

Contents

1	Encoding and escaping schemes	2
2	Conversion functions	4
3	Internal string functions	4
4	Possibilities, and things to do	5
5	l3str implementation	5
5.1	Helpers	6
5.1.1	A function unrelated to strings	6
5.1.2	Variables and constants	6
5.1.3	Escaping spaces	7
5.2	String conditionals	8
5.3	Conversions	10
5.3.1	Producing one byte or character	10
5.3.2	Mapping functions for conversions	11
5.3.3	Error-reporting during conversion	12
5.3.4	Framework for conversions	13
5.3.5	Byte unescape and escape	17
5.3.6	Native strings	18
5.3.7	8-bit encodings	20
5.4	Messages	23
5.5	Escaping definition files	24
5.5.1	Unescape methods	24
5.5.2	Escape methods	29
5.6	Encoding definition files	31
5.6.1	UTF-8 support	32
5.6.2	UTF-16 support	37
5.6.3	UTF-32 support	42
5.6.4	ISO 8859 support	46
	Index	63

The `l3str-convert` package: string encoding conversions*

The L^AT_EX3 Project[†]

Released 2013/01/08

1 Encoding and escaping schemes

Traditionally, string encodings only specify how strings of characters should be stored as bytes. However, the resulting lists of bytes are often to be used in contexts where only a restricted subset of bytes are permitted (*e.g.*, PDF string objects, URLs). Hence, storing a string of characters is done in two steps.

- The code points (“character codes”) are expressed as bytes following a given “encoding”. This can be UTF-16, ISO 8859-1, *etc.* See Table 1 for a list of supported encodings.¹
- Bytes are translated to T_EX tokens through a given “escaping”. Those are defined for the most part by the pdf file format. See Table 2 for a list of escaping methods supported.²

*This file describes v4339, last revised 2013/01/08.

[†]E-mail: latex-team@latex-project.org

¹Encodings and escapings will be added as they are requested.

Table 1: Supported encodings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the encoding in this list.

<i>⟨Encoding⟩</i>	description
<code>utf8</code>	UTF-8
<code>utf16</code>	UTF-16, with byte-order mark
<code>utf16be</code>	UTF-16, big-endian
<code>utf16le</code>	UTF-16, little-endian
<code>utf32</code>	UTF-32, with byte-order mark
<code>utf32be</code>	UTF-32, big-endian
<code>utf32le</code>	UTF-32, little-endian
<code>iso88591, latin1</code>	ISO 8859-1
<code>iso88592, latin2</code>	ISO 8859-2
<code>iso88593, latin3</code>	ISO 8859-3
<code>iso88594, latin4</code>	ISO 8859-4
<code>iso88595</code>	ISO 8859-5
<code>iso88596</code>	ISO 8859-6
<code>iso88597</code>	ISO 8859-7
<code>iso88598</code>	ISO 8859-8
<code>iso88599, latin5</code>	ISO 8859-9
<code>iso885910, latin6</code>	ISO 8859-10
<code>iso885911</code>	ISO 8859-11
<code>iso885913, latin7</code>	ISO 8859-13
<code>iso885914, latin8</code>	ISO 8859-14
<code>iso885915, latin9</code>	ISO 8859-15
<code>iso885916, latin10</code>	ISO 8859-16
Empty	Native (Unicode) string.

Table 2: Supported escapings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the escaping in this list.

<i>⟨Escaping⟩</i>	description
<code>bytes</code> , or empty	arbitrary bytes
<code>hex</code> , hexadecimal	byte = two hexadecimal digits
<code>name</code>	see <code>\pdfescapename</code>
<code>string</code>	see <code>\pdfescapestring</code>
<code>url</code>	encoding used in URLs

2 Conversion functions

`\str_set_convert:Nnnn`
`\str_gset_convert:Nnnn`

`\str_set_convert:Nnnn` $\langle str\ var \rangle$ $\{\langle string \rangle\}$ $\{\langle name\ 1 \rangle\}$ $\{\langle name\ 2 \rangle\}$

This function converts the $\langle string \rangle$ from the encoding given by $\langle name\ 1 \rangle$ to the encoding given by $\langle name\ 2 \rangle$, and stores the result in the $\langle str\ var \rangle$. Each $\langle name \rangle$ can have the form $\langle encoding \rangle$ or $\langle encoding \rangle / \langle escaping \rangle$, where the possible values of $\langle encoding \rangle$ and $\langle escaping \rangle$ are given in Tables 1 and 2, respectively. The default escaping is to input and output bytes directly. The special case of an empty $\langle name \rangle$ indicates the use of “native” strings, 8-bit for pdfTeX, and Unicode strings for the other two engines.

For example,

`\str_set_convert:Nnnn \l_foo_str { Hello! } { } { utf16/hex }`

results in the variable `\l_foo_str` holding the string FFFF00480065006C006C006F0021. This is obtained by converting each character in the (native) string `Hello!` to the UTF-16 encoding, and expressing each byte as a pair of hexadecimal digits. Note the presence of a (big-endian) byte order mark "FFFF", which can be avoided by specifying the encoding `utf16be/hex`.

An error is raised if the $\langle string \rangle$ is not valid according to the $\langle escaping\ 1 \rangle$ and $\langle encoding\ 1 \rangle$, or if it cannot be reencoded in the $\langle encoding\ 2 \rangle$ and $\langle escaping\ 2 \rangle$ (for instance, if a character does not exist in the $\langle encoding\ 2 \rangle$). Erroneous input is replaced by the Unicode replacement character "FFFD", and characters which cannot be reencoded are replaced by either the replacement character "FFFD" if it exists in the $\langle encoding\ 2 \rangle$, or an encoding-specific replacement character, or the question mark character.

`\str_set_convert:NnnnTF`
`\str_gset_convert:NnnnTF`

`\str_set_convert:NnnnTF` $\langle str\ var \rangle$ $\{\langle string \rangle\}$ $\{\langle name\ 1 \rangle\}$ $\{\langle name\ 2 \rangle\}$ $\{\langle true\ code \rangle\}$
 $\{\langle false\ code \rangle\}$

As `\str_set_convert:Nnnn`, converts the $\langle string \rangle$ from the encoding given by $\langle name\ 1 \rangle$ to the encoding given by $\langle name\ 2 \rangle$, and assigns the result to $\langle str\ var \rangle$. Contrarily to `\str_set_convert:Nnnn`, the conditional variant does not raise errors in case the $\langle string \rangle$ is not valid according to the $\langle name\ 1 \rangle$ encoding, or cannot be expressed in the $\langle name\ 2 \rangle$ encoding. Instead, the $\langle false\ code \rangle$ is performed.

`\c_max_char_int`

The maximum valid character code, 255 for pdfTeX, and 1114111 for XeTeX and LuaTeX.

3 Internal string functions

`__str_gset_other:Nn`

`__str_gset_other:Nn` $\langle tl\ var \rangle$ $\{\langle token\ list \rangle\}$

Converts the $\langle token\ list \rangle$ to an $\langle other\ string \rangle$, where spaces have category code “other”, and assigns the result to the $\langle tl\ var \rangle$, globally.

`_str_hexadecimal_use:NTF`

`_str_hexadecimal_use:NTF <token> {(true code)} {(false code)}`

If the $\langle token \rangle$ is a hexadecimal digit (upper case or lower case), its upper-case version is left in the input stream, *followed* by the $\langle true code \rangle$. Otherwise, the $\langle false code \rangle$ is left in the input stream.

T_EXhackers note: This function fails on some inputs if the escape character is a hexadecimal digit. We are thus careful to set the escape character to a known (safe) value before using it.

`_str_output_byte:n` ★

`_str_output_byte:n {(intexpr)}`

Expands to a character token with category other and character code equal to the value of $\langle intexpr \rangle$. The value of $\langle intexpr \rangle$ must be in the range $[-1, 255]$, and any value outside this range results in undefined behaviour. The special value -1 is used to produce an empty result.

4 Possibilities, and things to do

Encoding/escaping-related tasks.

- Describe the internal format in the code comments. Refuse code points in ["D800, "DFFF] in the internal representation?
- Add documentation about each encoding and escaping method, and add examples.
- The `hex` unescaping should raise an error for odd-token count strings.
- Decide what bytes should be escaped in the `url` escaping. Perhaps `!'()*-./0123456789_` are safe, and all other characters should be escaped?
- Automate generation of 8-bit mapping files.
- Change the framework for 8-bit encodings: for decoding from 8-bit to Unicode, use 256 integer registers; for encoding, use a tree-box.
- More encodings (see Heiko's `stringenc`). CESU?
- More escapings: shell escapes, lua escapes, etc?

5 l3str implementation

```
1 <*initex | package>
2 <@@=str>
3 \ProvidesExplPackage
4   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
5 \RequirePackage{l3str,l3tl-analysis,l3tl-build,l3flag}
```

5.1 Helpers

5.1.1 A function unrelated to strings

`\use_ii_i:nn` A function used to swap its arguments.

```
6 \cs_if_exist:NF \use_ii_i:nn
7 { \cs_new:Npn \use_ii_i:nn #1#2 { #2 #1 } }
```

(End definition for `\use_ii_i:nn`.)

5.1.2 Variables and constants

`\c_max_char_int` The maximum valid character code is 255 for pdfTeX, and 1114111 for other engines.

```
8 \int_const:Nn \c_max_char_int
9 { \pdfTeX_if_engine:TF { "FF } { "10FFFF } }
```

(End definition for `\c_max_char_int`. This variable is documented on page 4.)

`__str_tmp:w` Internal scratch space for some functions.

```
\l__str_internal_int 10 \cs_new_protected_nopar:Npn \__str_tmp:w { }
\l__str_internal_tl   11 \tl_new:N \l__str_internal_tl
                      12 \int_new:N \l__str_internal_int
```

(End definition for `__str_tmp:w`. This function is documented on page ??.)

`\g__str_result_tl` The `\g__str_result_tl` variable is used to hold the result of various internal string operations (mostly conversions) which are typically performed in a group. The variable is global so that it remains defined outside the group, to be assigned to a user-provided variable.

```
13 \tl_new:N \g__str_result_tl
```

(End definition for `\g__str_result_tl`. This variable is documented on page ??.)

`\c__str_replacement_char_int` When converting, invalid bytes are replaced by the Unicode replacement character "FFFD.

```
14 \int_const:Nn \c__str_replacement_char_int { "FFFD }
```

(End definition for `\c__str_replacement_char_int`. This variable is documented on page ??.)

`\c_forty_eight` We declare here some integer values which delimit ranges of ASCII characters of various types. This is mostly used in `l3regex`.

```
\c_fifty_eight
\c_sixty_five  15 \int_const:Nn \c_forty_eight { 48 }
\c_ninety_one  16 \int_const:Nn \c_fifty_eight { 58 }
\c_ninety_seven 17 \int_const:Nn \c_sixty_five { 65 }
\c_one_hundred_thirty_three 18 \int_const:Nn \c_ninety_one { 91 }
\c_one_hundred_thirty_seven 19 \int_const:Nn \c_ninety_seven { 97 }
20 \int_const:Nn \c_one_hundred_thirty_three { 123 }
21 \int_const:Nn \c_one_hundred_thirty_seven { 127 }
```

(End definition for `\c_forty_eight` and others. These variables are documented on page ??.)

`\g__str_alias_prop` To avoid needing one file per encoding/escaping alias, we keep track of those in a property list.

```

22 \prop_new:N \g__str_alias_prop
23 \prop_gput:Nnn \g__str_alias_prop { latin1 } { iso88591 }
24 \prop_gput:Nnn \g__str_alias_prop { latin2 } { iso88592 }
25 \prop_gput:Nnn \g__str_alias_prop { latin3 } { iso88593 }
26 \prop_gput:Nnn \g__str_alias_prop { latin4 } { iso88594 }
27 \prop_gput:Nnn \g__str_alias_prop { latin5 } { iso88599 }
28 \prop_gput:Nnn \g__str_alias_prop { latin6 } { iso885910 }
29 \prop_gput:Nnn \g__str_alias_prop { latin7 } { iso885913 }
30 \prop_gput:Nnn \g__str_alias_prop { latin8 } { iso885914 }
31 \prop_gput:Nnn \g__str_alias_prop { latin9 } { iso885915 }
32 \prop_gput:Nnn \g__str_alias_prop { latin10 } { iso885916 }
33 \prop_gput:Nnn \g__str_alias_prop { utf16le } { utf16 }
34 \prop_gput:Nnn \g__str_alias_prop { utf16be } { utf16 }
35 \prop_gput:Nnn \g__str_alias_prop { utf32le } { utf32 }
36 \prop_gput:Nnn \g__str_alias_prop { utf32be } { utf32 }
37 \prop_gput:Nnn \g__str_alias_prop { hexadecimal } { hex }

```

(End definition for `\g__str_alias_prop`. This variable is documented on page ??.)

