**PhUSE 2015**

**Paper DH03**

# Data Encoding: All Characters for All Countries

Donna Dutton, SAS Institute Inc., Cary, NC, USA

## ABSTRACT

The internal representation of data, formats, indexes, programs and catalogs collected for clinical studies is typically defined by the country(ies) in which the study is conducted. When data from multiple languages must be combined, the internal encoding must support all characters present, to prevent truncation and misinterpretation of characters. Migration to different operating systems with or without an encoding change make it impossible to use catalogs and existing data features such as indexes and constraints, unless the new operating system's internal representation is adopted. UTF-8 encoding on the Linux OS is used by SAS® Drug Development and SAS® Clinical Trial Data Transparency Solutions to support these requirements. UTF-8 encoding correctly represents all characters found in all languages by using multiple bytes to represent some characters. This paper explains transcoding data into UTF-8 without data loss, writing programs which work correctly with MBCS data, and handling format catalogs and programs.

## INTRODUCTION

The internal representation of data, formats, indexes, programs and catalogs collected for clinical studies is typically defined by the country or countries in which the study is conducted. The operating system on which the data was entered defines the internal data representation, such as Windows_64, HP_UX_64, Solaris_64, and Linux_X86_64.[1] Although SAS Software can read SAS data sets copied from one operating system to another, it impossible to use existing data features such as indexes and constraints, or SAS catalogs without converting them to the new operating system's data representation. In addition to operating system data representation differences, the character sets contained in the data and the encoding used to read and write them may differ. For example, Western European languages including English are frequently represented using the latin1 or latin9 character set, Eastern European languages may use latin2, Japanese data commonly uses Shift-JIS, and Chinese data may use Traditional Chinese (Big5) or Simplified Chinese (EUC-CN).

When data from multiple countries must be combined, whether within a single company or when sharing the data between companies, a character set must be chosen which includes all of the characters in any language used in the data. A character set is the group of characters to be represented, and a coded character set associates each character with a number. An encoding is used to read and write the data, and maps the bits and bytes of stored data to the correct characters via the coded character set. The character set which supports all languages and characters is called Unicode. UTF-8 is the most popular encoding used to read and write data in the Unicode character set, and is the encoding used in the SAS Drug Development and SAS Clinical Trial Data Transparency solutions.[2] These solutions run SAS software on a Linux server, but UTF-8 encoding is also available within Windows and other SAS implementations. UTF-8 encoding correctly represents all characters in the Unicode character set by using multiple bytes to represent some characters.

How can we tell what the encoding and operating system running our current SAS session is, if we don't already know? The PROC OPTIONS GROUP=LANGUAGECONTROL; statement will list all options related to language settings, including the session encoding. Or, we can be more specific and submit this code:

```
%put sysscp=&sysscp; /* prints the operating system (System Control Program) name */
%put encoding=%sysfunc(getoption(encoding));/* prints the current system encoding */
```

In order to correctly read data from another operating system and/or encoding in our current SAS session, we must determine the data representation defined by the operating system which produced it and the encoding with which it is stored. Beginning with version 6 for some SAS data sets (MBCS) and continuing with each later release, support was added to store the data set encoding as part of the data set header, and display it in PROC CONTENTS output. The following figures show PROC CONTENTS output from data sets created on different operating systems using

[1] EBCDIC data representations, used on IBM mainframe computers, are outside the scope of this paper.

[2] Other encoding forms, such as UTF-16 and UTF-32, as well as compression transformations, can also be used to represent the Unicode character set, but are outside the scope of this paper.

different encodings. All output shown in this paper was produced in SAS Drug Development 4.5.1 with the Linux operating system and SAS® 9.4 M1, or Windows SAS® 9.4 M2.



If we examine the log from the code used to produce this output, we will see some interesting notes.

```
NOTE: Data file WINLAT.TA.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.

WARNING: Integrity constraints cannot be processed for file WINLAT.TA because its
encoding does not match the session encoding or the file is in a format native to
another host, such as WINDOWS_64.

NOTE: Data file SJIS.CLASS.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.
```

## CROSS-ENVIRONMENT DATA ACCESS (CEDA)[3] LIMITATIONS

We see in these log notes that Cross-Environment Data Access (CEDA) was used automatically to read the data, making it easy for us to work with data from many operating systems and many encodings. However, the messages alert us to some of the limitations of CEDA, and in other cases we might see serious errors noting truncated data. When using CEDA, the transcoding of the data – the conversion of it from the original OS/encoding representation to the current system's OS/encoding representation - occurs in real-time each time the data is read. This results in an increase in the resources required to read the data, as noted in the log. In addition, some other limitations exist for the use of CEDA.

### INDEXES AND CONSTRAINTS

Data sets created on other operating systems or in different encodings can be read sequentially using CEDA. However, indexes created on another operating system or with a different encoding cannot be used by CEDA, even if the SAS index file (.sas7bndx) is present. Similarly, integrity constraints created on another operating system or with a different encoding cannot be used by CEDA, even if the SAS constraint file (.sas7bndx) is present. PROC CONTENTS with an OUT2= option can be used on the original system to capture the definitions of the indexes and constraints in SAS data sets. These data sets can then be copied to the target system and used to recreate the indexes and constraints there.[4]

[3] Cross-Environment Data Access (CEDA) was added in SAS V8.

[4] Examples of recreating indexes and constraints are beyond the scope of this paper.

**CATALOGS**

While CEDA can sequentially read data sets from another operating system and/or encoding, CEDA cannot read catalogs created on another operating system. All catalogs must be moved to a new system as follows:

- Write the catalog to a transport file using CPORT on the original operating system, copy it to the new operating system, then read it using CIMPORT on the new operating system, or
- Recreate the catalog in another manner on the new operating system. For example, FORMAT and compiled SAS Macro catalogs may be recreated by running the source code for the format or macro compilation on the new platform. Programs which create permanent or temporary formats having affected characters must also be transcoded in order to correctly create the formats when they are run in the new environment. This topic is covered in a later section of this paper.

