



The akshar package

Vu Van Dung

Version 0.2 — 2020/09/06

 <https://ctan.org/pkg/akshar>
 <https://github.com/jouleev/akshar>

Abstract

This package provides tools to deal with special characters in a Devanagari string.

Contents

1	Introduction	1
2	User manual	1
2.1	$\LaTeX 2_{\epsilon}$ macros	2
2.2	expl3 functions	3
3	Implementation	3
3.1	Variable declarations	3
3.2	Messages	4
3.3	Utilities	5
3.4	The <code>\akshar_convert:Nn</code> function and its variants	6
3.5	Other internal functions	7
3.6	Front-end $\LaTeX 2_{\epsilon}$ macros	9
	Index	12

1 Introduction

When dealing with processing strings in the Devanagari script, normal \LaTeX commands usually find some difficulties in distinguishing “normal” characters, like क, and “special” characters, for example ् or ी. Let’s consider this example code:

```
1 \ExplSyntaxOn
2 \tl_set:Nn \l_tmpa_tl { की}
3 \tl_count:N \l_tmpa_tl \c_space_token tokens.
4 \ExplSyntaxOff
```

2 tokens.

The output is 2, but the number of characters in it is only one! The reason is quite simple: the compiler treats ी as a normal character, and it shouldn’t do so.

To tackle that, this package provides expl3 functions to “convert” a given string, written in the Devanagari script, to a sequence of token lists. each of these token lists is a “true” Devanagari character. You can now do anything you want with this sequence; and this package does provide some front-end macros for some simple actions on the input string.

2 User manual

Due to the current implementation, all of these macros and functions are not expandable.

2.1 L^AT_EX 2_ε macros

\aksharStrLen \aksharStrLen {(token list)}

Return the number of Devanagari characters in the (token list).

There are 4 characters in नमस्कार.
expl3 returns 7, which is wrong.

```

1 There are \aksharStrLen{ नमस्कार} characters in नमस्कार.\par
2 \ExplSyntaxOn
3 \pkg{expl3}~returns~\tl_count:n { नमस्कार},~which~is~wrong.
4 \ExplSyntaxOff

```

\aksharStrHead \aksharStrHead {(token list)} {(n)}

Return the first character of the token list.

मं 1 \aksharStrHead { मंलीमडमं }

\aksharStrTail \aksharStrTail {(token list)} {(n)}

Return the last character of the token list.

मं 1 \aksharStrTail { लीमडमं }

\aksharStrChar \aksharStrChar {(token list)} {(n)}

Return the *n*-th character of the token list.

3rd character of नमस्कार is स्का.
It is not स.

```

1 3rd character of नमस्कार is \aksharStrChar{ नमस्कार}{3}.\par
2 \ExplSyntaxOn
3 It~is~not~\tl_item:nn { नमस्कार } {3}.
4 \ExplSyntaxOff

```

\aksharStrReplace \aksharStrReplace {(tl 1)} {(tl 2)} {(tl 3)}

\aksharStrReplace*

Replace all occurrences of (tl 2) in (tl 1) with (tl 3), and leaves the modified (tl 1) in the input stream.

The starred variant will replace only the first occurrence of (tl 2), all others are left intact.

expl3 output:
स्कास्कास्काडडस्कांलीस्कास्काड
\aksharStrReplace output:
स्कास्कास्काडडमंलीस्कास्काड

```

1 \ExplSyntaxOn
2 \pkg{expl3} ~ output:\par
3 \tl_set:Nn \l_tmpa_tl { ममडडमंलीमडमं }
4 \tl_replace_all:Nnn \l_tmpa_tl { म } { स्का }
5 \tl_use:N \l_tmpa_tl\par
6 \ExplSyntaxOff
7 \cs{aksharStrReplace} output:\par
8 \aksharStrReplace { ममडडमंलीमडमं } { म } { स्का }

```

expl3 output:
स्कांमडडमंलीमडमं
\aksharStrReplace* output:
ममंस्काडडमंलीमडमं

```

1 \ExplSyntaxOn
2 \pkg{expl3} ~ output:\par
3 \tl_set:Nn \l_tmpa_tl { ममंमडडमंलीमडमं }
4 \tl_replace_once:Nnn \l_tmpa_tl { मम } { स्का }
5 \tl_use:N \l_tmpa_tl\par
6 \ExplSyntaxOff
7 \cs{aksharStrReplace*} output:\par
8 \aksharStrReplace* { ममंमडडमंलीमडमं } { मम } { स्का }

```

\aksharStrRemove \aksharStrRemove {(tl 1)} {(tl 2)}

\aksharStrRemove*

Remove all occurrences of (tl 2) in (tl 1), and leaves the modified (tl 1) in the input stream.

