Paper 4645-2020

# Turn Yourself into a SAS® Internationalization Detective in One Day: A Macro for Analyzing SAS Data of Any Encoding

Houliang Li, HL SASBIPROS INC

## ABSTRACT

With the growing popularity of using Unicode in SAS®, it is more and more likely that you will be handed SAS data sets encoded in UTF-8. Or at least they claim those are genuine UTF-8 SAS data. How can you tell? What happens if you are actually required to generate output in plain old ASCII encoding from data sources of whatever encoding? This paper provides a solution that gives you all the details about the source data regarding potential multibyte Unicode and other non-ASCII characters in addition to ASCII control characters. Armed with the report, you can go back to your data provider and ask them to fix the problems that they did not realize they have been causing you. If that is not feasible, you can at least run your production jobs successfully against cleaned data after removing all the exotic characters.

## INTRODUCTION

Unicode is a computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems, according to Wikipedia. It concerns character variables and their values only. UTF-8, or utf-8, is the most popular implementation of Unicode. When you install SAS here in North America, whether on a PC or server or in a multitier deployment, UTF-8 has been one of the three encodings installed by default for many years, with the other two being Wlatin1 / Latin1 (English) and Shift-JIS (DBCS). Of course, you are free to install additional encodings for other languages such as French, German, Russian, and Chinese, etc. if you feel you may need to process data or generate output encoded in those languages. With the introduction of SAS Viya in recent years, UTF-8 became the default encoding, period! Although you do get the option to change to one of several encodings at the system level with more current versions of SAS Viya, being the default encoding says a lot about the importance of UTF-8. Don't be surprised if all SAS data sets from SAS Viya come in UTF-8 encoding. Say goodbye to your favorite Wlatin1 encoding on Windows or Latin1 encoding on Linux / Unix.

In reality, however, not many places are actively using SAS Viya at the moment, even though it is trending up. If you are somehow worried about this SAS Internationalization thing (I18N for short because there are 18 letters between the first letter I and last letter N), it is probably premature for the foreseeable future. For most companies and government agencies in North America, their data sources and output will still be in English, and Wlatin1 or Latin1 encoding should continue to serve them well.

If you are new, or relatively new, to the I18N band wagon or the concept of character encoding in general, encoding is literally mapping characters such as letters and symbols to numbers (called code points) on a character map (called a code page). The ASCII table is perhaps the most famous character map. Computers look smart and scary, but they are actually dumb. All they know and can handle is only bits, i.e., binary digits, or 0s and 1s. It is just that they can process those bits incredibly fast. ASCII, or American Standard Code for Information Interchange, is one of the earliest computer encoding schemes. It initially used only 7 bits, i.e., from 000 0000 to 111 1111, or 0 to 127 in decimal form, which means it could only represent a total of 128 unique characters. The first 32 (values 0 to 31) are control characters

and usually unprintable or invisible, and the last one is the "Delete" control character, also unprintable. The rest include uppercase and lowercase letters, decimal digits, some common symbols and punctuation marks, and the space. For example, the uppercase letter "S" is represented by the decimal number 83 (or 101 0011 in binary), which is its code point in ASCII. Similarly, the lowercase letter "s" has code point 115 (or 111 0011 in binary), and the space character, arguably the first printing or visible character in ASCII, occupies code point 32 (or 010 0000 in binary). Just Google "ASCII table" to see how your favorite letter or lucky number is represented in ASCII.

For the purpose of understanding this paper, we only need to know about three more encodings really: Wlatin1, Latin1, and UTF-8.

ISO 8859-1, also known in SAS as Latin1, adds one more bit to ASCII, so it ranges from 0000 0000 to 1111 1111 in binary, or 0 to 255 in decimal. That means it can represent 256 unique characters. The first 128 are the same ASCII characters, and the added 96 contain characters sufficient for the most common Western European languages.

Wlatin1, or Windows Latin1, is officially known as Windows 1252, or Code Page 1252 or cp1252. It is very similar to Latin1, except it has some additional symbols in the range 128 to 159 whereas Latin1 has nothing for those code points (hence only 96 additional characters out of 128 possible spots). Latin1 is the default encoding for SAS on Linux / Unix, while Wlatin1 is the default for Windows SAS. They are both exactly one byte in size. Table 1 below shows all the printable characters that are available in WLATIN1. LATIN1 includes all of the same characters except those enclosed in the blue rectangle:

| | _0 | _1 | _2 | _3 | _4 | _5 | _6 | _7 | _8 | _9 | _A | _B | _C | _D | _E | _F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2_ | (space) | ! | " | # | $ | % | & | ' | ( | ) | * | + | , | - | . | / |
| 3_ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | : | ; | < | = | > | ? |
| 4_ | @ | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O |
| 5_ | P | Q | R | S | T | U | V | W | X | Y | Z | [ | \ | ] | ^ | _ |
| 6_ | ` | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o |
| 7_ | p | q | r | s | t | u | v | w | x | y | z | { | \| | } | ~ | DEL |
| 8_ | € | | ‚ | ƒ | „ | … | † | ‡ | ˆ | ‰ | Š | ‹ | Œ | | Ž | |
| 9_ | | ‘ | ’ | “ | ” | • | – | — | ˜ | ™ | š | › | œ | | ž | Ÿ |
| A_ | NBSP | ¡ | ¢ | £ | ¤ | ¥ | ¦ | § | ¨ | © | ª | « | ¬ | (shy) | ® | ¯ |
| B_ | ° | ± | ² | ³ | ´ | µ | ¶ | · | ¸ | ¹ | º | » | ¼ | ½ | ¾ | ¿ |
| C_ | À | Á | Â | Ã | Ä | Å | Æ | Ç | È | É | Ê | Ë | Ì | Í | Î | Ï |
| D_ | Ð | Ñ | Ò | Ó | Ô | Õ | Ö | × | Ø | Ù | Ú | Û | Ü | Ý | Þ | ß |
| E_ | à | á | â | ã | ä | å | æ | ç | è | é | ê | ë | ì | í | î | ï |
| F_ | ð | ñ | ò | ó | ô | õ | ö | ÷ | ø | ù | ú | û | ü | ý | þ | ÿ |

