Paper SAS1902-2018

Have a Comprehensive understanding of SAS® K functions

Leo (Jiangtao) Liu, SAS Institute Inc.

Elizabeth Bales, SAS Institute Inc.

ABSTRACT

SAS® offers a variety of string functions that help you get the most out of your character data. However, the traditional string functions, such as SUBSTR and INDEX, assume that the length of a string in a SAS character column is always one byte. The K functions offer the power of SAS string function handling for all of your character data, no matter how long a character might be. This paper demonstrates the intrinsic characteristics of K functions by comparing them with the traditional SAS string functions. Your understanding of K functions will be further improved by looking at K functions from different perspectives. Based on a full exploration of the SAS K functions, you will be able to write SAS programs that can be run in any SAS environment using data from any language—"Write it once, run it anywhere!".

INTRODUCTION

This paper introduces K Functions, and how they work. Special focus is placed on K function behavior for SAS with a Double-Byte Character Set (DBCS) or Multi-Byte Character Set (MBCS) session encoding. Understanding K functions step by step and in depth will assist you in writing efficient SAS programs that require internationalization (I18N) to support multi-byte characters.

NON-K FUNCTIONS VERSUS K FUNCTIONS

A SAS string function is a component of the SAS programming language that can accept arguments, manipulate a string, and return a value that can be used in an assignment statement or elsewhere in expressions. The initial versions of SAS® string functions were designed to work just for the North American market. These traditional string functions assume the size of a character is always one byte, and that a specified position is always the byte position. This is true with a SAS Single-Byte Character Set (SBCS) session encoding.

As SAS began to support characters in more languages, new DBCS and MBCS encodings were added to SAS that support multi-byte data. DBCS and MBCS encodings require one byte, two bytes or even more bytes to represent a character. In the mid-1980s, a set of string manipulation functions for Kanji characters handling were released. The K functions were born!

After years of evolution, 'K' implies not only ideographic character support used in Japanese, Korean and Chinese, but also MBCS, such as UTF-8 with the SAS Unicode Server. The K functions also work very well for SBCS character data.

This paper uses the term "Non-K functions" to discuss the traditional SAS string functions that only support single-byte data. Non-K and K functions have very different usage scenarios because they have inherently different design purposes.

ACCURATE CHARACTER LENGTH

When working on SBCS characters, Non-K and K functions have the same behavior. The example below shows results from the functions LENGTH and KLENGTH when the input string contains single-byte characters. Both functions return the same value, a length of 3, for the SBCS string that is specified.

```
data _null_;
  len1 = length('SAS');
  len2 = klength('SAS');
  put len1= len2=;
run;
```

```
Result:
len1=3 len2=3
```

Example 1: LENGTH and KLENGTH Process SBCS Data

The behavior changes when the functions work on DBCS characters. The next example is run in SAS with a Shift-JIS session encoding, which supports Japanese characters. The string “中国語” contains three Japanese characters. Each character requires two bytes. KLENGTH returns a length of 3, indicating that there are three characters in the string. However, LENGTH returns a length of 6. The non-K function, LENGTH, simply calculates the number of bytes in the string, not the number of characters.

```
data _null_;
  len1 = length('中国語'); /* '中国語' is 3-DBCS-char string */
  len2 = klength('中国語');
  put len1= len2=;
run;

Result:
len1=6 len2=3
```

Example 2: LENGTH and KLENGTH Process DBCS data

Example 3: LENGTH and KLENGTH Process UTF-8 Data that shows the results from the same program when it is run in SAS with a UTF-8 session encoding. In UTF-8 most ideographic characters need three bytes. Therefore, when the Japanese string from the previous example is specified in the LENGTH function the byte length that is returned is 9. Obviously, LENGTH is improper for this language data in this environment if your SAS program needs to be aware of characters.

```
data _null_;
  len1 = length('中国語');
  len2 = klength('中国語');
  put len1= len2=;
run;

Result:
len1=9 len2=3
```

Example 3: LENGTH and KLENGTH Process UTF-8 Data

The examples in this section show us that the K function, KLENGTH, always returns the character length for character data regardless of the SAS session encoding. The Non-K function, LENGTH, is only designed to return the correct character length for SBCS data.

Note that the LENGTH function can be used with any SAS session encoding if you need to know the number of bytes that are in a string.

ACCURATE STRING BREAK

Substring, or copying a segment of a string, is a typical string manipulation operation. The following program attempts to get a substring with a length of 2 starting from the second character of the base string. The results shown below are from SAS running with UTF-8 session encoding. The two functions, KSUBSTR and SUBSTR, both return a portion of the base string. For both functions, the portion begins at the location that you specify as the second argument, which is the *position*. The result that is returned is determined by the value that you specify in the third argument, which is the *length*.

```
data _null_;
  str='中国語';
  put str= / str=hex.;
  str1=ksubstr(str, 2, 2);
  str2=substr(str, 2, 2);
  put str1= / str1= hex.;
  put str2= / str2= hex.;
run;

Result:
str = 中国語
str = E4B8ADE59BBDE8AA9E
str1= 国語
str1= E59BBDE8AA9E202020
str2= ▌▌    /* Garbage display here */
str2= B8AD20202020202020
```

Example 4: SUBSTR and KSUBSTR Process UTF-8 Data

For the non-K function, SUBSTR, the *position* represents the byte position where the segment will begin and the *length* expression specifies the number of bytes to return. The substring operation for SUBSTR is actually equivalent to cutting a piece from a byte sequence. In this case, the result that is returned begins at the second byte of the base string, which is the second byte of the first character in the base string. The result is two bytes long – 0xB8AD – which does not represent a valid UTF-8 character, and certainly does not contain two characters. The MBCS character has been corrupted!

For KSUBSTR, the *position* argument indicates the first character to include in the string segment. The *length* expression indicates the number of characters to copy to the segment. KSUBSTR returns the right result in this case. K function is the recommended solution here.

Table 1 below shows the characters of the original string followed by the bytes stored for the string when running SAS with a UTF-8 encoding. The Table also shows the character and byte offset values, followed by the results returned by KSUBSTR and SUBSTR in our example.

DBCS String	中			国			語		
UTF-8 (hex.)	E4	B8	AD	E5	9B	BD	E8	AA	9E
Char Offset	1			2			3		
Byte Offset	1	2	3	4	5	6	7	8	9
KSUBSTR Output				E5	9B	BD	E8	AA	9E
SUBSTR Output		**B8**	**AD**						

Table 1: Byte Offset and Character Offset

ACCURATE CHARACTER LOCATION

In the previous comparisons of K functions and non-K functions, you saw how choosing the wrong function can corrupt the result that is returned by the SAS function. This section demonstrates how a serious problems can occur when a byte value is interpreted in more than one way.