The starred variant will remove only the first occurrence of (tl 2), all others are left intact.

```

1 \ExplSyntaxOn
2 \pkg{expl3} ~ output:\par
3 \tl_set:Nn \l_tmpa_tl { ममडडमंकीममड}
4 \tl_remove_all:Nn \l_tmpa_tl { म}
5 \tl_use:N \l_tmpa_tl\par
6 \ExplSyntaxOff
7 \cs{aksharStrRemove} output:\par
8 \aksharStrRemove { ममडडमंकीममड} { म}

1 \ExplSyntaxOn
2 \pkg{expl3} ~ output:\par
3 \tl_set:Nn \l_tmpa_tl { मममडडमंकीममड}
4 \tl_remove_once:Nn \l_tmpa_tl { मम}
5 \tl_use:N \l_tmpa_tl\par
6 \ExplSyntaxOff
7 \cs{aksharStrRemove*} output:\par
8 \aksharStrRemove* { मममडडमंकीममड} { मम}

```

2.2 expl3 functions

This section assumes that you have a basic knowledge in \LaTeX programming. All macros in 2.1 directly depend on the following function, so it is much more powerful than all features we have described above.

```
\akshar_convert:Nn
\akshar_convert:(cn|Nx|cx)
```

```
\akshar convert:Nn <seq var> {<token list>}
```

This function converts `<token list>` to a sequence of characters, that sequence is stored in `<seq var>`.

```

1 \ExplSyntaxOn
2 \akshar_convert:Nn \l_tmpa_seq { नमस्कार }
न, म, स्का, and र
3 \seq_use:Nnnn \l_tmpa_seq { ~and~ } { ,~ } { ,~and~ }
4 \ExplSyntaxOff

```

3 Implementation

```
1 <@@=akshar>
2 <*package>
```

Declare the package. By loading fontspec, xparse, and in turn, expl3, are also loaded.

```
3 \RequirePackage{fontspec}
4 \ProvidesExplPackage {\aksharPackageName}
5   {\aksharPackageDate} {\aksharPackageVersion} {\aksharPackageDescription}
```

3.1 Variable declarations

```
\c__akshar_joining_tl
\c__akshar_diacritics_tl
```

These variables store the special characters we need to take into account:

- \c__akshar_joining_tl is the “connecting” character ँ.
- \c__akshar_diacritics_tl is the list of all diacritics: ा, ि, ೀ, ು, ೂ, ೃ, ೄ, ೅, ೆ, ೇ, ೈ, ೉, ೊ, ೋ, ೌ, ್, ೎, ೏, ೐, ೑, ೒, ೓, ೔, ೕ, ೖ, ೗, ೘, ೙, ೚, ೛, ೜, ೝ, ೞ, ೟, ೠ, ೡ, ೢ, ೣ, ೤, ೥, ೦, ೧, ೨, ೩, ೪, ೫, ೬, ೭, ೮, ೯, ೰, ೱ, ೲ, ೳ, ೴, ೵, ೶, ೷, ೸, ೹, ೺, ೻, ೼, ೽, ೾, ೿, ೼, ೽, ೾, ೿.

```

6 \tl_const:Nn \c__akshar_joining_tl { ീ }
7 \tl_const:Nn \c__akshar_diacritics_tl
8 {
9   ി, ി, ീ, ു, ൂ, ൃ, ൄ, ൅, െ, േ, ൈ, ൉, ൊ, ോ, ൌ, ്, ൎ, ൏, ൐, ൑, ൒, ൓, ൔ, ൕ,
10  ൖ, ൗ, ൘, ൙, ൚, ൛, ൜, ൝, ൞, ൟ, ൠ, ൡ, ൢ, ൣ, ൤, ൥, ൦, ൧, ൨, ൩, ൪, ൫, ൬, ൭, ൮,
11  ൯, ൰, ൱, ൲, ൳, ൴, ൵, ൶, ൷, ൸, ൹, ൺ, ൻ, ർ, ൽ, ൾ, ൿ, ൽ, ൾ, ൿ, ൽ, ൾ, ൿ, ൽ, ൾ, ൿ,

```

```

12   उ, क, न, प, र, ि, ः,
13 }

```

(End definition for `\c__akshar_joining_tl` and `\c__akshar_diacritics_tl`.)

`\l__akshar_prev_joining_bool` When we get to a normal character, we need to know whether it is joined, i.e. whether the previous character is the joining character. This boolean variable takes care of that.

```

14 \bool_new:N \l__akshar_prev_joining_bool

```

(End definition for `\l__akshar_prev_joining_bool`.)