`\g__str_error_bool` In conversion functions with a built-in conditional, errors are not reported directly to the user, but the information is collected in this boolean, used at the end to decide on which branch of the conditional to take.

```

38 \bool_new:N \g__str_error_bool

```

(End definition for `\g__str_error_bool`. This variable is documented on page ??.)

`str_byte` Conversions from one *<encoding>/<escaping>* pair to another are done within `x`-expanding assignments. Errors are signalled by raising the relevant flag.

```

39 \flag_new:n { str_byte }
40 \flag_new:n { str_error }

```

(End definition for `str_byte` and `str_error`. These variables are documented on page ??.)

5.1.3 Escaping spaces

`__str_gset_other:Nn` This function could be done by using `__str_to_other:n` within an `x`-expansion, but `__str_gset_other_loop:w` that would take a time quadratic in the size of the string. Instead, we can “leave the result behind us” in the input stream, to be captured into the expanding assignment. `__str_gset_other_end:w` This gives us a linear time.

```

41 \group_begin:
42 \char_set_lccode:nn { \* } { \ }
43 \char_set_lccode:nn { \A } { \A }
44 \tl_to_lowercase:n
45 {
46   \group_end:
47   \cs_new_protected:Npn \__str_gset_other:Nn #1#2
48   {
49     \tl_gset:Nx #1
50     {

```

```

51         \exp_after:wN \_str_gset_other_loop:w \tl_to_str:n {#2} ~ %
52         A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \q_stop
53     }
54 }
55 \cs_new:Npn \_str_gset_other_loop:w
56 #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 ~
57 {
58     \if_meaning:w A #9
59     \_str_gset_other_end:w
60     \fi:
61     #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * #9
62     \_str_gset_other_loop:w *
63 }
64 \cs_new:Npn \_str_gset_other_end:w \fi: #1 * A #2 \q_stop
65 { \fi: #1 }
66 }

```

(End definition for `_str_gset_other:Nn`. This function is documented on page 4.)

5.2 String conditionals

`_str_if_contains_char:NNT` Expects the *<token list>* to be an *<other string>*: the caller is responsible for ensuring that no (too-)special catcodes remain. Spaces with catcode 10 are ignored. Loop over the characters of the string, comparing character codes. The loop is broken if character codes match. Otherwise we return “false”.

`_str_if_contains_char:NNTF`

`_str_if_contains_char:nNTF`

`_str_if_contains_char_aux:NN`

`_str_if_contains_char_true:`

```

67 \prg_new_conditional:Npnn \_str_if_contains_char:NN #1#2 { T , TF }
68 {
69     \exp_after:wN \_str_if_contains_char_aux:NN \exp_after:wN #2
70     #1 { \_prg_break:n { ? \fi: } }
71     \_prg_break_point:
72     \prg_return_false:
73 }
74 \prg_new_conditional:Npnn \_str_if_contains_char:nN #1#2 { TF }
75 {
76     \_str_if_contains_char_aux:NN #2 #1 { \_prg_break:n { ? \fi: } }
77     \_prg_break_point:
78     \prg_return_false:
79 }
80 \cs_new:Npn \_str_if_contains_char_aux:NN #1#2
81 {
82     \if_charcode:w #1 #2
83     \exp_after:wN \_str_if_contains_char_true:
84     \fi:
85     \_str_if_contains_char_aux:NN #1
86 }
87 \cs_new_nopar:Npn \_str_if_contains_char_true:
88 { \_prg_break:n { \prg_return_true: \use_none:n } }

```

(End definition for `_str_if_contains_char:NNT` and `_str_if_contains_char:NNTF`. These functions are documented on page ??.)

`__str_octal_use:NTF` If the $\langle token \rangle$ is an octal digit, it is left in the input stream, *followed* by the $\langle true code \rangle$. Otherwise, the $\langle false code \rangle$ is left in the input stream.

T_EXhackers note: This function will fail if the escape character is an octal digit. We are thus careful to set the escape character to a known value before using it. T_EX dutifully detects octal digits for us: if #1 is an octal digit, then the right-hand side of the comparison is '1#1, greater than 1. Otherwise, the right-hand side stops as '1, and the conditional takes the **false** branch.

```

89 \prg_new_conditional:Npnn \__str_octal_use:N #1 { TF }
90 {
91   \if_int_compare:w \c_one < '1 \token_to_str:N #1 \exp_stop_f:
92     #1 \prg_return_true:
93   \else:
94     \prg_return_false:
95   \fi:
96 }

```

(End definition for `__str_octal_use:NTF`.)

`__str_hexadecimal_use:NTF` T_EX detects uppercase hexadecimal digits for us (see `__str_octal_use:NTF`), but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```

97 \prg_new_conditional:Npnn \__str_hexadecimal_use:N #1 { TF }
98 {
99   \if_int_compare:w \c_two < "1 \token_to_str:N #1 \exp_stop_f:
100     #1 \prg_return_true:
101   \else:
102     \if_case:w \__int_eval:w
103       \exp_after:wN ' \token_to_str:N #1 - 'a
104       \__int_eval_end:
105       A
106     \or: B
107     \or: C
108     \or: D
109     \or: E
110     \or: F
111     \else:
112       \prg_return_false:
113       \exp_after:wN \use_none:n
114     \fi:
115     \prg_return_true:
116   \fi:
117 }

```

(End definition for `__str_hexadecimal_use:NTF`.)

5.3 Conversions

5.3.1 Producing one byte or character

`\c__str_byte_0_tl` `\c__str_byte_1_tl` `\c__str_byte_255_tl` `\c__str_byte_-1_tl` For each integer N in the range $[0, 255]$, we create a constant token list which holds three character tokens with category code other: the character with character code N , followed by the representation of N as two hexadecimal digits. The value -1 is given a default token list which ensures that later functions give an empty result for the input -1 .

```

118 \group_begin:
119   \char_set_catcode_other:n { \c_zero }
120   \tl_set:Nx \l__str_internal_tl { \tl_to_str:n { 0123456789ABCDEF } }
121   \exp_args:No \tl_map_inline:nn { \l__str_internal_tl " }
122     { \char_set_lccode:nn {'#1} { \c_zero } }
123   \tl_map_inline:Nn \l__str_internal_tl
124     {
125       \tl_map_inline:Nn \l__str_internal_tl
126         {
127           \char_set_lccode:nn { \c_zero } {"#1##1}
128           \tl_to_lowercase:n
129             {
130               \tl_const:cx
131                 { c__str_byte_ \int_eval:n {"#1##1} _tl }
132                 { ^^@ #1 ##1 }
133             }
134         }
135     }
136 \group_end:
137 \tl_const:cn { c__str_byte_-1_tl } { { } \use_none:n { } }

```

(End definition for `\c__str_byte_0_tl`, `\c__str_byte_1_tl`, and `\c__str_byte_255_tl`. These functions are documented on page ??.)

`__str_output_byte:n` `__str_output_byte:w` `__str_output_hexadecimal:n` `__str_output_hexadecimal:w` `__str_output_end:` Those functions must be used carefully: feeding them a value outside the range $[-1, 255]$ will attempt to use the undefined token list variable `\c__str_byte_⟨number⟩_tl`. Assuming that the argument is in the right range, we expand the corresponding token list, and pick either the byte (first token) or the hexadecimal representations (second and third tokens). The value -1 produces an empty result in both cases.

```

138 \cs_new:Npn \__str_output_byte:n #1
139   { \__str_output_byte:w #1 \__str_output_end: }
140 \cs_new_nopar:Npn \__str_output_byte:w
141   {
142     \exp_after:wN \exp_after:wN
143     \exp_after:wN \use_i:nnn
144     \cs:w c__str_byte_ \int_use:N \__int_eval:w
145   }
146 \cs_new:Npn \__str_output_hexadecimal:n #1
147   { \__str_output_hexadecimal:w #1 \__str_output_end: }
148 \cs_new_nopar:Npn \__str_output_hexadecimal:w
149   {
150     \exp_after:wN \exp_after:wN

```

```

151     \exp_after:wN \use_none:n
152     \cs:w c__str_byte_ \int_use:N \__int_eval:w
153   }
154   \cs_new_nopar:Npn \__str_output_end:
155     { \__int_eval_end: _tl \cs_end: }

```

(End definition for __str_output_byte:n. This function is documented on page ??.)

__str_output_byte_pair_be:n Convert a number in the range [0,65535] to a pair of bytes, either big-endian or little-endian.

```

\__str_output_byte_pair_le:n
\__str_output_byte_pair:nnN
156 \cs_new:Npn \__str_output_byte_pair_be:n #1
157   {
158     \exp_args:Nf \__str_output_byte_pair:nnN
159       { \int_div_truncate:nn { #1 } { "100 } } {#1} \use:nn
160   }
161 \cs_new:Npn \__str_output_byte_pair_le:n #1
162   {
163     \exp_args:Nf \__str_output_byte_pair:nnN
164       { \int_div_truncate:nn { #1 } { "100 } } {#1} \use_ii_i:nn
165   }
166 \cs_new:Npn \__str_output_byte_pair:nnN #1#2#3
167   {
168     #3
169     { \__str_output_byte:n { #1 } }
170     { \__str_output_byte:n { #2 - #1 * "100 } }
171   }

```

(End definition for __str_output_byte_pair_be:n. This function is documented on page ??.)

5.3.2 Mapping functions for conversions

__str_convert_gmap:N This maps the function #1 over all characters in \g__str_result_tl, which should be a byte string in most cases, sometimes a native string.

```

\__str_convert_gmap_loop:NN
172 \cs_new_protected:Npn \__str_convert_gmap:N #1
173   {
174     \tl_gset:Nx \g__str_result_tl
175       {
176         \exp_after:wN \__str_convert_gmap_loop:NN
177         \exp_after:wN #1
178         \g__str_result_tl { ? \__prg_break: }
179         \__prg_break_point:
180       }
181   }
182 \cs_new:Npn \__str_convert_gmap_loop:NN #1#2
183   {
184     \use_none:n #2
185     #1#2
186     \__str_convert_gmap_loop:NN #1
187   }

```

(End definition for __str_convert_gmap:N. This function is documented on page ??.)

`__str_convert_gmap_internal:N` This maps the function #1 over all character codes in `\g__str_result_tl`, which must be in the internal representation.

```

188 \cs_new_protected:Npn \__str_convert_gmap_internal:N #1
189 {
190   \tl_gset:Nx \g__str_result_tl
191   {
192     \exp_after:wN \__str_convert_gmap_internal_loop:Nww
193     \exp_after:wN #1
194     \g__str_result_tl \s__tl \q_stop \__prg_break: \s__tl
195     \__prg_break_point:
196   }
197 }
198 \cs_new:Npn \__str_convert_gmap_internal_loop:Nww #1 #2 \s__tl #3 \s__tl
199 {
200   \use_none_delimit_by_q_stop:w #3 \q_stop
201   #1 {#3}
202   \__str_convert_gmap_internal_loop:Nww #1
203 }

```

(End definition for `__str_convert_gmap_internal:N`. This function is documented on page ??.)

5.3.3 Error-reporting during conversion

`__str_if_flag_error:nnx` When converting using the function `\str_set_convert:Nnnn`, errors should be reported to the user after each step in the conversion. Errors are signalled by raising some flag (typically `@@_error`), so here we test that flag: if it is raised, give the user an error, otherwise remove the arguments. On the other hand, in the conditional functions `\str_set_convert:NnnnTF`, errors should be suppressed. This is done by changing `__str_if_flag_error:nnx` into `__str_if_flag_no_error:nnx` locally.

```

204 \cs_new_protected:Npn \__str_if_flag_error:nnx #1
205 {
206   \flag_if_raised:nTF {#1}
207   { \__msg_kernel_error:nnx { str } }
208   { \use_none:nn }
209 }
210 \cs_new_protected:Npn \__str_if_flag_no_error:nnx #1#2#3
211 { \flag_if_raised:nT {#1} { \bool_gset_true:N \g__str_error_bool } }

```

(End definition for `__str_if_flag_error:nnx`. This function is documented on page ??.)

`__str_if_flag_times:nT` At the end of each conversion step, we raise all relevant errors as one error message, built on the fly. The height of each flag indicates how many times a given error was encountered. This function prints #2 followed by the number of occurrences of an error if it occurred, nothing otherwise.