Table 2: Conflict Between SBCS Character and Second Byte of DBCS Character shows a Japanese string, 'サービス[10]', that contains a mix of single-byte and double-byte characters. The first row in the table shows the characters of the Japanese string. The second row in the table shows the byte sequence that is stored for that string when SAS runs with a Shift-JIS session encoding. Row 3 shows the character position for each character, and row 4 shows the position of each byte. Note that the second byte of the second Japanese character, 'ー', has the same code point as the left square bracket character, '[', which requires one byte in the Shift-JIS encoding.

Japanese String	サ		ー		ビ		ス		[	1	0	]
Shift-JIS (hex.)	83	54	81	5b	83	72	83	58	5b	31	30	5d
Char Offset	1		2		3		4		5	6	7	8
Byte Offset	1	2	3	4	5	6	7	8	9	10	11	12

Table 2: Conflict Between SBCS Character and Second Byte of DBCS Character

The left square bracket character, '[', appears as the fifth character in the string. The SAS code below attempts to locate the position of that character using the INDEXC and KINDEXC functions:

```
data _null_;
  pos1 =  indexc('サービス[10]', '[]');
  pos2 = kindexc('サービス[10]', '[]');
  put pos1= pos2=;
run;
Result:
pos1=4 pos2=5
```

Example 5: INDEXC and KINDEXC Process DBCS Data

When KINDEXC is called to locate the square bracket it returns the correct position. However, INDEXC always assumes that each byte is a valid character and examines each byte of the string. Therefore, INDEXC returns 4 since the first occurrence of 0x5b is at byte position 4. Any SAS function that depends on this result could potentially split the DBCS character, 'ー'.

Some non-K functions can be very dangerous in SAS with a multi-byte session encoding. K functions can help your program get correct string searching results for any data in all environments.

UNDERSTANDING K FUNCTIONS INSIDE

In order to get a comprehensive understanding of K functions, let's take a look inside and observe these functions from different perspectives. This section examines several factors in detail that impact K functions.

THE IMPACT OF ENCODING

A character encoding is a collection of numeric values that can be stored in memory to represent characters. K functions must ensure that the functionality works well for any type of encoding.

SAS supports various languages and session encodings. The K function string manipulation behavior is often different depending on the SAS session encoding. Let's look at the results from one K function as an example.

KTRIM is a function that removes trailing blanks from a specified string. The examples below demonstrate how KTRIM processes blank characters for various SAS session encodings.

- Latin1
  Latin1 is the SAS session encoding used by SAS session on UNIX hosts for English. The single-byte space (0x20) is the only blank character that KTRIM removes. For SAS with a Latin1 session encoding, KTRIM and TRIM work the same way.
- Shift-JIS
  Shift-JIS is the session encoding used by SAS to represent Japanese on Windows and UNIX hosts. KTRIM needs to consider the Japanese ideographic space character, or the DBCS space, (0x8140) in addition to the single-byte space 0x20.

- GB2312
  On UNIX and Windows hosts, Simplified Chinese is supported by SAS using the GB2312 encoding. KTRIM needs to consider both the DBCS space and the single-byte space in Simplified Chinese. The single-byte space is still 0x20 in GB2312. However, DBCS space has a different character code (0xa1a1) in GB2312.

- Big5
  Big5 is the session encoding for Traditional Chinese SAS on UNIX and Windows hosts. KTRIM will remove the 0x20 as well as the Traditional Chinese DBCS space (0xa140) in Big5.

- EBCDIC
  On z/OS machines, the single-byte space is 0x40 in all EBCDIC SAS session encodings. KTRIM needs to remove the DBCS space (0x4040) inside the shift-out and shift-in (SO/SI) marks in EBCDIC DBCS encodings as well as the single-byte space.

- UTF-8
  UTF-8 is the session encoding of SAS Unicode Server that supports multi-lingual. The set of white-space characters is much larger than for the other encodings. UTF-8 not only includes the single-byte space (0x20) and the DBCS space characters, but also has a No-break space, Ogham space mark, Line feed, and Form feed, to name a few. You can find a complete list of Unicode space characters documented in the *SAS® 9.4 National Language Support (NLS): Reference Guide.* See the table titled "Unicode Spaces That Are Removed by KLEFT, KRIGHT, and KTRIM".

This is a prime example of how the K functions work with various encodings. K functions are the best solution for writing SAS programs that can process character data from any supported SAS session encoding. The diversity of encodings determines the complexity of the K functions.

CHARACTER BOUNDARY

A character is a minimal unit of text that has semantic value. The same character might have a different code value and length across different encodings. A significant feature of K functions is that they are character-based. Therefore, they need to recognize characters in different encodings.

The character boundary is like a box that surrounds each character. When a character boundary is identified, a character is identified. Identifying the character boundary allows K functions to manipulate a string character by character.

Let’s try to understand this concept by taking a look at the way K functions detect the character boundary in the following typical character encodings.

- SBCS encodings
  One character always occupies one byte in an SBCS encodings. Because character and byte boundaries are the same for SBCS encodings, SAS programs can manipulate the string freely without worrying about breaking characters. These assumptions enable you to write simpler and, possibly, faster performing programs.

- DBCS non-modal encodings
  DBCS non-modal encodings are used to support East Asian languages, such as Japanese and Chinese, on an operating system that supports ASCII. In a non-modal DBCS encoding, single-byte and double-byte characters occupy different code ranges. Two bytes are typically used to represent an ideographic character. However, some characters are represented by a single byte. For example, ASCII and half-width Katakana characters are represented by one byte. To find character boundaries, SAS maintains tables that map a character code-point (byte value) to the character that it represents. Using these tables to validate characters in a string, K functions can determine when an offset is at valid character boundary.

- DBCS modal encodings
  DBCS modal encodings are EBCDIC-based encodings that are supported by SAS running on a mainframe machine. For SBCS EBCDIC encodings there is no trouble with character boundaries. However, DBCS EBCDIC encodings are varying length encodings that support both single-byte and

double-byte characters. In DBCS a modal encoding, the single-byte and double-byte characters cannot be easily distinguished by the character-code range. In order to determine whether the bytes in a character string represent a DBCS or single-byte character, escape codes are used. These escape codes, called *Shift-out (SO)* and *Shift-in (SI),* surround the double-byte characters in the string to distinguish them from single-byte characters. Logic to detect character boundaries searches for a pair of *SO* and *SI* marks in order to locate DBCS characters in the string. The sample below shows the layout of a string that contains both single-byte and double-byte characters in the Japanese DBCS encoding IBM-939.