`\l__akshar_char_seq` This local sequence stores the output of the converter.

```

15 \seq_new:N \l__akshar_char_seq

```

(End definition for `\l__akshar_char_seq`.)

`\c__akshar_str_g_tl` Some self-descriptive constant variables.

`\c__akshar_str_seq_tl`

`\c__akshar_str_comma_tl`

```

16 \tl_const:Nx \c__akshar_str_g_tl { \tl_to_str:n {g} }

```

```

17 \tl_const:Nx \c__akshar_str_seq_tl { \tl_to_str:n {seq} }

```

```

18 \tl_const:Nx \c__akshar_str_comma_tl { \tl_to_str:n {,} }

```

(End definition for `\c__akshar_str_g_tl`, `\c__akshar_str_seq_tl`, and `\c__akshar_str_comma_tl`.)

`\l__akshar_tmpa_tl` Some temporary variables.

`\l__akshar_tmpb_tl`

`\l__akshar_tmpa_seq`

`\l__akshar_tmpb_seq`

`\l__akshar_tmpe_seq`

`\l__akshar_tmpe_seq`

`\l__akshar_tmpe_seq`

`\l__akshar_tmpe_int`

`\l__akshar_tmpe_int`

```

19 \tl_new:N \l__akshar_tmpa_tl

```

```

20 \tl_new:N \l__akshar_tmpb_tl

```

```

21 \seq_new:N \l__akshar_tmpa_seq

```

```

22 \seq_new:N \l__akshar_tmpb_seq

```

```

23 \seq_new:N \l__akshar_tmpe_seq

```

```

24 \seq_new:N \l__akshar_tmpe_seq

```

```

25 \seq_new:N \l__akshar_tmpe_seq

```

```

26 \int_new:N \l__akshar_tmpe_int

```

```

27 \int_new:N \l__akshar_tmpe_int

```

(End definition for `\l__akshar_tmpe_tl` and others.)

3.2 Messages

In `\akshar_convert:Nn` and friends, the argument needs to be a sequence variable. There will be an error if it isn't.

```

28 \msg_new:nnnn { akshar } { err_not_a_sequence_variable }
29 { #1 ~ is ~ not ~ a ~ valid ~ LaTeX3 ~ sequence ~ variable. }
30 {
31   You ~ have ~ requested ~ me ~ to ~ assign ~ some ~ value ~ to ~
32   the ~ control ~ sequence ~ #1, ~ but ~ it ~ is ~ not ~ a ~ valid ~
33   sequence ~ variable. ~ Read ~ the ~ documentation ~ of ~ expl3 ~
34   for ~ more ~ information. ~ Proceed ~ and ~ I ~ will ~ pretend ~
35   that ~ #1 ~ is ~ a ~ local ~ sequence ~ variable ~ (beware ~ that ~
36   unexpected ~ behaviours ~ may ~ occur).
37 }

```

In `\aksharStrChar`, we need to guard against accessing an ‘out-of-bound’ character (like trying to get the 8th character in a 5-character string.)

```

38 \msg_new:nnnn { akshar } { err_character_out_of_bound }
39 { Character ~ index ~ out ~ of ~ bound. }
40 {
41   You ~ are ~ trying ~ to ~ get ~ the ~ #2 ~ character ~ of ~ the ~
42   string ~ #1. ~ However ~ that ~ character ~ doesn't ~ exist. ~
43   Make ~ sure ~ that ~ you ~ use ~ a ~ number ~ between ~ and ~ not ~
44   including ~ 0 ~ and ~ #3, ~ so ~ that ~ I ~ can ~ return ~ a ~
45   good ~ output. ~ Proceed ~ and ~ I ~ will ~ return ~
46   \token_to_str:N \scan_stop:.
47 }

```

In \aksharStrHead and \aksharStrTail, the string must not be blank.

```

48 \msg_new:nnnn { akshar } { err_string_empty }
49 { The ~ input ~ string ~ is ~ empty. }
50 {
51   To ~ get ~ the ~ #1 ~ character ~ of ~ a ~ string, ~ that ~ string ~
52   must ~ not ~ be ~ empty, ~ but ~ the ~ input ~ string ~ is ~ empty.
53   Make ~ sure ~ the ~ string ~ contains ~ something, ~ or ~ proceed ~
54   and ~ I ~ will ~ use ~ \token_to_str:N \scan_stop:.
55 }

```

3.3 Utilities

\tl_if_in:NoTF When we get to a character which is not the joining one, we need to know if it is a diacritic. The current character is stored in a variable, so an expanded variant is needed. We only need it to expand only once.

```

56 \prg_generate_conditional_variant:Nnn \tl_if_in:Nn { No } { TF }

```

(End definition for \tl_if_in:NoTF.)

