Majerus.net Text automation library

Contents

Introduction

Majerus.Text is a COM/OLE Automation component designed to make working with strings and Unicode characters easier. It is bundled with Majerus.net ActiveScript Shell (axsh) and with Majerus.net PowerShell Tools.

The library is implemented as a single object containing all the core functionality.

JScript : `var text = new ActiveXObject("Majerus.Text");`
VBScript : `Set Text = CreateObject("Majerus.Text")`
PowerShell : `$text = New-Object -ComObject Majerus.Text`

Unicode, UTF-16 and UTF-32

Most modern text is encoded in Unicode, as it provides a single value for any character regardless of the language.

Windows uses UTF-16 internally, which uses 16-bit per character and provides a good compromise between memory space and number of characters that can be represented. The complete Unicode encoding would require 32-bit, but take twice as much memory for encoding characters from the

supplementary planes which are rarely used. These supplementary planes are not even used for complex Asian languages, they are only used as an extension for characters such as mathematical symbols, Egyptian hieroglyphs and such specific characters.

However, these supplementary planes characters can be encoded in UTF-16, but then use a pair of characters called surrogates, which makes it much harder for script developers to work with strings including these.
This is very efficient but breaks most assumptions… You cannot simply count the number of UTF-16 characters to know how many Unicode characters a string contains. You cannot just swap the characters order to reverse a string as that would also invert low and high surrogates of a single character, breaking these. You cannot access a character by index simply by considering a string as being an array of characters, as they are of variable length.

Majerus.Text works with UTF-16, but many of its features can expose the UTF-16 or hide it by grouping surrogates as single characters, effectively hiding the UTF-16 encoding complexity from you and letting you work as if you were working with UTF-32 instead.

A property lets you switch this behavior at any time and takes effect immediately (for any subsequent call).

`CharacterUnit`	Get or set what is considered a character. 1 cuCodeUnit means methods work at the WCHAR (UTF-16 code unit) level. 2 cuCodePoint means methods work at the Unicode Code Point level.

For most methods changing behavior depending on the state of the CharacterUnit property, the naive mode provided by CharacterUnit set to 1 (cuCodeUnit) will perform faster than their surrogates-aware mode provided by CharacterUnit set to 2 (cuCodePoint).

Note if you need to use both modes, you can toggle it at any time, but you can also simply instantiate two Text objects, one in each mode. The setting only applies to the object the property belongs to, it does not affect any other instance of the Text object.

Accessing characters

`GetCharCount (strInput)`	Returns the number of characters in a string
`GetCharAt (strInput, nIndex)`	Returns the character at the specified index
`GetCharCodeAt (strInput, nIndex)`	Returns an integer representing the Unicode encoding of the character at the specified location

GetCharCount returns the number of characters. When CharacterUnit is 2 (cuCodePoint), surrogate pairs are counted as one instead of two.

GetCharAt returns a string containing the single character at position nIndex. When CharacterUnit is 2 (cuCodePoint), nIndex is the Unicode Code Point position instead of UTF-16 Code Unit position, and the returned string could contain a surrogate pair, effectively two UTF-16 Code Units.

GetChatCodeAt works similarly to GetCharAt, but returns the UTF-16 or UTF-32 value of the character at the requested position.

Text effects

`Uppercase (strInput)`	Convert a string to capital letters
`Lowercase (strInput)`	Convert a string to small letters
`SmallCaps (strInput, bUseCompatibilityLookalikes=false, bAllowDecomposition=false, bUseCompatibilityNormalization=false)`	Convert small letters of a string to small capitals *(See note below for compatibility lookalikes and decomposition and compatibility normalization)*
`Underline (strInput, bDoubleUnderline=false)`	Add underline to text using Unicode low line diacritical marks
`Overline (strInput, bDoubleOverline=false)`	Add overline to text using Unicode overline diacritical marks
`Strikethrough (strInput)`	Add strikethrough to text using Unicode stroke overlay diacritical marks

These three methods convert the given string to uppercase, lowercase or small capitals versions. This does not apply only to the A to Z letters, but to all symbols that supports cases as well.