IBM-930 Str	A	B	***SO***	中		文		***SI***	C	D	***SO***	日		本		***SI***
Byte Offset	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Table 3: EBCDIC Encoding

- UTF-8
  UTF-8 is also multi-byte and variant-width character encoding, with a maximum of four bytes for one character. SAS supports all UTF-8 characters regardless of the length. The following table shows the number of bytes that are needed in order to represent the characters for the languages in the list. The first byte of each UTF-8 character contains flag bits that allow K functions to identify the length of the character. Therefore, it is possible for K functions to go forward and backward in a UTF-8 string to locate the characters boundaries.

Character length	Language
1 byte	US_ASCII Characters
2 bytes	East and West European, Baltic, Greek, Turkish, Cyrillic, Hebrew, Arabic, and other supported character sets.
3 bytes	Chinese, Japanese, Korean (CJK), Thai, Indic, and certain control characters
4 bytes	Emoji characters, less common CJK characters and various historic scripts

Table 4: UTF-8 Character Length by Language

Understanding character boundaries can help you realize how characters are represented in computer memory. K functions ensure that characters are not broken, which provides consistent results for data in any encoding.

I18N COMPATIBILITY OF STRING FUNCTIONS

In order to understand which string functions are appropriate in certain situations, SAS has devised a system of I18N levels. Each string function is assigned an I18N compatibility level from 0 to 2. Most K functions have full Level 2 I18N compatibility whereas non-K functions might only work in certain situations. The I18N Compatibility Standard is described below:

- I18N Level 0
  This function is designed for SBCS data. Do not use this function to process DBCS/MBCS data.
- I18N Level 1
  This function requires encoding specific logic. It should be avoided when processing DBCS/MBCS data. The function might not work correctly under certain circumstances.
- I18N Level 2
  This function can be used for SBCS, DBCS, and MBCS (UTF-8) data.

Reference 1: K Functions Matrix of Properties includes I18N compatibility information for each function that is listed. See the column “I18N Level” for information.

BYTE VERSUS CHARACTER SEMANTICS

A function uses character semantics if the parameters, internal processing and return value are all expressed as character units for any type of character encoding. Otherwise, if a function's parameters and return value are defined in terms of byte units, the function is identified as one that uses byte semantics. Please refer to the column "Function Semantics" in the matrix table (Reference 1: K Functions Matrix of Properties) to query the semantics of function.

When talking about string offset, there are also two definitions, byte-based and character-based. Since DBCS and MBCS characters occupy more than one byte, a byte-base offset is not equivalent to a character-based offset for a string containing multi-byte characters. Being aware of byte semantics and char semantics can help you understand how to select the SAS string functions that you need to use.

When you run SAS with a DBCS or UTF-8 session encoding, the result from a function that uses character semantics could differ completely from the result of a similar function that uses byte semantics. For example, the REVERSE function is a byte-based function that reverses a character expression. KREVERSE has the same functionality, but uses character semantics. The example below, run in a SAS UTF-8 session, shows the difference between REVERSE and KREVERSE:

```
data _null_;
  str1 =  reverse('中国语');
  str2 = kreverse('中国语');
  put str1= / str2=;
run;

Result:
str1=▌▌▌▌▌▌  /* Garbage at the leading string */
str2=语国中
```

Example 6: REVERSE and KREVERSE Process UTF-8 Data

In this example, REVERSE returns un-recognized characters where the Japanese characters should have been. In Table 5 below, the hex row below the "Actual output" row shows the hex values of the string returned by REVERSE. The resulting string is neither the reverse order of the string nor valid characters in UTF-8.

Expecting result	语			国			中		
(hex.)	E8	AF	AD	E5	9B	BD	E4	B8	AD
Actual output	(corrupted garbage characters)								
(hex.)	AD	AF	E8	BD	9B	E5	AD	B8	E4

Table 5: The Byte Sequence Outputted by REVERSE

The logic of byte semantics always look at a string as a byte stream.

Semantics and I18N compatibility level are different concepts of string functions. Character semantic functions should be fully I18N compatible. However, some functions that use byte semantics are still fully I18N compatible.

For example, KSUSTRB is both I18N compatible, and it will never corrupt a character, and uses byte semantics. The arguments represent byte position and number of bytes. This example shows output from KSUBSTRB under SAS with a UTF-8 session encoding:

```
data _null_;
  str1 = ksubstrb('中国语', 2, 6);
  put str1=;
run;
```

```
Result:
str1=国
```

Example 7: KSUBSTRB Process UTF-8 Data Example

In this example, the position argument is 2, which is in the middle of the first Japanese character. The byte length of substring is 6, ending at the first byte of the third Japanese character. However, KSUBSTRB does not interpret either the position or the length in a way that would return invalid characters. The third row in the table below shows the byte stream that is initially extracted. Rather than returning a string that contains invalid characters, KSUBSTRB adjusts the beginning and end of the string to remove bytes that do not represent valid characters, as shown in the fourth row. The fifth row shows the actual result that is returned.

str =	中			国			语		
(hex. In UTF-8)	E4	B8	AD	E5	9B	BD	E8	AF	AD
ksubstrb(str, 2, 6)		B8	AD	E5	9B	BD	E8		
boundary adjust				E5	9B	BD			
actual output				国					

Table 6: The Byte Sequence Outputted by KSUBSTRB

NEW STAGE WITH VARCHAR

The new data type VARCHAR is being introduced into SAS products. Its emergence leads to a new milestone for SAS string functions. The existing character data type in SAS is defined in terms of bytes. However, the VARCHAR data type uses character rather than byte semantics.

Many non-K and K functions support VARCHAR columns. Every function that is available for VARCHAR must support character semantics when a VARCHAR is specified. A string function that supports VARCHAR can be used with characters from any encoding. VARCHAR represents a convergence of byte and character semantics when processing SAS string variables.

The example below shows how K and non-K functions can return identical results for VARCHAR data that contains Japanese characters. The example was run in SAS with a Shift-JIS session encoding:

```
data _null_;
  length str varchar(100);
  str='中国語';
  len1 = length(str);
  len2 = klength(str);
  put len1= / len2=;

  str1 =  substr(str, 2, 2);
  str2 = ksubstr(str, 2, 2);
  put str1= / str2=;

  str1 =  reverse(str);
  str2 = kreverse(str);
  put str1= / str2=;

  str='サービス[10]';
  pos1 =  indexc(str, '[]');
  pos2 = kindexc(str, '[]');
  put pos1= / pos2=;
```

```
run;

Result:
len1=3
len2=3
str1=国語
str2=国語
str1=語国中
str2=語国中
pos1=5
pos2=5
```

Example 8: VARCHAR Data Type Example

The column “VARCHAR Support” in the table (Reference 1: K Functions Matrix of Properties) shows the non-K and K functions that support VARCHAR columns. SAS string functions automatically determine the processing logic for string input depending on whether the function input is VARCHAR or CHAR. Passing VARCHAR data to a function that does not support VARCHAR might cause unexpected errors.