Table 1: The Wlatin1 Character Set

Just like Wlatin1 and Latin1, UTF-8 also has its first 128 characters identical to ASCII. However, that is where the similarity ends. While both Wlatin1 and Latin1 use 8 bits (or one

byte) and can represent up to 256 characters (remember Latin1 does not use all the available code points), UTF-8 uses 8 bits / one byte for the 128 ASCII characters only. It does not utilize the remaining 128 code points for anything at all! What makes UTF-8 so powerful is that it can use 16, 24, even 32 bits (or two, three, even four bytes!) to represent all the other characters in the world. It is an MBCS (Multi Byte Character Set). All those characters in Wlatin1 and Latin1 beyond ASCII have their corresponding representations in UTF-8 as either a two-byte or three-byte code point. In addition, UTF-8 has a unique bit pattern for each of those multibyte representations. For example, the first byte will always begin with bits 110, 1110, or 11110, indicating either two, three, or four bytes in total for the character. The remaining bytes always start with bits 10 to indicate they are continuing with the same character. After removing those indicator bits, a 16-bit / two-byte UTF-8 representation has 11 bits left (110X XXXX 10XX XXXX) to represent 2**11 = 2048 characters. The first 128 code points overlap ASCII as represented by the one-byte UTF-8, so the net increase is actually 2048 − 128 = 1920 characters. Similarly, a 24-bit / three-byte UTF-8 encoding can represent a total of 2**16 = 65536 characters (again notice the overlap with 2-byte UTF-8), and a 32-bit / four-byte UTF-8 representation can encode 2**21, or 2,097,152 characters. Clearly, two million+ is sufficient to represent all written human languages, dead or alive, plus a lot of other symbols such as emojis. So far, Unicode has actually defined less than 1.2 million code points. There are simply not enough alphabets and logograms and other symbols in the world to encode! I know you all like this emoji: 11110000:10011111:10011000:10000011 (hexadecimal equivalent: F0 9F 98 83), which is 😃 the smiley face. But do you also like this emoji? 11110000:10011111:10010010:10101001 (hexadecimal equivalent: F0 9F 92 A9). Maybe you do and have even used it too, but you need to figure it out first!

Just in case you are not so familiar with hexadecimal representations, it is a more efficient (read: shorter) way of representing binary values. It ranges from 0 to 15, with each of the 16 values denoted by decimal digits 0 through 9 and continuing from letters A through F (or a through f, since case does not matter), i.e. A = 10, B = 11, and F = 15, etc. Each hexadecimal (hex for short) symbol / digit corresponds to exactly four binary digits. For example, the letter "S" is 0101 0011 in binary, which adds up to 2**6 + 2**4 + 2**1 + 1 = 83 (in decimal). Its hex equivalent is 53, which adds up to 5 * 16**1 + 3 = 83 (in decimal). We know its code point in Wlatin1 / Latin1 / UTF-8 is 83. Everything matches perfectly. To really bring the concept home, let's examine a multibyte UTF-8 character. Given the following binary representation in UTF-8 (Please note: colons between bytes are for easy identification purposes only and not required): *1110*0010:*10*0001001:*10*100100, or its hex equivalent E2 89 A4, our task is to figure out its code point in Unicode. For the binary representation, we need to first remove the italicized bits because they are multibyte indicators. Next we assemble ALL the remaining bits (depicted in Bold) in the same order as listed and treat the resultant binary string as a single number, which should reveal the code point for the character: 0010 001001 100100 = 10 0010 0110 0100 = 2**13 + 2**9 + 2**6 + 2**5 + 2**2 = 8192 + 512 + 64 + 32 + 4 = 8804. This code point is the "less than or equal to" sign ≤ in Unicode. Please remember that the hex equivalent, E2 89 A4, cannot be directly converted into a decimal value to yield the code point because it is equal to the 24-bit binary string which contains several multibyte indicator bits. The nominal values of binary and hex representations are equal to a character's code point only in the ASCII range, like those for the letters "S" and "s". As a final note, you can look up the actual character in any encoding via a search engine like Google, be it Wlatin1 or Latin1 or UTF-8, by using either the binary string or hex value or code point. For example, the one-byte hex value AE (code point 174) in Wlatin1 and Latin1 represents the registered trademark symbol ®, which has the same code point in UTF-8 but a two-byte hex value of C2 AE.