**MISCELLANEOUS OTHER CEDA LIMITATIONS** [5]

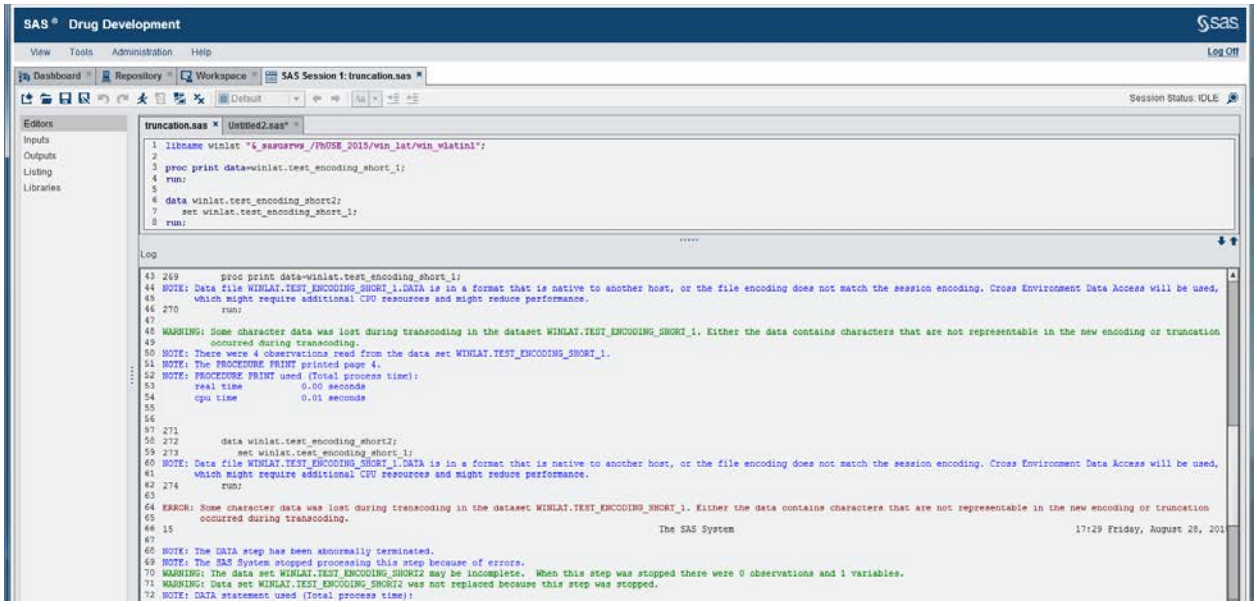
In addition to the limitations already covered, several other limitations exist, such as:

- Data sets cannot be updated in place using CEDA.
  - DATA step MODIFY statement
  - PROC SQL INSERT INTO and UPDATE clause
  - PROC APPEND, when the BASE= data set has the non-native data representation
  - PROC FSEDIT
- Data Step Views and Stored Programs cannot be processed.
- SAS data set Audit Trails cannot be read or updated.
- Data created by Version 6 SAS or earlier versions cannot be processed.

**INSUFFICIENT LENGTH FOR CHARACTER VARIABLES**

This is probably the most serious issue which must be addressed when moving data between different encodings. It can result in truncated and incorrect data, and in some cases, inability to read the data set at all. Let's attempt to use CEDA to read some data from Windows having the wlatin1 encoding on Linux having the UTF-8 encoding. The character variable WORD in the data set is 4 bytes in length, and contains these values:

- doña
- brød
- über
- será





[5] Discussion of alternatives to these CEDA limitations is beyond the scope of this paper.

We can see from the messages and the output above that SAS was not able to correctly handle the special characters in our data. When asked to read and print the data, SAS did the best it could, with the result of showing us truncated data. However, when asked to create a new data set in a DATA step when reading truncated data, SAS triggers a fatal error and stops the step. This prevents the truncated data from being incorporated into a new data set. What happened here, and what can we do about it?

## CHARACTER ENCODING

As we saw in the output from PROC CONTENTS, an attribute called Encoding is displayed. One of our examples shows a Windows data set with the wlatin1 encoding. An encoding tells the computer how to read the binary numbers stored as bytes in a character string and map them with the character set used by that encoding. For example, the character set latin-1(ISO/EIC 8859-1) includes 191 characters from the Latin script and is widely used throughout Western Europe and the Americas. In its 256 code points, it includes the upper and lower case English characters, the digits 0 through 9, and characters such as the Danish characters æ, ø, and å, the German character ü, the Spanish characters ñ and á, and some special characters such as ©. All of these characters and symbols are represented by a specific numeric value in a single stored byte. The encodings wlatin1 for Windows and latin1 for Unix translate these numbers into characters, 1 character per byte. A Single Byte Character Set (SBCS) represents all characters in the set in a single byte. The first 128 code points of wlatin1, latin1 and most other encodings are retained from the original ASCII standard of the 1960's. Characters represented by code points 129-256 are sometimes referred to as Extended ASCII. The various latinX encodings (the ISO 8859 family) vary in the characters represented in the extended ASCII codes. For example, latin1 does not include the symbol for the Euro currency, while the newer latin9 does.

However, the characters in languages such as Japanese, Chinese and Korean (JCK) cannot be represented in a single byte and 256 code points; multi-byte character sets (MBCS) are necessary to correctly represent all characters in these languages. MBCS are sometimes also called DBCS (double byte character sets) because the first MBCS's defined used a maximum of two bytes. For example, Shift JIS (SJIS) for Japanese represents all characters defined by Japanese Industrial Standards and the basic ASCII characters in one or two bytes, but does not represent all characters in some other languages.

The Unicode character set was created to attempt to support all languages and represent all possible characters, and UTF-8 (Universal character set Transformation Format-8-bit) is the most common encoding used with it. UTF-8 is a variable-length encoding that uses from 1 to 4 8-bit code units per character and is capable of representing all possible characters in the Unicode character set. The first 128 codes of the UTF-8 character set are the same as the ASCII character set and almost all other non-EBCDIC encodings, and represent the following characters in single bytes:

- all of the letters of the English alphabet (uppercase and lowercase)
- the digits 0-9
- various punctuation characters and control codes

Additional characters, such as the extended ASCII European characters mentioned above, are represented in UTF-8 in 2 bytes. Japanese and other characters may require up to 4 bytes to be represented.

And this is why we see truncation in some cases when we attempt to read data created in one encoding using CEDA in another encoding. A character variable in the new encoding may require more space than has been allocated to it to correctly represent the value it contained in the original data set.