```

212 \cs_new_protected:Npn \__str_if_flag_times:nT #1#2
213 { \flag_if_raised:nT {#1} { #2~(x \flag_height:n {#1} ) } }

```

(End definition for `__str_if_flag_times:nT`.)

5.3.4 Framework for conversions

Most functions in this module expect to be working with “native” strings. Strings can also be stored as bytes, in one of many encodings, for instance UTF8. The bytes themselves can be expressed in various ways in terms of T_EX tokens, for instance as pairs of hexadecimal digits. The questions of going from arbitrary Unicode code points to bytes, and from bytes to tokens are mostly independent.

Conversions are done in four steps:

- “unescape” produces a string of bytes;
- “decode” takes in a string of bytes, and converts it to a list of Unicode characters in an internal representation, with items of the form

$\langle bytes \rangle \backslash s_t1 \langle Unicode\ code\ point \rangle \backslash s_t1$

where we have collected the $\langle bytes \rangle$ which combined to form this particular Unicode character, and the $\langle Unicode\ code\ point \rangle$ is in the range [0, "10FFFF].

- “encode” encodes the internal list of code points as a byte string in the new encoding;
- “escape” escapes bytes as requested.

The process is modified in case one of the encoding is empty (or the conversion function has been set equal to the empty encoding because it was not found): then the unescape or escape step is ignored, and the decode or encode steps work on tokens instead of bytes. Otherwise, each step must ensure that it passes a correct byte string or internal string to the next step.

The input string is stored in $\backslash g_str_result_t1$, then we: unescape and decode; encode and escape; exit the group and store the result in the user’s variable. The various conversion functions all act on $\backslash g_str_result_t1$. Errors are silenced for the conditional functions by redefining $\backslash_str_if_flag_error:nx$ locally.

```

\str_set_convert:Nnnn
\str_gset_convert:Nnnn
\str_set_convert:NnnnTF
\str_gset_convert:NnnnTF
\__str_convert:nNNnnn
214 \cs_new_protected_nopar:Npn \str_set_convert:Nnnn
215   { \__str_convert:nNNnnn { } \tl_set_eq:NN }
216 \cs_new_protected_nopar:Npn \str_gset_convert:Nnnn
217   { \__str_convert:nNNnnn { } \tl_gset_eq:NN }
218 \prg_new_protected_conditional:Npnn
219   \str_set_convert:Nnnn #1#2#3#4 { T , F , TF }
220   {
221     \bool_gset_false:N \g__str_error_bool
222     \__str_convert:nNNnnn
223     { \cs_set_eq:NN \__str_if_flag_error:nxx \__str_if_flag_no_error:nxx }
224     \tl_set_eq:NN #1 {#2} {#3} {#4}
225     \bool_if:NTF \g__str_error_bool \prg_return_false: \prg_return_true:
226   }
227 \prg_new_protected_conditional:Npnn
228   \str_gset_convert:Nnnn #1#2#3#4 { T , F , TF }
229   {

```

```

230 \bool_gset_false:N \g__str_error_bool
231 \__str_convert:nNNnnn
232 { \cs_set_eq:NN \__str_if_flag_error:nmx \__str_if_flag_no_error:nmx }
233 \tl_gset_eq:NN #1 {#2} {#3} {#4}
234 \bool_if:NTF \g__str_error_bool \prg_return_false: \prg_return_true:
235 }
236 \cs_new_protected:Npn \__str_convert:nNNnnn #1#2#3#4#5#6
237 {
238   \group_begin:
239   #1
240   \__str_gset_other:Nn \g__str_result_tl {#4}
241   \exp_after:wN \__str_convert:wwwnn
242   \tl_to_str:n {#5} /// \q_stop
243   { decode } { unescape }
244   \prg_do_nothing:
245   \__str_convert_decode_:
246   \exp_after:wN \__str_convert:wwwnn
247   \tl_to_str:n {#6} /// \q_stop
248   { encode } { escape }
249   \use_ii_i:nn
250   \__str_convert_encode_:
251   \group_end:
252   #2 #3 \g__str_result_tl
253 }

```

(End definition for `\str_set_convert:Nnnn` and `\str_gset_convert:Nnnn`. These functions are documented on page 4.)

`__str_convert:wwwnn`
`__str_convert:NNnNN`

The task of `__str_convert:wwwnn` is to split $\langle encoding \rangle / \langle escaping \rangle$ pairs into their components, #1 and #2. Calls to `__str_convert:nnn` ensure that the corresponding conversion functions are defined. The third auxiliary does the main work.

- #1 is the encoding conversion function;
- #2 is the escaping function;
- #3 is the escaping name for use in an error message;
- #4 is `\prg_do_nothing:` for unescaping/decoding, and `\use_ii_i:nn` for encoding/escaping;
- #5 is the default encoding function (either “decode” or “encode”), for which there should be no escaping.

Let us ignore the native encoding for a second. In the unescaping/decoding phase, we want to do #2#1 in this order, and in the encoding/escaping phase, the order should be reversed: #4#2#1 does exactly that. If one of the encodings is the default (native), then the escaping should be ignored, with an error if any was given, and only the encoding, #1, should be performed.

```

254 \cs_new_protected:Npn \__str_convert:wwwnn
255   #1 / #2 // #3 \q_stop #4#5

```

```

256 {
257   \__str_convert:nnn {enc} {#4} {#1}
258   \__str_convert:nnn {esc} {#5} {#2}
259   \exp_args:Ncc \__str_convert:NNnNN
260   { \__str_convert_#4_#1: } { \__str_convert_#5_#2: } {#2}
261 }
262 \cs_new_protected:Npn \__str_convert:NNnNN #1#2#3#4#5
263 {
264   \if_meaning:w #1 #5
265   \tl_if_empty:nF {#3}
266   { \_msg_kernel_error:nnx { str } { native-escaping } {#3} }
267   #1
268   \else:
269   #4 #2 #1
270   \fi:
271 }

```

(End definition for `__str_convert:wwwnn`. This function is documented on page 4.)

`__str_convert:nnn` The arguments of `__str_convert:nnn` are: `enc` or `esc`, used to build filenames, the type of the conversion (unescape, decode, encode, escape), and the encoding or escaping name. If the function is already defined, no need to do anything. Otherwise, filter out all non-alphanumerics in the name, and lowercase it. Feed that, and the same three arguments, to `__str_convert:nnnn`. The task is then to make sure that the conversion function `#3_#1` corresponding to the type `#3` and filtered name `#1` is defined, then set our initial conversion function `#3_#4` equal to that.

How do we get the `#3_#1` conversion to be defined if it isn't? Two main cases.

First, if `#1` is a key in `\g__str_alias_prop`, then the value `\l__str_internal_tl` tells us what file to load. Loading is skipped if the file was already read, *i.e.*, if the conversion command based on `\l__str_internal_tl` already exists. Otherwise, try to load the file; if that fails, there is an error, use the default empty name instead.

Second, `#1` may be absent from the property list. The `\cs_if_exist:cF` test is automatically false, and we search for a file defining the encoding or escaping `#1` (this should allow third-party `.def` files). If the file is not found, there is an error, use the default empty name instead.

In all cases, the conversion based on `\l__str_internal_tl` is defined, so we can set the `#3_#1` function equal to that. In some cases (*e.g.*, `utf16be`), the `#3_#1` function is actually defined within the file we just loaded, and it is different from the `\l__str_internal_tl`-based function: we mustn't clobber that different definition.

```

272 \cs_new_protected:Npn \__str_convert:nnn #1#2#3
273 {
274   \cs_if_exist:cF { \__str_convert_#2_#3: }
275   {
276     \exp_args:Nx \__str_convert:nnnn
277     { \__str_convert_lowercase_alphanum:n {#3} }
278     {#1} {#2} {#3}
279   }
280 }
281 \cs_new_protected:Npn \__str_convert:nnnn #1#2#3#4

```

```

282 {
283   \cs_if_exist:cF { __str_convert_#3_#1: }
284   {
285     \prop_get:NnNF \g__str_alias_prop {#1} \l__str_internal_tl
286     { \tl_set:Nn \l__str_internal_tl {#1} }
287     \cs_if_exist:cF { __str_convert_#3_ \l__str_internal_tl : }
288     {
289       \file_if_exist:nTF { l3str-#2- \l__str_internal_tl .def }
290       {
291         \group_begin:
292         \__str_load_catcodes:
293         \file_input:n { l3str-#2- \l__str_internal_tl .def }
294         \group_end:
295       }
296       {
297         \tl_clear:N \l__str_internal_tl
298         \__msg_kernel_error:nxxx { str } { unknown-#2 } {#4} {#1}
299       }
300     }
301     \cs_if_exist:cF { __str_convert_#3_#1: }
302     {
303       \cs_gset_eq:cc { __str_convert_#3_#1: }
304       { __str_convert_#3_ \l__str_internal_tl : }
305     }
306   }
307   \cs_gset_eq:cc { __str_convert_#3_#4: } { __str_convert_#3_#1: }
308 }

```

(End definition for `__str_convert:nnn`. This function is documented on page 4.)

`__str_convert_lowercase_alphanum:n` This function keeps only letters and digits, with upper case letters converted to lower case.
`__str_convert_lowercase_alphanum_loop:N`

```

309 \cs_new:Npn \__str_convert_lowercase_alphanum:n #1
310 {
311   \exp_after:wN \__str_convert_lowercase_alphanum_loop:N
312   \tl_to_str:n {#1} { ? \__prg_break: }
313   \__prg_break_point:
314 }
315 \cs_new:Npn \__str_convert_lowercase_alphanum_loop:N #1
316 {
317   \use_none:n #1
318   \if_int_compare:w '#1 < \c_ninety_one
319   \if_int_compare:w '#1 < \c_sixty_five
320   \if_int_compare:w \c_one < 1#1 \exp_stop_f:
321   #1
322   \fi:
323   \else:
324     \__str_output_byte:n { '#1 + \c_thirty_two }
325   \fi:
326   \else:

```



```

327     \if_int_compare:w '#1 < \c_one_hundred_twenty_three
328     \if_int_compare:w '#1 < \c_ninety_seven
329     \else:
330         #1
331     \fi:
332 \fi:
333 \fi:
334 \__str_convert_lowercase_alphanum_loop:N
335 }

```

(End definition for __str_convert_lowercase_alphanum:n. This function is documented on page ??.)

__str_load_catcodes: Since encoding files may be loaded at arbitrary places in a T_EX document, including within verbatim mode, we set the catcodes of all characters appearing in any encoding definition file.

```

336 \cs_new_protected:Npn \__str_load_catcodes:
337 {
338     \char_set_catcode_escape:N \
339     \char_set_catcode_group_begin:N \{
340     \char_set_catcode_group_end:N \}
341     \char_set_catcode_math_toggle:N \$
342     \char_set_catcode_alignment:N &
343     \char_set_catcode_parameter:N #
344     \char_set_catcode_math_superscript:N ^
345     \char_set_catcode_ignore:N %
346     \char_set_catcode_space:N ~
347     \tl_map_function:nN { abcdefghijklmnopqrstuvwxyz_:\<DEFILNPSTUX }
348     \char_set_catcode_letter:N
349     \tl_map_function:nN { 0123456789"'"?*\+-.(),!/<>[];= }
350     \char_set_catcode_other:N
351     \char_set_catcode_comment:N %
352     \int_set:Nn \tex_endlinechar:D {32}
353 }

```

(End definition for __str_load_catcodes:.)

5.3.5 Byte unescape and escape

Strings of bytes may need to be stored in auxiliary files in safe “escaping” formats. Each such escaping is only loaded as needed. By default, on input any non-byte is filtered out, while the output simply consists in letting bytes through.

__str_filter_bytes:n In the case of pdfT_EX, every character is a byte. For Unicode-aware engines, test the character code; non-bytes cause us to raise the flag **str_byte**. Spaces have already been given the correct category code when this function is called.

```

354 \pdfTeX_if_engine:TF
355 { \cs_new_eq:NN \__str_filter_bytes:n \use:n }
356 {
357     \cs_new:Npn \__str_filter_bytes:n #1
358     {
359         \__str_filter_bytes_aux:N #1

```

```

360         { ? \__prg_break: }
361         \__prg_break_point:
362     }
363     \cs_new:Npn \__str_filter_bytes_aux:N #1
364     {
365         \use_none:n #1
366         \if_int_compare:w '#1 < 256 \exp_stop_f:
367             #1
368         \else:
369             \flag_raise:n { str_byte }
370         \fi:
371         \__str_filter_bytes_aux:N
372     }
373 }

```

(End definition for __str_filter_bytes:n. This function is documented on page ??.)

__str_convert_unescape_: The simplest unescaping method removes non-bytes from \g__str_result_tl.