## MANIPULATING THE ENCODING ATTRIBUTE OF SAS DATA SETS

Now that you know enough about SAS encoding, we can run some SAS code to experience firsthand how SAS deals with encoding, particularly when the SAS session encoding is different than the data set encoding. Below is a simple SAS program with only eight steps:

```
data test;
   string = "registered mark " || byte(174) ||" is at column 17 - the end";
run;

proc print;
run;

proc contents data=test;
run;

proc datasets library=work nolist;
   modify test / correctencoding="utf-8";
quit;

proc print;
run;

data _null_;
   set test;
   do column=16 to 18;
      char = substr(string, column, 1);
      put column= char= char=$hex2.;
   end;
run;

proc contents data=test;
run;

data testcopy;
   set test;
run;
```

We will run the code in a default SAS session on Windows, e.g. by selecting the "SAS 9.4 (English)" menu item, so the session encoding will be Wlatin1. After the first DATA step, the character variable string, with a length of 43 (you can count up the letters and spaces and dash!), should have the symbol ® at position 17. When we run the PRINT procedure, we do see the symbol at position 17, as shown in Figure 1.

| Obs | string |
|---|---|
| 1 | registered mark ® is at column 17 - the end |

Figure 1: PROC PRINT Showing the Variable Value and Position of the Symbol ®

The CONTENTS procedure confirms that the temporary data set test has the encoding of Wlatin1, as expected. See Figure 2.

The CONTENTS Procedure

| | | | |
|---|---|---|---|
| Data Set Name | WORK.TEST | Observations | 1 |
| Member Type | DATA | Variables | 1 |
| Engine | V9 | Indexes | 0 |
| Created | 09/03/2019 21:47:14 | Observation Length | 43 |
| Last Modified | 09/03/2019 21:47:14 | Deleted Observations | 0 |
| Protection | | Compressed | NO |
| Data Set Type | | Sorted | NO |
| Label | | | |
| Data Representation | WINDOWS_64 | | |
| Encoding | wlatin1 Western (Windows) | | |

Figure 2: PROC CONTENTS Output Showing the Data Set Encoding (Last Row)

Everything has been good so far. Now when we run the DATASETS procedure to change the encoding of the test data set from Wlatin1 to UTF-8, we do get the following warning:

WARNING: CORRECTENCODING was successful. MODIFY RUN group closed because the ENCODING value does not match the session encoding value.

There seems to be no cause for alarm since we are indeed changing the data set's encoding from the default session encoding to something else. The next PRINT procedure, however, gives us our first surprise!

| Obs | string |
|---|---|
| 1 | registered mark  is at column 17 - the end |

Figure 3: PROC PRINT Showing the Altered Variable Value with ® Missing

For some reason, the ® symbol is gone, with what looks like a blank space now in its place (Figure 3). What is more, we get an ominous warning in the log:

WARNING: Some character data was lost during transcoding in the dataset WORK.TEST. Either the data contains characters that are not representable in the new encoding or truncation occurred during transcoding.

That's weird! Let's find out what is in that space with the next DATA step (Figure 4). The corresponding code is provided below again for easy reference:

```
data _null_;
   set test;
   do column=16 to 18;
      char = substr(string, column, 1);
      put column= char= char=$hex2.;
   end;
run;
```

```
column=16 char=  char=20
column=17 char=▯ char=1A
column=18 char=  char=20
```

Figure 4: SAS Log Listing the Characters around Column 17 Where ® Used to Be

With this DATA step, we not only get the same warning about transcoding as above during the PRINT procedure, but the character at column 17, where ® used to be, is now a vertical rectangle, and its hex value is 1A. Looking up in our trusty ASCII table, we learn that 1A, with its code point at 26, means "Substitute" and is a control character. The SAS software chooses to display this character as a rectangle in the log, but presents it as a space (more like an unprintable character!) in the PRINT output. The ASCII table also tells us that the two characters on either side of column 17 are actually space characters, just like in the first PRINT output as well as in the first DATA step where the string variable is assigned.

We know that the PRINT procedure does not change the contents of the input SAS data set, so the only place the change can start has to be in the DATASETS procedure. It turns out that when we change the encoding attribute of a SAS data set, it is not just a simple label swap. It sows the seeds for future underlying bits changes to reflect the new encoding designation. In subsequent steps whenever this data set is accessed, SAS compares the encoding in the data set header with the session encoding and sees that they are not the same so it transcodes the data from UTF-8 (the artificially designated encoding of the data set) to WLATIN1 (the session encoding where data is to be processed). In this case, all the characters in the string variable are in the ASCII range except the ® symbol at column 17. SAS does not need to do anything about the ASCII characters because it is an exact match between Wlatin1 and UTF-8 for those characters. For the symbol at column 17, however, since it has a code point of 174 and is beyond ASCII, Wlatin1 and UTF-8 treat it very differently. While it is a legitimate one-byte character in Wlatin1 encoding (hex value AE, or 1010 1110 in binary), code point 174 needs to be a two-byte character in UTF-8 with a hex value of C2 AE. SAS detects the character AE as NOT a legitimate byte in the assigned encoding of UTF-8. As a result, SAS substitutes this invalid character conveniently with the one-byte "Substitute" control character (hex 1A) wherever it is encountered. Naturally, SAS spits out warnings about its actions and discoveries, although the warning from the DATASETS procedure is too generic to be helpful. The warning from both the PRINT procedure and the DATA step is spot on, although you do need to know which part the warning is specifically referring to. In any case, the "Substitute" character is what you have to deal with in place of the bad character from now on.

When we run the CONTENTS procedure again (Figure 5), we do see that the test data set now has the UTF-8 encoding:

| | | | |
|---|---|---|---|
| **Data Set Name** | WORK.TEST | **Observations** | 1 |
| **Member Type** | DATA | **Variables** | 1 |
| **Engine** | V9 | **Indexes** | 0 |
| **Created** | 09/03/2019 21:47:14 | **Observation Length** | 43 |
| **Last Modified** | 09/03/2019 21:47:14 | **Deleted Observations** | 0 |
| **Protection** | | **Compressed** | NO |
| **Data Set Type** | | **Sorted** | NO |
| **Label** | | | |
| **Data Representation** | WINDOWS_64 | | |
| **Encoding** | utf-8 Unicode (UTF-8) | | |

Figure 5**: PROC CONTENTS Output Displaying the Data Set's New Encoding**

This procedure also generates the following SAS note:

NOTE: Data file WORK.TEST.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.