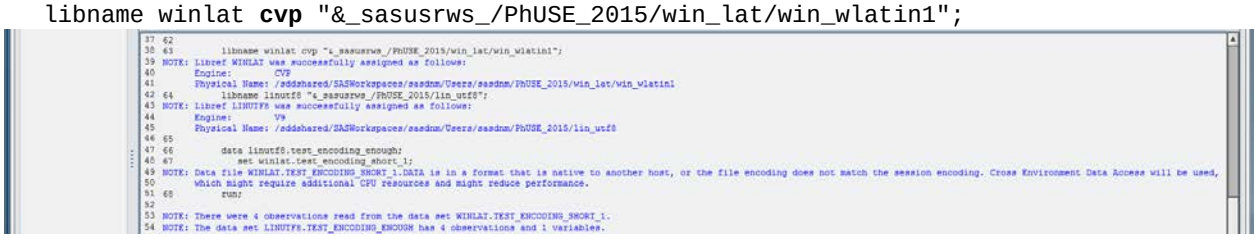
So, what do we do?

## CREATING THE TRANSCODED DATA

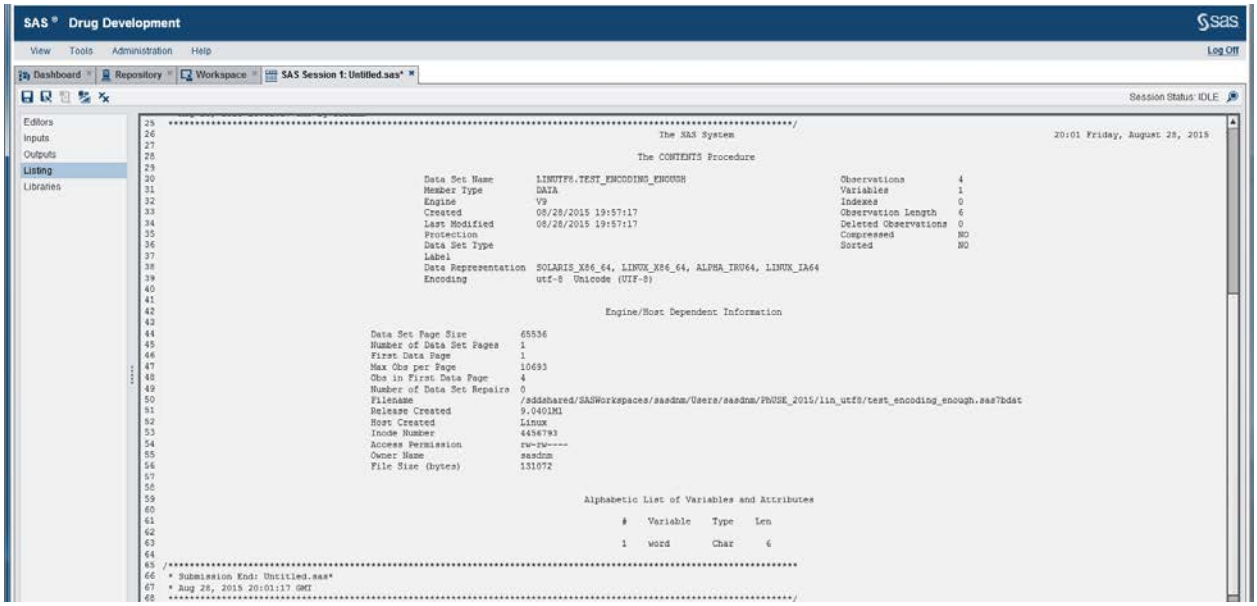
What we need is a way to tell SAS to give these character variables some extra space when reading a data set from a different encoding, so that truncation does not occur. Enter another acronym: CVP – Character Variable Padding. The CVP read-only libname engine was created for just this situation, and is documented in the SAS® 9.4 National Language Support (NLS): Reference Guide. When the CVP engine is added to a LIBNAME statement, a default or specified amount of space is added to each character variable read from a data set, allowing room for the expansion of characters in the new encoding.

Returning to the data we tried to read earlier, if we replace our basic LIBNAME statement with this one:

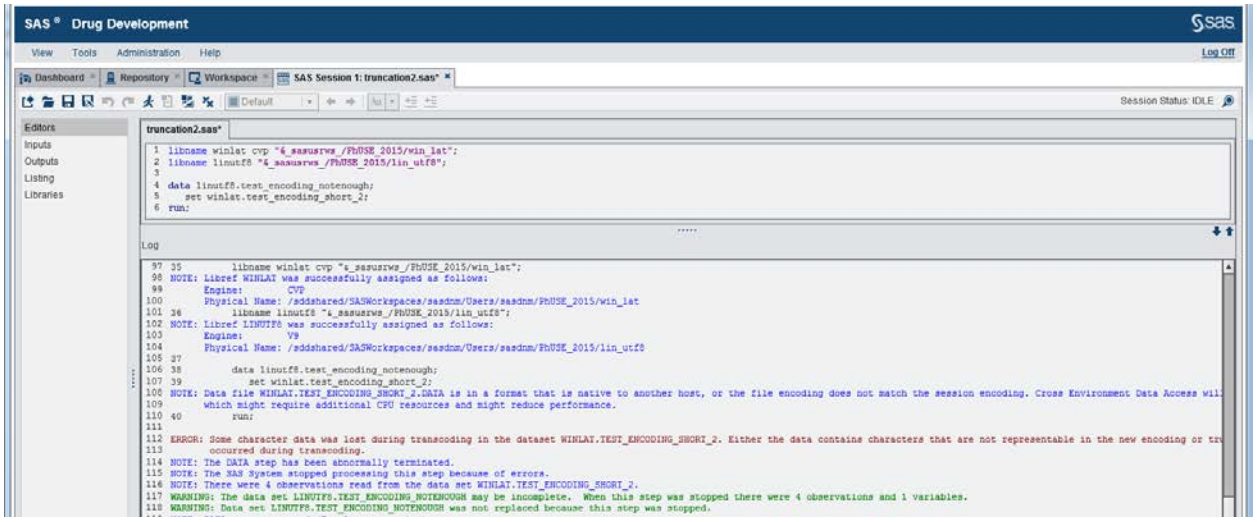
```
libname winlat cvp "&_sasusrws_/PhUSE_2015/win_lat/win_wlatin1";
```



We see that although CEDA is still used to read the data, we no longer see any warnings or errors about truncation. Examining the PROC CONTENTS output of the new data set, we see:




Our new data set has the data representation of our Linux operating system, and the UTF-8 encoding of our current SAS Session. In addition, the length of the character variable WORD has been increased from 4 to 6 bytes, allowing enough space to store all of the values without truncation. The default amount used by the CVP engine to increase the size of character variables is 1.5 times the current length. But what would happen if this is not enough? Let's say we have another value in our data set of ¡áí!. Three of the four characters require 2 bytes in UTF-8 encoding.

