Sunday, February 21, 2016 10:01:45 PM        Observation
Length    0
Last Modified          Sunday, February 21, 2016 10:01:45 PM        Deleted
Observations
Protection                                                          Compressed
NO
Data Set Type                                                       Sorted
NO
Label
Data Representation    LINUX_IA64
Encoding               UTF-8  Unicode (UTF-8)
```

Output 2. Output from PROC CONTENTS

While the FDA does not explicitly state requirements for data encoding, there are related standards that must be followed. Per the technical conformance guide, “Variable names, as well as variable and dataset labels should include American Standard Code for Information Interchange (ASCII) text codes only” (Rui, 2016). As for the contents of the data itself, the agency requires only the English language and expects sponsors to remove or convert non-ASCII characters collected in research before submission (U.S. Food and Drug Administration, 2015). This is an important requirement of which to be aware, as the use of free text fields in UTF-8 encoded source data has high potential to capture non-ASCII characters. Also, for reasons on which this paper will elaborate, it can be assumed that the extended-ASCII character set is included in the list of character that should be removed or converted.

HOW SAS TRANSCODES AND WHY ERRORS HAPPEN

In many situations in which SAS needs to transcode, SAS does the work for you. This is because of Cross-Environment Data Access (CEDA). CEDA allows SAS to do the work behind the scenes. Assuming no issues are encountered, you are left with a simple note in the log.

```
NOTE: Data file RAWDATA.AE.DATA is in a format that is native to another
host, or the file encoding does not match the session encoding. Cross
Environment Data Access will be used, which might require additional CPU
resources and might reduce performance.
```

Output 1. Log message showing note generated by CEDA.

In other situations, you may not be so lucky. SAS may be unable to transcode the data. To make matters worse, the log errors generated by SAS do not explicitly state where or why SAS encountered the issue.

```
ERROR: Some character data was lost during transcoding in the dataset
DATA.AE. Either the data contains characters that are not representable in
the new encoding or truncation occurred during transcoding.
```

Output 2. Log message showing the error generated by SAS when it is unable to transcode.

Though the error does not guide you to the issue, it gives some context as to why the issue could be happening. Transcoding errors can happen in SAS for two reasons:

1. The dataset being read into SAS has a character that is not representable in the session encoding.
2. Truncation occurred during transcoding.

The first potential issue has a much clearer explanation than the second. As mentioned before, UTF-8 encoding supports more than 120,000 characters. This is vastly greater than the 256 characters handled by Latin1. If there is no character to transcode to, then the character cannot be transcoded. For example, if in a UTF-8 encoded dataset, the Greek character “Δ” exists – then Latin1 has no compatible conversion for this, and thus SAS forces the error as the conversion was not successful.

As for the second issue, to understand it, we must first understand what is being truncated. When a SAS length is set on a character variable, the length is not in reference to the number of *characters* that may be contained, but rather the number of *bytes*. For an SBCS, these two things are synonymous. If 1 byte = 1 character, then a length of $4 will allow both 4 characters and 4 bytes. But in UTF-8 encoding, 1 character may consist of 1 to 4 bytes. For example, if we look at the word “sofá" on the byte level:

- LATIN1

```
  01110011 : 01101111 : 01100110 : 11100001
|    s     |    o     |    f     |    á     |
```

Figure 1. Outline of byte level data in Latin1 encoding.

- UTF-8

```
  01110011 : 01101111 : 01100110 : 11000011 : 10100001
|    s     |    o     |    f     |          á          |
```

Figure 2. Outline of byte level data in UTF-8 encoding.

Because Latin1 encoding is a SBCS, each character is represented by one single byte. The character “á” is within the extended ASCII character set, and thus Latin1 encoding is capable of representing it. In this encoding, the string “sofá” is made up by 4 bytes, and therefore a length of $4 is acceptable.

The trouble arises when we try to transcode this data to UTF-8 without any compensation for variable lengths. While the character “á” is made up on one byte in Latin1, in UTF-8 it is made up of 2 bytes. Given that the variable length is $4, the last byte of the string “sofá” will be truncated:

```
  01110011 : 01101111 : 01100110 : 11000011
|     s    |     o    |     f    |     ?    |
```

Figure 3. Outline of truncated byte level data in UTF-8 encoding.

Note that this truncation issue occurs when transcoding from Latin1 to UTF-8. Truncation itself is not of concern when moving from UTF-8 to Latin1, as the Latin1 will only require identical, if not shorter lengths to represent the same data as UTF-8 because it requires one byte per character. That being said, this assumes that the lengths in the UTF-8 encoded data are not already truncating any values. The string “sofá” can still be captured in UTF-8 data natively with a length of $4, but the last byte of “á” will be truncated off and the character will not display. When trying to transcode this truncated string from UTF-8 to Latin1, SAS will again error – but this time not because of the truncation itself, but because the first byte of “á” has no direct counterpart in Latin1, and thus the character is non-representable.

THE DANGER OF ENCODING = ANY | ASCIIANY

One thing that makes the issue of encoding differences particularly frustrating is how buried the problem itself can be. Clinical trials can collect a massive amount of data, and within the data may be a single character in an inconvenient location that renders SAS unable to transcode. The “encoding=any” or “encoding=asciiany” options will allow you to import your data and work with it in your session, but it is very important to understand how this works.

First, let us look at the syntax to use “encoding=asciiany”:

```
data myDS;
  set utf8.myDS (encoding=asciiany);
run;
```

The syntax is simple and requires little effort to import the data into SAS with no errors. But what is SAS doing behind the scenes to make this work? Once again, let us look at UTF-8 truncated string of “sofá” with a length of $4:

```
  01110011 : 01101111 : 01100110 : 11000011
|     s    |     o    |     f    |     ?    |
```

Figure 3. Outline of truncated byte level data in UTF-8 encoding.

By using the option “encoding=asciiany”, SAS ignores the UTF-8 encoding all together. Instead, SAS will read the data and interpret every byte using their single byte ASCII coded values. SAS cannot magically make up the data lost in truncation, so it needs to interpret the value as something. While the byte “11000011” does not have a coded value in UTF-8, it does have an ASCII coded value. Therefore, SAS will interpret the string as:

```
  01110011 : 01101111 : 01100110 : 11000011
|     s    |     o    |     f    |     Ã    |
```

Figure 3. Interpretation of truncated byte level data in UTF-8 encoding using “encoding=asciiany”.

Now it should be clear that this option should not be used as a *fix,* but rather a temporary workaround to be able to access the data. To make matters worse, this option will not only impact truncated characters like in our string “sofá”, but rather *any character made up of 2 bytes or greater* in the UTF-8 encoded data. For example:

Table 1

Binary	UTF-8	ASCII
11000011:10100001	á	Ã¡
11000011:10111100	ü	Ã¼

Binary	UTF-8	ASCII
11000011:10111000	ø	Ã¸
11000011:10110001	ñ	Ã±

Table 1. UTF-8 encoded double byte characters and how they will be converted using "encoding=asciiany"

IDENTIFYING AND ISOLATING YOUR ISSUES

With mountains of data on hand, it is important to have a tool to isolate and reveal the characters potentially causing issues. The code examples displayed in this paper hinge on modifications of the following syntax (Ziem, 2011).