BEST PRACTICE OF USING K FUNCTIONS

In prior sections, you have learned about multi-byte string handling with K functions: basic concepts, working principle and character semantic. To use SAS string functions more efficiently in practice, you need to know the best strategy of whether to choose K functions or non-K functions and how to use K functions with better performance.

CHOOSING THE RIGHT STRING FUNCTIONS

Using K functions is absolutely a good first step toward making your program internationalized. However, you cannot simply replace every non-K function with a K function with a similar name. You must first understand the purpose and behavior of the function in order to get the results that you expect.

This compatibility issue varies from function to function. Some K functions have different number of arguments than the similarly named non-K function, or the arguments may be defined differently. Some K functions might have slightly different functionality than the non-K counterpart.

The table Reference 2: K Functions versus Non-K Functions Compatibility lists non-K functions with their K function equivalent. The table also explains any differences between the functions. This table will help you choose the appropriate function to use.

Most of the differences between non-K and K functions is simply character versus byte semantic. In that case, you can simply replace non-K function with the K function equivalent.

For example:

```
INDEX/KINDEX
INDEXC/KINDEXC
LENGTH/KLENGTH
REVERSE/KREVERSE
TRANSLATE/KTRANSLATE
VERIFY/KVERIFY
```

In some cases, the behavior of the non-K function is actually identical to the K function and no change is needed. For example, both LOWCASE and UPCASE are fully I18N compliant and can be used in a multi-byte SAS session. Other non-K and K functions might handle characters, such as double-byte space

characters, that your data will never contain. You might choose to continue using the LEFT, RIGHT, and STRIP functions instead of replacing them with KLEFT, KRIGHT, and KSTRIP.

However, other non-K and K functions have extra differences, such as different arguments or behavior. You need read the specification in the table carefully, and choose them appropriately.

USE K FUNCTIONS IN DBCS/MBCS TRUNCATION CASES

Data truncation frequently occurs when you put DBCS/MBCS data into a SAS character variable, and the length of the variable is not large enough. It also might occur when you migrate your SAS program or Data Set from a regional SAS encoding to SAS with a UTF-8 session encoding. This is because CJK (Chinese, Japanese and Korean) characters in DBCS system usually occupy two bytes in a regional encoding, but require three bytes in UTF-8. A character variable might have enough bytes to store characters in a DBCS encoding, but might fail to store the same number of characters when in UTF-8.

To understand what DBCS/MBCS data truncation is, please see Example 9: Data Truncation Example.

```
data _null_;
  length res $8;
  dbstr = '中国語';
  len1 = KLENGTH(dbstr);
  put len1= dbstr=;
  res = dbstr;
  len2 = KLENGTH(res);
  put len2= res =;
  res = KTRUNCATE(dbstr);
  len3 = KLENGTH(res);
  put len3= res =;
run;

Result in DBCS session:
len1=3 dbstr=中国語     /* '中国語' occupies 6 bytes in DBCS session.  */
len2=3 res = 中国語     /* No truncation occurs.                       */
len3=3 res = 中国語
Result in UTF-8 session:
len1=3 dbstr=中国語     /* '中国語' occupies 9 bytes in UTF-8 session. */
len2=3 res = 中国?      /* Truncation occurs in UTF-8 encoding.        */
len3=2 res = 中国
```

Example 9: Data Truncation Example

Variable *res* is a character variable with a fixed length of 8. The character variable, *dbstr,* contains three Chinese characters which require six bytes in SAS with a DBCS session encoding that supports Chinese characters. The value of *dbstr* is successfully assigned to the variable *res.*

However, when the same program is run in SAS with a UTF-8 session encoding, the length of the characters assigned to *dbstr* is 9. When SAS attempts to assign the string in *dbstr,* which has a length of 9, directly to *res*, which is only eight bytes long, the string is truncated. There is a trailing garbage byte (displayed as '?') at the end of *res*.

To prevent the truncation, call KTRUNCATE to handle the variable assignment. In case 3, we called it before assigning *dbstr* to *res*. The input string is trimmed to two characters and fit in the result variable, and no trailing garbage bytes occur.

Other K functions could also handle the DBCS characters truncation case properly. Replacing KTRUNCATE with KSTRIP also prevents the trailing garbage bytes. However, the STRIP function does not fix the problem. DBCS/MBCS character truncation handling is a benefit of K functions. This is also another reason that you should use K functions to process DBCS/MBCS data.

USE KUPDATE AS A REPLACEMENT OF SUBSTR AT THE LEFT SIDE OF '='

To update or correct some text in a string, we usually put SUBSTR at the left side of the assignment operator '='. The result of the expression on the right of the assignment statement is assigned to the segment of the string identified by SUBSTR on the left.

```
data _null_;
  a = 'Starbuck';
  substr(a, 5, 4) = 'gaze';
  put a=;
run;

Result:
a=Stargaze
```

Example 10: Use SUBSTR at the Left Side of '=' to Update the String

However, as mentioned earlier, the SUBSTR function does not handle multi-byte characters correctly. Also, K functions cannot be used on the left side of an assignment statement. Therefore, SUBSTR cannot be replaced by KSUBSTR in the example above.

When you need to internationalize the code in Example 10, the correct counterpart K function to use is KUPDATE. Example 11 shows how to use KUPDATE completing the same functionality.

```
data _null_;
  a = 'Starbuck';
  a = kupdate(a, 5, 4, 'gaze');
  put a=;
run;

Result:
a=Stargaze
```

Example 11: Use KUPDATE Update Part of the Input String

HANDLING VARIOUS SPACES WITH K FUNCTIONS

Before knowing about various spaces, you need to understand the concept of full-width and half-width characters. With fixed-width fonts, a half-width character occupies half the width of a full-width character. Generally, an ideographic, or CJK character, is full-width and a Latin character is half-width. To help align full-width CJK characters with Latin characters, regional DBCS encodings also include full-width Latin alphabet, numeric and space characters available in CJK character set.

Take the full-width ideographic space 0xE38080 (UTF-8 code point) as an example. It has different code points in different encodings as the following table shows. K functions can handle this.