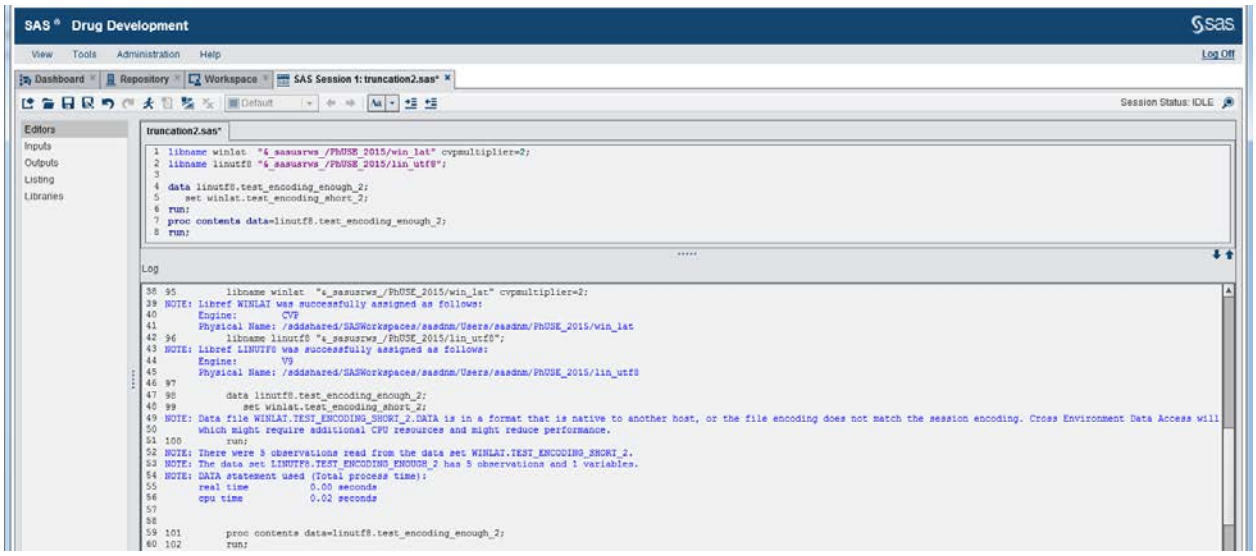


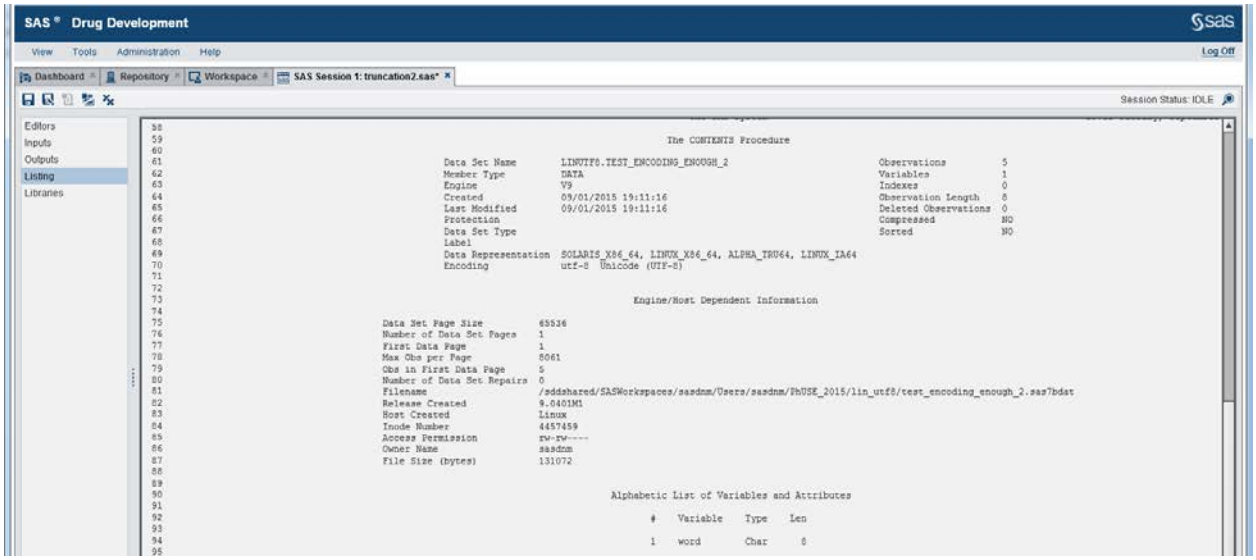

Once again, we experience truncation. There are two options which can be used to give control over the amount of space used to pad the character variables:

`CVPBYTES=` a fixed number of bytes between 0 and 32,766 is added to each character variable

`CVPMULTIPLIER=` each character variable's length is multiplied by a value between 1 and 5. Default=1.5

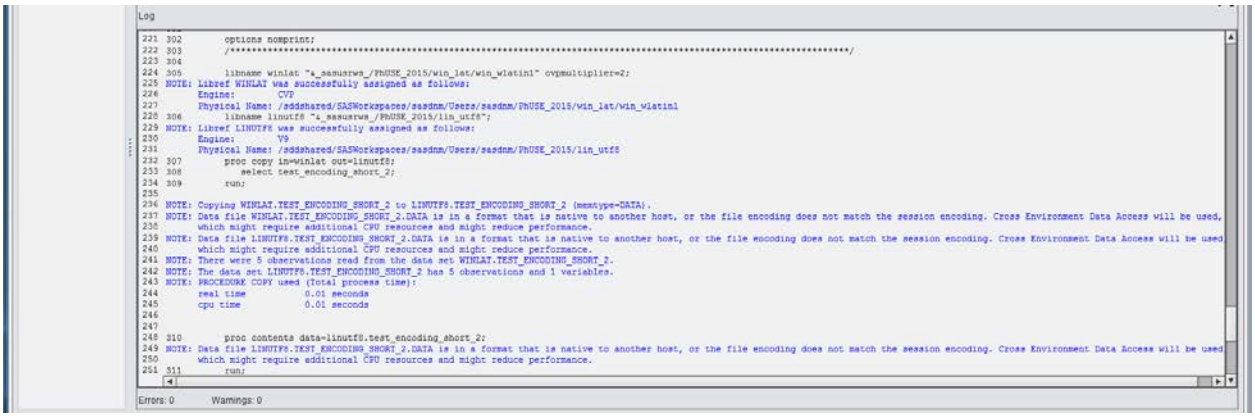
These two options imply the use of the CVP engine, and cannot be used together.