```
<variable>=prxchange('s/[\x20-\x7F]//', -1, <source>);
```

The function PRXCHANGE allows you to use Perl regular expressions. A regular expression is a special text string for describing search patterns. In a programming language like Perl, they can also be used to do things such as global replaces. The syntax of PRXCHANGE is defined as (PRXCHANGE Function, n.d.):

- PRXCHANGE(perl-regular-expression | regular-expression-id, times, source)

In the code example, the fields are utilized as follows:

- perl-regular-expression | regular-expression-id

  Perl regular expression executing a global replace that specifies to remove the lower-ASCII character set, defined by hexadecimal character codes. With only slight changes to this statement, the macro can be modified to do a number of different things. In its current state, it removes the lower ASCII character set, but it can be changed to:

    Replace values outside the lower-ASCII character with other values (see "Locating issues in the data and adding preventative checks")

    Remove characters outside the lower ASCII character set (see "Cleaning it all out")

- Times

  In these examples, our goal is to remove or replace all targeted characters in the string. By specifying '-1', this tells the function to replace matching patterns until the end of source is reached. In these examples, our goal is to remove or replace all targeted characters in the string.

- Source

  This is the target string or variable on which the Perl regular expression will be executed.

Following is an example of how this function may be used to remove ASCII characters from a string and leave only the non-ASCII characters:

```
data example1;
  var1="ßet ÿöû c@ñ'† get ®id of mÊ";
  var2=prxchange('s/[\x20-\x7F]//', -1, var1);
run;

proc print data=example1;
run;
```

Output 3 shows the ASCII characters being removed

```
Obs                  var1                      var2

 1      ßet ÿöû c@ñ'† get ®id of mÊ     ßÿöûñ†®Ê
```

Output 3. Output from a PROC PRINT Statement.

VAR1 contains a string of characters consisting of a mix between characters in the lower-ASCII and extended-ASCII character set. Within VAR2, all lower-ASCII characters have been removed. The benefit of this is to troubleshoot data that may be problematic. If you are to fix the problem, you first must understand what is causing it in the first place. If a transcoding error is encountered, the problem is not coming from lower-ASCII characters, but rather characters outside the lower-ASCII character range. This syntax is a first step towards isolating those characters.

PUT THE CODE TO WORK

This function can be very powerful in creating scripts that help show you where your issue characters are, isolate the issue characters, or remove them completely. While these steps will not fix the fundamental problem, they can give a strong leg up towards understanding the problem and deciding an approach. The following code examples all use this example dataset:

```
Obs     var1         var2     var3           var4      var5     var6     var7

 1      ßet          ÿöû      c@ñ'†          get†      ®id      Øf       mÊ
 2      ßµ†          whÿ      wØµld          ÿØµ       want†    †ø       anÿwaÿ?
 3      ßecause      sØme     charac†ers     dØn'†     like     †Ø       †ranscØde
```

Input 1. Data to be used in code examples

ISOLATING YOUR ISSUES

At this point, we understand that the lower-ASCII character set, represented by a single byte in both Latin1 and UTF-8 encodings, is not the source of the issue. These characters have no issue transcoding between encodings, but rather the characters outside this list may give you trouble. Gathering a list of the unique characters within your dataset allows you to examine what could be causing the issue and decide your next move. The following macro will isolate the unique characters outside of the lower-ASCII character set.

```
%macro isolate(inds=);
  *Gather character variable names;
  proc contents data=&inds. noprint out=vars (where=(type=2) keep=name
type);
  run;

  *Compress macro for removing non-ascii characters;
  %macro rmeasci(var);
    &var.=prxchange('s/[\x20-\x7F]//', -1, &var.);
  %mend;

  *Loop macro;
  %macro dataset;
  %do j=1 %to %eval(&cnt.);
     %let cds=%scan(&vars.,&j);
      %rmeasci(&cds);
  %end;
  %mend dataset;
```

```
    *Run macro through dataset;
    data non_&inds. (keep=bad_char where=(not missing(bad_char)));
      set &inds.;
      %dataset;
      bad_char=cats(&vars2.);
    run;

    proc sql noprint;
      select distinct bad_char into: bad_char separated by '' from
  non_&inds.;
    run;

    data non_&inds.2 (rename=(string=bad_char));vi
      length STRING $1000;
      /*Set length of string*/
      string="&bad_char.";
      l1=length(string);
      /* Eliminate duplicates in the string */
      do i = 1 to l1-1;
        t1=substr(string,i,1);
        if t1 ^= '00'x then do j = i+1 to l1;
          if t1=substr(string,j,1) then substr(string,j,1)='00'x;
        end;
      end;
      * the COMPRESS function is used to remove the nulls;
      string=strip(compress(string,'00'x));
      keep string;
    run;

    proc print data=non_&inds.2;
    run;
  %mend isolate;
```

The output of the macro shows you the unique list of non-ASCII characters in your data.