```

\__str_convert_unescape_bytes:
374 \pdfTeX_if_engine:TF
375 { \cs_new_protected_nopar:Npn \__str_convert_unescape_: { } }
376 {
377     \cs_new_protected_nopar:Npn \__str_convert_unescape_:
378     {
379         \flag_clear:n { str_byte }
380         \tl_gset:Nx \g__str_result_tl
381         { \exp_args:No \__str_filter_bytes:n \g__str_result_tl }
382         \__str_if_flag_error:nxx { str_byte } { non-byte } { bytes }
383     }
384 }
385 \cs_new_eq:NN \__str_convert_unescape_bytes: \__str_convert_unescape_:

```

(End definition for __str_convert_unescape_:. This function is documented on page ??.)

__str_convert_escape_: The simplest form of escape leaves the bytes from the previous step of the conversion unchanged.

```

386 \cs_new_protected_nopar:Npn \__str_convert_escape_: { }
387 \cs_new_eq:NN \__str_convert_escape_bytes: \__str_convert_escape_:

```

(End definition for __str_convert_escape_:. This function is documented on page ??.)

5.3.6 Native strings

__str_convert_decode_: Convert each character to its character code, one at a time.

```

\__str_decode_native_char:N
388 \cs_new_protected_nopar:Npn \__str_convert_decode_:
389 { \__str_convert_gmap:N \__str_decode_native_char:N }
390 \cs_new:Npn \__str_decode_native_char:N #1
391 { #1 \s__tl \__int_value:w '#1 \s__tl }

```

(End definition for __str_convert_decode_:. This function is documented on page ??.)

`__str_convert_encode_:` The conversion from an internal string to native character tokens is very different in pdfTeX and in other engines. For Unicode-aware engines, we need the definitions to be read when the null byte has category code 12, so we set that inside a group.

```

392 \group_begin:
393   \char_set_catcode_other:n { 0 }
394   \pdfTeX_if_engine:TF

```

`__str_encode_native_char:n` Since pdfTeX only supports 8-bit characters, and we have a table of all bytes, the conversion can be done in linear time within an x-expanding assignment. Look out for character codes larger than 255, those characters are replaced by ?, and raise a flag, which then triggers a pdfTeX-specific error.

```

395 {
396   \cs_new_protected_nopar:Npn \__str_convert_encode_:
397   {
398     \flag_clear:n { str_error }
399     \__str_convert_gmap_internal:N \__str_encode_native_char:n
400     \__str_if_flag_error:nnx { str_error }
401     { pdfTeX-native-overflow } { }
402   }
403   \cs_new:Npn \__str_encode_native_char:n #1
404   {
405     \if_int_compare:w #1 < \c_two_hundred_fifty_six
406       \__str_output_byte:n {#1}
407     \else:
408       \flag_raise:n { str_error }
409       ?
410     \fi:
411   }
412   \__msg_kernel_new:nnnn { str } { pdfTeX-native-overflow }
413   { Character-code-too-large-for-pdfTeX. }
414   {
415     The-pdfTeX-engine-only-supports-8-bit-characters:~
416     valid-character-codes-are-in-the-range-[0,255].~
417     To-manipulate-arbitrary-Unicode,~use-LuaTeX-or-XeTeX.
418   }
419 }

```

`__str_encode_native_loop:w`
`__str_encode_native_flush:`
`__str_encode_native_filter:N`

In Unicode-aware engines, since building particular characters cannot be done expandably in TeX, we cannot hope to get a linear-time function. However, we get quite close using the `l3tl-build` module, which abuses `\toks` to reach an almost linear time. Use the standard lowercase trick to produce an arbitrary character from the null character, and add that character to the end of the token list being built. At the end of the loop, put the token list together with `__tl_build_end:`. Note that we use an x-expanding assignment because it is slightly faster. Unicode-aware engines will never incur an overflow because the internal string is guaranteed to only contain code points in `[0, "10FFFF]`.

```

420 {
421   \cs_new_protected_nopar:Npn \__str_convert_encode_:
422   {

```

```

423     \int_zero:N \l__tl_build_offset_int
424     \__tl_gbuild_x:Nw \g__str_result_tl
425     \exp_after:wN \__str_encode_native_loop:w
426     \g__str_result_tl \s__tl { \q_stop \__prg_break: } \s__tl
427     \__prg_break_point:
428     \__tl_build_end:
429   }
430   \cs_new_protected:Npn \__str_encode_native_loop:w #1 \s__tl #2 \s__tl
431   {
432     \use_none_delimit_by_q_stop:w #2 \q_stop
433     \tex_lccode:D \l__str_internal_int \__int_eval:w #2 \__int_eval_end:
434     \tl_to_lowercase:n { \__tl_build_one:n { ^^@ } }
435     \__str_encode_native_loop:w
436   }
437 }

```

End the group to restore the catcode of the null byte.

```

438 \group_end:

```

(End definition for `__str_convert_encode_`:. This function is documented on page ??.)

5.3.7 8-bit encodings

This section will be entirely rewritten: it is not yet clear in what situations 8-bit encodings are used, hence I don't know what exactly should be optimized. The current approach is reasonably efficient to convert long strings, and it scales well when using many different encodings. An approach based on csnames would have a smaller constant load time for each individual conversion, but has a large hash table cost. Using a range of `\count` registers works for decoding, but not for encoding: one possibility there would be to use a binary tree for the mapping of Unicode characters to bytes, stored as a box, one per encoding.

Since the section is going to be rewritten, documentation lacks.

All the 8-bit encodings which l3str supports rely on the same internal functions.

`__str_declare_eight_bit_encoding:nnn` This declares the encoding $\langle name \rangle$ to map bytes to Unicode characters according to the $\langle mapping \rangle$, and map those bytes which are not mentioned in the $\langle mapping \rangle$ either to the replacement character (if they appear in $\langle missing \rangle$), or to themselves.

All the 8-bit encoding definition file start with `__str_declare_eight_bit_encoding:nnn { $\langle encoding name \rangle$ } { $\langle mapping \rangle$ } { $\langle missing bytes \rangle$ }`. The $\langle mapping \rangle$ argument is a token list of pairs $\{ \langle byte \rangle \} \{ \langle Unicode \rangle \}$ expressed in uppercase hexadecimal notation. The $\langle missing \rangle$ argument is a token list of $\{ \langle byte \rangle \}$. Every $\langle byte \rangle$ which does not appear in the $\langle mapping \rangle$ nor the $\langle missing \rangle$ lists maps to the same code point in Unicode.

```

439 \cs_new_protected:Npn \__str_declare_eight_bit_encoding:nnn #1#2#3
440   {
441     \tl_set:Nn \l__str_internal_tl {#1}
442     \cs_new_protected_nopar:cpn { __str_convert_decode_#1: }
443     { \__str_convert_decode_eight_bit:n {#1} }
444     \cs_new_protected_nopar:cpn { __str_convert_encode_#1: }
445     { \__str_convert_encode_eight_bit:n {#1} }

```

```

446 \tl_const:cn { c__str_encoding_#1_tl } {#2}
447 \tl_const:cn { c__str_encoding_#1_missing_tl } {#3}
448 }

```

(End definition for __str_declare_eight_bit_encoding:nnn.)

```

\__str_convert_decode_eight_bit:n
\__str_decode_eight_bit_load:nn
\__str_decode_eight_bit_load_missing:n
\__str_decode_eight_bit_char:N
449 \cs_new_protected:Npn \__str_convert_decode_eight_bit:n #1
450 {
451   \group_begin:
452     \int_zero:N \l__str_internal_int
453     \exp_last_unbraced:Nx \__str_decode_eight_bit_load:nn
454       { \tl_use:c { c__str_encoding_#1_tl } }
455     { \q_stop \__prg_break: } { }
456     \__prg_break_point:
457     \exp_last_unbraced:Nx \__str_decode_eight_bit_load_missing:n
458       { \tl_use:c { c__str_encoding_#1_missing_tl } }
459     { \q_stop \__prg_break: }
460     \__prg_break_point:
461     \flag_clear:n { str_error }
462     \__str_convert_gmap:N \__str_decode_eight_bit_char:N
463     \__str_if_flag_error:nxx { str_error } { decode-8-bit } {#1}
464   \group_end:
465 }
466 \cs_new_protected:Npn \__str_decode_eight_bit_load:nn #1#2
467 {
468   \use_none_delimit_by_q_stop:w #1 \q_stop
469   \tex_dimen:D "#1 = \l__str_internal_int sp \scan_stop:
470   \tex_skip:D \l__str_internal_int = "#1 sp \scan_stop:
471   \tex_toks:D \l__str_internal_int \exp_after:wN { \__int_value:w "#2 }
472   \tex_advance:D \l__str_internal_int \c_one
473   \__str_decode_eight_bit_load:nn
474 }
475 \cs_new_protected:Npn \__str_decode_eight_bit_load_missing:n #1
476 {
477   \use_none_delimit_by_q_stop:w #1 \q_stop
478   \tex_dimen:D "#1 = \l__str_internal_int sp \scan_stop:
479   \tex_skip:D \l__str_internal_int = "#1 sp \scan_stop:
480   \tex_toks:D \l__str_internal_int \exp_after:wN
481     { \int_use:N \c__str_replacement_char_int }
482   \tex_advance:D \l__str_internal_int \c_one
483   \__str_decode_eight_bit_load_missing:n
484 }
485 \cs_new:Npn \__str_decode_eight_bit_char:N #1
486 {
487   #1 \s_tl
488   \if_int_compare:w \tex_dimen:D '#1 < \l__str_internal_int
489     \if_int_compare:w \tex_skip:D \tex_dimen:D '#1 = '#1 \exp_stop_f:
490     \tex_the:D \tex_toks:D \tex_dimen:D
491   \fi:
492   \fi:

```

```

493   \_int_value:w '#1 \s_tl
494 }

```

(End definition for _str_convert_decode_eight_bit:n. This function is documented on page ??.)

```

\_str_convert_encode_eight_bit:n
  \_str_encode_eight_bit_load:nn
  \_str_encode_eight_bit_char:n
  \_str_encode_eight_bit_char_aux:n

```

```

495 \cs_new_protected:Npn \_str_convert_encode_eight_bit:n #1
496 {
497   \group_begin:
498     \int_zero:N \l__str_internal_int
499     \exp_last_unbraced:Nx \_str_encode_eight_bit_load:nn
500     { \tl_use:c { c__str_encoding_#1_tl } }
501     { \q_stop \_prg_break: } { }
502     \_prg_break_point:
503     \flag_clear:n { str_error }
504     \_str_convert_gmap_internal:N \_str_encode_eight_bit_char:n
505     \_str_if_flag_error:nx { str_error } { encode-8-bit } {#1}
506   \group_end:
507 }
508 \cs_new_protected:Npn \_str_encode_eight_bit_load:nn #1#2
509 {
510   \use_none_delimit_by_q_stop:w #1 \q_stop
511   \tex_dimen:D "#2 = \l__str_internal_int sp \scan_stop:
512   \tex_skip:D \l__str_internal_int = "#2 sp \scan_stop:
513   \exp_args:NNf \tex_toks:D \l__str_internal_int
514   { \_str_output_byte:n { "#1 } }
515   \tex_advance:D \l__str_internal_int \c_one
516   \_str_encode_eight_bit_load:nn
517 }
518 \cs_new:Npn \_str_encode_eight_bit_char:n #1
519 {
520   \if_int_compare:w #1 > \c_max_register_int
521     \flag_raise:n { str_error }
522   \else:
523     \if_int_compare:w \tex_dimen:D #1 < \l__str_internal_int
524       \if_int_compare:w \tex_skip:D \tex_dimen:D #1 = #1 \exp_stop_f:
525       \tex_the:D \tex_toks:D \tex_dimen:D #1 \exp_stop_f:
526       \exp_after:wN \exp_after:wN \exp_after:wN \use_none:nn
527       \fi:
528     \fi:
529     \_str_encode_eight_bit_char_aux:n {#1}
530   \fi:
531 }
532 \cs_new:Npn \_str_encode_eight_bit_char_aux:n #1
533 {
534   \if_int_compare:w #1 < \c_two_hundred_fifty_six
535     \_str_output_byte:n {#1}
536   \else:
537     \flag_raise:n { str_error }
538   \fi:
539 }

```

(End definition for `_str_convert_encode_eight_bit:n`. This function is documented on page ??.)

5.4 Messages

General messages, and messages for the encodings and escapings loaded by default (“native”, and “bytes”).