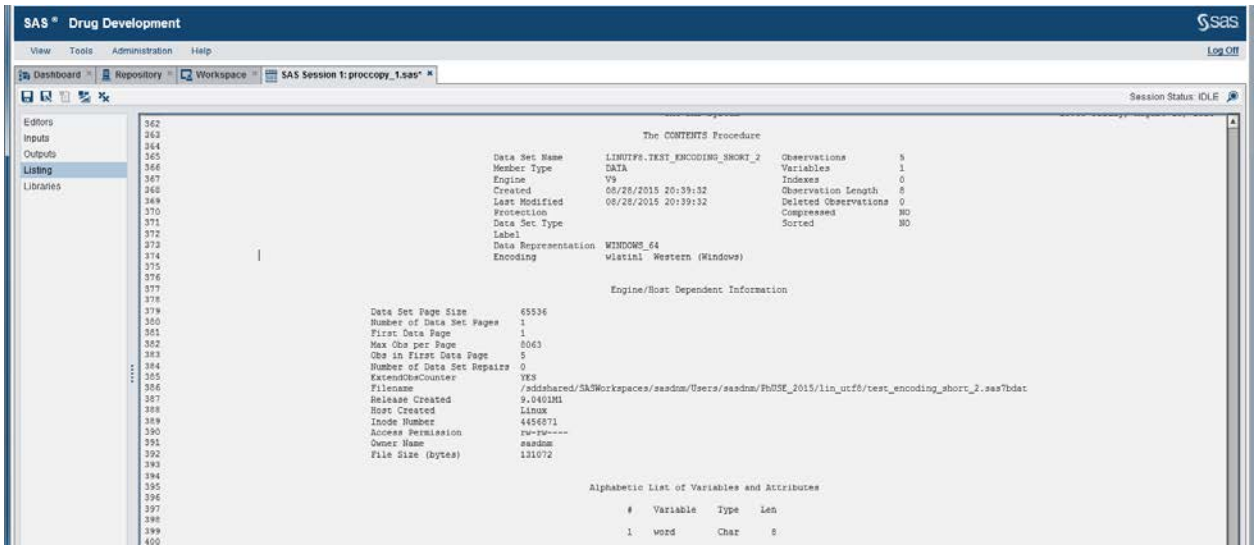



As you can see, it may take some experimenting to find the correct amount of space needed for various situations, and reviewing the log for truncation messages is critical when transcoding data.

In the previous example, we used a DATA step to create our transcoded data set. But what if we have hundreds of data sets in our library that need to be transcoded? PROC COPY, of course, right?



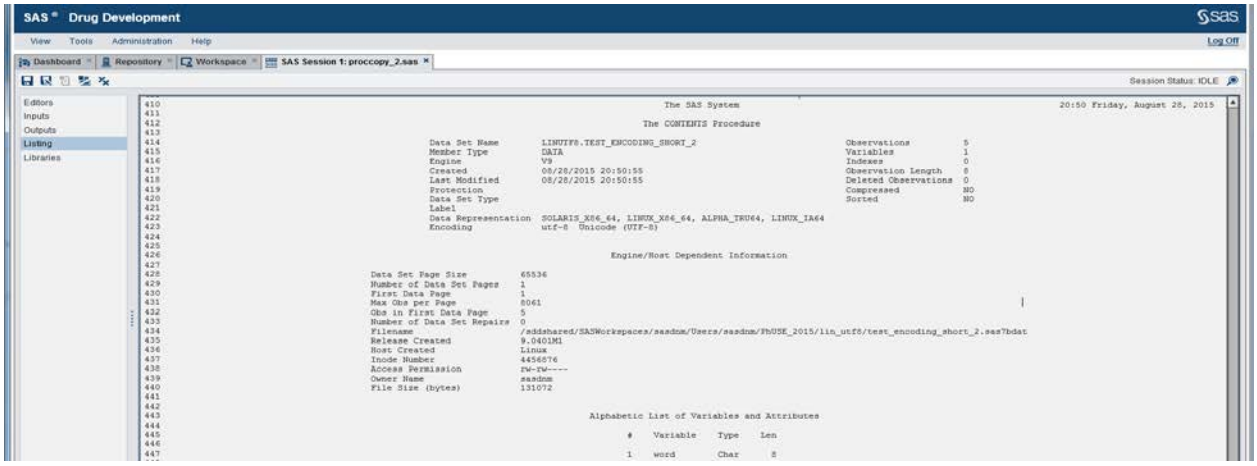



Hmm. Why is the new data set still in the Windows data representation with an encoding of wlatin1, even though the size of the variable WORD has been increased?

By default, PROC COPY creates an exact copy of the input data set, preserving all data set attributes, including the data representation and encoding. That is, it creates a clone of the input data set. The option NOCLONE tells PROC COPY to adopt the current operating system data representation, the current SAS Session encoding, and other relevant attributes, unless options are specified to override those attributes. The statement

```
proc copy noclone in=winlat out=linutf8;
```

yields the desired results. See Base SAS® 9.4 Procedures Guide, Fourth Edition for details about the data set attributes affected by this option.




Now that we have our data sets transcoded, we can get on with our normal programming activities, right? Not so fast…

## WORKING WITH MULTI-BYTE CHARACTERS

As SAS programmers, we are all very accustomed to using the many character handling functions available in the SAS language. However, there are some facts about many of our frequently used functions that we may have never noticed. Let's take a look at the documentation for the SUBSTR function:

### SUBSTR (RIGHT OF =) FUNCTION

**Extracts a substring from an argument.**

| | |
|---|---|
| **Category:** | Character |
| **Restriction:** | This function is assigned an I18N Level 0 status, and is designed for SBCS data. Do not use this function to process DBCS or MBCS data. For more information, see Internationalization Compatibility. |
| **Tip:** | DBCS equivalent functions are KSUBSTR and KSUBSTRB. |

*Syntax*
<*variable*=>**SUBSTR**(*string*, *position* <, *length*>)

It's possible that you have never even noticed the restrictions associated with many functions related to multi-byte character data, but now that you are working in an encoding in this category, it is very important. The SUBSTR function's third argument, LENGTH, specifies the number of bytes of data to return, not the number of characters to return. For SBCS data, these are the same, but not for MBCS data.

The links above for Internationalization Compatibility and the functions mentioned in the tip section take us to the SAS 9.4 National Language Support (NLS): Reference Guide, Fourth Edition, which is an indispensible resource when working with UTF-8 encoded data. I18N is the abbreviation for Internationalization. There are three levels of classification for functions relative to I18N:

I18N Level 0 – The function is designed for SBCS data only.