\seq_set_split:Nxx A variant we will need in __akshar_var_if_global.

```

57 \cs_generate_variant:Nn \seq_set_split:Nnn { Nxx }

```

(End definition for \seq_set_split:Nxx.)

\msg_error:nnx Some variants of l3msg functions that we will need when issuing error messages.
\msg_error:nnnxx

```

58 \cs_generate_variant:Nn \msg_error:nnn { nnx }
59 \cs_generate_variant:Nn \msg_error:nnnnn { nnnxx }

```

(End definition for \msg_error:nnx and \msg_error:nnnxx.)

__akshar_tl_if_in_ncomma:NNTF This conditional is essentially \tl_if_in:Nn, but if #2 is a comma this conditional always return false.

```

60 \prg_new_conditional:Npnn \__akshar_tl_if_in_ncomma:NN #1 #2 { T, F, TF }
61 {
62   \tl_if_eq:NNTF \c__akshar_str_comma_tl #2 { \prg_return_false: }
63   { \tl_if_in:NoTF #1 {#2} { \prg_return_true: } { \prg_return_false: } }
64 }

```

(End definition for __akshar_tl_if_in_ncomma:NNTF.)

__akshar_var_if_global:NTF This conditional checks if #1 is a global sequence variable or not. In other words, it returns true iff #1 is a control sequence in the format \g_<name>_seq. If it is not a sequence variable, this function will (TODO) issue an error message.

```

65 \prg_new_conditional:Npnn \__akshar_var_if_global:N #1 { T, F, TF }
66 {
67   \bool_if:nTF
68   { \exp_last_unbraced:Nf \use_iii:nnn { \cs_split_function:N #1 } }
69   {
70     \msg_error:nnx { akshar } { err_not_a_sequence_variable }
71     { \token_to_str:N #1 }
72     \prg_return_false:
73   }
74   {
75     \seq_set_split:Nxx \l__akshar_tmpb_seq { \token_to_str:N _ }
76     { \exp_last_unbraced:Nf \use_i:nnn { \cs_split_function:N #1 } }
77     \seq_get_left:NN \l__akshar_tmpb_seq \l__akshar_tmpa_tl
78     \seq_get_right:NN \l__akshar_tmpb_seq \l__akshar_tmpb_tl
79     \tl_if_eq:NNTF \c__akshar_str_seq_tl \l__akshar_tmpb_tl
80     {
81       \tl_if_eq:NNTF \c__akshar_str_g_tl \l__akshar_tmpa_tl
82       { \prg_return_true: } { \prg_return_false: }

```

```

83     }
84     {
85         \msg_error:nxx { akshar } { err_not_a_sequence_variable }
86         { \token_to_str:N #1 }
87         \prg_return_false:
88     }
89 }
90 }

```

(End definition for `__akshar_var_if_global:NTF.`)

`__akshar_int_append_ordinal:n` Append st, nd, rd or th to interger #1. Will be needed in error messages.

```

91 \cs_new:Npn \__akshar_int_append_ordinal:n #1
92 {
93     #1
94     \int_case:nnF { #1 }
95     {
96         { 11 } { th }
97         { 12 } { th }
98         { 13 } { th }
99         { -11 } { th }
100        { -12 } { th }
101        { -13 } { th }
102    }
103    {
104        \int_compare:nNnTF { #1 } > { -1 }
105        {
106            \int_case:nnF { #1 - 10 * ( #1 / 10 ) }
107            {
108                { 1 } { st }
109                { 2 } { nd }
110                { 3 } { rd }
111            } { th }
112        }
113        {
114            \int_case:nnF { ( - #1 ) - 10 * ( ( - #1 ) / 10 ) }
115            {
116                { 1 } { st }
117                { 2 } { nd }
118                { 3 } { rd }
119            } { th }
120        }
121    }
122 }

```

(End definition for `__akshar_int_append_ordinal:n.`)

3.4 The `\akshar_convert:Nn` function and its variants

`\akshar_convert:Nn` This converts #2 to a sequence of true Devanagari characters. The sequence is set to #1, which should be a sequence variable.

```

\akshar_convert:cn
\akshar_convert:Nx
\akshar_convert:cx
123 \cs_new:Npn \akshar_convert:Nn #1 #2
124 {

```

Clear anything stored in advance. We don't want different calls of the function to conflict with each other.

```

125     \seq_clear:N \l__akshar_char_seq
126     \bool_set_false:N \l__akshar_prev_joining_bool

```

Loop through every token of the input.

```

127     \tl_map_variable:NNn {#2} \l__akshar_map_tl
128     {
129         \__akshar_tl_if_in_ncomma:NNTF
130         \c__akshar_diacritics_tl \l__akshar_map_tl
131     }

