uCalc String Library

uCalc String Library consists of a set of string-handling routines with names that will seem very familiar, especially to those who use either .NET's Microsoft.VisuaBasic.Strings module, VB 6, or PowerBASIC, but with support for uCalc Patterns, which infuses each routine with advanced parsing capabilities.

If you are familiar with such routines as InStr, Mid(), Left(), Right(), etc, you already know most of what you need to know to get started. This documentation focuses mainly on the additional functionality that uCalc brings to the table. The general description of each routine is similar to what you may be used to. The main differences are that text can be accessed or manipulated using special patterns, instead of just plain strings; and numeric arguments can be measured either in terms of characters, tokens, blocks, or expressions; and all routines have an additional Options parameter, which lets you select such things as whether the function call should be whitespace-sensitive and case-sensitive. By default case-sensitivity and white-space sensitivity is off, and tokens (not characters) are the unit of measurement, except where indicated otherwise. Some routines might be more familiar particularly to PowerBASIC users. As is common in BASIC dialects, character positions start at 1 (not 0).

For details about uCalc Patterns, visit http://www.ucalc.com/docs/Patterns.htm

Note: uCalc String Library routines are preceded by "uc" when called from your source code. However, when called from uCalc Console Calculator or uCalc Transform, the routines are not preceded by "uc". So for instance you would use ucInStr() if you are calling it from your source code, whereas you'd call InStr() if you are using uCalc Transform. With uCalc Fast Math Parser your source code would use ucInStr(), whereas your end-user expressions would use InStr(). In order for your uCalcFMP expressions to use the String Library, first place a call to ucLoadStringLibInFMP()

New: ucDelim, ucSplit, ucEquals, ucRange

ucInStr (*Start, MainString, Pattern* [, *Options*])

Returns the location of text within *MainString* that matches *Pattern*.

Start - Starting character position from where to begin the search
MainString - String within which to search for *Pattern*
Pattern - String or pattern to find within *MainString*

Example: InStr with default setting

ucInStr(1, MainString, "a test")

```
THIS IS A   TEST, "This is a test", (a  test), a  test, a test.
         ^
```

```
        Returns 9
```

Explanation: With the default setting, `"a test"` (third argument) is a pattern and not a literal string. This pattern consists of the tokens "a" followed by "test". Spacing between the two does not matter, nor does casing (upper case/lower case) by default.

Example: InStr with special patterns

ucInStr(1, MainString, "({whatever})")

```
THIS IS A   TEST, "This is a test", (a  test), a  test, a test.
                                    ^
                                    Returns 37
```

ucInStr(1, MainString, "is {etc:2}")

```
THIS IS A   TEST, "This is a test", (a  test), a  test, a test.
     ^
     Returns 6
```

Explanation: In the first part, it matches the first occurrence of a pair of parenthesis with some text inside. `{whatever}` is a pattern variable. The name in this example was arbitrarily selected; instead of {whatever}, it could have been named {MyPattern}, or {item} just the same. The second pattern matches the token "is" followed by the next two tokens. {etc} is another pattern variable. The ":2" in {etc} tells it to pick up the next two tokens. See http://www.ucalc.com/docs/Patterns.htm for more details on uCalc Patterns.

Example: InStr with ucChar property

ucInStr(1, MainString, "a test", **ucChar**)

```
THIS IS A   TEST, "This is a test", (a  test), a  test, a test.
                          ^
                          Returns 28
```

Explanation: **ucChar** makes it search character by character (the way traditional VB's InStr() function would do it), instead of token by token (which is the default in uCalc). This is similar to not using uCalc Patterns.

Example: InStr with ucCase (case-sensitive) property

ucInStr(1, MainString, "a test", **ucCase**)

```
THIS IS A   TEST, "This is a test", (a  test), a  test, a test.
                                     ^
                                     Returns 38
```

Explanation: Here we introduce **ucCase**, which makes the search case sensitive. It skips over `"A TEST"`, since it is all caps. Note that it also skips over the second literal occurrence, because it is within quotes. By default comparisons are done token by token, and `"This is a test"`

(including surrounding quotes) represents one token, so it does not match the two separate tokens: "a" followed by "test". Likewise, ucInStr(1, MainString, "is") would match the "IS" token following "THIS", and not the "IS" that occurs within the word "THIS".