I18N Level 1 – The function should be avoided, if possible, when working with MBCS data. These functions might not work correctly with MCBS data under certain circumstances

I18N Level 2 – This function can be used for SBCS and MBCS data.

The documentation for each function, as seen above, contains the I18N compatibility level of the function. Also, the section Internationalization Compatibility for SAS String Functions in the SAS 9.4 National Language Support (NLS): Reference Guide contains a list of all SAS String Functions and their I18N level, including the special 'K' functions which were created specifically to handle MBCS character data correctly. They are known as the 'K' functions because the letter K was added as a prefix to the names of the standard SAS functions.[6] For example:

### KSUBSTR Function

**Extracts a substring from an argument.**

| | |
|---|---|
| **Category:** | DBCS |
| **Restriction:** | This function is assigned an I18N Level 2 status and designed for use with SBCS, DBCS, and MBCS (UTF-8). For more information, see Internationalization Compatibility. |

*Syntax*
**KSUBSTR**(*argument, position*<,*n*> )

The KSUBSTR function's second and third arguments refer to **characters**, not bytes. The starting position of the substring operation begins with the character specified by position and is the number of characters specified by n, which for UTF-8 encoded data, could be anywhere between one and four bytes. As an example, data from the SASHELP.CLASS data set from a Shift-JIS encoded system was copied to our UTF-8 system and transcoded. The screen shot below shows that for these values, three bytes were used to represent the first character of the Japanese names.


[6] The 'K' originally referred to Kanji characters, a Japanese form of writing using Chinese-derived characters. These characters are stored in UTF-8 encoding using multiple bytes. The 'K' persisted to indicate functions designed to work with any MBCS data.

Existing 'K' function counterparts of standard SAS character functions include:

| | | | |
|---|---|---|---|
| KCOMPARE | KCOMPRESS | KCOUNT | KINDEX |
| KINDEXB | KINDEXCB | KINDEXC | KLEFT |
| KLENGTH | KLOWCASE | KREVERSE | KRIGHT |
| KSCAN | KSTRCAT | KSTRIP | KSUBSTR |
| KSUBSTRB | KTRANSLATE | KTRIM | KTRUNCATE |
| KUPCASE | KUPDATE | KUPDATEB | KUPDATES |
| KVERIFYB | KVERIFY | | |

In addition, these specialized 'K' functions provide the ability to perform various types of data conversion:

| | |
|---|---|
| KCVT | Converts data from one type of encoding data to another encoding data |
| KPROPCASE | Converts Chinese, Japanese, Korean, Taiwanese (CJKT) characters |
| KPROPCHAR | Converts special characters to normal characters |
| KPROPDATA | Removes or converts unprintable characters |

Also, a number of SAS-supplied autocall macros also have 'K' counterparts:

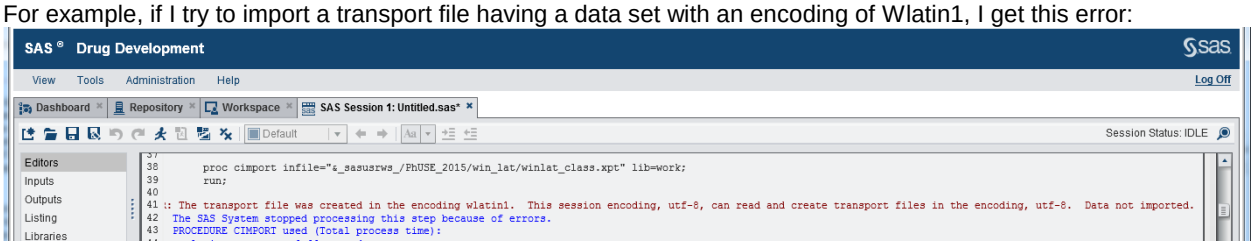
| | | | | |
|---|---|---|---|---|
| %KLOWCASE | %QKLOWCASE | %KTRIM | %QKTRIM | %KVERIFY |

## FORMAT CATALOGS

SAS catalogs must be created using the OS data representation of the current session in order to be read. If there is no difference in the encoding of the source and target systems, then PROC CPORT/PROC CIMPORT can be used to create and then import a SAS transport file containing formats. However, for SAS releases prior to SAS® 9.4 M3 (released JUL2015), PROC CIMPORT can only address the difference in OS data representations. It cannot transcode between encodings for any type of SAS file or catalog.

**WARNING:** PROC CIMPORT can detect encoding differences between SAS data sets in a transport file and the current session, and will trigger an error and stop the import process if the data set encoding does not match the current session encoding. However, catalogs do not store an indication of the encoding with which they were created, so PROC CIMPORT is unable to determine a mismatch in encodings. The catalog is imported, but any formats containing characters having a mismatch between the encoding with which the format was created and the current session encoding will not be correct. No warning or error is generated!

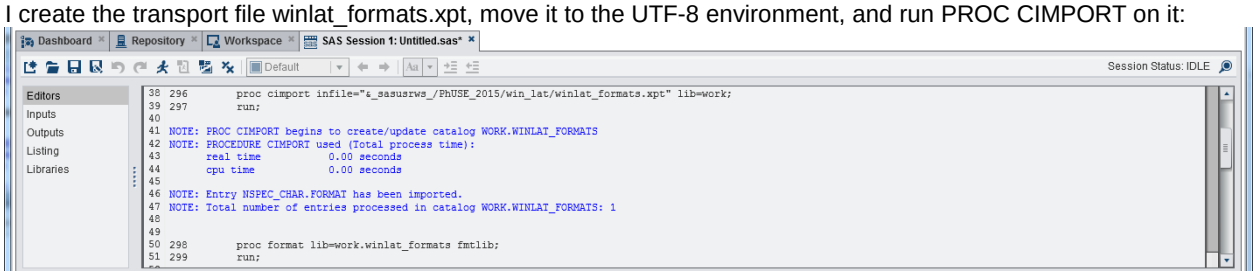
For example, if I try to import a transport file having a data set with an encoding of Wlatin1, I get this error:



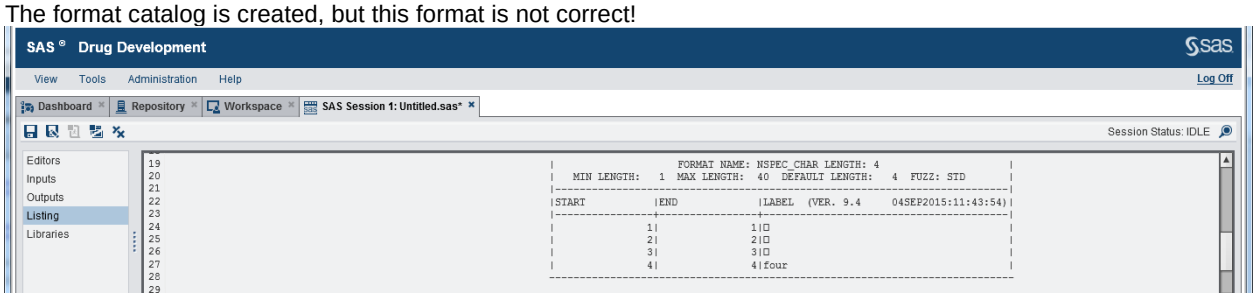

Let's say that on our Windows wlatin1 environment, we have the catalog SASUSER.WINLAT_FORMATS. In the catalog, I have a format which was created by this SAS code:

```
proc format lib=sasuser.winlat_formats;
value nspec_char
 1='ú'
 2='ñ'
 3='ø'
 4='four';
run;
```

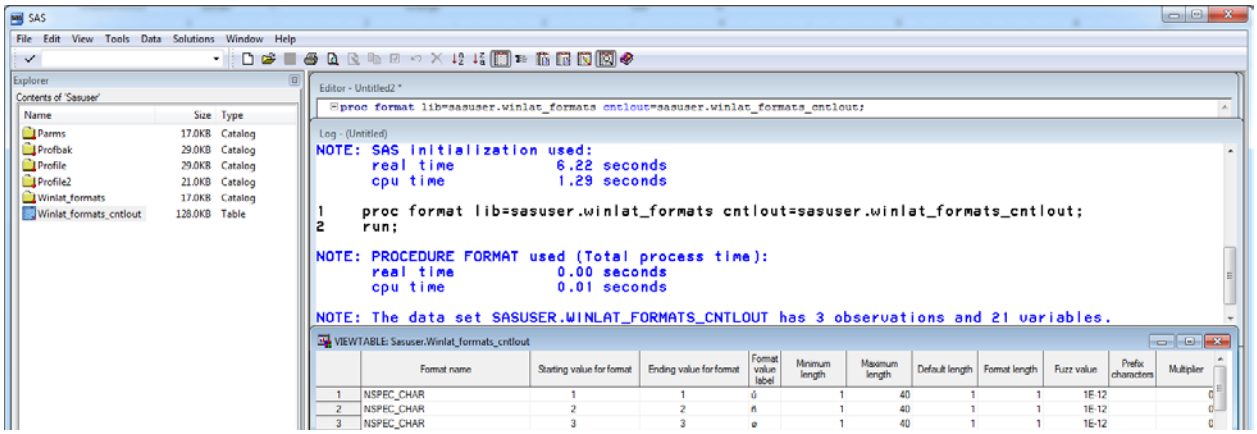
I create the transport file winlat_formats.xpt, move it to the UTF-8 environment, and run PROC CIMPORT on it:



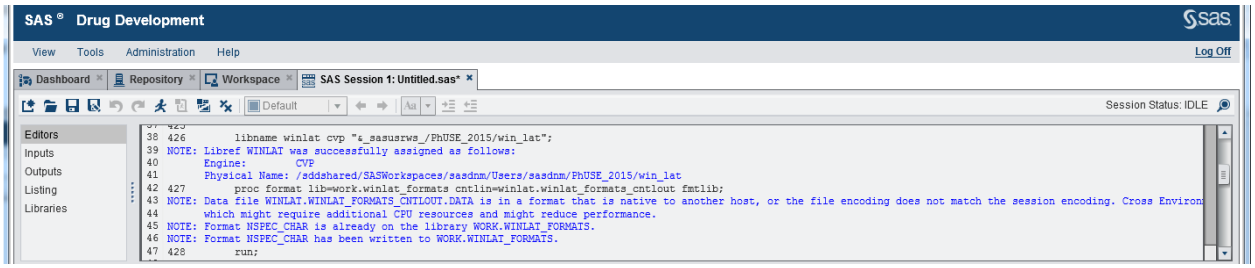
The format catalog is created, but this format is not correct!



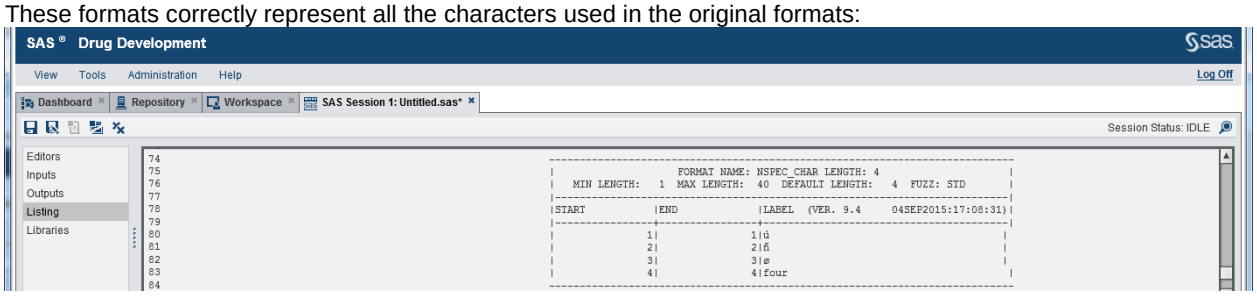

To successfully create the format in the new environment, I must either rerun the program used to create it, or use the PROC FORMAT CNTLOUT= and CNTLIN= options. If I have a lot of formats, using the PROC FORMAT options would definitely be the fastest solution, since the Output Control data set will contain the information about all of the formats in the catalog. In my SAS Windows environment, I create the Output Control data set:




My CNTLOUT data set has a wlatin1 encoding, and so will need to be transcoded once I transfer it to my Linux UTF-8 environment. We already know how to do that using the CVP libname option. I use that option on the LIBNAME statement pointing to the copied data set, which will be used as the CNTLIN Input Control data set to create a Linux UTF-8 format catalog containing all of the formats described in that data set.

