

The akshar package

Vu Van Dung

Version 0.1 — 2020/05/17

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	1
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	10

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

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 `\aksharStrHead { मंळीमड}`

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

Return the last character of the token list.

मं `\aksharStrTail { ळीमडमं}`

`\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 नमस्कारis \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:

स्कास्कास्काडडस्कांळीस्कास्काड

`\aksharStrReplace` output:

स्कास्कास्काडडमंळीमड

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

expl3 output:

स्कांममडडमंळीमड

`\aksharStrReplace*` output:

ममस्काडडमंळीमड

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


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

```

44 \msg_new:nnnn { akshar } { err_string_empty }
45 { The ~ input ~ string ~ is ~ empty. }
46 {
47   To ~ get ~ the ~ #1 ~ character ~ of ~ a ~ string, ~ that ~ string ~
48   must ~ not ~ be ~ empty, ~ but ~ the ~ input ~ string ~ is ~ empty.
49   Make ~ sure ~ the ~ string ~ contains ~ something, ~ or ~ proceed ~
50   and ~ I ~ will ~ use ~ \token_to_str:N \scan_stop:.
51 }

```

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.

```

52 \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`.

```

53 \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`

```

54 \cs_generate_variant:Nn \msg_error:nnn { nnx }
55 \cs_generate_variant:Nn \msg_error:nnnxx { nnnxx }

```

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

`__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`.
`\c__akshar_str_g_tl` If it is not a sequence variable, this function will (TODO) issue an error message.
`\c__akshar_str_seq_tl`

```

56 \tl_const:Nx \c__akshar_str_g_tl { \tl_to_str:n {g} }
57 \tl_const:Nx \c__akshar_str_seq_tl { \tl_to_str:n {seq} }
58 \prg_new_conditional:Npnn \__akshar_var_if_global:N #1 { T, F, TF }
59 {
60   \bool_if:nTF
61   { \exp_last_unbraced:Nf \use_iii:nnn { \cs_split_function:N #1 } }
62   {
63     \msg_error:nnx { akshar } { err_not_a_sequence_variable }
64     { \token_to_str:N #1 }
65     \prg_return_false:
66   }
67   {
68     \seq_set_split:Nxx \l__akshar_tmpb_seq { \token_to_str:N _ }
69     { \exp_last_unbraced:Nf \use_i:nnn { \cs_split_function:N #1 } }
70     \seq_get_left:NN \l__akshar_tmpb_seq \l__akshar_tmpa_tl
71     \seq_get_right:NN \l__akshar_tmpb_seq \l__akshar_tmpb_tl
72     \tl_if_eq:NNTF \c__akshar_str_seq_tl \l__akshar_tmpb_tl
73     {
74       \tl_if_eq:NNTF \c__akshar_str_g_tl \l__akshar_tmpa_tl
75       { \prg_return_true: } { \prg_return_false: }
76     }
77     {
78       \msg_error:nnx { akshar } { err_not_a_sequence_variable }
79       { \token_to_str:N #1 }
80       \prg_return_false:
81     }
82   }
83 }

```

(End definition for `__akshar_var_if_global:NTF`, `\c__akshar_str_g_tl`, and `\c__akshar_str_seq_tl`.)

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

```

84 \cs_new:Npn \__akshar_int_append_ordinal:n #1
85 {
86   #1
87   \int_case:nnF { #1 }
88   {
89     { 11 } { th }
90     { 12 } { th }
91     { 13 } { th }
92     { -11 } { th }
93     { -12 } { th }
94     { -13 } { th }
95   }
96   {
97     \int_compare:nNnTF { #1 } > { -1 }
98     {
99       \int_case:nnF { #1 - 10 * ( #1 / 10 ) }
100      {
101        { 1 } { st }
102        { 2 } { nd }
103        { 3 } { rd }
104      } { th }
105    }
106    {
107      \int_case:nnF { (- #1) - 10 * ((- #1) / 10) }
108      {
109        { 1 } { st }
110        { 2 } { nd }
111        { 3 } { rd }
112      } { th }
113    }
114  }
115 }

```

(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. The assignment is local.