Again, no cause for alarm. The data set test does have an encoding, UTF-8, that is different than the session encoding of Wlatin1 following the DATASETS procedure. Hopefully things will be fine from now on. Let's run the last DATA step, displayed below, and call it a day:

```
data testcopy;
   set test;
run;
```

Here is the log:

```
22   data testcopy;
23      set test;
NOTE: Data file WORK.TEST.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.
24   run;

ERROR: Some character data was lost during transcoding in the dataset WORK.TEST. Either the
       data contains characters that are not representable in the new encoding or truncation
       occurred during transcoding.
NOTE: The DATA step has been abnormally terminated.
NOTE: The SAS System stopped processing this step because of errors.
WARNING: The data set WORK.TESTCOPY may be incomplete.  When this step was stopped there were 0
         observations and 1 variables.
NOTE: DATA statement used (Total process time):
      real time           0.02 seconds
      cpu time            0.00 seconds
```

Figure 6: SAS Log with a CEDA NOTE, Transcoding ERROR and Failure WARNING

What a nasty surprise! Why can't we even copy the data? Did we do something wrong again? Actually, no. It is just the fact that SAS does not know if a data set's encoding has been altered improperly. It looks at the header of the current data set each time it needs to do something with it. If transcoding is needed (in this case, yes!), it does that. If transcoding fails (in this case, unfortunately yes), it sends an error. By persistently displaying errors and warnings about transcoding and data truncation, it is forcing users to go back and identify the root cause of their data problems, such as inappropriately changing the data set encoding without fully understanding its consequences. Well, we have no choice at this point but to try a different approach.

## DETECTING UNEXPECTED CHARACTERS IN SAS DATA OF ANY ENCODING

The DATASETS procedure above demonstrates the detrimental effect of changing the data set encoding after its creation. Is it possible to change the encoding attribute without compromising data? Here is an interesting scenario. Your data provider has recently handed you some SAS data sets and claims they are encoded in UTF-8. You have tried to use the data but got some transcoding / truncation errors in a Wlatin1 SAS session. Some data sets won't even open in a SAS program or batch job even though you can view them interactively in SAS, with a warning perhaps. However, when you do the same things in a UTF-8 SAS session, everything seems fine. You strongly suspect there is something wrong with the data sets or their encoding, for example, when you see a single-byte ® symbol following a product name in a Wlatin1 SAS session but the data set has been marked as encoded in UTF-8 where ® should be two bytes. Suppose you are required to produce SAS output in Wlatin1 encoding only because your customers demand it. It is time to expand our investigative tools beyond

using simple SUBSTR() functions and PUT statements, with the hope that we can come up with a robust solution.

Fortunately, the smart folks at SAS are on our side. The DATASETS procedure does provide an encoding option that will not alter your data, regardless of the original encoding. The new encoding, ASCIIANY, "specifies that no transcoding occurs when the mixed encodings are ASCII encodings," according to SAS documentation. ASCIIANY effectively treats each byte as its own single byte character, as if everything in a character variable had been encoded in Wlatin1 or Latin1. For example, the registered trademark symbol ® is encoded as C2 AE in a UTF-8 data set. When changing the data set's encoding to ASCIIANY, SAS effectively treats the two bytes as two separate characters in their own right instead of being parts of a two-byte character. In other words, in the new encoding, SAS now sees one character represented by C2, and another character represented by AE. Since we are opening the data set in a Wlatin1 SAS session, C2 is Â, the Latin capital letter A with circumflex. We already know AE by itself is our old friend ®. We can demonstrate this with some simple SAS code.

Run the steps below in a UTF-8 SAS session by choosing the "SAS 9.4 (Unicode Support)" option:

```
libname temp "c:\temp";

data temp.test;
   length string $ 44;
   string = "registered mark " || 'C2AE'x || " is at column 17 - the end";
run;

proc print data=temp.test;
run;
```

The PRINT procedure shows that the string variable is assigned correctly (see Figure 7). Please note that we have expanded the variable length from 43 to 44. This is to accommodate the required two-byte representation in UTF-8 for the ® symbol. In fact, even if we did not define the length explicitly, SAS would still correctly create the string variable with 44 bytes because it knows how long string needs to be to hold the assigned text value. You can try it for yourself.

| Obs | string |
|---|---|
| 1 | registered mark ® is at column 17 - the end |

Figure 7: PROC PRINT Displaying the Variable Value and Position of the Symbol ®

Now run the steps below in a Wlatin1 SAS session, i.e. "SAS 9.4 (English)":

```
libname temp "c:\temp";

proc print data=temp.test;
run;

proc contents data=temp.test;
run;
```

If you were expecting to see the character Â (hex C2) from Wlatin1 encoding in the PRINT output, you would be disappointed, although the CONTENTS procedure does confirm that the data set test was created with UTF-8 encoding. The PRINT output is exactly the same as in Figure 7. The secret lies in the SAS note that accompanies both the PRINT and CONTENTS procedures:

NOTE: Data file TEMP.TEST.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.