```

It is a diacritic. We append the current diacritic to the last item of the sequence instead of pushing the diacritic to a new sequence item.

```

132         \seq_pop_right:NN \l__akshar_char_seq \l__akshar_tmpa_tl
133         \seq_put_right:Nx \l__akshar_char_seq
134         { \l__akshar_tmpa_tl \l__akshar_map_tl }
135     }
136     {
137         \tl_if_eq:NNTF \l__akshar_map_tl \c__akshar_joining_tl
138         {

```

In this case, the character is the joining character, ٠. What we do is similar to the above case, but `\l__akshar_prev_joining_bool` is set to true so that the next character is also appended to this item.

```

139         \seq_pop_right:NN \l__akshar_char_seq \l__akshar_tmpa_tl
140         \seq_put_right:Nx \l__akshar_char_seq
141         { \l__akshar_tmpa_tl \l__akshar_map_tl }
142         \bool_set_true:N \l__akshar_prev_joining_bool
143     }
144     {

```

Now the character is normal. We see if we can push to a new item or not. It depends on the boolean variable.

```

145         \bool_if:NNTF \l__akshar_prev_joining_bool
146         {
147             \seq_pop_right:NN \l__akshar_char_seq \l__akshar_tmpa_tl
148             \seq_put_right:Nx \l__akshar_char_seq
149             { \l__akshar_tmpa_tl \l__akshar_map_tl }
150             \bool_set_false:N \l__akshar_prev_joining_bool
151         }
152         {
153             \seq_put_right:Nx
154             \l__akshar_char_seq { \l__akshar_map_tl }
155         }
156     }
157 }
158 }
```

Set #1 to `\l__akshar_char_seq`. The package automatically determines whether the variable is a global one or a local one.

```

159     \__akshar_var_if_global:NNTF #1
160     { \seq_gset_eq:NN #1 \l__akshar_char_seq }
161     { \seq_set_eq:NN #1 \l__akshar_char_seq }
162 }
```

Generate variants that might be helpful for some.

```

163 \cs_generate_variant:Nn \akshar_convert:Nn { cn, Nx, cx }
```

(End definition for `\akshar_convert:Nn`. This function is documented on page 3.)

3.5 Other internal functions

`__akshar_seq_push_seq:NN` Append sequence #1 to the end of sequence #2. A simple loop will do.

```

164 \cs_new:Npn \__akshar_seq_push_seq:NN #1 #2
165 { \seq_map_inline:Nn #2 { \seq_put_right:Nn #1 { ##1 } } }
```

(End definition for `__akshar_seq_push_seq:NN`.)

`__akshar_replace:NnnnN` If #5 is `\c_false_bool`, this function replaces all occurrences of #3 in #2 by #4 and stores the output sequence to #1. If #5 is `\c_true_bool`, the replacement only happens once.

The algorithm used in this function: We will use `\l__akshar_tmpa_int` to store the “current position” in the sequence of #3. At first it is set to 1.

We will store any subsequence of #2 that may match #3 to a temporary sequence. If it doesn't match, we push this temporary sequence to the output, but if it matches, #4 is pushed instead.

We loop over #2. For each of these loops, we need to make sure the `\l__akshar_tmpa_int`-th item must indeed appear in #3. So we need to compare that with the length of #3.

- If now `\l__akshar_tmpa_int` is greater than the length of #3, the whole of #3 has been matched somewhere, so we reinitialize the integer to 1 and push #4 to the output.

Note that it is possible that the current character might be the start of another match, so we have to compare it to the first character of #3. If they are not the same, we may now push the current mapping character to the output and proceed; otherwise the current character is pushed to the temporary variable.

- Otherwise, we compare the current loop character of #2 with the `\l__akshar_tmpa_int`-th character of #3.
 - If they are the same, we still have a chance that it will match, so we increase the “iterator” `\l__akshar_tmpa_int` by 1 and push the current mapping character to the temporary sequence.
 - If they are the same, the temporary sequence won't match. Let's push that sequence to the output and set the iterator back to 1. Note that now the iterator has changed. Who knows whether the current character may start a match? Let's compare it to the first character of #3, and do as in the case of `\l__akshar_tmpa_int` is greater than the length of #3.