Example: InStr with **ucSpace** (whitespace-sensitive) property

ucInStr(1, MainString, "a test", **ucSpace**)

```
THIS IS A   TEST, "This is a test", (a  test), a  test, a test.
                                                        ^
                                                        Returns 57
```

Explanation: Here the pattern is whitespace-sensitive. The given pattern has exactly one space, so it skips over occurrences where there is more than one space between "a" and "test".

Example: InStr with **ucBlock** property

ucInStr(1, MainString, "a test", **ucBlock**)

```
THIS IS A   TEST, "This is a test", (a  test), a  test, a test.
                                               ^
                                               Returns 48
```

ucInStr(1, MainString, "(a test)", **ucBlock**)

```
THIS IS A   TEST, "This is a test", (a  test), a  test, a test.
                                    ^
                                    Returns 37
```

Explanation: So far, we've searched character by character, or token by token. Here, we are searching block by block. A block consists of either one token, or if the token was defined with the block property, the block starts with that token and ends with the closing token associated with it. By default the following block token pairs are defined **(** and **)**, **{** and **}**, and **(** and **)**. The first part of the example skips over "(a test)", and matches the following occurrence because (a test), including the parenthesis, represents one block unit, and it is not equal to "a test". In the second example "(a test)" as a block matches the given pattern "(a test)".

Example: InStr with combined properties

ucInStr(1, MainString, "testing 123")

```
TESTING   123, testing   123, TESTING 123, "testing 123", (testing
123), testing 123.
^
Returns 1
```

ucInStr(1, MainString, "testing 123", **ucCase**)

```
TESTING   123, testing   123, TESTING 123, "testing 123", (testing
123), testing 123.
               ^
               Returns 16
```

ucInStr(1, MainString, "testing 123", **ucChar**)

```
TESTING   123, testing   123, TESTING 123, "testing 123", (testing
123), testing 123.
                           ^
                           Returns 31
```

ucInStr(1, MainString, "testing 123", **ucCase+ucChar**)

```
TESTING   123, testing   123, TESTING 123, "testing 123", (testing
123), testing 123.
                                           ^
                                           Returns 47
```

ucInStr(1, MainString, "(a test)", **ucCase+ucSpace**)

```
TESTING   123, testing   123, TESTING 123, "testing 123", (testing
123), testing 123.
                                                          ^
                                                          Returns 60
```

ucInStr(1, MainString, "(a test)", **ucCase+ucSpace+ucBlock**)

```
TESTING   123, testing   123, TESTING 123, "testing 123", (testing
123), testing 123.

^

Returns 74
```

Note: Not all combinations will produce a logically meaningful result. For instance ucChar cannot be combined with ucToken or ucBlock, because a character search is exclusive of token patterns.

---

ucLeft (*MainString*, *n* [, *Options*])

Returns the left-most *n* characters, tokens, blocks, or expressions in *MainString*.

Example: Counting tokens with ucLeft() (which is the default, and not characters)

ucLeft(MainString, 5)

```
This "is a test" using  (the quick) brown fox, and more.

Returns: [the text that's highlighted]
```

Explanation: By default functions from this library count tokens (not characters). The 5 left-most tokens are:

1. This
2. "
3. Is
4. A
5. Test

This example is not spacing-sensitive.

Example: Counting tokens with ucLeft() (which is the default, and not characters)

ucLeft(MainString, 5, **ucQuote**)
`This "is a test" using  (the quick) brown fox, and more.`

`Returns: [the text that's highlighted]`

Explanation: By default functions from this library count tokens (not characters). The 5 left-most tokens are:

1. `This`
2. `"is a test"`
3. `using`
4. `(`
5. `the`

This example is not spacing-sensitive. When ucQuote is on, text within quotes (including the quotes) count as one token.