Table 7 shows some pairs of full-width and half-width Latin alphabet, numeric and space characters and the code points for those characters in various encodings.

Encodings name	Half-width	Full-width	Half-width	Full-width	Half-width	Full-width	SBCS space	Full-width space
	A	A	a	a	1	1		
UTF-8	41	EFBCA1	61	EFBD81	31	EFBC91	20	E38080
Shift-JIS	41	8260	61	8281	31	8250	20	8140
GB2312	41	A3C1	61	A3E1	31	A3B1	20	A1A1

Table 7: Full-Width and Half-Width Characters

In K functions, KTRIM, KLEFT, KRIGHT and KCOMPRESS are space handling related functions. The equivalent non-K functions also handle space characters but do not know about full-width and Unicode spaces. If you need to handle various space characters in your program, you need to use these K functions.

Example 12 shows SAS code that calls COMPRESS and KCOMPRESS. KCOMPRESS removes DBCS spaces from DBCS string which could not be detected by COMPRESS.

```
data _null_;
  dbstr = 'a b c   d e f   g h i';  /* DBCS space separated words */
  str1 = COMPRESS(dbstr);
  str2 = KCOMPRESS(dbstr);
  put str1= / str2 =;
run;

Result:
str1 = a b c   d e f   g h i
str2 = a b c d e f g h i
```

Example 12: KCOMPRESS Compresses DBCS String

(Reference 3: Unicode Spaces Definition) shows the Unicode spaces could be handled by KTRIM, KLEFT, KRIGHT and KCOMPRESS.

SIMILAR FUNCTION NAMES, DIFFERENT FUNCTIONS

From the naming, you can see that all K functions name have a 'K' prefix. The 'K' version and the corresponding non-K version without 'K' prefix, usually implied they are a pair of functions, and should have similar functionality. This is very likely to be true. But, in some cases, they are not pair of functions like their name implied, and this is due to some historical reason which we cannot change. You need to distinguish these functions in SAS programming.

Here are two samples of functions that have this issue.

PROPCASE versus KPROPCASE

The functions PROPCASE and KPROPCASE provide one example of a pair of functions that have similar names but behave very differently.

PROPCASE was designed for proper case or title case. It converts a string like "INTRODUCTION TO THE SCIENCE OF ASTRONOMY" to "Introduction To The Science Of Astronomy" which is usually used in article title. PROPCASE is already an internationalized function that correctly supports SBCS, DBCS,

and MBCS characters. For example, PROPCASE was expanded in SAS 9.4 to support locale-sensitive lowercase and uppercase in Turkish language.

On the other hand, KPROPCASE was designed for converting CJK characters. Using KPROPCASE, you can convert between Half-Katakana and Full-Katakana, Full-alphabet and Half-alphabet, or Katakana and Romaji characters. KPROPCASE also supports uppercase and lowercase. It has completely different functionalities compared to PROPCASE.

PROPCASE is usually used with ASCII characters or Latin characters in western European languages, while KPROPCASE is used with CJK characters in East Asian languages. Their naming frequently confuses users. They both are I18N level 2 functions, no need to replace or switch PROPCASE to KPROPCASE in any circumstance.

COUNT versus KCOUNT

Another pair of functions with similar names are COUNT and KCOUNT.

The COUNT function counts the number of times a substring occurs within a string. The purpose of KCOUNT is very different. KCOUNT is used to count the number of DBCS/MBCS characters in a source string.

COUNT supports SBCS characters well, but does not handle DBCS and MBCS characters correctly. If your data contains multi-byte characters, you can use KCOUNTX as a replacement function. KCOUNTX is a new I18N function which was introduced in the latest SAS release: 9.4m5 and SAS Viya 3.3.

BINARY DATA IN K FUNCTIONS

Most of the K functions are designed to use character semantics. Those functions do not handle binary data. However, there are two K functions that do accept binary data: KCVT and KPROPDATA. They are designed to transcode an input string from a specified encoding to another encoding. Both functions have input and output encoding arguments. Both functions also accept a string as input that is in a specified encoding which does not match with current SAS session encoding. They have no CHAR or BYTE semantics properties in the table Reference 1: K Functions Matrix of Properties.

THE PERFORMANCE OF K FUNCTIONS

From prior sections, you can see K functions need to consider various conditions when processing DBCS/MBCS data. For some data, they might have performance degradation compared to non-K functions. This is one reason we chose to create a new set of functions rather than internationalizing original non-K functions. In performance-sensitive SAS programs that process a very large data set, you should be careful to choose the appropriate string functions to complete the task.

For an example, when the LENGTH function calculates the length of a string, it costs constant time because it is simply counting the number of bytes. However, in order to count the number of characters in a string, KLENGTH is required to check every byte in the string to calculate the character-length. The cost of linear time depends on the string length and the characters in the string. When multi-byte characters are present, KLENGTH is much slower than LENGTH.

Data normalization for better performance

In Base SAS, we usually use DATA STEP to read observations and variables from serialized data sets. An observation in a SAS data set might have two types of variables: character and numeric. The character variable has a fixed length.

When we read those character variables from a SAS data set to a local string variable or pass to a function, the string regularly has leading or trailing blanks. As we know, K functions often need to iterate over every character in the string to get the expected result. But many trailing blanks will cause K functions run into long time loops. This meaninglessly consumes computer resources. To remove those blanks, you can use one of the string functions to trim them off before processing the string. We call this process string data normalization.

Example 13 shows an example of string variable normalization. We added '*' mark at the beginning and end of the string to show the trailing blanks. As you can see, the variable *model* has unnecessary trailing blanks. Since the trailing blanks are SBCS characters, we can use the STRIP function to trim them off. Then, we get clean result in variable *r2*.

```
data cars;  /* Data normalization example */
  set sashelp.cars(firstobs=1 obs=1);
  r1 = '*' || LEFT(model) || '*';
  len1 = LENGTH(r1);
  r2 = '*' || LEFT(STRIP(model)) || '*';
  len2 = LENGTH(r2);
  put r1 = len1= / r2 = len2= ;
run;

Result:
r1=*MDX                                          * len1=42
r2=*MDX* len2=5
```

Example 13. Data Normalization Example

You could also use a function combination like KTRIM(KLEFT()) to trim the leading and trailing blanks. KLEFT first puts all blanks at the right side of the string, and then KTRIM trims them off. The problem is KTRIM and KLEFT will consider DBCS blanks in a DBCS session or Unicode spaces in a Unicode SAS session. This will make the normalization operation much slower compared to STRIP. If you are not concerned about stripping out multi-byte spaces, using STRIP is a better option.