SAS knows that you are trying to print a UTF-8 data set in a Wlatin1 session, so it performs a little covert conversion, i.e. transcoding, behind the scenes for you whenever possible. In this case, it recognizes that C2 AE in UTF-8 is ®, which happens to have a code point in Wlatin1 encoding (AE), so it proceeds to print the symbol in the output using the Wlatin1 equivalent while also spitting out the NOTE to notify you about what is happening, although again it falls short in fully explaining its heroic deeds. Please note that Cross Environment Data Access, or CEDA, includes transcoding data on the fly. Unfortunately, joyful moments like this are quite rare considering the sheer number of characters encoded in UTF-8 versus the paltry few offered by the 128 code points in the extended ASCII range that Wlatin1 is capable of. It bears repeating that a Wlatin1 SAS session can only display characters encoded in Wlatin1, which are less than 256 in total. Most times you will get an error or warning, depending on your actions, about transcoding and data truncation when dealing with data initially created in another encoding like UTF-8. Whenever a Wlatin1 SAS session encounters a typical Unicode character such as Chinese word or Cyrillic letter or modern emoji, you will be abruptly reminded, once again, about SAS encoding and I18N.

Next, we will continue to run the code below in the same Wlatin1 SAS session:

```
proc datasets library=temp nolist;
   modify test / correctencoding=asciiany;
quit;

proc contents data=temp.test;
run;

proc print data=temp.test;
run;
```

There are no errors or warnings from these steps, and no SAS note about transcoding either. The DATASETS procedure shows the new encoding, us-ascii (see Figure 8), and the PRINT procedure (see Figure 9) delivers the much-anticipated letter A with a coolie hat! Finally, we are getting what we expected!

| Data Set Name | TEMP.TEST | Observations | 1 |
|---|---|---|---|
| Member Type | DATA | Variables | 1 |
| Engine | V9 | Indexes | 0 |
| Created | 09/04/2019 09:14:57 | Observation Length | 44 |
| Last Modified | 09/04/2019 09:14:57 | Deleted Observations | 0 |
| Protection | | Compressed | NO |
| Data Set Type | | Sorted | NO |
| Label | | | |
| Data Representation | WINDOWS_64 | | |
| Encoding | us-ascii ASCII (ANSI) | | |

Figure 8**: PROC CONTENTS Output Displaying the Data Set's New Encoding**

| Obs | string |
|---|---|
| 1 | registered mark Â® is at column 17 - the end |

Figure 9: PROC PRINT Showing the Variable Value and Identities of C2 and AE

Eagle-eyed SAS users will notice that in the data set with a new encoding label, the symbol ® is no longer at column 17 in the string variable. It has been pushed back one spot to column 18. Of all the surprises so far, this may be the least of them all.

Based on what we have learned so far, we are now ready to build our I18N detection SAS program that will alert us to any Unicode, or other encoding, characters in our SAS data, regardless of the encoding label that is currently attached to the data set. We will run the detection macro in a Wlatin1 SAS session, but you should be able to run it in a Latin1 session with minor path modifications only. We will use the SAS log to provide detailed information about any unexpected characters in the data. By "unexpected" we mean any ASCII control characters (code points 0 to 31 plus 127) which shouldn't be there in the first place, and any multibyte Unicode characters, or any non-ASCII characters of other encodings. We will replace the offending bytes with plain space characters after gathering all the useful information. At the end of the program, the data will be devoid of any problematic bytes, and an Excel report will detail the issues found which can be presented to the data provider for any corrective measures if desired.