```
Obs      bad_char

1       ß†ÿöûñ®ØÊµø
```

Output 4. Output from PROC PRINT showing unique non-ASCII characters

LOCATING ISSUES IN THE DATA AND ADDING PREVENTATIVE CHECKS

Isolating the characters causing issues helps, but it does not show you where the characters are in the data. It is also important to understand which observations and variables themselves have the non-ASCII character. First, this requires a modification to our non-ascii function.

```
<variable>=prxchange('s/[^\x20-\x7F]/XX/', -1, <source>);
```

Two changes have been made. The “^” was added before “\x20” to specify that we are replacing characters outside the ASCII character set. Also, “XX” was added after the forward back slash as the “replace with” value. The following macro will loop through a dataset, replace all characters outside the lower-ASCII character set with XX, and then compare the changes back to the original dataset – allowing you to view the individual records where non-ascii characters were discovered.

```
%macro compare(inds=);
  *Gather character variable names;
  proc contents data=&inds. noprint out=vars (where=(type=2) keep=name 
type);
  run;

  *Compress macro for removing non-ascii characters;
  %macro rmeasci(var);
      &var.=prxchange('s/[^\x20-\x7F]/XX/', -1, &var.);
  %mend;

  *Loop macro;
  %macro dataset;
  %do j=1 %to %eval(&cnt.);
     %let cds=%scan(&vars.,&j);
      %rmeasci(&cds);
  %end;
  %mend dataset;

  *Set macro variables for character variables and counts;
  proc sql noprint;
    select name into: vars separated by ' ' from vars;
    select count("&vars.",' ')+1 as cnt into: cnt from vars;
  quit;

  *Run macro through dataset;
  data non_&inds.;
    set &inds.;
    %dataset;
  run;

  *Compare results back to original;
  proc compare base=&inds. compare=non_&inds. outbase outcomp outnoequal 
noprint out=non_diff;
  run;

  *Print discrepancies;
  proc print data=non_diff;
    var _type_ _obs_ &vars.;
  run;

  *Remove working datasets;
  proc datasets library=work nolist;
    delete non_&nds. non_diff vars;
  run;
%mend compare;
```

The print resulting from the macro shows each issue record back to back. The original observations show up on the BASE lines. On the COMPARE lines, characters outside of the lower-ASCII character set have been replaced with “XX”.

```
Obs _TYPE_   _OBS_  var1      var2   var3         var4    var5    var6  var7

 1  BASE      1     ßet       ÿöû    c@ñ'†        get     ®id     Øf    mÊ
 2  COMPARE   1     XXeXX     XXXXXX c@XX'XX      geXX    XXid    XXf   mXXXX
 3  BASE      2     ßµ†       whÿ    wØµld        ÿØµ     want    †ø
anÿwaÿ?
 4  COMPARE   2     XXXXXX    whXX   wXXXXld      XXXXXX  wanXX   XXXX
anXXwaXX?XX
 5  BASE      3     ßecause   sØme   charac†ers   dØn'†   like    †Ø
†ranscØde
 6  COMPARE   3     XXecause  sXXme  characXXers  dXXn'XX like    XXXX
XXranscXXdeXX
```

Output 5. Output from PROC PRINT showing original record and non-ASCII characters replaced with XX

CLEANING IT ALL OUT

What if you are left without an option? The database is locked, extended-ASCII characters or non-ASCII characters are sprinkled throughout, and the agreement is that you will remove them at the CDISC level. To be clear, this is *far* from ideal. If fixing the issues at the source is not an option, then you should consider developing an algorithm to replace the non-ASCII characters with acceptable ASCII counterparts (i.e. “µ” with “u”). In this scenario, removing the non-ASCII characters is not the *best* option; it is the *only one available*. Additionally, for this to truly be an appropriate case, the characters to be removed would more likely be invisible characters – things such as carriage returns or control characters within the extended-ASCII character set. It is *not* recommended to use this method if the characters removed are of some sort of value. This again requires a slight modification to our non-ASCII function.