```

540 \_msg_kernel_new:nnn { str } { unknown-esc }
541 { Escaping~scheme~'#1'~(filtered:~'#2')~unknown. }
542 \_msg_kernel_new:nnn { str } { unknown-enc }
543 { Encoding~scheme~'#1'~(filtered:~'#2')~unknown. }
544 \_msg_kernel_new:nnnn { str } { native-escaping }
545 { The~'native'~encoding~scheme~does~not~support~any~escaping. }
546 {
547   Since~native~strings~do~not~consist~in~bytes,~
548   none~of~the~escaping~methods~make~sense.~
549   The~specified~escaping,~'#1',~will be ignored.
550 }
551 \_msg_kernel_new:nnn { str } { file-not-found }
552 { File~'l3str-#1.def'~not~found. }

```

Message used when the “bytes” unescaping fails because the string given to `\str_set_convert:Nnnn` contains a non-byte. This cannot happen for the pdfTeX engine, since that engine only supports 8-bit characters. Messages used for other escapings and encodings are defined in each definition file.

```

553 \pdfTeX_if_engine:F
554 {
555   \_msg_kernel_new:nnnn { str } { non-byte }
556   { String~invalid~in~escaping~'#1':~it~may~only~contain~bytes. }
557   {
558     Some~characters~in~the~string~you~asked~to~convert~are~not~
559     8-bit~characters.~Perhaps~the~string~is~a~'native'~Unicode~string?~
560     If~it~is,~try~using~\
561     \
562     \iow_indent:n
563     {
564       \iow_char:N\str_set_convert:Nnnn \
565       \ \ <str-var>~\{~<string>~\}~\{~native~\}~\{~<target~encoding>~\}
566     }
567   }
568 }

```

Those messages are used when converting to and from 8-bit encodings.

```

569 \_msg_kernel_new:nnnn { str } { decode-8-bit }
570 { Invalid~string~in~encoding~'#1'. }
571 {
572   LaTeX~came~across~a~byte~which~is~not~defined~to~represent~
573   any~character~in~the~encoding~'#1'.
574 }
575 \_msg_kernel_new:nnnn { str } { encode-8-bit }
576 { Unicode~string~cannot~be~converted~to~encoding~'#1'. }

```

```

577 {
578   The~encoding~'#1'~only~contains~a~subset~of~all~Unicode~characters.~
579   LaTeX~was~asked~to~convert~a~string~to~that~encoding,~but~that~
580   string~contains~a~character~that~'#1'~does~not~support.
581 }
582 </initex | package>

```

5.5 Escaping definition files

Several of those encodings are defined by the pdf file format. The following byte storage methods are defined:

- **bytes** (default), non-bytes are filtered out, and bytes are left untouched (this is defined by default);
- **hex** or **hexadecimal**, as per the pdfTeX primitive `\pdfescapehex`
- **name**, as per the pdfTeX primitive `\pdfescapename`
- **string**, as per the pdfTeX primitive `\pdfescapestring`
- **url**, as per the percent encoding of urls.

5.5.1 Unescape methods

`__str_convert_unescape_hex:` Take chars two by two, and interpret each pair as the hexadecimal code for a byte.
`__str_unescape_hex_auxi:N` Anything else than hexadecimal digits is ignored, raising the flag. A string which contains
`__str_unescape_hex_auxii:N` an odd number of hexadecimal digits gets 0 appended to it: this is equivalent to appending
a 0 in all cases, and dropping it if it is alone.

```

583 <*hex>
584 \cs_new_protected_nopar:Npn \__str_convert_unescape_hex:
585 {
586   \group_begin:
587     \flag_clear:n { str_error }
588     \int_set:Nn \tex_escapechar:D { 92 }
589     \tl_gset:Nx \g__str_result_tl
590     {
591       \__str_output_byte:w "
592       \exp_last_unbraced:Nf \__str_unescape_hex_auxi:N
593       { \tl_to_str:N \g__str_result_tl }
594       0 { ? 0 - \c_one \__prg_break: }
595       \__prg_break_point:
596       \__str_output_end:
597     }
598     \__str_if_flag_error:nnx { str_error } { unescape-hex } { }
599   \group_end:
600 }
601 \cs_new:Npn \__str_unescape_hex_auxi:N #1
602 {

```



```

603     \use_none:n #1
604     \__str_hexadecimal_use:NTF #1
605     { \__str_unescape_hex_auxii:N }
606     {
607         \flag_raise:n { str_error }
608         \__str_unescape_hex_auxi:N
609     }
610 }
611 \cs_new:Npn \__str_unescape_hex_auxii:N #1
612 {
613     \use_none:n #1
614     \__str_hexadecimal_use:NTF #1
615     {
616         \__str_output_end:
617         \__str_output_byte:w " \__str_unescape_hex_auxi:N
618     }
619     {
620         \flag_raise:n { str_error }
621         \__str_unescape_hex_auxii:N
622     }
623 }
624 \__msg_kernel_new:nnnn { str } { unescape-hex }
625 { String~invalid~in~escaping~'hex':~only~hexadecimal~digits~allowed. }
626 {
627     Some~characters~in~the~string~you~asked~to~convert~are~not~
628     hexadecimal~digits~(0-9,~A-F,~a-f)~nor~spaces.
629 }
630 </hex>

```

(End definition for __str_convert_unescape_hex:. This function is documented on page ??.)

__str_convert_unescape_name: The __str_convert_unescape_name: function replaces each occurrence of # followed by two hexadecimal digits in \g__str_result_tl by the corresponding byte. The url function is identical, with escape character % instead of #. Thus we define the two together. The arguments of __str_tmp:w are the character code of # or % in hexadecimal, the name of the main function to define, and the name of the auxiliary which performs the loop.

The looping auxiliary #3 finds the next escape character, reads the following two characters, and tests them. The test __str_hexadecimal_use:NTF leaves the upper-case digit in the input stream, hence we surround the test with __str_output_byte:w " and __str_output_end:. If both characters are hexadecimal digits, they should be removed before looping: this is done by \use_i:nnn. If one of the characters is not a hexadecimal digit, then feed "#1 to __str_output_byte:w to produce the escape character, raise the flag, and call the looping function followed by the two characters (remove \use_i:nnn).

```

631 <*name | url>
632 \cs_set_protected:Npn \__str_tmp:w #1#2#3
633 {
634     \cs_new_protected:cpn { __str_convert_unescape_#2: }

```

```

635     {
636         \group_begin:
637         \flag_clear:n { str_byte }
638         \flag_clear:n { str_error }
639         \int_set:Nn \tex_escapechar:D { 92 }
640         \tl_gset:Nx \g__str_result_tl
641         {
642             \exp_after:wN #3 \g__str_result_tl
643             #1 ? { ? \__prg_break: }
644             \__prg_break_point:
645         }
646         \__str_if_flag_error:nnx { str_byte } { non-byte } { #2 }
647         \__str_if_flag_error:nnx { str_error } { unescape-#2 } { }
648     \group_end:
649 }
650 \cs_new:Npn #3 ##1##2##3
651 {
652     \__str_filter_bytes:n {##1}
653     \use_none:n ##3
654     \__str_output_byte:w "
655     \__str_hexadecimal_use:NTF ##2
656     {
657         \__str_hexadecimal_use:NTF ##3
658         { }
659         {
660             \flag_raise:n { str_error }
661             * \c_zero + '#1 \use_i:nn
662         }
663     }
664     {
665         \flag_raise:n { str_error }
666         0 + '#1 \use_i:nn
667     }
668     \__str_output_end:
669     \use_i:nnn #3 ##2##3
670 }
671 \__msg_kernel_new:nnnn { str } { unescape-#2 }
672 { String~invalid~in~escaping~'#2'. }
673 {
674     LaTeX~came~across~the~escape~character~'#1'~not~followed~by~
675     two~hexadecimal~digits.~This~is~invalid~in~the~escaping~'#2'.
676 }
677 }
678 </name | url>
679 <*name>
680 \exp_after:wN \__str_tmp:w \c_hash_str { name }
681 \__str_unescape_name_loop:wNN
682 </name>
683 <*url>
684 \exp_after:wN \__str_tmp:w \c_percent_str { url }

```

```
685 \__str_unescape_url_loop:wNN
```

```
686 \url
```

(End definition for __str_convert_unescape_name:. This function is documented on page ??.)

```
\_str_convert_unescape_string:
\_str_unescape_string_newlines:wN
\_str_unescape_string_loop:wNNN
\_str_unescape_string_repeat:NNNNN
```

The **string** escaping is somewhat similar to the **name** and **url** escapings, with escape character \. The first step is to convert all three line endings, ^^J, ^^M, and ^^M^^J to the common ^^J, as per the PDF specification. This step cannot raise the flag.

Then the following escape sequences are decoded.

\n Line feed (10)

\r Carriage return (13)

\t Horizontal tab (9)

\b Backspace (8)

\f Form feed (12)

\(Left parenthesis

\) Right parenthesis

\\ Backslash

\ddd (backslash followed by 1 to 3 octal digits) Byte ddd (octal), subtracting 256 in case of overflow.

If followed by an end-of-line character, the backslash and the end-of-line are ignored. If followed by anything else, the backslash is ignored, raising the error flag.

```
687 <*string>
```

```
688 \group_begin:
```

```
689 \char_set_lccode:nn {'\*} {'\\}
```

```
690 \char_set_catcode_other:N \^^J
```

```
691 \char_set_catcode_other:N \^^M
```

```
692 \tl_to_lowercase:n
```

```
693 {
```

```
694 \cs_new_protected_nopar:Npn \__str_convert_unescape_string:
```

```
695 {
```

```
696 \group_begin:
```

```
697 \flag_clear:n { str_byte }
```

```
698 \flag_clear:n { str_error }
```

```
699 \int_set:Nn \tex_escapechar:D { 92 }
```

```
700 \tl_gset:Nx \g__str_result_tl
```

```
701 {
```

```
702 \exp_after:wN \__str_unescape_string_newlines:wN
```

```
703 \g__str_result_tl \__prg_break: ^^M ?
```

```
704 \__prg_break_point:
```

```
705 }
```

```
706 \tl_gset:Nx \g__str_result_tl
```

```
707 {
```

```

708         \exp_after:wN \_str_unescape_string_loop:wNNN
709         \g__str_result_tl * ?? { ? \_prg_break: }
710         \_prg_break_point:
711     }
712     \_str_if_flag_error:nxx { str_byte } { non-byte } { string }
713     \_str_if_flag_error:nxx { str_error } { unescape-string } { }
714     \group_end:
715 }
716 \cs_new:Npn \_str_unescape_string_loop:wNNN #1 *#2#3#4
717 {
718     {
719         \_str_filter_bytes:n {#1}
720         \use_none:n #4
721         \_str_output_byte:w '
722         \_str_octal_use:NTF #2
723         {
724             \_str_octal_use:NTF #3
725             {
726                 \_str_octal_use:NTF #4
727                 {
728                     \if_int_compare:w #2 > \c_three
729                     - 256
730                     \fi:
731                     \_str_unescape_string_repeat:NNNNNN
732                 }
733                 { \_str_unescape_string_repeat:NNNNNN ? }
734             }
735             { \_str_unescape_string_repeat:NNNNNN ?? }
736         }
737     }
738     \str_case_x:nnn {#2}
739     {
740         { \c_backslash_str } { 134 }
741         { ( } { 50 }
742         { ) } { 51 }
743         { r } { 15 }
744         { f } { 14 }
745         { n } { 12 }
746         { t } { 11 }
747         { b } { 10 }
748         { ^J } { 0 - \c_one }
749     }
750     {
751         \flag_raise:n { str_error }
752         0 - \c_one \use_i:nn
753     }
754 }
755 \_str_output_end:
756 \use_i:nn \_str_unescape_string_loop:wNNN #2#3#4
757 }

```

```

758 \cs_new:Npn \__str_unescape_string_repeat:NNNNNN #1#2#3#4#5#6
759 { \__str_output_end: \__str_unescape_string_loop:wNNN }
760 \cs_new:Npn \__str_unescape_string_newlines:wN #1 ^M #2
761 {
762   #1
763   \if_charcode:w ^J #2 \else: ^J \fi:
764   \__str_unescape_string_newlines:wN #2
765 }
766 \__msg_kernel_new:nnnn { str } { unescape-string }
767 { String~invalid~in~escaping~'string'. }
768 {
769   LaTeX~came~across~an~escape~character~'\c_backslash_str'~
770   not~followed~by~any~of:~'n',~'r',~'t',~'b',~'f',~'(',~')',~
771   '\c_backslash_str',~one~to~three~octal~digits,~or~the~end~
772   of~a~line.
773 }
774 \group_end:
775 \</string>

```

(End definition for __str_convert_unescape_string:. This function is documented on page ??.)

5.5.2 Escape methods

Currently, none of the escape methods can lead to errors, assuming that their input is made out of bytes.