Here is the full macro:

```
libname temp "C:\Path\To\My\Data";

%macro detect_bad_chars(library=temp, dataset=test, maxlength=2000);
/* Note: maxlength is the defined length of the longest character variable
         in the data set. It can be obtained by querying the dictionary.  */

/* Change the data set encoding to ASCIIANY to prevent transcoding */
proc datasets library=&library nolist;
   modify &dataset / correctencoding=asciiany;
quit;

/* Identify all character variables and put them into macro variables */
%let charvarcount = 0;

data _null_;
   length varname $ 32 vartype $ 1;
   set &library..&dataset;

   do until (varname = " ");
      call vnext(varname, vartype);

      if vartype = "C" and varname not in ("varname", "vartype") then do;
         count + 1;
         call symput("varname" || compress(count), trim(varname));
      end;
   end;

   call symputx("charvarcount", count);
   stop;
run;
```

```
/* Continue processing the data set only if it has char variables */
%if &charvarcount = 0 %then %do;
   %put The data set &dataset has no character variables. No need to check.;
   %goto EXIT;
%end;

%let badcharfound = 0;
data unexpected_chars (keep=dataset row variable value badchars)
     &library..&dataset (drop=dataset row variable value badchars note
                              noteprinted varlength byteindex byte bytehex
                              firsthexchar badcharfound count);
   attrib dataset label="Data Set Name" length=$32
          row label="Row Number"
          variable label="Variable Name" length=$32
          value label="Variable Value (Unexpected Bytes Replaced with Space)"
                length=$&maxlength
          badchars length=$1024 label=
                   "Unexpected Bytes and Their Locations in the Variable"
          note length=$100;
   retain badcharfound noteprinted 0;
   set &library..&dataset end=last;

%do index = 1 %to &charvarcount;
   badchars = "";
   count = 0;
   varlength = lengthn(&&varname&index);

   do byteindex = 1 to varlength;
      byte = substr(&&varname&index, byteindex, 1);
      bytehex = put(byte, $hex2.);
      firsthexchar = substr(bytehex, 1, 1);

      if firsthexchar in ("0", "1", "8", "9", "A", "B", "C", "D", "E", "F")
         or bytehex = "7F" then do;

         if firsthexchar in ("0", "1") or bytehex = "7F" then
            put "Found and replaced one ASCII control/nonprinting character";
         else do;
            if noteprinted = 0 then do;
               put "Note: If an extended ASCII character does not print " /
                   "or the printed character makes no sense, it may be " /
                   "part of a multibyte Unicode character. If two or " /
                   "three consecutive messages talk about finding and " /
                   "replacing a character in the extended ASCII range for " /
                   "the same variable in the same row and their byte " /
                   "positions are consecutive (e.g. 25, 26 or 40, 41, 42), "/
                   "these bytes probably form a genuine multibyte Unicode "/
                   "character. Otherwise, the data set may have just been " /
                   "mislabelled as utf-8 after its creation in another " /
                   "encoding. This message appears only once." /;

               noteprinted = 1;
            end;

            put "Found and replaced one non-ASCII character --> " byte "<--";
         end;

         note = "(row=" || compress(_n_) || " variable=&&varname&index " ||
```

```
               "column=" || compress(byteindex) || " hexvalue='" ||
               put(substr(&&varname&index, byteindex,1), $hex2.) || "')";
         put note;

         if badchars = "" then
            badchars = "Variable length: " || compress(varlength) ||
                       " -- Unexpected characters:";

         if count then
            badchars = trim(badchars) || " |";

         badchars = trim(badchars) || " Column " || compress(byteindex) ||
                    " - Hex Value '" || bytehex || "'";
         count + 1;
         badcharfound = 1;
         substr(&&varname&index, byteindex, 1) = " "; /* replace with space*/
         dataset = "&dataset";
         row = _n_;
         variable = "&&varname&index";
         value = &&varname&index;
      end;
   end;

   if lengthn(badchars) then
      output unexpected_chars;
%end;

   if last and badcharfound then
      call symput("badcharfound", "1");

   output &library..&dataset;
run;

/* Create an Excel report on ASCII control characters and Unicode found */
%if &badcharfound = 0 %then %do;
   %put The data set &dataset contains no unexpected characters.;
   %put As a result, no report will be created.;
   %goto EXIT;
%end;

title bold justify=left "Unexpected Characters Found in %upcase(&dataset)";
ods listing close;
ods excel file="C:\Path\To\My\Report\Unexpected_Characters_in_&dataset..xlsx"
          options(start_at="1,2"
                  embedded_titles="yes"
                  frozen_headers="4"
                  autofilter="2-3"
                  row_repeat="1-4"
                  sheet_name="Unexpected Characters");

proc print data=unexpected_chars label noobs ;
run;

ods excel close;
ods listing;

%EXIT:
%mend;
```

This SAS macro does not have many steps, but it does require an advanced understanding of SAS programming if you need to change it. After assigning the required library to point to the location of our target SAS data set, we immediately change the data set encoding to ASCIIANY to prevent any transcoding of data. Next, we identify and tag every character variable in the data set so we can check their values in each record. A macro loop is used inside the next DATA step to generate code for each character variable. For example, if the data set has 10 character variables, the DATA step will have 10 separate code blocks to check on each of them. It can be a VERY long DATA step when fully expanded, but why do we care? Code generation is one of the major functions of SAS MACRO programming. Within each block for a particular character variable, we use a DO loop to traverse the text value of the variable, byte by byte. If any ASCII control characters are found, they are flagged in the log (but not displayed since most of them are not printable) and also written to a special data set. Figure 10 shows some examples of ASCII control characters as reported to the log. If you look up the hex value 09 in ASCII, you will find that it is the horizontal tab. Technically it is printable, but you will never know it's there just by looking at it.

```
Found and replaced one ASCII control/nonprinting character
(row=1 variable=DataPageName column=39 hexvalue='09')
Found and replaced one ASCII control/nonprinting character
(row=2 variable=DataPageName column=39 hexvalue='09')
Found and replaced one ASCII control/nonprinting character
(row=3 variable=DataPageName column=39 hexvalue='09')
```

Figure 10: SAS Log Displaying ASCII Control Characters Found

Similarly, if any multibyte Unicode characters, or other non-ASCII characters of esoteric encodings are found, they are also flagged AND displayed in the log using the corresponding extended ASCII characters (see Figure 11). A special note is printed to the log reminding the user that those printed extended ASCII characters may not make sense at all if they are actually part of a multibyte character – they are not supposed to be meaningful in that case! Naturally, those non-ASCII instances are also written to the special data set, one variable / record combination per row. Figure 12 displays the reminder:

```
Found and replaced one non-ASCII character --> º <--
(row=3132 variable=CGISS1 column=24 hexvalue='BA')
Found and replaced one non-ASCII character --> Ð <--
(row=3132 variable=CGISS1 column=25 hexvalue='D0')
Found and replaced one non-ASCII character --> ¾ <--
(row=3132 variable=CGISS1 column=26 hexvalue='BE')
Found and replaced one non-ASCII character --> Ð <--
(row=3132 variable=CGISS1 column=28 hexvalue='D0')
Found and replaced one non-ASCII character --> ½ <--
(row=3132 variable=CGISS1 column=29 hexvalue='BD')
Found and replaced one non-ASCII character --> Ð <--
(row=3132 variable=CGISS1 column=30 hexvalue='D0')
```