Most fonts, such as fonts used in the Windows Console, do not support characters introduced recently in the Unicode specifications for small capitals versions of F, Q and S. To use look-alikes available in most fonts instead of correct Unicode 13.0 small capital letters, set bUseCompatibilityLookalikes to true.

By default, SmallCaps does not replace precomposed characters with diacritics (accents, cedilla, …) as there are no precomposed equivalents. Use bAllowDecomposition = true to turn them into decomposed equivalents that can use the small caps characters with combining diacritics. Additionally, bUseCompatibilityNormalization can be set to true to allow simplifying characters to compatible form ones.

Changing strings

Replacing

`Replace (strInput, strFind, strReplacement)`	Replaces all matches of a substring by another string

If strFind is not found, is an empty string, or is identical to strReplacement, the string is returned unmodified.

Reversing

`Reverse (strInput)`	Reverse the characters of a string

Reverse makes it easy to invert the order of characters in a string. If CharacterUnit is 2 (cuCodePoint), it will keep surrogate pairs grouped properly. If CharacterUnit is 1 (cuCodeUnit), it will simply reverse the order of UTF-16 characters, which is faster but would break any character from other planes.

Trimming

`Trim (strInput, strCharacters=“ ”)`	Removes all leading and trailing occurrences of a set of characters from a string
`TrimEnd (strInput, strCharacters=“ ”)`	Removes all trailing occurrences of a set of characters from a string
`TrimStart (strInput, strCharacters=“ ”)`	Removes all leading occurrences of a set of characters from a string

Trimming consists of removing any character from a set of characters from the end, start, or both sides of a string, until the first character not in the set is encountered.

The default value for strCharacters is not just the space character, but a set of whitespace characters. If CharacterUnit is 2 (cuCodePoint), strCharacters can be used to specify characters from supplementary planes and each will be processed properly as a single character, not as two individual surrogates to trim if found.

Cropping

`CropEnd (strInput, nNumber)`	Return a specified number of characters from the left side of a string
`CropStart (strInput, nNumber)`	Return a specified number of characters from the right side of a string
`CropEven (strInput, nNumber)`	Return a specified number of characters from the middle of a string

Cropping removes characters from either or both sides of a string to return a string of the requested number of characters at most. If the string is shorter than the requested number of characters, the string is returned unchanged.

This is typically used to show only the beginning or the end of some text when it cannot exceed a predefined space.

If CharacterUnit is 2 (cuCodePoint), each surrogate pair will count as a single character. The returned string could be longer than nNumber but the number of Unicode characters will be nNumber, and surrogate pairs will not be separated.

Padding

`PadEnd (strInput, nNumber, character=“ ”)`	Extend a string to the specified number of characters by adding padding characters on the right side
`PadStart (strInput, nNumber, character=“ ”)`	Extend a string to the specified number of characters by adding padding characters on the left side
`PadEven (strInput, nNumber, character=“ ”)`	Extend a string to the specified number of characters by adding padding characters on both sides
`Padding (nNumber, character=“ ”)`	Return a padding string of the specified number of characters

Padding is used to make sure a string takes at least the specified number of characters, a longer string will not be shortened, but a shorter string will be extended using the specified padding character to be at least nNumber long. By default, strings will be padded with the space character, but any character can be provided.

If CharacterUnit is 2 (cuCodePoint), each surrogate pair will count as a single character and the padding character can be a surrogate pair.

Fitting

`FitLeft (strInput, nNumber, character=" ")`	Extend or crop a string to fit the specified number of characters, aligning it to the left
`FitRight (strInput, nNumber, character=" ")`	Extend or crop a string to fit the specified number of characters, aligning it to the right
`FitCenter (strInput, nNumber, character=" ")`	Extend or crop a string to fit the specified number of characters, centering it
`FitJustify (strInput, nNumber, character=" ")`	Extend or crop a string to fit the specified number of characters, justifying it

Fitting will either crop or pad the string to make it exactly the requested length.

If CharacterUnit is 2 (cuCodePoint), each surrogate pair will count as a single character, the padding character can be a surrogate pair, and surrogate pairs will not be separated.