`\akshar_convert:cn`
`\akshar_convert:Nx`
`\akshar_convert:cx`

```

116 \cs_new:Npn \akshar_convert:Nn #1 #2
117 {

```

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

```

118   \seq_clear:N \l__akshar_char_seq
119   \bool_set_false:N \l__akshar_prev_joining_bool

```

Loop through every token of the input.

```

120   \tl_map_variable:NNn {#2} \l__akshar_map_tl
121   {
122     \tl_if_in:NoTF \c__akshar_diacritics_tl {\l__akshar_map_tl}
123     {

```

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.

```

124       \seq_pop_right:NN \l__akshar_char_seq \l__akshar_tmpa_tl
125       \seq_put_right:Nx \l__akshar_char_seq
126       { \l__akshar_tmpa_tl \l__akshar_map_tl }
127     }
128   {
129     \tl_if_eq:NNTF \l__akshar_map_tl \c__akshar_joining_tl
130     {

```

In this case, the character is the joining character, ॐ. 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.

```

131         \seq_pop_right:NN \l__akshar_char_seq \l__akshar_tmpa_tl
132         \seq_put_right:Nx \l__akshar_char_seq
133         { \l__akshar_tmpa_tl \l__akshar_map_tl }
134         \bool_set_true:N \l__akshar_prev_joining_bool
135     }
136     {

```

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

```

137         \bool_if:NTF \l__akshar_prev_joining_bool
138         {
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_false:N \l__akshar_prev_joining_bool
143         }
144         {
145             \seq_put_right:Nx
146             \l__akshar_char_seq { \l__akshar_map_tl }
147         }
148     }
149 }
150

```

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

```

151     \__akshar_var_if_global:NTF #1
152     { \seq_gset_eq:NN #1 \l__akshar_char_seq }
153     { \seq_set_eq:NN #1 \l__akshar_char_seq }
154 }

```

Generate variants that might be helpful for some.

```

155 \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.

```

156 \cs_new:Npn \__akshar_seq_push_seq:NN #1 #2
157 { \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.

```

158 \cs_new:Npn \__akshar_replace:NnnN #1 #2 #3 #4 #5
159 {
160   \akshar_convert:Nn \l__akshar_tmpe_seq {#2}
161   \akshar_convert:Nn \l__akshar_tmpe_seq {#3}
162   \akshar_convert:Nn \l__akshar_tmpe_seq {#4}
163   \seq_clear:N \l__akshar_tmpe_seq
164   \seq_clear:N \l__akshar_tmpe_seq
165   \int_set:Nn \l__akshar_tmpe_int { 1 }
166   \int_set:Nn \l__akshar_tmpe_int { 0 }
167   \seq_map_variable:Nnn \l__akshar_tmpe_seq \l__akshar_map_tl
168   {
169     \int_compare:nNnTF { \l__akshar_tmpe_int } > { 0 }
170     { \seq_put_right:NV \l__akshar_tmpe_seq \l__akshar_map_tl }
171     {
172       \int_compare:nNnTF
173       { \l__akshar_tmpe_int } = { 1 + \seq_count:N \l__akshar_tmpe_seq }
174       {
175         \bool_if:NT {#5}
176         { \int_incr:N \l__akshar_tmpe_int }
177         \seq_clear:N \l__akshar_tmpe_seq
178         \__akshar_seq_push_seq:NN
179         \l__akshar_tmpe_seq \l__akshar_tmpe_seq
180         \int_set:Nn \l__akshar_tmpe_int { 1 }
181         \tl_set:Nx \l__akshar_tmpe_tl
182         { \seq_item:Nn \l__akshar_tmpe_seq { 1 } }
183         \tl_if_eq:NNTF \l__akshar_map_tl \l__akshar_tmpe_tl
184         {
185           \int_incr:N \l__akshar_tmpe_int
186           \seq_put_right:NV \l__akshar_tmpe_seq \l__akshar_map_tl
187         }
188         {
189           \seq_put_right:NV \l__akshar_tmpe_seq \l__akshar_map_tl
190         }
191       }
192     }
193     \tl_set:Nx \l__akshar_tmpe_tl
194     {
195       \seq_item:Nn \l__akshar_tmpe_seq { \l__akshar_tmpe_int }
196     }
197     \tl_if_eq:NNTF \l__akshar_map_tl \l__akshar_tmpe_tl
198     {
199       \int_incr:N \l__akshar_tmpe_int
200       \seq_put_right:NV \l__akshar_tmpe_seq \l__akshar_map_tl
201     }
202     {
203       \int_set:Nn \l__akshar_tmpe_int { 1 }

```



```

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

```

(End definition for `__akshar_replace:NnnnN`.)