In Base SAS 9.4, we introduced a new function KSTRIP as a replacement of the cases like KTRIM(KLEFT()) to trim the leading and trailing blanks in DBCS/MBCS SAS session. KSTRIP could only trim SBCS spaces. However, it can handle DBCS data truncation case properly.

The following example shows the performance difference between KTRIM(KLEFT()) and KSTRIP().

```
data _null_;
  length str200 $200.;
  length r $5;
  count = 50000000;
  str200 = "apple";

  t1 = time();
  do i = 1 to count;
    r = KTRIM(KLEFT(str200));
  end;
  t2 = time();
  time1 = t2 - t1;

  t1 = time();
  do i = 1 to count;
    r = KSTRIP(str200);
  end;
  t2 = time();
```

```
  time2 = t2 - t1;

  put "KTRIM(KLEFT()) cost: " time1 4.2 " second.";
  put "KSTRIP() cost: " time2 4.2 " second.";
run;

Result:
KTRIM(KLEFT()) cost: 2.85 second.
KSTRIP() cost: 1.37 second.
```

Example 14: Use KSTRIP Instead of KTRIM(KLEFT()) in DBCS Environment for Better Performance

Using non-K functions process SBCS data

K functions are not required if you are quite sure that your input data does not contain multi-byte data. For example, for data that only contains ZIP codes, serial number or phone numbers, use the non-K functions for any string manipulation that is needed.

Using non-K functions to process UTF-8 data in some circumstances

K functions are required to guarantee that the right result is returned when processing character data in SAS with a UTF-8 session encoding. However, if the returned result that you want is simply TRUE or FALSE, a non-K function might return an acceptable response and perform better.

The example below was run in SAS with a UTF-8 session encoding. SAS code compares the results when INDEX and KINDEX are called to check if a certain substring exists in the base string.

```
data _null_;
  str1 = "赛仕软件提供了很多字符处理函数。";
  str2 = "函数";
  index1 = INDEX(str1, str2);
  index2 = KINDEX(str1, str2);
  if (index1) then
    put "'str1' contains 'str2' at byte position: " index1;
  else
    put "'str1' does NOT contains 'str2' at byte position: " index1;
  if (index2) then
    put "'str1' contains 'str2' at character position: " index2;
  else
    put "'str1' does NOT contains 'str2' at character position: " index2;
run;

Result:
'str1' contains 'str2' at byte position: 27
'str1' contains 'str2' at character position: 14
```

Example 15: Use INDEX to Check MBCS Data in UTF-8 Session

In this case, the SAS code is simply checking for a nonzero return value to verify the presence of the string. INDEX can be used this way in SAS with a UTF-8 encoding as long as the substring contains valid

UTF-8 characters. Unlike DBCS encodings, there is no code point conflict in UTF-8 between 1-byte characters and the bytes of a multi-byte character.

CONCLUSION

The key value of K functions is to simplify SAS programming logic to deal with strings in different encodings by introducing character semantic programming. K functions allow your program to target consumers around the world. Non-K functions are fundamental string functions for single-byte character encodings. The process of understanding K functions is the process of understanding character semantics. With careful use of non-K functions and K functions, users can write internationalized SAS programs without suffering the performance overhead. K functions are the only solution for all languages in any SAS environment.

APPENDIX

K FUNCTIONS PROPERTIES MATRIX

No.	Function	I18N Level	Function semantics	Environment Support	VARCHAR Support	Description
1	CAT	2	BYTE	BASE, DS2, CAS	YES	Does not remove leading or trailing "blanks", and returns a concatenated character string.
	KSTRCAT	2	CHAR	BASE, DS2, CAS	YES	Concatenates two or more character expressions.
2	COMPARE	0	BYTE	BASE, DS2, CAS	YES	Returns the position of the leftmost character by which two strings "differ," or returns 0 if there is no difference.
	KCOMPARE	2	CHAR	BASE, DS2, CAS	YES	Returns the result of a comparison of character expressions.
3	COMPRESS	0	BYTE	BASE, DS2, CAS	YES	Returns a character string with specified characters removed from the original string.
	KCOMPRESS	2	CHAR	BASE, DS2, CAS	YES	Removes specified characters from a character expression.
4	COUNT	0	BYTE	BASE, DS2, CAS	YES	Counts the number of times that a specified substring appears within a character string.
	KCOUNTX	2	CHAR	BASE, DS2, CAS	YES	Counts the number of times that a specified substring appears within a character string.
5	KCOUNT	2	CHAR	BASE, DS2, CAS	YES	Returns the number of DBCS/MBCS characters in an expression.
6	INDEX	0	BYTE	BASE, DS2, CAS	YES	Searches a character expression for a string of "characters" and returns the position of the string's first character for the first occurrence of the string.
	KINDEX	2	CHAR	BASE, DS2, CAS	YES	Searches a character expression for a string of characters.
7	KINDEXB	2	BYTE	BASE	YES	Searches a character expression for a string of characters.
8	INDEXC	0	BYTE	BASE, DS2, CAS	YES	Searches a character expression for any of the specified "characters" and returns the position of that character.
	KINDEXC	2	CHAR	BASE, DS2, CAS	YES	Searches a character expression for specified characters.
9	KINDEXCB	2	BYTE	BASE	YES	Searches a character expression for specified characters.
10	LEFT	2	BYTE	BASE, DS2, CAS	YES	Left-aligns a character string.
	KLEFT	2	CHAR	BASE, DS2, CAS	YES	Left-aligns a character expression by removing unnecessary leading DBCS blanks and SO/SI.