Example: Counting characters with ucLeft()

ucLeft(MainString, 5, **ucChar**)
`This "is a test" using  (the quick) brown fox, and more.`

`Returns: "This " (without quotes, but including space)`

Explanation: The 5 left-most characters are:

1. `T`
2. `h`
3. `i`
4. `s`
5. `[space]`

Example: Counting blocks with ucLeft()

ucLeft(MainString, 5, ucQuote+**ucBlock**)
`This "is a test" using  (the quick) brown fox, and more.`

`Returns: [the text that's highlighted]`

Explanation: The 5 left-most blocks are:

1. `This`
2. `"is a test"`
3. `using`

4. `(the quick)`
5. `brown`

Example: Counting tokens, space-sensitive with ucLeft()

ucLeft(MainString, 5, ucQuote+**ucSpace**)
`This "is a test" using  (the quick) brown fox, and more.`

`Returns: [`*`the text that's highlighted`*`]`

Explanation: The 5 left-most whitespace-sensitive tokens are:
1. `This`
2. `[space]`
3. `"is a test"`
4. `[space]`
5. `using`

---

ucRight (*MainString*, *n* [, *Options*])

Returns the right-most *n* characters, tokens, blocks, or expressions in *MainString*.

The rules for ucRight are very similar to those for ucLeft. One important difference to consider (especially if speed is a consideration) is that unlike ucLeft, which parses only until it reaches the number of characters or tokens specified, ucRight always parses the entire *MainString* text before returning the right-most n characters/tokens.

Examples: See ucLeft for the concept of counting by tokens, and blocks, etc.

---

ucLen (*MainString* [, *Options*])

Returns the length of a string in terms of characters, tokens, blocks, or expressions.

Example: counting tokens

ucLen(MainString, ucQuote)
`This "is a test" using  (the quick) brown fox, and more.`

`Returns: 13`

Explanation: The counted tokens are:
1. `This`
2. `"is a test"`
3. `Using`

4. (
5. The
6. Quick
7.)
8. Brown
9. Fox
10. ,
11. And
12. More
13. .

Example: Counting characters with ucLen

ucLen(MainString, **ucChar**)
```
This "is a test" using  (the quick) brown fox, and more.

Returns: 56
```

Explanation: This count each character much the same way VB's Len() function would.

Example: counting space-sensitive tokens with ucLen

ucLen(MainString, ucQuote+**ucSpace**)
```
This "is a test" using  (the quick) brown fox, and more.

Returns: 21
```

Explanation: The counting here is similar to the first ucLen example except that each block of whitespace also counts as one unit. The space before "using" for instance counts as one, and the two consecutive spaces after "using" also count as one.

Example: Counting blocks with ucLen

ucLen(MainString, ucQuote+**ucBlock**)
```
This "is a test" using  (the quick) brown fox, and more.

Returns: 10
```

Explanation: The text blocks that are counted are:
1. This
2. "is a test"
3. using
4. (the quick)
5. brown
6. fox
7. ,
8. and

9. more
10. .

---

ucUCase(*MainString* [, Pattern] [, *Options*])

Returns an uppercase version of MainString.

Pattern – ucUCase without the optional arguments behaves the same way as VB's UCase() function. If the optional pattern argument is used, then only text within MainString that matches *Pattern* is changed to uppercase.

Example:

MainString value:

```
This "is a test" using  (the quick) brown "fox", (and) more.
```

ucUCase(MainString, "({text})")

```
Returns:
This "is a test" using  (THE QUICK) brown "fox", (AND) more.
```

ucUCase(MainString, "{Q}{text}{Q}")

```
Returns:
This "IS A TEST" using  (the quick) brown "FOX", (and) more.
```

---

ucUCase(MainString, "", ucSkipOver("({item})")

```
Returns:
THIS "IS A TEST" USING  (the quick) BROWN "FOX", (and) MORE.
```

Explanation: Only text matching the pattern within the given string was changed to uppercase. SkipOver() causes it to match everything except the pattern passed to SkipOver.