FitJustify will not add padding characters at the beginning or end of a string, but instead insert them between words to keep text evenly spaced while touching both sides.

Conversions

`RemoveDiacritics (strInput, bCompatibilityNormalization=false)`	Remove diacritical marks from characters *(and simplify characters to compatible form ones if bCompatibilityNormalization is true)*
`MathVariant (strInput, variant, bAllowDecomposition=false, bUseCompatibilityNormalization=false)`	Replace alphanumeric characters by mathematical symbols variants. *(See note below for decomposition and compatibility normalization)*
`NumberToRoman (number, bLowercase=false, bUseRomanNumeralChars=false, bUsePrecomposedChars=false)`	Convert a number to a roman numerals string
`RomanToNumber (strRomanNumeral)`	Convert a roman numerals string to its numeric value
`HalfwidthToFullwidth (strInput)`	Convert Japanese and Hangul half-width to full-width
`FullwidthToHalfwidth (strInput)`	Convert Japanese and Hangul full-width to half-width
`HiraganaToKatakana (strInput)`	Convert Japanese hiraganas to katakanas
`KatakanaToHiragana (strInput)`	Convert Japanese katakanas to hiraganas
`SimplifiedToTraditional (strInput)`	Convert Simplified Chinese to Traditional Chinese
`TraditionalToSimplified (strInput)`	Convert Traditional Chinese to Simplified Chinese
`Precompose (strInput, bCompatibilityNormalization=false)`	Convert a string to Unicode normalized precomposed form C (canonical) or KC (compatible).

`Decompose (strInput, bCompatibilityNormalization=false)`	Convert a string to Unicode normalized decomposed form D (canonical) or KD (compatible). For example, Å (U+212B) is decomposed as **A** (U+0041) and ˚ (U+0E0A). Decomposed form can always be converted to precomposed form. Compatibility equivalence can be distinct appearances and cannot always be turned back to precomposed form, for example, **fi** (U+FB01) is decomposed as **f** (U+0066) and **i** (U+0069).

Valid variants for MathVariant are "normal", "bold", "italic", "bold-italic", "double-struck", "fraktur", "bold-fraktur", "script", "bold-script", "sans-serif", "bold-sans-serif", "sans-serif-italic", "sans-serif-bold-italic", "monospace" and "stretched".

By default, MathVariant does not replace precomposed characters with diacritics (accents, cedilla, …) as there are no precomposed equivalents. Use bAllowDecomposition = true to turn them into decomposed equivalents that can use the symbols variants. Additionally, bUseCompatibilityNormalization can be set to true to allow simplifying characters to compatible form ones.

Comparisons

`Compare (strString1, strString2)`	Compare strings linguistically

Strings will be compared linguistically, which means according to their sort order, and taking into consideration meaning. For example, “Item 9” will be sorted before “Item 10” even though “9” is after “1”. It performs a linguistical comparison, which means that for example "Masse" and "Maße" are considered equal and therefore return 0 (crEqual).

It returns -1 if strString1 comes before strString2, 0 if strings are identical, and 1 if strString1 comes after strString2.

Automation-specific

`StringToArray (strInput)`	Convert a string into an array of its characters values
`ArrayToString (arrayOfValues)`	Convert an array of characters values into a string
`FromCharCode (charValue, …)`	Returns a string from a number of Unicode character values
`Concat (strings…)`	Returns a string value containing the concatenation of supplied strings
`GetChars (strInput)`	Get an enumerable characters collection from a string
`IsTextString (strInput)`	Get whether a string is a well-formed text string (no isolated surrogate or lone byte) *(strings can contain unpaired surrogates, making them invalid UTF-16 sequences, or can even contain an odd number of bytes if representing Basic Byte Strings instead of text, this function checks whether the string is well-formed)*

These are mostly used for Active Scripting as they are tightly linked to Automation types.

Named characters

Chars	Named characters

The Chars property is an object that exposes characters as properties named using their common literal names. For example, Chars.times returns the “×” (multiplication sign) character.

They are too numerous to list here, but HTML5 literals provides a good reference.