The complexity of this algorithm is $O(m \max(n, p))$, where m, n, p are the lengths of the sequences created from #2, #3 and #4. As #3 and #4 are generally short strings, this is (almost) linear to the length of the original sequence #2.

```

166 \cs_new:Npn \l__akshar_replace:NnnN #1 #2 #3 #4 #5
167 {
168   \akshar_convert:Nn \l__akshar_tmpe_seq {#2}
169   \akshar_convert:Nn \l__akshar_tmpe_seq {#3}
170   \akshar_convert:Nn \l__akshar_tmpe_seq {#4}
171   \seq_clear:N \l__akshar_tmpe_seq
172   \seq_clear:N \l__akshar_tmpe_seq
173   \int_set:Nn \l__akshar_tmpe_int { 1 }
174   \int_set:Nn \l__akshar_tmpe_int { 0 }
175   \seq_map_variable:Nnn \l__akshar_tmpe_seq \l__akshar_map_tl
176   {
177     \int_compare:nNnTF { \l__akshar_tmpe_int } > { 0 }
178     { \seq_put_right:NV \l__akshar_tmpe_seq \l__akshar_map_tl }
179     {
180       \int_compare:nNnTF
181       { \l__akshar_tmpe_int } = { 1 + \seq_count:N \l__akshar_tmpe_seq }
182       {
183         \bool_if:NT {#5}
184         { \int_incr:N \l__akshar_tmpe_int }
185         \seq_clear:N \l__akshar_tmpe_seq
186         \__akshar_seq_push_seq:NN
187         \l__akshar_tmpe_seq \l__akshar_tmpe_seq
188         \int_set:Nn \l__akshar_tmpe_int { 1 }
189         \tl_set:Nx \l__akshar_tmpe_tl
190         { \seq_item:Nn \l__akshar_tmpe_seq { 1 } }
191         \tl_if_eq:NNTF \l__akshar_map_tl \l__akshar_tmpe_tl
192         {
193           \int_incr:N \l__akshar_tmpe_int
194           \seq_put_right:NV \l__akshar_tmpe_seq \l__akshar_map_tl
195         }
196         {
197           \seq_put_right:NV \l__akshar_tmpe_seq \l__akshar_map_tl
198         }
199       }

```



```

200 {
201   \tl_set:Nx \l__akshar_tmpa_tl
202   {
203     \seq_item:Nn \l__akshar_tmpd_seq { \l__akshar_tmpa_int }
204   }
205   \tl_if_eq:NNTF \l__akshar_map_tl \l__akshar_tmpa_tl
206   {
207     \int_incr:N \l__akshar_tmpa_int
208     \seq_put_right:NV \l__akshar_tmpb_seq \l__akshar_map_tl
209   }
210   {
211     \int_set:Nn \l__akshar_tmpa_int { 1 }
212     \__akshar_seq_push_seq:NN
213       \l__akshar_tmpa_seq \l__akshar_tmpb_seq
214     \seq_clear:N \l__akshar_tmpb_seq
215     \tl_set:Nx \l__akshar_tmpa_tl
216       { \seq_item:Nn \l__akshar_tmpd_seq { 1 } }
217     \tl_if_eq:NNTF \l__akshar_map_tl \l__akshar_tmpa_tl
218     {
219       \int_incr:N \l__akshar_tmpa_int
220       \seq_put_right:NV
221         \l__akshar_tmpb_seq \l__akshar_map_tl
222     }
223     {
224       \seq_put_right:NV
225         \l__akshar_tmpa_seq \l__akshar_map_tl
226     }
227   }
228 }
229 }
230 }
231 \__akshar_seq_push_seq:NN \l__akshar_tmpa_seq \l__akshar_tmpb_seq
232 \__akshar_var_if_global:NTF #1
233 { \seq_gset_eq:NN #1 \l__akshar_tmpa_seq }
234 { \seq_set_eq:NN #1 \l__akshar_tmpa_seq }
235 }

```

(End definition for `__akshar_replace:NnnnN`.)

3.6 Front-end $\text{\LaTeX}2_{\epsilon}$ macros

`\aksharStrLen` Expands to the length of the string.

```

236 \NewDocumentCommand \aksharStrLen {m}
237 {
238   \akshar_convert:Nn \l__akshar_tmpa_seq {#1}
239   \seq_count:N \l__akshar_tmpa_seq
240 }

```

(End definition for `\aksharStrLen`. This function is documented on page 2.)

`\aksharStrChar` Returns the n -th character of the string.

```

241 \NewDocumentCommand \aksharStrChar {mm}
242 {
243   \akshar_convert:Nn \l__akshar_tmpa_seq {#1}
244   \bool_if:nTF
245     {
246       \int_compare_p:nNn {#2} > {0} &&
247       \int_compare_p:nNn {#2} < {1 + \seq_count:N \l__akshar_tmpa_seq}
248     }
249     { \seq_item:Nn \l__akshar_tmpa_seq { #2 } }
250     {
251       \msg_error:nnnxx { akshar } { err_character_out_of_bound }
252       { #1 } { \__akshar_int_append_ordinal:n { #2 } }
253       { \int_eval:n { 1 + \seq_count:N \l__akshar_tmpa_seq } }
254       \scan_stop:
255     }
256 }

```

(End definition for `\aksharStrChar`. This function is documented on page 2.)