---

ucLCase(*MainString* [, Pattern] [, *Options*])

Returns a lower case version of MainString.

See ucUCase.

---

ucMCase(*MainString* [, Pattern] [, *Options*])

Returns a mixed case version of MainString.

See ucUCase.

---

ucMid (*MainString*, *Start* [, *Count*] [, *Options*])

Returns a subset of a string.

Start – Starting position (counted by character, token, block, or expression)
Count – Length (in terms of characters, tokens, blocks, or expressions) of substring that should be returned.

Note: In this version, the second arg is required for it to work; use -1 to make it go to the end. [to be fixed.]

Example:

ucMid(MainString, 6)

```
THIS IS A   TEST, "This is a test", (a  test), a  test, a test.
^    ^  ^   ^    ^ ^
1    2  3   4    5 6
```

Returns

```
"This is a test", (a  test), a  test, a test.
```

ucMid(MainString, 6, 4, ucQuote)

```
THIS IS A   TEST, "This is a test", (a  test), a  test, a test.
^    ^  ^   ^    ^ ^
1    2  3   4    5 6
```

Returns

```
"This is a test", (a

The 4 tokens counted are
   1. "This is a test"
   2. ,
   3. (
   4. a
```

ucMid(MainString, 6, 4, ucChar)

```
THIS IS A   TEST, "This is a test", (a  test), a  test, a test.
^^^^^^
123456
```

---

ucReplace (*MainString*, *Pattern* , *Replacement* [, *Options*])

Replaces all occurrences of one string pattern within MainString with another string.

Pattern – Pattern of text to find within MainString
Replacement – Replacement string for text that matches the Pattern argument. This string can contain pattern variables from the Pattern argument.

Example: ucReplace with default setting

ucReplace(MainString, "is {tokens:2}", "was {tokens}?")
```
THIS IS A   TEST, This is a test, (is a  test).
```

Returns
```
THIS was A   TEST? This was a test?, (was a  test?).
```

ucReplace(MainString, "is {tokens:2}", "was {tokens}?", ucSkipOver("This {etc},"))
```
THIS IS A   TEST, This is a test, (is a  test).
```

Returns
```
THIS IS A   TEST, This is a test, (was a  test?).
```

---

ucTally(*MainString*, *Pattern* [, *Options*])

Counts the number of occurrences of text matching a given pattern within a string. The count can be based on characters, tokens, blocks, or expressions. You can select whether or not to make the result whitespace-sensitive.

Examples:

ucTally(MainString, "is")
```
THIS IS A   TEST, "This is a test", (is a  test).
```

Returns: 3

ucTally(MainString, "is", ucQuote)
```
THIS IS A   TEST, "This is a test", (is a  test).
```

Returns: 2

ucTally(MainString, "is", ucChar)
```
THIS IS A   TEST, "This is a test", (is a  test).
```

Returns: 5

---

ucExtract(*MainString*, *Pattern* [, *Options*])

Extracts characters, tokens, blocks, or expressions up to a given pattern in a string. Similar to PowerBASIC's Extract, except that that one extracts only characters.

Example:

ucExtract(MainString, "<{tag}>{word:2}</{tag}>")
`This is <i>Test<br></i>.  This is <b>another test</b>. Testing 123.`

Returns: `This is <i>Test<br></i>.  This is`

Explanation: Returns text leading up to the first occurrence of text matching a pattern consisting of an HTML tag pair, containing exactly two words (tokens). It skips over Test which has 4 tokens ("Test", "<", "br", and ">"), and continues up until the second tag pair.

---

ucRemain(*MainString*, *Pattern* [, *Options*])

Returns text following a given pattern in a string. Similar to PowerBASIC's Remain, except that that one only deals with characters.

Example:

ucRemain(MainString, "<{tag}>{anything}</{tag}>")
`This is <i>Test<br></i>.  This is <b>another test</b>. Testing 123.`

Returns: `.  This is <b>another test</b>. Testing 123.`

Explanation: This returns text following (but not including) text that matches the pattern.