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

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

```

228 \NewExpandableDocumentCommand \aksharStrLen {m}
229 {
230     \akshar_convert:Nn \l__akshar_tmpa_seq {#1}
231     \seq_count:N \l__akshar_tmpa_seq
232 }

```

(End definition for `\aksharStrLen`. This function is documented on page 1.)

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

```

233 \NewExpandableDocumentCommand \aksharStrChar {mm}
234 {
235     \akshar_convert:Nn \l__akshar_tmpa_seq {#1}
236     \bool_if:nTF
237     {
238         \int_compare_p:nNn {#2} > {0} &&
239         \int_compare_p:nNn {#2} < {1 + \seq_count:N \l__akshar_tmpa_seq}
240     }
241     { \seq_item:Nn \l__akshar_tmpa_seq { #2 } }
242     {
243         \msg_error:nnnx { akshar } { err_character_out_of_bound }
244         { #1 } { \__akshar_int_append_ordinal:n { #2 } }
245         { \int_eval:n { 1 + \seq_count:N \l__akshar_tmpa_seq } }
246         \scan_stop:
247     }
248 }

```

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

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

```

249 \NewExpandableDocumentCommand \aksharStrHead {m}
250 {
251     \akshar_convert:Nn \l__akshar_tmpa_seq {#1}
252     \int_compare:nNnTF { \seq_count:N \l__akshar_tmpa_seq } = {0}
253     {
254         \msg_error:nnn { akshar } { err_character_out_of_bound }
255         { first }

```

```

256     \scan_stop:
257     }
258     { \seq_item:Nn \l__akshar_tmpa_seq { 1 } }
259 }

```

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

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

```

260 \NewExpandableDocumentCommand \aksharStrTail {m}
261 {
262     \akshar_convert:Nn \l__akshar_tmpa_seq {#1}
263     \int_compare:nNnTF { \seq_count:N \l__akshar_tmpa_seq } = {0}
264     {
265         \msg_error:nnn { akshar } { err_character_out_of_bound }
266         { first }
267         \scan_stop:
268     }
269     { \seq_item:Nn \l__akshar_tmpa_seq { \seq_count:N \l__akshar_tmpa_seq } }
270 }

```

(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*`

```

271 \NewExpandableDocumentCommand \aksharStrReplace {smmm}
272 {
273     \IfBooleanTF {#1}
274     {
275         \__akshar_replace:NnnN \l__akshar_tmpa_seq
276         {#2} {#3} {#4} \c_true_bool
277     }
278     {
279         \__akshar_replace:NnnN \l__akshar_tmpa_seq
280         {#2} {#3} {#4} \c_false_bool
281     }
282     \seq_use:Nn \l__akshar_tmpa_seq {}
283 }

```

(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*`

```

284 \NewExpandableDocumentCommand \aksharStrRemove {smm}
285 {
286     \IfBooleanTF {#1}
287     {
288         \__akshar_replace:NnnN \l__akshar_tmpa_seq
289         {#2} {#3} {} \c_true_bool
290     }
291     {
292         \__akshar_replace:NnnN \l__akshar_tmpa_seq
293         {#2} {#3} {} \c_false_bool
294     }
295     \seq_use:Nn \l__akshar_tmpa_seq {}
296 }

```

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

```

297 </package>

```

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.