`\aksharStrHead` Return the first character of the string.

```
257 \NewDocumentCommand \aksharStrHead {m}
258 {
259   \akshar_convert:Nn \l__akshar_tmpa_seq {#1}
260   \int_compare:nNnTF { \seq_count:N \l__akshar_tmpa_seq } = {0}
261   {
262     \msg_error:nnn { akshar } { err_character_out_of_bound }
263     { first }
264     \scan_stop:
265   }
266   { \seq_item:Nn \l__akshar_tmpa_seq { 1 } }
267 }
```

(End definition for `\aksharStrHead`. This function is documented on page 2.)

`\aksharStrTail` Return the last character of the string.

```
268 \NewDocumentCommand \aksharStrTail {m}
269 {
270   \akshar_convert:Nn \l__akshar_tmpa_seq {#1}
271   \int_compare:nNnTF { \seq_count:N \l__akshar_tmpa_seq } = {0}
272   {
273     \msg_error:nnn { akshar } { err_character_out_of_bound }
274     { last }
275     \scan_stop:
276   }
277   { \seq_item:Nn \l__akshar_tmpa_seq { \seq_count:N \l__akshar_tmpa_seq } }
278 }
```

(End definition for `\aksharStrTail`. This function is documented on page 2.)

`\aksharStrReplace` Replace occurrences of #3 of a string #2 with another string #4.
`\aksharStrReplace*`

```
279 \NewDocumentCommand \aksharStrReplace {smmm}
280 {
281   \IfBooleanTF {#1}
282   {
283     \__akshar_replace:NnnnN \l__akshar_tmpa_seq
284     {#2} {#3} {#4} \c_true_bool
285   }
286   {
287     \__akshar_replace:NnnnN \l__akshar_tmpa_seq
288     {#2} {#3} {#4} \c_false_bool
289   }
290   \seq_use:Nn \l__akshar_tmpa_seq {}
291 }
```

(End definition for `\aksharStrReplace` and `\aksharStrReplace*`. These functions are documented on page 2.)

`\aksharStrRemove` Remove occurrences of #3 in #2. This is just a special case of `\aksharStrReplace`.
`\aksharStrRemove*`

```
292 \NewDocumentCommand \aksharStrRemove {smm}
293 {
294   \IfBooleanTF {#1}
295   {
296     \__akshar_replace:NnnnN \l__akshar_tmpa_seq
297     {#2} {#3} {} \c_true_bool
298   }
299   {
300     \__akshar_replace:NnnnN \l__akshar_tmpa_seq
301     {#2} {#3} {} \c_false_bool
302   }
303   \seq_use:Nn \l__akshar_tmpa_seq {}
304 }
```

(End definition for `\aksharStrRemove` and `\aksharStrRemove*`. These functions are documented on page 2.)

305 `\endpackage`

Index

Underlined page numbers point to the definition, all others indicate the places where it is used or described.