---

ucRetain(*MainString*, *Pattern* [, *Options*])

Returns a string consisting of only text that matches a given pattern. Similar to PowerBASIC's Retain() function except that that one deals only with characters.

Example:

ucRetain(ucFile("\Test\ucFile\cdcatalog.xml"), "<artist>{etc}</artist>", ucNth(22))

Returns: <artist>Luciano Pavarotti</artist>

Explanation: This example is meant to work with cdcatalog.xml, downloaded from http://www.w3schools.com/xml/cd_catalog.xml . ucFile() is a quick way to obtain all the text from one file. ucNth(22) tells it to retain only the 22th occurrence of the pattern of text between <artist> and </artist>.

---

ucRange(*start, finish, Expression*)

Returns a string consisting of a computed range of values. The variable **x** is the default iterator.

Example:

ucRange(65, 67, "Chr(x)")

Returns: ABC

ucRange (65, 67, "Chr(x)+Chr(x+32)", ucDelim(", ", "{", "}"))

Returns: {Aa, Bb, Cc}

Explanation: The expression in quotes is evaluated with values between 65 and 67. A concatenation of the ASCII Chr() codes for each is returned. In the second example, a upper and lower case pairs are computed, and they are separated with a delimiter defined by ucDelim. The first argument of ucDelim is the separator. The optional second and third arguments are the opening and closing string that will surround the computed range.

---

ucEquals(*String1, String2*)

Compares two strings. This powerful function compares two strings, not character by character (unless you use ucChar), but by pattern. Just as with other library routines, you can toggle space, quote, block, and case sensitivity, and you can also conveniently use ucSkip() to have it disregard certain parts when doing the operation (string comparison in this case).

Example:

ucEquals("THIS is a test", "this is a test") ' True
ucEquals("THIS, is a test?", "This is a test.") ' False
ucEquals("THIS, is a test?", "This is a test.", ucSkip("{ , | ? | . | <{tag}> }")) ' True

ucSetStringDefaults(*Options*)

All library routines allow a last argument that configures properties such as case sensitivity, delimiters, etc. If you will generally be using the same combination of properties, instead of passing it to each call, you can configure it before placing calls. Multiple properties are combined using the "+" operator.

Examples:

```
ucSetStringDefaults(ucSkip("<{tag}>"))
ucEquals("This <b>is</b> a test", "This is a TEST")  ' True
ucEquals("This <b>is</b> a test", "This <i>is</i> a test")  ' True
ucEquals("This <b>is</b> a test", "This is a test!")  ' False

ucSetStringDefaults(ucCase+ucBlock+ucQuote)
ucLen("This (is a) ""test of tests""  ")   ' Returns 3
ucLeft("This (is a) ""test of tests""  ", 2)   ' Returns "This (is a)"
ucEqual("THIS IS A TEST", "This is a test")  ' False

' In uCalc ConsoleCalculator
C:\> uCalc
.> SetStringDefaults(ucChar)
.> Len("This is a test")
. 14
.> Left("This is a test", 3)
. Thi
```

Other functions, which can be used in the Options argument here (or with all the other routines):

ucStart(Position)
ucLimit(Limit)
ucStartAfter(number)
ucStopAfter(number)
ucBetween(a, b)
ucNth(number)
ucPos(x)
ucLength(x)
ucText(x)
ucDelim(separator [, OpeningString, ClosingString])

ucSplit

This splits a text into parts. By default it splits the text into tokens. But you can use some of the same properties described above, or you can split based on a pattern. Once the text is split, you can retrieve the parts with ucStringItem(n), where n is a value between 1 and the number of parts, which is ucStringItemCount().

Example:

```
ucSplit("This (is a) 'test'.")
Print ucStringItemCount() ' This would return 9

Print ucStringItem(1)   '  "This"
Print ucStringItem(2)   '   ("
Print ucStringItem(3)   '  is
Print ucStringItem(4)   '  a
Etc.

ucSplit("This<a> is<b> a<x>test<yz>.", "<{tag}>")
```

Here <{tag}> is the delimiter. The values returned by ucStringItem() would be:

This
Is
A
Test