No.	Function	I18N Level	Function semantics	Environment Support	VARCHAR Support	Description
11	LENGTH	2	BYTE	BASE, DS2, CAS	YES	Returns the byte length of a non-blank character string excluding trailing "blanks" and returns 1 for a blank character string.
	LENGTHC	2	BYTE	BASE, DS2, CAS	*NO*	Returns the byte length of a character string, including trailing blanks.
12	LENGTHN	2	BYTE	BASE, DS2, CAS	*NO*	Returns the length of a character string excluding trailing blanks.
	KLENGTH	2	CHAR	BASE, DS2, CAS	YES	Returns the character length of a string.
13	LOWCASE	2	BYTE	BASE, DS2, CAS	YES	Converts all letters in an argument to lowercase.
	KLOWCASE	2	BYTE	BASE, DS2, CAS	YES	Same as lowercase.
14	REVERSE	0	BYTE	BASE, DS2, CAS	YES	Reverses a character string.
	KREVERSE	2	CHAR	BASE, DS2, CAS	YES	Reverses a character string based on character semantic.
15	RIGHT	2	BYTE	BASE, DS2, CAS	YES	Right aligns a character expression.
	KRIGHT	2	CHAR	BASE, DS2, CAS	YES	Right-aligns a character expression by trimming trailing DBCS blanks and SO/SI.
16	SCAN	0	BYTE	BASE, DS2, CAS	YES	Returns the nth word from a character string.
	KSCAN	2	CHAR	BASE, DS2, CAS	YES	Returns the nth word from a character string based on character semantic.
17	SUBSTR (right of =)	0	BYTE	BASE, DS2, CAS	YES	Extracts a substring from an argument.
	KSUBSTR	2	CHAR	BASE, DS2, CAS	YES	Extracts a substring from an argument based on character semantic.
18	KSUBSTRB	2	BYTE	BASE	YES	Extracts a substring from an argument according to the byte position of the substring in the argument.
19	SUBSTRN	0	BYTE	BASE	*NO*	Returns a substring allowing a result with a length of zero.
	KSUBSTRN	2	CHAR	BASE	YES	Returns a substring, allowing a result with a length of zero based on character semantic.
20	SUBSTR (left of =)	0	BYTE	BASE, DS2, CAS	YES	Replaces character value contents.
	KUPDATE	2	CHAR	BASE, DS2, CAS	YES	Inserts, deletes, and replaces character value contents.

No.	Function	I18N Level	Function semantics	Environment Support	VARCHAR Support	Description
21	KUPDATEB	2	BYTE	BASE	YES	Inserts, deletes, and replaces the contents of the character value according to the byte position of the character value in the argument.
22	KUPDATES	2	CHAR	BASE, DS2, CAS	YES	Inserts, deletes, and replaces character value contents.
23	STRIP	2	BYTE	BASE, DS2, CAS	YES	Removes leading and trailing blanks from a character string.
	KSTRIP	2	BYTE	BASE, DS2, CAS	YES	Removes leading and trailing blanks from a character string.
24	TRANSLATE	0	BYTE	BASE, DS2, CAS	YES	Replaces specific characters in a character string.
	KTRANSLATE	2	CHAR	BASE, DS2, CAS	YES	Replaces specific characters in a character string based on character semantic.
25	TRIM	2	BYTE	BASE, DS2, CAS	YES	Removes trailing blanks from a character "string," and returns one blank if the string is missing.
	KTRIM	2	CHAR	BASE, DS2, CAS	YES	Removes trailing DBCS blanks and SO/SI from character expressions.
26	UPCASE	2	BYTE	BASE, DS2, CAS	YES	Converts all letters in an argument to uppercase.
	KUPCASE	2	BYTE	BASE, DS2, CAS	YES	Same as uppercase.
27	VERIFY	2	BYTE	BASE, DS2, CAS	YES	Returns the position of the first character in a string that is not in any of several other strings.
	KVERIFY	2	CHAR	BASE, DS2, CAS	YES	Returns the position of the first character that is unique to an expression.
28	KVERIFYB	2	BYTE	BASE	YES	Returns the position of the first character that is unique to an expression.
29	KCVT	1		BASE	*NO*	Converts data from one type of encoding data to another type of encoding data. So, KCVT has no semantics definition.
30	PROPCASE	2	CHAR	BASE, DS2, CAS	YES	Converts all words in an argument to proper case.
31	KPROPCASE	2	CHAR	BASE	YES	Converts Chinese, Japanese, Korean, Taiwanese (CJKT) characters.
32	KPROPCHAR	2	CHAR	BASE	YES	Converts special characters to normal characters.
33	KPROPDATA	2		BASE	*NO*	Converts the input data string to the current SAS session encoding and removes or converts unprintable characters.
34	KTRUNCATE	2	CHAR	BASE	YES	Truncates a string to a specified length in byte unit without breaking multi-byte characters.

Reference 1: K Functions Matrix of Properties

K FUNCTIONS VERSUS NON-K FUNCTIONS COMPATIBILITY TABLE

SAS Function	Equivalent K Function	Description of feature	Comment
CAT	KSTRCAT	Concatenates character strings without removing leading or trailing blanks.	If your application defines the semantic of DBCS blanks as spaces or needs 'smart' handling of DBCS data on z/OS, use KSTRCAT. Otherwise, CAT is allowed.
COMPARE	KCOMPARE	Returns the result of a comparison of character strings.	COMPARE arguments include the strings to compare and a modifier to customize the results of the comparison. KCOMPARE does not support the modifier but does include arguments to set the position to start the comparison and number of bytes to examine.
COMPRESS	KCOMPRESS	Removes specific characters from a character string.	COMPRESS supports a modifier to customize the results of the compression. KCOMPRESS does not support the modifier argument.
COUNT	KCOUNTX (Available in SAS Viya)	Returns the number of times a specified substring occurs in a string.	COUNT will work if the search string contains single-byte characters and there is more than one single-byte character. The functionality and argument list of COUNT differs from KCOUNT.
	KCOUNT	Returns the number of national characters in a string.	KCOUNT can be used to determine whether non-K functions are appropriate.
INDEX	KINDEX or KINDEXB	Searches a character expression for a string of characters.	If the search string is found, KINDEX returns the *character* position of the substring while KINDEXB returns the *byte* position.
INDEXC	KINDEXC or KINDEXCB	Searches a character expression for specific characters.	If the search string is found, KINDEXC returns the *character* position of the substring while KINDEXCB returns the *byte* position.
LEFT	KLEFT	Left aligns a character expression by removing unnecessary leading DBCS blanks and SO/SI.	KLEFT treats DBCS spaces as blank characters. If your application defines the semantic of DBCS blanks as spaces, you must use the KLEFT function. Otherwise, if DBCS space is just another character, you can use the LEFT function.
LENGTH or LENGTHN	KLENGTH	Returns the length of an argument.	For missing or blank strings, KLENGTH returns the same result as LENGTHN. For a non-blank string, KLENGTH returns the number of characters in the string while LENGTH and LENGTHN return the number of bytes.
LOWCASE	KLOWCASE	Converts all letters in an argument to lowercase.	LOWCASE and KLOWCASE are same, just different alias of same function.