```

\__str_convert_escape_hex: Loop and convert each byte to hexadecimal.
  \__str_escape_hex_char:N
776 <*hex>
777 \cs_new_protected_nopar:Npn \__str_convert_escape_hex:
778 { \__str_convert_gmap:N \__str_escape_hex_char:N }
779 \cs_new:Npn \__str_escape_hex_char:N #1
780 { \__str_output_hexadecimal:n { '#1' } }
781 \</hex>

```

(End definition for __str_convert_escape_hex:. This function is documented on page ??.)

```

\__str_convert_escape_name: For each byte, test whether it should be output as is, or be “hash-encoded”. Roughly,
  \__str_escape_name_char:N bytes outside the range [“2A,”7E] are hash-encoded. We keep two lists of exceptions:
  \__str_if_escape_name:NTF characters in \c__str_escape_name_not_str are not hash-encoded, and characters in
  \c__str_escape_name_str the \c__str_escape_name_str are encoded.
  \c__str_escape_name_not_str

```

```

782 <*name>
783 \str_const:Nn \c__str_escape_name_not_str { ! " $ & ' } %$
784 \str_const:Nn \c__str_escape_name_str { {}/ <> [] }
785 \cs_new_protected_nopar:Npn \__str_convert_escape_name:
786 { \__str_convert_gmap:N \__str_escape_name_char:N }
787 \cs_new:Npn \__str_escape_name_char:N #1
788 {
789   \__str_if_escape_name:NTF #1 {#1}
790   { \c_hash_str \__str_output_hexadecimal:n { '#1' } }
791 }
792 \prg_new_conditional:Npnn \__str_if_escape_name:N #1 { TF }

```

```

793 {
794   \if_int_compare:w '#1 < "2A \exp_stop_f:
795     \__str_if_contains_char:NNTF \c__str_escape_name_not_str #1
796     \prg_return_true: \prg_return_false:
797   \else:
798     \if_int_compare:w '#1 > "7E \exp_stop_f:
799     \prg_return_false:
800   \else:
801     \__str_if_contains_char:NNTF \c__str_escape_name_str #1
802     \prg_return_false: \prg_return_true:
803   \fi:
804 \fi:
805 }
806 \</name>

```

(End definition for __str_convert_escape_name:. This function is documented on page ??.)

_str_convert_escape_string: Any character below (and including) space, and any character above (and including) del, are converted to octal. One backslash is added before each parenthesis and backslash.

```

\__str_escape_string_char:N
\__str_if_escape_string:N
\c__str_escape_string_str
807 <*string>
808 \str_const:Nx \c__str_escape_string_str
809 { \c_backslash_str ( ) }
810 \cs_new_protected_nopar:Npn \__str_convert_escape_string:
811 { \__str_convert_gmap:N \__str_escape_string_char:N }
812 \cs_new:Npn \__str_escape_string_char:N #1
813 {
814   \__str_if_escape_string:NNTF #1
815   {
816     \__str_if_contains_char:NNTF
817       \c__str_escape_string_str #1
818       { \c_backslash_str }
819     #1
820   }
821   {
822     \c_backslash_str
823     \int_div_truncate:nn {'#1} {64}
824     \int_mod:nn { \int_div_truncate:nn {'#1} \c_eight } \c_eight
825     \int_mod:nn {'#1} \c_eight
826   }
827 }
828 \prg_new_conditional:Npnn \__str_if_escape_string:N #1 { TF }
829 {
830   \if_int_compare:w '#1 < "21 \exp_stop_f:
831   \prg_return_false:
832   \else:
833     \if_int_compare:w '#1 > "7E \exp_stop_f:
834     \prg_return_false:
835   \else:
836     \prg_return_true:
837   \fi:

```

```

838     \fi:
839   }
840 </string>

```

(End definition for `_str_convert_escape_string`:. This function is documented on page ??.)

`_str_convert_escape_url`: This function is similar to `_str_convert_escape_name`:, escaping different characters.

```

\__str_escape_url_char:N
\__str_if_escape_url:NTF

```

```

841 <*url>
842 \cs_new_protected_nopar:Npn \_str_convert_escape_url:
843   { \_str_convert_gmap:N \_str_escape_url_char:N }
844 \cs_new:Npn \_str_escape_url_char:N #1
845   {
846     \_str_if_escape_url:NTF #1 {#1}
847     { \c_percent_str \_str_output_hexadecimal:n { '#1 } }
848   }
849 \prg_new_conditional:Npnn \_str_if_escape_url:N #1 { TF }
850   {
851     \if_int_compare:w '#1 < "41 \exp_stop_f:
852     \_str_if_contains_char:nNTF { "-.<> } #1
853     \prg_return_true: \prg_return_false:
854   \else:
855     \if_int_compare:w '#1 > "7E \exp_stop_f:
856     \prg_return_false:
857   \else:
858     \_str_if_contains_char:nNTF { [ ] } #1
859     \prg_return_false: \prg_return_true:
860   \fi:
861 \fi:
862   }
863 </url>

```

(End definition for `_str_convert_escape_url`:. This function is documented on page ??.)

5.6 Encoding definition files

The **native** encoding is automatically defined. Other encodings are loaded as needed. The following encodings are supported:

- UTF-8;
- UTF-16, big-, little-endian, or with byte order mark;
- UTF-32, big-, little-endian, or with byte order mark;
- the ISO 8859 code pages, numbered from 1 to 16, skipping the inexistent ISO 8859-12.

5.6.1 utf-8 support

864 `<*utf8>`

```
__str_convert_encode_utf8:
    _str_encode_utf_viii_char:n
    _str_encode_utf_viii_loop:wnnw
```

Loop through the internal string, and convert each character to its UTF-8 representation. The representation is built from the right-most (least significant) byte to the left-most (most significant) byte. Continuation bytes are in the range [128, 191], taking 64 different values, hence we roughly want to express the character code in base 64, shifting the first digit in the representation by some number depending on how many continuation bytes there are. In the range [0, 127], output the corresponding byte directly. In the range [128, 2047], output the remainder modulo 64, plus 128 as a continuation byte, then output the quotient (which is in the range [0, 31]), shifted by 192. In the next range, [2048, 65535], split the character code into residue and quotient modulo 64, output the residue as a first continuation byte, then repeat; this leaves us with a quotient in the range [0, 15], which we output shifted by 224. The last range, [65536, 1114111], follows the same pattern: once we realize that dividing twice by 64 leaves us with a number larger than 15, we repeat, producing a last continuation byte, and offset the quotient by 240 for the leading byte.

How is that implemented? `__str_encode_utf_vii_loop:wnnw` takes successive quotients as its first argument, the quotient from the previous step as its second argument (except in step 1), the bound for quotients that trigger one more step or not, and finally the offset used if this step should produce the leading byte. Leading bytes can be in the ranges [0, 127], [192, 223], [224, 239], and [240, 247] (really, that last limit should be 244 because Unicode stops at the code point 1114111). At each step, if the quotient `#1` is less than the limit `#3` for that range, output the leading byte (`#1` shifted by `#4`) and stop. Otherwise, we need one more step: use the quotient of `#1` by 64, and `#1` as arguments for the looping auxiliary, and output the continuation byte corresponding to the remainder `#2 - 64#1 + 128`. The bizarre construction `\c_minus_one + \c_zero *` removes the spurious initial continuation byte (better methods welcome).

```
865 \cs_new_protected_nopar:cpn { __str_convert_encode_utf8: }
866   { __str_convert_gmap_internal:N __str_encode_utf_viii_char:n }
867 \cs_new:Npn __str_encode_utf_viii_char:n #1
868   {
869     __str_encode_utf_viii_loop:wnnw #1 ; \c_minus_one + \c_zero * ;
870     { 128 } { \c_zero }
871     { 32 } { 192 }
872     { 16 } { 224 }
873     { 8 } { 240 }
874   \q_stop
875 }
876 \cs_new:Npn __str_encode_utf_viii_loop:wnnw #1; #2; #3#4 #5 \q_stop
877   {
878     \if_int_compare:w #1 < #3 \exp_stop_f:
879       __str_output_byte:n { #1 + #4 }
880       \exp_after:wN \use_none_delimit_by_q_stop:w
881     \fi:
882     \exp_after:wN __str_encode_utf_viii_loop:wnnw
883       \int_value:w \int_div_truncate:nn {#1} {64} ; #1 ;
```



```

884      #5 \q_stop
885      \__str_output_byte:n { #2 - 64 * ( #1 - \c_two ) }
886    }

```

(End definition for __str_convert_encode_utf8:. This function is documented on page ??.)

```

\l__str_missing_flag
\l__str_extra_flag
\l__str_overlong_flag
\l__str_overflow_flag

```

When decoding a string that is purportedly in the UTF-8 encoding, four different errors can occur, signalled by a specific flag for each (we define those flags using \flag_clear_new:n rather than \flag_new:n, because they are shared with other encoding definition files).

- “Missing continuation byte”: a leading byte is not followed by the right number of continuation bytes.
- “Extra continuation byte”: a continuation byte appears where it was not expected, *i.e.*, not after an appropriate leading byte.
- “Overlong”: a Unicode character is expressed using more bytes than necessary, for instance, "C0"80 for the code point 0, instead of a single null byte.
- “Overflow”: this occurs when decoding produces Unicode code points greater than 1114111.

We only raise one L^AT_EX3 error message, combining all the errors which occurred. In the short message, the leading comma must be removed to get a grammatically correct sentence. In the long text, first remind the user what a correct UTF-8 string should look like, then add error-specific information.

```

887 \flag_clear_new:n { str_missing }
888 \flag_clear_new:n { str_extra }
889 \flag_clear_new:n { str_overlong }
890 \flag_clear_new:n { str_overflow }
891 \__msg_kernel_new:nmm { str } { utf8-decode }
892 {
893   Invalid-UTF-8~string: \exp_last_unbraced:Nf \use_none:n
894   \__str_if_flag_times:nT { str_missing } { ,~missing-continuation-byte }
895   \__str_if_flag_times:nT { str_extra } { ,~extra-continuation-byte }
896   \__str_if_flag_times:nT { str_overlong } { ,~overlong-form }
897   \__str_if_flag_times:nT { str_overflow } { ,~code-point-too-large }
898   .
899 }
900 {
901   In~the~UTF-8~encoding,~each~Unicode~character~consists~in~
902   1~to~4~bytes,~with~the~following~bit~pattern: \\
903   \iow_indent:n
904   {
905     Code~point~\ \ \ <~128:~0xxxxxxx \\
906     Code~point~\ \ \ <~2048:~110xxxxx~10xxxxxx \\
907     Code~point~\ \ \ <~65536:~1110xxxx~10xxxxxx~10xxxxxx \\
908     Code~point~ \ <~1114112:~11110xxx~10xxxxxx~10xxxxxx~10xxxxxx \\
909   }
910   Bytes~of~the~form~10xxxxxx~are~called~continuation~bytes.

```

```

911 \flag_if_raised:nT { str_missing }
912 {
913     \\\
914     A~leading~byte~(in~the~range~[192,255])~was~not~followed~by~
915     the~appropriate~number~of~continuation~bytes.
916 }
917 \flag_if_raised:nT { str_extra }
918 {
919     \\\
920     LaTeX~came~across~a~continuation~byte~when~it~was~not~expected.
921 }
922 \flag_if_raised:nT { str_overlong }
923 {
924     \\\
925     Every~Unicode~code~point~must~be~expressed~in~the~shortest~
926     possible~form.~For~instance,~'0xC0'~'0x83'~is~not~a~valid~
927     representation~for~the~code~point~3.
928 }
929 \flag_if_raised:nT { str_overflow }
930 {
931     \\\
932     Unicode~limits~code~points~to~the~range~[0,1114111].
933 }
934 }

```

(End definition for `\l__str_missing_flag` and others. These variables are documented on page ??.)