```
<variable>=prxchange('s/[^\x20-\x7F]//', -1, <source>);
```

Now, instead of replacing characters with “XX”, the characters non-ASCII characters are replaced with nothing. The following macro will loop through each variable on each observation of your dataset and simply remove the non-ASCII characters encountered.

```
%macro cleanse(inds=);
  *Gather character variable names;
  proc contents data=&inds. noprint out=vars (where=(type=2) keep=memlabel
name type);
  run;

  *Grab dataset label to reapply;
  proc sql noprint;
    select distinct strip(memlabel) into: domlabel from vars;
  quit;

  *Compress macro for removing characters outside the lower-ASCII character
set;
  %macro rmeasci(var);
    &var.=prxchange('s/[^\x20-\x7F]//', -1, &var.);
  %mend;

  *Loop macro;
  %macro dataset;
  %do i=1 %to %eval(&cnt.);
     %let cds=%scan(&vars.,&i);
      %rmeasci(&cds);
  %end;
  %mend dataset;
```

```
    *Set macro variables for character variables and counts;
    proc sql noprint;
      select name into: vars separated by ' ' from vars;
      select count("&vars.",' ')+1 as cnt into: cnt from vars;
    quit;
    *Run macro through dataset;
    data &inds. (label="&domlabel.");
      set &inds.;
      %dataset;
    run;

    proc datasets library=work nolist;
      delete vars;
    run;
  %mend cleanse;
```

The dataset resulting from the macro is stripped of characters outside the lower-ASCII character set.

```
Obs     var1       var2     var3          var4     var5     var6     var7

 1      e                   c@'           ge       id        f       m
 2                 wh       wld                    wan               anwa?
 3      ecause     sme      characers     dn'      like              ranscde
```

Output 6. Output from PROC PRINT showing original only ASCII characters remaining

EXAMPLES AND TIPS FOR RESOLVING ENCODING ISSUES

Some of the most common sources for special characters are MH, AE, CM and CO related data sources as they contain free-text fields. And within these, the characters that are most commonly known to cause issues downstream are control characters (examples line feed - ^J; carriage return - ^M). These are usually hidden in your data and will not be printed, so they are really hard to find normally.

Below is an example of a case where such hidden characters were identified using the compare macro discussed above:

```
Obs     _TYPE_      _OBS_        CMINDC_CODV

 1      BASE         185     Hypertension
 2      COMPARE      185     Hypertension XXXX
 3      BASE         197     Chronic respiratory failure
 4      COMPARE      197     Chronic respiratory failureXXXX
 5      BASE         198     Nutrition improvement
 6      COMPARE      198     Nutrition improvementXXXX XXXX
```

Output 7. Output from PROC PRINT showing location of control characters

You can replace these by using the cleanse macro discussed in this paper. If you are absolutely sure that the only issue with your data is related to control characters then you can simply use the compress function to clean them out as follows:

```
  var1=compress(var1,,’kw’);
```

Here 'kw' stands for 'keep writable', this tells SAS to clean out all the non-printable characters in the variable in question. The key here is to first know what type of special characters are in your data and in which fields in order for you to determine the best way to fix the problem.

SWITCHING A SESSION ENCODING

So far, we have dealt with situations where the session encoding is different from the file encoding. However, the easiest way to rid yourself of encoding errors is to match the encoding of your SAS session to the encoding of your dataset. The instructions for doing this vary between versions of SAS and your operating system, but are readily available through SAS support. Though this is the simplest solution, there are some problems to take into consideration.

First, you may not have control over this. If you own your own distribution of SAS and have administration privileges on your machine, then yes – you can change the configuration. On the other hand, if you are working on your company system then this change has much larger implications, and the decision should be thoroughly thought through. In an ideal world, your SAS system would be set up so that you can easily interchange between Latin1/Wlatin1 encoding and UTF-8, as this would cover most of your encoding needs.

With these thoughts in mind, from this point forward we will assume that your SAS session encoding has been switched to UTF-8.

CONSIDERATIONS FOR WORKING IN A UTF-8 ENCODED SESSION

Below is an example of vital signs data for temperature that might be recorded as "97°F", the "°" is again part of the extended-ascii character list and could be replaced with the word "degrees" or can be dropped completely since it is understood that the temperature is collected in degrees. The replacement or dropping of this character seems like a simple enough task, however, this can produce unexpected results if not done with proper understanding of the UTF-8 encoded session, as demonstrated here:

```
data vitals;
  var1="97°F";
  var1_translate=translate(var1,'','°');
run;
```

Resultant dataset:

```
Obs     var1      var1_translate

 1      97°F       97   F
```

Output 8. Output from PROC PRINT showing issues with replacement of characters

If examined closely, you can see that there are 2 blank spaces introduced between 97 and F when we used translate function to replace "°" with a space. The reason is that this SAS function is byte-based and therefore replaces "°", which is 2-bytes in UTF-8 session, with a double space. SAS introduced K-functions to deal with such situations. These are character-based functions and can easily correct this result for us. You can use 'ktranslate' to replace "°" with a single space as shown below:

```
data vitals;
  var1="97°F";
  var1_translate=translate(var1,'','°');
  var1_ktranslate=ktranslate(var1,'','°');
run;
```

Resultant dataset:

```
Obs    var1     var1_translate   var1_ktranslate

 1     97°F      97  F            97 F
```

Output 8. Output from PROC PRINT showing fixed replacement of characters

There are several k-functions that help with such scenarios. Please see table below for a few such SAS functions that have a k-function counterpart:

Regular SAS Function	**K Function**
LEFT/RIGHT	KLEFT/KRIGHT
TRIM	KTRIM
STRIP	KSTRIP
LENGTH	KLENGTH
SCAN	KSCAN
SUBSTR	KSUBSTR
UPCASE/LOWCASE/PROPCASE	KUPCASE/KLOWCASE/KPROPCASE
REVERSE	KREVERSE

Table 1. Functions with available K counterparts

Not all K-functions support 4-byte characters, below is a list of 4-byte character functions that was released in SAS 9.4:

```
KSUBSTR
KCOMPRESS
KREVERSER
KTRANSLATE
KTRUNCATE
KUPDATE
KRIGHT
%KCMPRES macro function
```

Table 2. K Functions that support 4-byte characters as of SAS 9.4

Another common issue you might run into in UTF-8 encoding is when trying to import flat files into your session. If we are importing a Latin1 encoded file that contains extended-ascii text into UTF-8, then we should tell the SAS session how to read these values in correctly, otherwise this will produce unexpected results. Consider the example where a csv file containing lab data is read into UTF-8 encoded session:

These are the contents of the labs.csv file being read into a UTF-8 encoded session:

lbunit	**lborres**
µmol/L	0.9

Input 2. Example of labs.csv

Below is the SAS code that would help you read this file in, however you can clearly see that there is something wrong with your resultant SAS dataset.