A		B	
akshar commands:		bool commands:	
\akshar_convert:Nn 1, 3, 4, 6, <u>123</u> , 168, 169, 170, 238, 243, 259, 270		\bool_if:NTF 145, 183	
akshar internal commands:		\bool_if:nTF 67, 244	
\l__akshar_char_seq		\bool_new:N 14	
..... 7, <u>15</u> , 125, 132, 133, 139, 140, 147, 148, 154, 160, 161		\bool_set_false:N 126, 150	
\c__akshar_diacritics_tl . 3, <u>6</u> , 130		\bool_set_true:N 142	
__akshar_int_append_ordinal:n <u>91</u> , 252		\c_false_bool 7, 288, 301	
\c__akshar_joining_tl 3, <u>6</u> , 137		\c_true_bool 7, 284, 297	
\l__akshar_map_tl		C	
..... 127, 130, 134, 137, 141, 149, 154, 175, 178, 191, 194, 197, 205, 208, 217, 221, 225		cs commands:	
\l__akshar_prev_joining_bool ..		\cs_generate_variant:Nn	
..... 7, <u>14</u> , 126, 142, 145, 150	 57, 58, 59, 163	
__akshar_replace:NnnN		\cs_new:Npn <u>91</u> , 123, 164, 166	
..... <u>166</u> , 283, 287, 296, 300		\cs_split_function:N 68, 76	
__akshar_seq_push_seq:NN		E	
..... <u>164</u> , 186, 212, 231		exp commands:	
\c__akshar_str_comma_tl <u>16</u> , 62		\exp_last_unbraced:Nf 68, 76	
\c__akshar_str_g_tl <u>16</u> , 81		I	
\c__akshar_str_seq_tl <u>16</u> , 79		\IfBooleanTF 281, 294	
__akshar_tl_if_in_ncomma:NNTF <u>60</u> , 129		int commands:	
\l__akshar_tmpa_int 7, 8, <u>19</u> , 173, 181, 188, 193, 203, 207, 211, 219		\int_case:nnTF 94, 106, 114	
\l__akshar_tmpa_seq <u>19</u> , 171, 187, 197, 213, 225, 231, 233, 234, 238, 239, 243, 247, 249, 253, 259, 260, 266, 270, 271, 277, 283, 287, 290, 296, 300, 303		\int_compare:nNnTF	
\l__akshar_tmpa_tl <u>19</u> , 77, 81, 132, 134, 139, 141, 147, 149, 189, 191, 201, 205, 215, 217	 104, 177, 180, 260, 271	
\l__akshar_tmpb_int		\int_compare_p:nNn 246, 247	
..... <u>19</u> , 174, 177, 184		\int_eval:n 253	
\l__akshar_tmpb_seq		\int_incr:N 184, 193, 207, 219	
..... <u>19</u> , 75, 77, 78, 172, 178, 185, 194, 208, 213, 214, 221, 231		\int_new:N 26, 27	
\l__akshar_tmpb_tl <u>19</u> , 78, 79		\int_set:Nn 173, 174, 188, 211	
\l__akshar_tmpe_seq ... <u>19</u> , 168, 175		M	
\l__akshar_tmpe_seq <u>19</u> , 169, 181, 190, 203, 216		msg commands:	
\l__akshar_tmpe_seq ... <u>19</u> , 170, 187		\msg_error:nnn	
__akshar_var_if_global 5	 <u>58</u> , 58, 70, 85, 262, 273	
__akshar_var_if_global:NTF ...		\msg_error:nnnn <u>58</u> , 59, 251	
..... <u>65</u> , 159, 232		\msg_new:nnnn 28, 38, 48	
\aksharPackageDate 5		N	
\aksharPackageDescription 5		\NewDocumentCommand	
\aksharPackageName 4	 236, 241, 257, 268, 279, 292	
\aksharPackageVersion 5		P	
\aksharStrChar 2, 4, <u>241</u>		prg commands:	
\aksharStrHead 2, 5, <u>257</u>		\prg_generate_conditional_ variant:Nnn 56	
\aksharStrLen 2, <u>236</u>		\prg_new_conditional:Npnn .. 60, 65	
\aksharStrRemove 2, 3, <u>292</u>		\prg_return_false: 62, 63, 72, 82, 87	
\aksharStrRemove* 2, 3, <u>292</u>		\prg_return_true: 63, 82	
\aksharStrReplace 2, 10, <u>279</u>		\ProvidesExplPackage 4	
\aksharStrReplace* 2, <u>279</u>		R	
\aksharStrTail 2, 5, <u>268</u>		\RequirePackage 3	
		S	
		scan commands:	
		\scan_stop: .. 46, 54, 254, 264, 275	
		seq commands:	
		\seq_clear:N 125, 171, 172, 185, 214	
		\seq_count:N	
		181, 239, 247, 253, 260, 271, 277	
		\seq_get_left:NN 77	
		\seq_get_right:NN 78	

```

\seq_gset_eq:Nn ..... 160, 233
\seq_item:Nn .....
.... 190, 203, 216, 249, 266, 277
\seq_map_inline:Nn ..... 165
\seq_map_variable:NNn ..... 175
\seq_new:N .. 15, 21, 22, 23, 24, 25
\seq_pop_right:NN .... 132, 139, 147
\seq_put_right:Nn .....
..... 133, 140, 148, 153,
165, 178, 194, 197, 208, 220, 224
\seq_set_eq:Nn ..... 161, 234
\seq_set_split:Nnn ..... 57, 57, 75
\seq_use:Nn ..... 290, 303

T

tl commands:
\tl_const:Nn ..... 6, 7, 16, 17, 18

\tl_if_eq:NNTF .....
... 62, 79, 81, 137, 191, 205, 217
\tl_if_in:Nn ..... 5, 56
\tl_if_in:NnTF ..... 56, 63
\tl_map_variable:NNn ..... 127
\tl_new:N ..... 19, 20
\tl_set:Nn ..... 189, 201, 215
\tl_to_str:n ..... 16, 17, 18

token commands:
\token_to_str:N . 46, 54, 71, 75, 86

U

use commands:
\use_i:nnn ..... 76
\use_iii:nnn ..... 68

```