```

\__str_convert_decode_utf8:
    \_str_decode_utf_viii_start:N
    \_str_decode_utf_viii_continuation:wN
    \_str_decode_utf_viii_aux:wNnnwN
    \_str_decode_utf_viii_overflow:w
\__str_decode_utf_viii_end:

```

Decoding is significantly harder than encoding. As before, lower some flags, which are tested at the end (in bulk, to trigger at most one L^AT_EX3 error, as explained above). We expect successive multi-byte sequences of the form *⟨start byte⟩ ⟨continuation bytes⟩*. The `_start` auxiliary tests the first byte:

- [0, "7F]: the byte stands alone, and is converted to its own character code;
- ["80, "BF]: unexpected continuation byte, raise the appropriate flag, and convert that byte to the replacement character "FFFD;
- ["C0, "FF]: this byte should be followed by some continuation byte(s).

In the first two cases, `\use_none_delimit_by_q_stop:w` removes data that only the third case requires, namely the limits of ranges of Unicode characters which can be expressed with 1, 2, 3, or 4 bytes.

We can now concentrate on the multi-byte case and the `_continuation` auxiliary. We expect #3 to be in the range ["80, "BF]. The test for this goes as follows: if the character code is less than "80, we compare it to —"C0, yielding `false`; otherwise to "C0, yielding `true` in the range ["80, "BF] and `false` otherwise. If we find that the byte is not a continuation range, stop the current slew of bytes, output the replacement character, and continue parsing with the `_start` auxiliary, starting at the byte we just tested. Once we know that the byte is a continuation byte, leave it behind us in the input stream, compute what code point the bytes read so far would produce, and feed that number to the `_aux` function.

The `_aux` function tests whether we should look for more continuation bytes or not. If the number it receives as `#1` is less than the maximum `#4` for the current range, then we are done: check for an overlong representation by comparing `#1` with the maximum `#3` for the previous range. Otherwise, we call the `_continuation` auxiliary again, after shifting the “current code point” by `#4` (maximum from the range we just checked).

Two additional tests are needed: if we reach the end of the list of range maxima and we are still not done, then we are faced with an overflow. Clean up, and again insert the code point "FFFD" for the replacement character. Also, every time we read a byte, we need to check whether we reached the end of the string. In a correct UTF-8 string, this happens automatically when the `_start` auxiliary leaves its first argument in the input stream: the end-marker begins with `_prg_break:`, which ends the loop. On the other hand, if the end is reached when looking for a continuation byte, the `\use_none:n #3` construction removes the first token from the end-marker, and leaves the `_end` auxiliary, which raises the appropriate error flag before ending the mapping.

```

935 \cs_new_protected_nopar:cpn { __str_convert_decode_utf8: }
936 {
937   \flag_clear:n { str_error }
938   \flag_clear:n { str_missing }
939   \flag_clear:n { str_extra }
940   \flag_clear:n { str_overlong }
941   \flag_clear:n { str_overflow }
942   \tl_gset:Nx \g__str_result_tl
943   {
944     \exp_after:wN \__str_decode_utf_viii_start:N \g__str_result_tl
945     { \_prg_break: \__str_decode_utf_viii_end: }
946     \__prg_break_point:
947   }
948   \__str_if_flag_error:nmx { str_error } { utf8-decode } { }
949 }
950 \cs_new:Npn \__str_decode_utf_viii_start:N #1
951 {
952   #1
953   \if_int_compare:w '#1 < "C0 \exp_stop_f:
954     \s__tl
955     \if_int_compare:w '#1 < "80 \exp_stop_f:
956       \__int_value:w '#1
957     \else:
958       \flag_raise:n { str_extra }
959       \flag_raise:n { str_error }
960       \int_use:N \c__str_replacement_char_int
961     \fi:
962   \else:
963     \exp_after:wN \__str_decode_utf_viii_continuation:wwN
964     \int_use:N \__int_eval:w '#1 - "C0 \exp_after:wN \__int_eval_end:
965     \fi:
966     \s__tl
967     \use_none_delimit_by_q_stop:w {"80} {"800} {"10000} {"110000} \q_stop
968     \__str_decode_utf_viii_start:N

```

```

969 }
970 \cs_new:Npn \__str_decode_utf_viii_continuation:wwN
971   #1 \s__tl #2 \__str_decode_utf_viii_start:N #3
972 {
973   \use_none:n #3
974   \if_int_compare:w '#3 <
975     \if_int_compare:w '#3 < "80 \exp_stop_f: - \fi:
976     "C0 \exp_stop_f:
977     #3
978     \exp_after:wN \__str_decode_utf_viii_aux:wNnnwN
979     \int_use:N \__int_eval:w
980     #1 * "40 + '#3 - "80
981     \exp_after:wN \__int_eval_end:
982   \else:
983     \s__tl
984     \flag_raise:n { str_missing }
985     \flag_raise:n { str_error }
986     \int_use:N \c__str_replacement_char_int
987   \fi:
988   \s__tl
989   #2
990   \__str_decode_utf_viii_start:N #3
991 }
992 \cs_new:Npn \__str_decode_utf_viii_aux:wNnnwN
993   #1 \s__tl #2#3#4 #5 \__str_decode_utf_viii_start:N #6
994 {
995   \if_int_compare:w #1 < #4 \exp_stop_f:
996     \s__tl
997     \if_int_compare:w #1 < #3 \exp_stop_f:
998       \flag_raise:n { str_overlong }
999       \flag_raise:n { str_error }
1000     \int_use:N \c__str_replacement_char_int
1001   \else:
1002     #1
1003     \fi:
1004   \else:
1005     \if_meaning:w \q_stop #5
1006       \__str_decode_utf_viii_overflow:w #1
1007     \fi:
1008     \exp_after:wN \__str_decode_utf_viii_continuation:wwN
1009     \int_use:N \__int_eval:w #1 - #4 \exp_after:wN \__int_eval_end:
1010   \fi:
1011   \s__tl
1012   #2 {#4} #5
1013   \__str_decode_utf_viii_start:N
1014 }
1015 \cs_new:Npn \__str_decode_utf_viii_overflow:w #1 \fi: #2 \fi:
1016 {
1017   \fi: \fi:
1018   \flag_raise:n { str_overflow }

```

```

1019     \flag_raise:n { str_error }
1020     \int_use:N \c__str_replacement_char_int
1021   }
1022 \cs_new_nopar:Npn \__str_decode_utf_viii_end:
1023 {
1024   \s__tl
1025   \flag_raise:n { str_missing }
1026   \flag_raise:n { str_error }
1027   \int_use:N \c__str_replacement_char_int \s__tl
1028   \__prg_break:
1029 }
(End definition for \__str_convert_decode_utf8:. This function is documented on page ??.)
1030 </utf8>

```

5.6.2 utf-16 support

The definitions are done in a category code regime where the bytes 254 and 255 used by the byte order mark have catcode 12.

```

1031 <*utf16>
1032 \group_begin:
1033   \char_set_catcode_other:N ^^fe
1034   \char_set_catcode_other:N ^^ff

```

`__str_convert_encode_utf16:` When the endianness is not specified, it is big-endian by default, and we add a byte-order mark. Convert characters one by one in a loop, with different behaviours depending on the character code.

```

\__str_convert_encode_utf16be:
\__str_convert_encode_utf16le:
\__str_encode_utf_xvi_aux:N
\__str_encode_utf_xvi_char:n

```

- [0, "D7FF]: converted to two bytes;
- ["D800, "DFFF] are used as surrogates: they cannot be converted and are replaced by the replacement character;
- ["E000, "FFFF]: converted to two bytes;
- ["10000, "10FFFF]: converted to a pair of surrogates, each two bytes. The magic "D7C0 is "D800 – "10000/"400.

For the duration of this operation, `__str_tmp:w` is defined as a function to convert a number in the range [0, "FFFF] to a pair of bytes (either big endian or little endian), by feeding the quotient of the division of #1 by "100, followed by #1 to `__str_encode_utf_xvi_be:nn` or its `le` analog: those compute the remainder, and output two bytes for the quotient and remainder.

```

1035 \cs_new_protected_nopar:cpn { __str_convert_encode_utf16: }
1036 {
1037   \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n
1038   \tl_gput_left:Nx \g__str_result_tl { ^^fe ^^ff }
1039 }
1040 \cs_new_protected_nopar:cpn { __str_convert_encode_utf16be: }
1041 { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n }

```

```

1042 \cs_new_protected_nopar:cpn { __str_convert_encode_utf16le: }
1043 { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_le:n }
1044 \cs_new_protected:Npn \__str_encode_utf_xvi_aux:N #1
1045 {
1046   \flag_clear:n { str_error }
1047   \cs_set_eq:NN \__str_tmp:w #1
1048   \__str_convert_gmap_internal:N \__str_encode_utf_xvi_char:n
1049   \__str_if_flag_error:nxx { str_error } { utf16-encode } { }
1050 }
1051 \cs_new:Npn \__str_encode_utf_xvi_char:n #1
1052 {
1053   \if_int_compare:w #1 < "D800 \exp_stop_f:
1054     \__str_tmp:w {#1}
1055   \else:
1056     \if_int_compare:w #1 < "10000 \exp_stop_f:
1057       \if_int_compare:w #1 < "E000 \exp_stop_f:
1058         \flag_raise:n { str_error }
1059         \__str_tmp:w { \c__str_replacement_char_int }
1060       \else:
1061         \__str_tmp:w {#1}
1062       \fi:
1063     \else:
1064       \exp_args:Nf \__str_tmp:w { \int_div_truncate:nn {#1} {"400} + "D7C0 }
1065       \exp_args:Nf \__str_tmp:w { \int_mod:nn {#1} {"400} + "DC00 }
1066     \fi:
1067   \fi:
1068 }

```

(End definition for __str_convert_encode_utf16:, __str_convert_encode_utf16be:, and __str_convert_encode_utf16l:.)
 These functions are documented on page ??.)

\l__str_missing_flag When encoding a Unicode string to UTF-16, only one error can occur: code points in the range ["D800,"DFFF], corresponding to surrogates, cannot be encoded. We use the
 \l__str_extra_flag all-purpose flag @@_error to signal that error.
 \l__str_end_flag

When decoding a Unicode string which is purportedly in UTF-16, three errors can occur: a missing trail surrogate, an unexpected trail surrogate, and a string containing an odd number of bytes.

```

1069 \flag_clear_new:n { str_missing }
1070 \flag_clear_new:n { str_extra }
1071 \flag_clear_new:n { str_end }
1072 \__msg_kernel_new:nnnn { str } { utf16-encode }
1073 { Unicode~string~cannot~be~expressed~in~UTF-16:~surrogate. }
1074 {
1075   Surrogate~code~points~(in~the~range~[U+D800,~U+DFFF])~
1076   can~be~expressed~in~the~UTF-8~and~UTF-32~encodings,~
1077   but~not~in~the~UTF-16~encoding.
1078 }
1079 \__msg_kernel_new:nnnn { str } { utf16-decode }
1080 {
1081   Invalid~UTF-16~string: \exp_last_unbraced:Nf \use_none:n

```

```

1082     \_str_if_flag_times:nT { str_missing } { ,~missing~trail~surrogate }
1083     \_str_if_flag_times:nT { str_extra } { ,~extra~trail~surrogate }
1084     \_str_if_flag_times:nT { str_end } { ,~odd-number-of~bytes }
1085   .
1086 }
1087 {
1088   In~the~UTF-16~encoding,~each~Unicode~character~is~encoded~as~
1089   2~or~4~bytes: \\
1090   \iow_indent:n
1091   {
1092     Code~point~in~[U+0000,~U+D7FF]:~two~bytes \\
1093     Code~point~in~[U+D800,~U+DFFF]:~illegal \\
1094     Code~point~in~[U+E000,~U+FFFF]:~two~bytes \\
1095     Code~point~in~[U+10000,~U+10FFFF]:~
1096     a~lead~surrogate~and~a~trail~surrogate \\
1097   }
1098   Lead~surrogates~are~pairs~of~bytes~in~the~range~[0xD800,~0xDBFF],~
1099   and~trail~surrogates~are~in~the~range~[0xDC00,~0xDFFF] .
1100   \flag_if_raised:nT { str_missing }
1101   {
1102     \\\
1103     A~lead~surrogate~was~not~followed~by~a~trail~surrogate.
1104   }
1105   \flag_if_raised:nT { str_extra }
1106   {
1107     \\\
1108     LaTeX~came~across~a~trail~surrogate~when~it~was~not~expected.
1109   }
1110   \flag_if_raised:nT { str_end }
1111   {
1112     \\\
1113     The~string~contained~an~odd~number~of~bytes.~This~is~invalid:~
1114     the~basic~code~unit~for~UTF-16~is~16~bits~(2~bytes).
1115   }
1116 }

```

(End definition for \l__str_missing_flag, \l__str_extra_flag, and \l__str_end_flag. These variables are documented on page ??.)

_str_convert_decode_utf16: As for UTF-8, decoding UTF-16 is harder than encoding it. If the endianness is unknown, check the first two bytes: if those are "FE and "FF in either order, remove them and use the corresponding endianness, otherwise assume big-endianness. The three endianness cases are based on a common auxiliary whose first argument is 1 for big-endian and 2 for little-endian, and whose second argument, delimited by the scan mark \s__stop, is expanded once (the string may be long; passing \g__str_result_tl as an argument before expansion is cheaper).

The _str_decode_utf_xvi:Nw function defines _str_tmp:w to take two arguments and return the character code of the first one if the string is big-endian, and the second one if the string is little-endian, then loops over the string using _str_decode_utf_xvi_pair:NN described below.