```
proc import datafile="<your path here>/labs.csv”
                   out=labs
                   dbms=csv replace;
                   getnames=yes;
run;
```

Resultant dataset:

```
Obs    lbunit          lborres

 1      �mol/L            0.9
```

Output 9. Output from PROC PRINT showing issue with µ

This happens because the flat file stores the extended-ascii character “µ” in a 1-byte character, while the UTF-8 encoded session that is used to read the file in is expecting to see this as a 2-byte character. The result is that SAS reads in and stores gibberish. The below modification to your SAS code will transcode your results correctly (please note that the lrecl value is set to 32767 on purpose, otherwise SAS defaults to an lrecl value of 256):

```
filename ldat "<your path here>/labs.csv" encoding="latin1" lrecl=32767;
proc import datafile=ldat
                       out=labs
                       dbms=csv replace;
                       getnames=yes;
run;
```

Resultant dataset:

```
Obs    lbunit          lborres

 1      µmol/L            0.9
```

Output 10. Output from PROC PRINT showing fixed µ

However, this value of “µ” must now be replaced with a value from the lower-ascii character list in order to meet the FDA data submission guidelines. As explained earlier, this can be easily accomplished by replacing it with “u” (and don’t forget to use the K-functions!).

It is very common to see ‘truncation occurred’ warnings when CEDA tries to transcode your Latin1 or Wlatin1 encoded SAS data into an UTF-8 encoded session. This happens because the variable length is not enough to transcode the information correctly in UTF-8. In other words, some data that is a single byte in Latin1/Wlatin1 encoded file is now two or more bytes UTF-8 encoding, so more length is needed to store the same character value in the UTF-8 encoded data. We have discussed a case earlier where a flat file was read in by using the ‘encoding=’ option to tell the UTF-8 session to expect a Latin1 encoded file in order to adjust the lengths while transcoding into UTF-8. The way to achieve a similar result with SAS datasets is by using the CVP (character variable padding) engine. CVP pads character variable lengths with more bytes to increase their size, thus addressing the truncation issue. You can use CVP explicitly as shown below:

```
libname <your libname here> cvp <your data location here>;
```

This results in a default variable length expansion of 1.5 times, which would be sufficient for most data. However, in some situations you may have shorter length variables that contain a character that requires 4 bytes in UTF-8 encoding, and thus requires more padding. In these cases, you will have to use a higher expansion rate. By implicitly specifying CVPBYTES= or the CVPMULTIPLIER= options you can achieve this. CVPBYTES operates by expanding the character variables by the number of bytes specified (values can be 0 to 32766; remember lrecl=32767 from the flat file example?). CVPMULTIPLIER on the other hand multiplies the character variable lengths by the factor provided (values can go from 1 to 5). You would use these as shown below:

```
libname <your libname here> cvp <your path here> cvpbytes=10;
                            (OR)
libname <your libname here> cvp <your path here> cvpmultiplier=2;
```

In the case of cvpbytes=10, if a character variable length in a dataset is 8, after expansion it will have a length of 18 (8+10). In case you use cvpmultiplier=2, the same character variable of length 8 will have a new length of 16 (8*2).

It must be kept in mind that there are some caveats to using CVP:

- It only supports SAS datasets
- Dataset sizes are increased, which will increase the amount of storage needed for your data
- SAS performance is reduced
- It will not work on variables with the attribute ‘transcode=no’
- Indexing does not work when CVP is in use.

CONCLUSION

When working with UTF-8 encoded data, most of the time SAS does the work for you. Cross-Environment Data Access is a very useful tool built into SAS that transcodes your data behind the scenes, but due to the nature of encoding, it cannot work flawlessly each and every time. SAS cannot transcode a character if it is not representable by both the source encoding and the session encoding. Additionally, with the multi-byte nature of UTF-8, there is a risk for truncation of values – which is not of concern in Latin1 encoding.

Despite the many benefits of UTF-8 encoding, caution must be taken when working on a CDISC submission to the FDA. Ultimately, whether a Latin1 or UTF-8 session encoding is used to create your CDISC datasets is not important. If a UTF-8 encoded session is used, then transcoding issues when working with UTF-8 encoded source data will be avoided – but considerations must be taken to handle UTF-8 encoded data appropriately. For many functions, SAS has k-function counterparts that are specifically built to handle MBCS data. Additionally, if you read Latin1 encoded data into a UTF-8 encoded session, you must consider if there are extended-ASCII characters in the dataset. If so, the CVP engine may need to be used on your LIBNAME statement.

If a Latin1 session encoding is used, then any errors that occur upon import of the source data must be addressed and understood, as work-arounds like the “encoding=asciiany” option are not acceptable solution. The bottom line is that non-ASCII characters must be removed or converted at some point during development.

Given that issues with non-ASCII characters and transcoding route back to the source, it is best to start these checks at the source itself. If non-ASCII characters are never entered into the data in the first place, then issues downstream will be prevented before they can occur. If the “bad” characters make it into the data anyway, then the source data is still the optimal place to have the issue fixed.

Unfortunately, this is not always an option – but the problem still needs to be fixed somewhere. One potential option to eliminate non-ASCII characters at the SDTM level is to determine replacements. Many characters may have an ASCII counterpart that would be an acceptable replacement. For example, “μ” could be replaced with “u” without too much concern. If the non-ASCII characters that exist in the source data are all invisible characters, then these could likely be eliminated without significant impact to the data or information loss. Ultimately, decisions must be made based on the issue at hand, and any changes made at the SDTM level should be documented in the Study Data Reviewers Guide (SDRG).

Given the FDA requirement of restricting data to the ASCII character set and submitting data in the English language, additional review measures should be put in place throughout the course of a trial to ensure submission readiness. Not only can the code presented in this paper be used to evaluate and understand problems with encoding at hand, but also to place preventive checks on your data throughout the process of developing your datasets, both CDISC and non-CDISC. Understanding and properly handling any encoding issues encountered in your data is crucial to maintaining quality and traceability. Attacking the issues early on helps ensure that data sent to the FDA will be compatible downstream and will help prevent any last minute delays before submission.

REFERENCES

- Dutton, D. (2015). Retrieved from http://www.lexjansen.com/phuse/2015/DH/DH03_ppt.pdf
- PRXCHANGE Function. (n.d.). Retrieved from SAS CUSTOMER SUPPORT: http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a002601591.htm
- Rui, Li. (2016, March 7). personal communication.
- Sample 39036: Eliminating duplicate characters in a string. (2010, June 25). Retrieved from SAS CUSTOMER SUPPORT: http://support.sas.com/kb/39/036.html
- Ziem, A. (2011, March 30). Strip Non-Printable ASCII Characters (SAS). Retrieved from Heuristic Andrew: https://heuristically.wordpress.com/2011/03/30/strip-non-printable-ASCII-characters-sas/
- COMPRESS Function. Retrieved from SAS CUSTOMER SUPPORT: http://support.sas.com/documentation/cdl/en/lrdict/64316/HTML/default/viewer.htm#a000212246.htm
- K function counterparts for regular functions (Internationalization Compatibility for SAS String Functions). Retrieved from SAS CUSTOMER SUPPORT: http://support.sas.com/documentation/cdl/en/nlsref/61893/HTML/default/viewer.htm#a003274281.htm
- K functions that support 4-byte characters. Retrieved from SAS CUSTOMER SUPPORT: http://support.sas.com/kb/55/413.html
- Specify an encoding for reading external files. Retrieved from SAS DOCUMENTATION:

  http://documentation.sas.com/?docsetId=nlsref&docsetTarget=n0utx4w7x4exijn1wt7pk0srrxz5.htm&docsetVersion=9.4&locale=en
- Avoiding Character Data Truncation by Using the CVP Engine. Retrieved from SAS CUSTOMER SUPPORT: http://support.sas.com/documentation/cdl/en/nlsref/61893/HTML/default/viewer.htm#a002604734.htm

CONTACT INFORMATION

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

Name:	Michael Stackhouse
Enterprise:	Covance, Inc.
Work Phone:	(610) 513-4609
E-mail:	Michael.Stackhouse@Chiltern.com

Name:	Lavanya Pogula
Enterprise:	Covance, Inc.
Work Phone:	(330) 812-0535
E-mail:	Lavanya.Pogula@Chiltern.com