SAS Function	Equivalent K Function	Description of feature	Comment
REVERSE	KREVERSE	Reverses a character expression.	REVERSE is not internationalized, change REVERSE to KREVERSE, their arguments are same.
RIGHT	KRIGHT	Right aligns a character expression by trimming trailing DBCS blanks and SO/SI.	KRIGHT treats DBCS spaces as blank characters. If your application defines the semantic of DBCS blanks as spaces, you must use the KRIGHT function. Otherwise, if DBCS space is just another character, you can use the RIGHT function.
SCAN	KSCAN	Selects a specified word from a character expression.	The SCAN argument that specifies modifiers is not supported in KSCAN. However, the basic functionality of KSCAN is compatible.
STRIP	KSTRIP (Available in 9.3m2 and 9.4)	The STRIP function returns the argument with all leading and trailing blanks removed.	Behavior of KSTRIP is the same as STRIP. In addition, KSTRIP also considers the DBCS character truncation case.
SUBSTR (left of =)	See comments	Replaces character value contents.	The K functions do not work the same way as SUBSTR on the left side of the equal sign. KSUBSTR or KSUBSTRB can be used on the left of the = in a conditional statement when the value is not saved. - Otherwise, use KUPDATE, KUPDATEB on the right of the =.
SUBSTR (right of =)	See comments	Extracts a substring from an argument.	Change might depend on usage. Available functions are KSUBSTR, KSUBSTRB.
TRANSLATE	KTRANSLATE	Replaces specific characters in a character expression.	In general, KTRANSLATE handles non-binary data. If data contains binary data, the result is unexpected. TRANSLATE does not distinguish binary from printable characters.
TRIM	KTRIM	Removes trailing DBCS blanks and SO/SI from character expressions.	KTRIM treats DBCS spaces as blank characters. If your application defines the semantic of DBCS blanks as spaces, you must use the KTRIM function. Otherwise, if DBCS space is just another character, you can use the TRIM function.
UPCASE	KUPCASE	Converts all single-byte letters in an argument to uppercase.	LOWCASE and KLOWCASE are same, just different alias of same function.
	KUPDATE or KUPDATEB (right of =)	Inserts, deletes, and replaces the contents of the character value.	I18N string update function which as replacement of SUBSTR.
	KUPDATES (Available in 9.4)	Inserts, deletes, and replaces the contents of the character value.	Little difference comparing to KUPDATE in string output format. See full compatibility table for detail.
VERIFY	KVERIFY or KVERIFYB	Returns the position of the first character in a	KVERIFY returns the *character* position of the first character as described. KVERIFYB returns the *byte* position. For example, if

SAS Function	Equivalent K Function	Description of feature	Comment
		string that is not in any of several other strings	the string can contain the characters 'abcë' and the variable contains 'abcë€', KVERIFY returns 5, KVERIFYB returns 6 in UTF-8 session.

Reference 2: K Functions versus Non-K Functions Compatibility

UNICODE SPACES DEFINITION

No.	KCOMPRESS	KLEFT KRIGHT and KTRIM	Comment	General category
1		0x0009	Character tabulation	Other control
2		0x000a	Line feed	Other control
3		0x000b	Line tabulation	Other control
4		0x000c	Form feed	Other control
5		0x000d	Carriage return	Other control
6	0x0020	0x0020	Single-byte space, 0x40 in EBCDIC	Separator space
7		0x0085	Next line	Other control
9	0x00a0	0x00a0	No-break space	Separator space
10		0x1680	Ogham space mark	Separator space
11		0x180e	Mongolian vowel separator	Other Format
12	0x2000	0x2000	EN QUAD	Separator space
13	0x2001	0x2001	EM QUAD	Separator space
14	0x2002	0x2002	EN SPACE	Separator space
15	0x2003	0x2003	EM SPACE	Separator space
16	0x2004	0x2004	THREE-PER-EM SPACE	Separator space
17	0x2005	0x2005	FOUR-PER-EM SPACE	Separator space
18	0x2006	0x2006	SIX-PER-EM SPACE	Separator space
19	0x2007	0x2007	FIGURE SPACE	Separator space
20	0x2008	0x2008	PUNCTUATION SPACE	Separator space
21	0x2009	0x2009	THIN SPACE	Separator space
22	0x200a	0x200a	HAIR SPACE	Separator space
23	0x200b		ZERO WIDTH SPACE	Other Format
24		0x2029	PARAGRAPH SEPARATOR	Separator paragraph
25	0x202f	0x202f	Narrow no-break space	Separator space
26		0x205f	Medium mathematical space	Separator space
27	0x3000	0x3000	Ideographic space	Separator space

Reference 3: Unicode Spaces Definition

REFERENCES

"Internationalization and localization" Wikipedia. Available https://en.wikipedia.org/wiki/Internationalization_and_localization. Accessed on November 7, 2017

"Category: Character sets" Wikipedia. Available https://en.wikipedia.org/wiki/Category:Character_sets. Accessed on October 17, 2016

"Character encoding" Wikipedia. Available https://en.wikipedia.org/wiki/Character_encoding. Accessed on November 12, 2017

"Extended Unix Code" Wikipedia. Available https://en.wikipedia.org/wiki/Extended_Unix_Code. Accessed on October 28, 2017

Whistler, Ken, Davis, Mark, Freytag, Asmus. "Unicode Character Encoding Model." November 11, 2008 Available http://www.unicode.org/reports/tr17/.

"Whitespace character" Wikipedia. Available https://en.wikipedia.org/wiki/Whitespace_character. Accessed on November 26, 2017

Bales, Elizabeth, Zheng, Wei. "SAS® and UTF-8: Ultimately the Finest." 2017 Available https://www.sas.com/content/dam/SAS/support/en/technical-papers/SAS0296-2017.pdf

ACKNOWLEDGMENTS

We would like to express our gratitude to the people who helped make this paper possible. Alfred Liu, Li (Lane) Li, Kansun Xia, Hunter (Yuntao) Dou, and Sandy McNeill supported our efforts in researching and writing this paper. Scott Dobner and Dan Harrell provided valuable feedback during the review.

Finally, we would like to acknowledge Shinichiro Kayano who first added DBCS support to SAS® many years ago. His work stands the test of time.

RECOMMENDED READING

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

SAS Institute Inc. 2017. "Internationalization Compatibility for SAS String Functions" in SAS® 9.4 National Language Support (NLS): Reference Guide, Fifth Edition. Cary, NC: SAS Institute Inc. Available http://support.sas.com.

SAS Institute Inc. 2017. "VARCHAR Data Type in String Functions" in SAS® 9.4 National Language Support (NLS): Reference Guide, Fifth Edition. Cary, NC: SAS Institute Inc. Available http://support.sas.com.

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

CONTACT INFORMATION

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

Leo (Jiangtao) Liu
SAS Research and Development (Beijing) Co., Ltd.
+86 10 83193702
Jiangtao.liu@sas.com