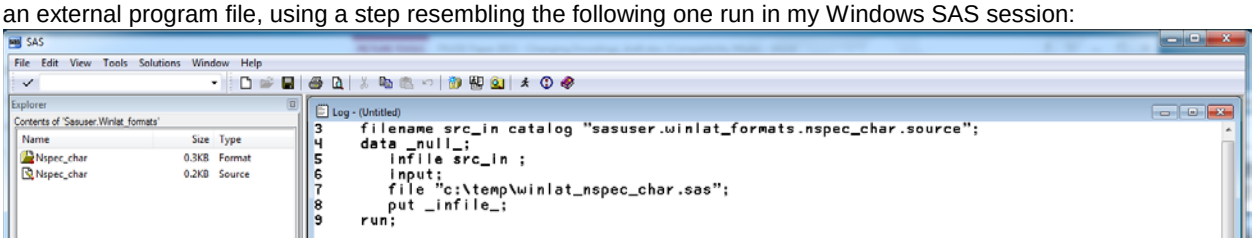



These formats correctly represent all the characters used in the original formats:



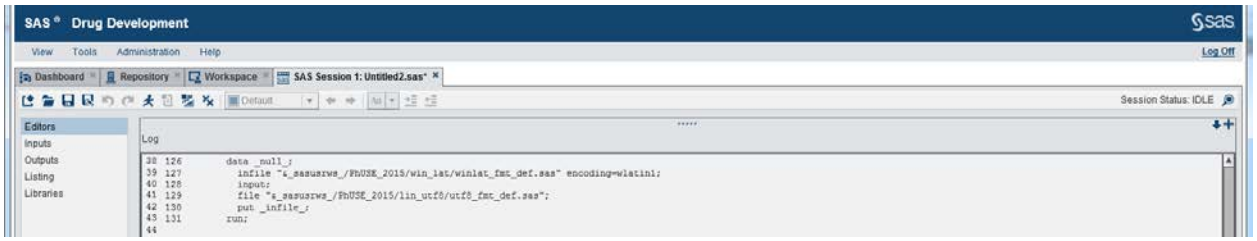

If I have catalog .source entries or .sas files containing my format programs, how would I handle those?

Since PROC CPORT/CIMPORT cannot transcode data, the catalog .source entries would first need to be written to an external program file, using a step resembling the following one run in my Windows SAS session:

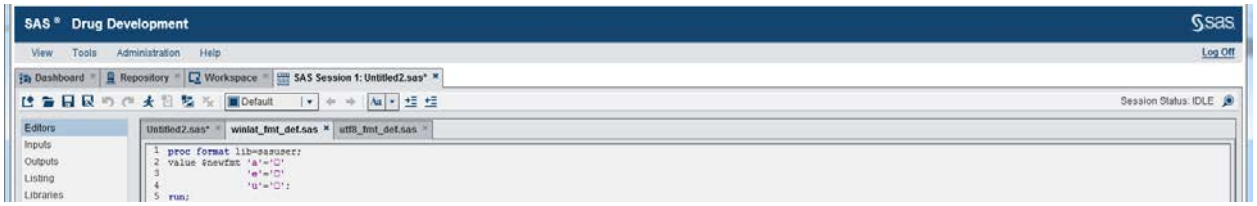


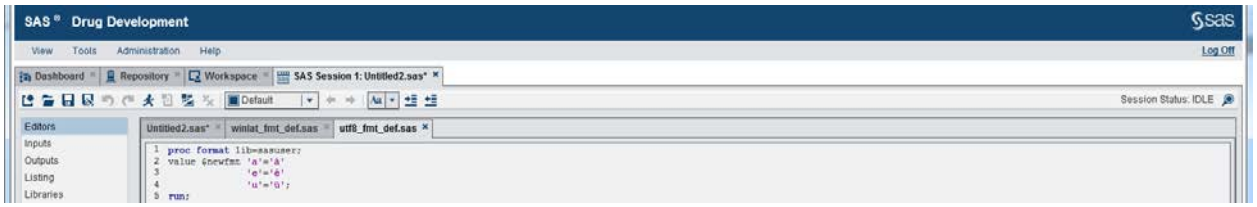

If there are many source entries, this could be encapsulated in a macro which would write each .source entry to a file. Then, we could copy our standard program files along with the converted .source entries to the target system in preparation for executing them to create the new formats. But we are not ready to run these programs yet.

External files are also created with a specific encoding, but do not typically display or contain its value. If the encoding of the source system is different from that of the current SAS session, we must rely on our knowledge of their source and explicitly state their encoding in the SAS statements which read them. For example, this code creates a correct program to run in a UTF-8 SAS session and successfully create a format containing any wlatin1 characters.




Examining the two programs within the Linux UTF-8 environment clearly shows that the original program does not correctly represent the characters in this encoding, while the new program does.




## CONCLUSION

The combination of data from a variety of different sources is very desirable for many types of analysis, and SAS software provides a variety of tools to facilitate it. When the data to be combined include character data in many languages, special handling is necessary to successfully represent all characters within the SAS environment being used to analyze the combined data. The first consideration is the selection of an encoding that is capable of handling all of the necessary characters. If all characters from all countries are to be supported, UTF-8 is an excellent choice. We must then ensure that no data is truncated when reading the data, which may require expanding the size of character variables using the CVP (Character Variable Padding) engine. In many cases, it makes sense to create a new version of the data stored in the data representation and encoding of the current session. As we continue to work with the data in its new encoding, we must keep in mind that in MBCS encodings, one character may take more than one byte of storage, and remember to use the I18N Level 2 functions, and especially the 'K'-functions, when writing programs that manipulate character data. Format programs and catalog entries require special handling to ensure that they are correctly transcoded and new valid formats are created.

## REFERENCES

SAS® 9.4 National Language Support (NLS): Reference Guide
Base SAS® 9.4 Procedures Guide, Fourth Edition
SAS® 9.4 Functions and CALL Routines: Reference, Fourth Edition

## ACKNOWLEDGMENTS

Manfred Kiefer, Principal Domain Specialist in the SAS Institute Inc. Software Globalization Division
Elizabeth Bales, Senior Manager in the SAS Institute Inc. Software Globalization Division

## RECOMMENDED READING

SAS Institute Inc. 2013. SAS Institute technical paper. “Multilingual Computing with SAS® 9.4” Available at http://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.

Kiefer, Manfred. 2012. SAS® Encoding: Understanding the Details. Cary, NC: SAS Institute Inc.

Haralambous, Yannis. 2007. Fonts & Encodings: From Unicode to Advanced Typography and Everything in Between. Beijing: O'Reilly.

Gillam, Richard. 2003. Unicode Demystified: A Practical Programmer's Guide to the Encoding Standard. Boston: Addison-Wesley.

## CONTACT INFORMATION

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

Donna Dutton
Principal Technical Consultant
Health & Life Sciences, SAS Solutions OnDemand
SAS Institute Inc.
Work Phone: 919-531-1159
Email: donna.dutton@sas.com