Figure 11: SAS Log Showing Non-ASCII Characters Found

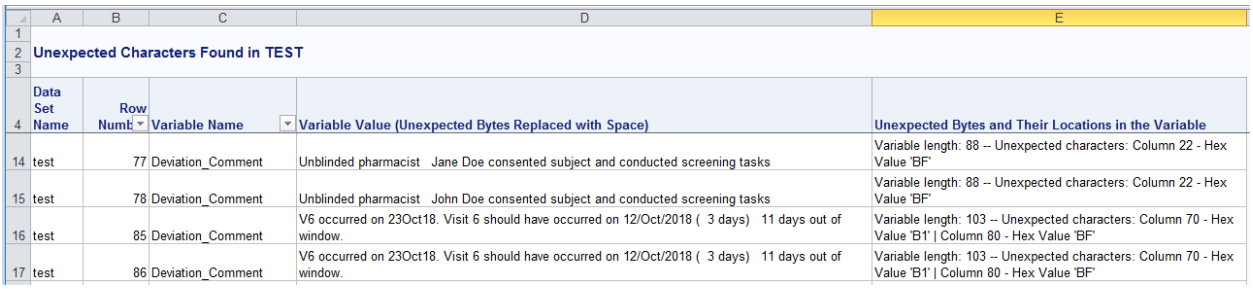
```
Note: If an extended ASCII character does not print
or the printed character makes no sense, it may be
part of a multibyte Unicode character. If two or
three consecutive messages talk about finding and
replacing a character in the extended ASCII range for
the same variable in the same row and their byte
positions are consecutive (e.g. 25, 26 or 40, 41, 42),
these bytes probably form a genuine multibyte Unicode
character. Otherwise, the data set may have just been
mislabelled as utf-8 after its creation in another
encoding. This message appears only once.
```

Figure 12: Custom Note Explaining Nonsensical Characters and Unicode

A warning is in place here. Since our goal is to remove all offending, unexpected characters from the SAS data, we need to make sure that the temporary variable names we use in this

big DATA step do NOT overlap any existing variable names in the data set. One way to do this is to use a special prefix or postfix on all temporary variable names so they have almost no chance of an overlap. It is not done in the macro above due to formatting and readability considerations for this paper.

Once all the unexpected characters have been identified and removed from the data set, an Excel report is generated using SAS ODS. This report provides detailed information about ALL changes made to the data, such as what was replaced and where, down to the byte location in an individual variable on a specific record. Figure 13 is part of a sample report:



| | A | B | C | D | E |
|---|---|---|---|---|---|
| 1 | | | | | |
| 2 | **Unexpected Characters Found in TEST** | | | | |
| 3 | | | | | |
| 4 | Data Set Name | Row Numb | Variable Name | Variable Value (Unexpected Bytes Replaced with Space) | Unexpected Bytes and Their Locations in the Variable |
| 14 | test | 77 | Deviation_Comment | Unblinded pharmacist  Jane Doe consented subject and conducted screening tasks | Variable length: 88 -- Unexpected characters: Column 22 - Hex Value 'BF' |
| 15 | test | 78 | Deviation_Comment | Unblinded pharmacist  John Doe consented subject and conducted screening tasks | Variable length: 88 -- Unexpected characters: Column 22 - Hex Value 'BF' |
| 16 | test | 85 | Deviation_Comment | V6 occurred on 23Oct18. Visit 6 should have occurred on 12/Oct/2018 (  3 days)  11 days out of window. | Variable length: 103 -- Unexpected characters: Column 70 - Hex Value 'B1' \| Column 80 - Hex Value 'BF' |
| 17 | test | 86 | Deviation_Comment | V6 occurred on 23Oct18. Visit 6 should have occurred on 12/Oct/2018 (  3 days)  11 days out of window. | Variable length: 103 -- Unexpected characters: Column 70 - Hex Value 'B1' \| Column 80 - Hex Value 'BF' |

Figure 13: Excel Report on Unexpected Characters Found and Subsequent Changes

Using the macro is easy. All you need to do is to first define your library where your target SAS data sets reside and then specify where you want your final report to go (in the first ODS EXCEL statement). After that, you can just call the macro by providing the libref and data set name. Please remember that the macro is designed to run in a Wlatin1 or Latin1 SAS session, not a UTF-8 session.

## DISCUSSION

SAS has been providing National Language Support (NLS) for many years. It has steadily evolved and expanded over the decades by adding native languages and DBCS (Double Byte Character Set), and finally Unicode in the form of UTF-8. Even though SAS Viya, supposedly the latest and greatest SAS offering, runs UTF-8 by default, most SAS users are not actually affected, now or in the near future. Sure, the idea is cool and forward-thinking. By adopting Unicode, users from all over the world no longer need to worry about language barriers in their SAS data. All writing systems are already represented in Unicode, along with a whole lot of symbols and icons. It will be the ideal world. Unfortunately, it will probably take many years before it can become a reality, if at all. Most companies are not global companies, and all governments are expected to support their own citizens primarily. That means running software, SAS included, geared towards their customers and citizens in the language they use should make more sense. Besides, the language selected, and encoding in SAS, need to match your data sources upstream and customers and users downstream. Until everyone has switched to Unicode, you are only courting trouble by being an early adopter. Do your vendors and customers have an incentive to adopt Unicode too? If plain old English / Wlatin1 or Latin1 is good for 99% of your use cases, why do you want to jump to Unicode just to accommodate the 1% special cases? It is not like there are no costs involved. An overly simplified analogy would be for a local store to make / put up additional signs in Japanese when almost all your customers speak English only.