A	
akshar commands:	
\akshar_convert:Nn	1, 3, 4, 6, <u>116</u> , 160, 161, 162, 230, 235, 251, 262
akshar internal commands:	
\l__akshar_char_seq 7, <u>14</u> , 118, 124, 125, 131, 132, 139, 140, 146, 152, 153
\c__akshar_diacritics_tl	. 3, <u>6</u> , 122
__akshar_int_append_ordinal:n <u>84</u> , 244
\c__akshar_joining_tl 3, <u>6</u> , 129
\l__akshar_map_tl 120, 122, 126, 129, 133, 141, 146, 167, 170, 183, 186, 189, 197, 200, 209, 213, 217
\l__akshar_prev_joining_bool 6, <u>13</u> , 119, 134, 137, 142
__akshar_replace:NnnN <u>158</u> , 275, 279, 288, 292
__akshar_seq_push_seq:NN <u>156</u> , 178, 204, 223
\c__akshar_str_g_tl <u>56</u>
\c__akshar_str_seq_tl <u>56</u>
\l__akshar_tmpa_int	7, 8, <u>15</u> , 165, 173, 180, 185, 195, 199, 203, 211
\l__akshar_tmpa_seq <u>15</u> , 163, 179, 189, 205, 217, 223, 225, 226, 230, 231, 235, 239, 241, 245, 251, 252, 258, 262, 263, 269, 275, 279, 282, 288, 292, 295
\l__akshar_tmpa_tl <u>15</u> , 70, 74, 124, 126, 131, 133, 139, 141, 181, 183, 193, 197, 207, 209
\l__akshar_tmpb_int <u>15</u> , 166, 169, 176
\l__akshar_tmpb_seq <u>15</u> , 68, 70, 71, 164, 170, 177, 186, 200, 205, 206, 213, 223
\l__akshar_tmpb_tl <u>15</u> , 71, 72
\l__akshar_tmpe_seq	... <u>15</u> , 160, 167
\l__akshar_tmpe_seq <u>15</u> , 161, 173, 182, 195, 208
\l__akshar_tmpe_seq	... <u>15</u> , 162, 179
__akshar_var_if_global 5
__akshar_var_if_global:NTF <u>56</u> , 151, 224
\aksharStrChar 2, 4, <u>233</u>
\aksharStrHead 2, 4, <u>249</u>
\aksharStrLen 1, <u>228</u>
\aksharStrRemove 2, 3, <u>284</u>
\aksharStrRemove* 2, 3, <u>284</u>
\aksharStrReplace 2, 10, <u>271</u>
\aksharStrReplace* 2, <u>271</u>
\aksharStrTail 2, 4, <u>260</u>
B	
bool commands:	
\bool_if:NTF 137, 175
\bool_if:nTF 60, 236
\bool_new:N 13
\bool_set_false:N 119, 142
\bool_set_true:N 134
\c_false_bool 7, 280, 293
\c_true_bool 7, 276, 289
C	
cs commands:	
\cs_generate_variant:Nn 53, 54, 55, 155
\cs_new:Npn 84, 116, 156, 158
\cs_split_function:N 61, 69
E	
exp commands:	
\exp_last_unbraced:Nf 61, 69
I	
\IfBooleanTF 273, 286
int commands:	
\int_case:nnTF 87, 99, 107
\int_compare:nNnTF 97, 169, 172, 252, 263
\int_compare_p:nNn 238, 239
\int_eval:n 245
\int_incr:N 176, 185, 199, 211
\int_new:N 22, 23
\int_set:Nn 165, 166, 180, 203
M	
msg commands:	
\msg_error:nnn <u>54</u> , 54, 63, 78, 254, 265
\msg_error:nnnn <u>54</u> , 55, 243
\msg_new:nnnn 24, 34, 44
N	
\NewExpandableDocumentCommand 228, 233, 249, 260, 271, 284
P	
prg commands:	
\prg_generate_conditional_	- variant:Nnn 52
\prg_new_conditional:Npnn 58
\prg_return_false: 65, 75, 80
\prg_return_true: 75
\ProvidesExplPackage 4
R	
\RequirePackage 3
S	
scan commands:	
\scan_stop:	.. 42, 50, 246, 256, 267
seq commands:	
\seq_clear:N	118, 163, 164, 177, 206
\seq_count:N 173, 231, 239, 245, 252, 263, 269
\seq_get_left:NN 70
\seq_get_right:NN 71
\seq_gset_eq:NN 152, 225
\seq_item:Nn 182, 195, 208, 241, 258, 269
\seq_map_inline:Nn 157
\seq_map_variable:NNn 167
\seq_new:N	.. 14, 17, 18, 19, 20, 21
\seq_pop_right:NN 124, 131, 139

```

\seq_put_right:Nn .....
    ..... 125, 132, 140, 145,
    157, 170, 186, 189, 200, 212, 216
\seq_set_eq:NN ..... 153, 226
\seq_set_split:Nnn ..... 53, 53, 68
\seq_use:Nn ..... 282, 295

T

tl commands:
\tl_const:Nn ..... 6, 7, 56, 57
\tl_if_eq:NNTF .....
    ..... 72, 74, 129, 183, 197, 209
\tl_if_in:Nn ..... 52

\tl_if_in:NnTF ..... 52, 122
\tl_map_variable:NNn ..... 120
\tl_new:N ..... 15, 16
\tl_set:Nn ..... 181, 193, 207
\tl_to_str:n ..... 56, 57

token commands:
\token_to_str:N . 42, 50, 64, 68, 79

U

use commands:
\use_i:nnn ..... 69
\use_iii:nnn ..... 61

```