If you are among the small minority of corporations that have global operations, switching to Unicode may make sense. If that is the case, SAS NLS provides a respectable set of tools to help you do your job in the new world. K functions (including macro functions), for example, can handle UTF-8 data with ease. By adopting K functions where available, your SAS programs will become encoding-proof – you won't need to update them in the future when processing data of a new encoding. In addition, SAS has added classification markers to all string

functions and call routines in the documentation to indicate their level of compatibility with I18N. A level of 2 means the function can be used with data of all encodings, just like a K function, whereas a level of 0 means the function is good only for SBCS (Single Byte Character Set) like Wlatin1 and Latin1. Level 0 functions may produce unexpected results or error messages when processing DBCS and MBCS data. While going through the SAS 9.4 NLS documentation, pay close attention to the following functions: ENCODCOMPAT, KCVT, KPROPDATA, BASECHAR, and %VALIDCHS. You will recognize that some of them can be used to perform certain tasks we have implemented but with important differences.

In any case, the I18N detection macro does a good job in identifying all characters and bytes that fall outside of the regular printable / visible ASCII range. It will shed light on why your SAS programs fail while processing ostensibly "normal" data sets that others have provided to you. We have demonstrated the possibility to change a data set's encoding to UTF-8 after its creation in another encoding. I am sure someone may be able to label a data set Wlatin1 after the fact too. It is also likely that the data comes from a database that has a different encoding than Wlatin1 or Latin1. Unfortunately, it is not always easy to obtain the original encoding information from your data provider – they may not have firsthand knowledge either. Sometimes they may not want to reveal to you how they have created the data. This macro can be your first step in addressing any potential data issues related to I18N. If you are good at SAS programming, you can modify the macro to suit your particular needs. For example, you may add a parameter to control whether to replace the exotic bytes or to identify them only. You can also invoke the macro in a macro DO loop or DATA step CALL EXECUTE routine to analyze an entire SAS library. Depending on your familiarity with the data and your business needs, you may decide to create an exemptions list to allow certain extended ASCII characters to remain and only replace others. The report currently has only one worksheet, but nothing can stop you from adding another sheet or two to flag other issues with the data, either at the variable level or data set level. Remember, the macro works for you!

One important caveat to note is that everything this paper talks about relates to encodings of the ASCII type usually found on Windows and Linux / Unix machines. The author has never worked with EBCDIC data and knows precious little about it. If you intend to adapt the macro for EBCDIC-based data on the mainframe platform, proceed with caution and at your own risk.

## CONCLUSION

SAS Internationalization (I18N) is an interesting topic. To most SAS users, it is probably a new topic too. I am not sure many of them have ever had to deal with issues arising out of those exotic characters and symbols along with the unfamiliar foreign language involved. It can feel intimidating, even dreadful. However, as the paper has demonstrated, it all comes down to the basics. As long as you have a solid understanding of ASCII and character encoding in general, extending that knowledge to Unicode or a language-specific encoding like Russian or Chinese is not such a big stretch. In the process you will gain deeper insight into how computers and software work, while effortlessly noticing how widely Unicode is already supported in other common applications. You will become a much better SAS programmer and more valuable resource to your employer and colleagues. It is almost like mastering a second language. Your perspective will change, not to mention the handy perks that come with your expanded capabilities.

While the provided I18N detection macro may serve your immediate needs well without change, you are encouraged to try to fully understand its logic and maybe modify it. Like any seasoned detective worthy of his gumshoeing badge, you should aim to expand your tools and skills beyond the macro. Going through the latest SAS NLS documentation helps a lot, so does reviewing the references listed at the end of this paper. Becoming an I18N detective in

one day is indeed possible with the macro as your sidekick, but I know deep down you have bigger aspirations. Go for it. The exotics will be rooting for you!

## REFERENCES


SAS Institute Inc. SAS® 9.4 National Language Support (NLS): Reference Guide, Fifth Edition.

Kiefer, Manfred. 2013. "Multilingual Computing in SAS® 9.4"
Technical Paper. Available at
https://support.sas.com/resources/papers/Multilingual_Computing_with_SAS_94.pdf

Bouedo, Mickaël and Beatrous, Stephen. 2013.
"Internationalization 101: Give Some International Flavor to Your SAS® Applications"
Proceedings of the SAS Global Forum 2013 Conference, Cary, NC: SAS Institute Inc.
Available at
http://support.sas.com/resources/papers/proceedings13/025-2013.pdf

Bales, Elizabeth and Zheng, Wei. 2017.
"SAS® and UTF-8: Ultimately the Finest. Your Data and Applications Will Thank You!"
Proceedings of the SAS Global Forum 2017 Conference, Cary, NC: SAS Institute Inc.
Available at
https://www.sas.com/content/dam/SAS/support/en/technical-papers/SAS0296-2017.pdf

Poppe, Frank. 2017. "Îf ÿøù have to process difficult characters: UTF-8 encoding and SAS®"
Proceedings of the SAS Global Forum 2017 Conference, Cary, NC: SAS Institute Inc.
Available at
https://support.sas.com/resources/papers/proceedings17/1163-2017.pdf

Liu, Leo (Jiangtao) and Bales, Elizabeth. 2018.
"Have a Comprehensive understanding of SAS® K functions"
Proceedings of the SAS Global Forum 2018 Conference, Cary, NC: SAS Institute Inc.
Available at
https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/1902-2018.pdf


## ACKNOWLEDGMENTS


I would like to express my sincere gratitude to Elizabeth Bales who provided detailed and valuable feedback, and Mickael Bouedo who reviewed the draft and suggested various improvements. Reading their published SAS papers on the subject has filled a lot of gaps in my understanding. Big thanks are also due to Jody Carlton for encouraging me to write this paper and arranging for expert reviews. This has been a truly unique learning experience for me.


## CONTACT INFORMATION

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


Houliang Li
Frederick, MD
Houliang_Li@yahoo.com