```

1117 \cs_new_protected_nopar:cpn { __str_convert_decode_utf16be: }
1118 { \__str_decode_utf_xvi:Nw 1 \g__str_result_tl \s__stop }
1119 \cs_new_protected_nopar:cpn { __str_convert_decode_utf16le: }
1120 { \__str_decode_utf_xvi:Nw 2 \g__str_result_tl \s__stop }
1121 \cs_new_protected_nopar:cpn { __str_convert_decode_utf16: }
1122 {
1123   \exp_after:wN \__str_decode_utf_xvi_bom:NN
1124   \g__str_result_tl \s__stop \s__stop \s__stop
1125 }
1126 \cs_new_protected:Npn \__str_decode_utf_xvi_bom:NN #1#2
1127 {
1128   \str_if_eq_x:nnTF { #1#2 } { ^^ff ^^fe }
1129   { \__str_decode_utf_xvi:Nw 2 }
1130   {
1131     \str_if_eq_x:nnTF { #1#2 } { ^^fe ^^ff }
1132     { \__str_decode_utf_xvi:Nw 1 }
1133     { \__str_decode_utf_xvi:Nw 1 #1#2 }
1134   }
1135 }
1136 \cs_new_protected:Npn \__str_decode_utf_xvi:Nw #1#2 \s__stop
1137 {
1138   \flag_clear:n { str_error }
1139   \flag_clear:n { str_missing }
1140   \flag_clear:n { str_extra }
1141   \flag_clear:n { str_end }
1142   \cs_set:Npn \__str_tmp:w ##1 ##2 { ' ## #1 }
1143   \tl_gset:Nx \g__str_result_tl
1144   {
1145     \exp_after:wN \__str_decode_utf_xvi_pair:NN
1146     #2 \q_nil \q_nil
1147     \__prg_break_point:
1148   }
1149   \__str_if_flag_error:nnx { str_error } { utf16-decode } { }
1150 }

```

(End definition for `__str_convert_decode_utf16:`, `__str_convert_decode_utf16be:`, and `__str_convert_decode_utf16le:`. These functions are documented on page ??.)

```

\__str_decode_utf_xvi_pair:NN
\__str_decode_utf_xvi_quad:NNwNN
\__str_decode_utf_xvi_pair_end:Nw
\__str_decode_utf_xvi_error:nnN
\__str_decode_utf_xvi_extra:NNw

```

Bytes are read two at a time. At this stage, `\@@_tmp:w #1#2` expands to the character code of the most significant byte, and we distinguish cases depending on which range it lies in:

- ["D8,"DB] signals a lead surrogate, and the integer expression yields 1 (ε -TeX rounds ties away from zero);
- ["DC,"DF] signals a trail surrogate, unexpected here, and the integer expression yields 2;
- any other value signals a code point in the Basic Multilingual Plane, which stands for itself, and the `\if_case:w` construction expands to nothing (cases other than 1 or 2), leaving the relevant material in the input stream, followed by another call to the `_pair` auxiliary.

The case of a lead surrogate is treated by the `_quad` auxiliary, whose arguments `#1`, `#2`, `#4` and `#5` are the four bytes. We expect the most significant byte of `#4#5` to be in the range `["DC", "DF"]` (trail surrogate). The test is similar to the test used for continuation bytes in the UTF-8 decoding functions. In the case where `#4#5` is indeed a trail surrogate, leave `#1#2#4#5 \s__tl <code point> \s__tl`, and remove the pair `#4#5` before looping with `__str_decode_utf_xvi_pair:NN`. Otherwise, of course, complain about the missing surrogate.

The magic number "D7F7 is such that $"D7F7" * 400 = "D800" * 400 + "DC00" - "10000$.

Every time we read a pair of bytes, we test for the end-marker `\q_nil`. When reaching the end, we additionally check that the string had an even length. Also, if the end is reached when expecting a trail surrogate, we treat that as a missing surrogate.

```

1151 \cs_new:Npn \__str_decode_utf_xvi_pair:NN #1#2
1152 {
1153   \if_meaning:w \q_nil #2
1154     \__str_decode_utf_xvi_pair_end:Nw #1
1155   \fi:
1156   \if_case:w
1157     \__int_eval:w ( \__str_tmp:w #1#2 - "D6 ) / \c_four \__int_eval_end:
1158   \or: \exp_after:wN \__str_decode_utf_xvi_quad:NNwNN
1159   \or: \exp_after:wN \__str_decode_utf_xvi_extra:NNw
1160   \fi:
1161   #1#2 \s__tl
1162   \int_eval:n { "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 } \s__tl
1163   \__str_decode_utf_xvi_pair:NN
1164 }
1165 \cs_new:Npn \__str_decode_utf_xvi_quad:NNwNN
1166   #1#2 #3 \__str_decode_utf_xvi_pair:NN #4#5
1167 {
1168   \if_meaning:w \q_nil #5
1169     \__str_decode_utf_xvi_error:nnn { missing } #1#2
1170     \__str_decode_utf_xvi_pair_end:Nw #4
1171   \fi:
1172   \if_int_compare:w
1173     \if_int_compare:w \__str_tmp:w #4#5 < "DC \exp_stop_f:
1174       \c_zero = \c_one
1175     \else:
1176       \__str_tmp:w #4#5 < "E0 \exp_stop_f:
1177     \fi:
1178   #1 #2 #4 #5 \s__tl
1179   \int_eval:n
1180   {
1181     ( "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 - "D7F7 ) * "400
1182     + "100 * \__str_tmp:w #4#5 + \__str_tmp:w #5#4
1183   }
1184   \s__tl
1185   \exp_after:wN \use_i:nnn
1186 \else:
1187   \__str_decode_utf_xvi_error:nnn { missing } #1#2

```

```

1188     \fi:
1189     \__str_decode_utf_xvi_pair:NN #4#5
1190   }
1191   \cs_new:Npn \__str_decode_utf_xvi_pair_end:Nw #1 \fi:
1192   {
1193     \fi:
1194     \if_meaning:w \q_nil #1
1195     \else:
1196       \__str_decode_utf_xvi_error:nNN { end } #1 \prg_do_nothing:
1197     \fi:
1198     \__prg_break:
1199   }
1200   \cs_new:Npn \__str_decode_utf_xvi_extra:NNw #1#2 \s__tl #3 \s__tl
1201   { \__str_decode_utf_xvi_error:nNN { extra } #1#2 }
1202   \cs_new:Npn \__str_decode_utf_xvi_error:nNN #1#2#3
1203   {
1204     \flag_raise:n { str_error }
1205     \flag_raise:n { str_#1 }
1206     #2 #3 \s__tl
1207     \int_use:N \c__str_replacement_char_int \s__tl
1208   }

```

(End definition for `__str_decode_utf_xvi_pair:NN`, `__str_decode_utf_xvi_quad:NNwNN`, and `__str_decode_utf_xvi_pair_end:Nw`. These functions are documented on page ??.)

Restore the original catcodes of bytes 254 and 255.

```

1209 \group_end:
1210 </utf16>

```

5.6.3 utf-32 support

The definitions are done in a category code regime where the bytes 0, 254 and 255 used by the byte order mark have catcode “other”.

```

1211 <utf32>
1212 \group_begin:
1213   \char_set_catcode_other:N ^^00
1214   \char_set_catcode_other:N ^^fe
1215   \char_set_catcode_other:N ^^ff

```

`__str_convert_encode_utf32:` Convert each integer in the comma-list `\g__str_result_tl` to a sequence of four bytes. The functions for big-endian and little-endian encodings are very similar, but the `__str_output_byte:n` instructions are reversed.

```

\__str_convert_encode_utf32be:
\__str_convert_encode_utf32le:
\__str_encode_utf_xxxii_be:n
\__str_encode_utf_xxxii_be_aux:nn
\__str_encode_utf_xxxii_le:n
\__str_encode_utf_xxxii_le_aux:nn

```

```

1216   \cs_new_protected_nopar:cpn { __str_convert_encode_utf32: }
1217   {
1218     \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_be:n
1219     \tl_gput_left:Nx \g__str_result_tl { ^^00 ^^00 ^^fe ^^ff }
1220   }
1221   \cs_new_protected_nopar:cpn { __str_convert_encode_utf32be: }
1222   { \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_be:n }
1223   \cs_new_protected_nopar:cpn { __str_convert_encode_utf32le: }
1224   { \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_le:n }

```

```

1225 \cs_new:Npn \__str_encode_utf_xxxii_be:n #1
1226 {
1227   \exp_args:Nf \__str_encode_utf_xxxii_be_aux:nn
1228   { \int_div_truncate:nn {#1} { "100 } } {#1}
1229 }
1230 \cs_new:Npn \__str_encode_utf_xxxii_be_aux:nn #1#2
1231 {
1232   ^^00
1233   \__str_output_byte_pair_be:n {#1}
1234   \__str_output_byte:n { #2 - #1 * "100 }
1235 }
1236 \cs_new:Npn \__str_encode_utf_xxxii_le:n #1
1237 {
1238   \exp_args:Nf \__str_encode_utf_xxxii_le_aux:nn
1239   { \int_div_truncate:nn {#1} { "100 } } {#1}
1240 }
1241 \cs_new:Npn \__str_encode_utf_xxxii_le_aux:nn #1#2
1242 {
1243   \__str_output_byte:n { #2 - #1 * "100 }
1244   \__str_output_byte_pair_le:n {#1}
1245   ^^00
1246 }

```

(End definition for `__str_convert_encode_utf32:`, `__str_convert_encode_utf32be:`, and `__str_convert_encode_utf32l:`. These functions are documented on page ??.)

str_overflow There can be no error when encoding in UTF-32. When decoding, the string may not
str_end have length $4n$, or it may contain code points larger than "10FFFF". The latter case often happens if the encoding was in fact not UTF-32, because most arbitrary strings are not valid in UTF-32.

```

1247 \flag_clear_new:n { str_overflow }
1248 \flag_clear_new:n { str_end }
1249 \__msg_kernel_new:nnnn { str } { utf32-decode }
1250 {
1251   Invalid~UTF-32~string: \exp_last_unbraced:Nf \use_none:n
1252   \__str_if_flag_times:nT { str_overflow } { ,~code~point~too~large }
1253   \__str_if_flag_times:nT { str_end } { ,~truncated~string }
1254   .
1255 }
1256 {
1257   In~the~UTF-32~encoding,~every~Unicode~character~
1258   (in~the~range~[U+0000,~U+10FFFF])~is~encoded~as~4~bytes.
1259   \flag_if_raised:nT { str_overflow }
1260   {
1261     \\\
1262     LaTeX~came~across~a~code~point~larger~than~1114111,~
1263     the~maximum~code~point~defined~by~Unicode.~
1264     Perhaps~the~string~was~not~encoded~in~the~UTF-32~encoding?
1265   }
1266   \flag_if_raised:nT { str_end }

```

```

1267     {
1268         \\\
1269         The~length~of~the~string~is~not~a~multiple~of~4.~
1270         Perhaps~the~string~was~truncated?
1271     }
1272 }

```

(End definition for `str_overflow` and `str_end`. These variables are documented on page ??.)

`__str_convert_decode_utf32:` The structure is similar to UTF-16 decoding functions. If the endianness is not given, `_str_convert_decode_utf32be:` test the first 4 bytes of the string (possibly `\s__stop` if the string is too short) for the presence of a byte-order mark. If there is a byte-order mark, use that endianness, and remove the 4 bytes, otherwise default to big-endian, and leave the 4 bytes in place. The `_str_convert_decode_utf32le:` `_str_decode_utf_xxxii_bom:NNNN` and `_str_decode_utf_xxxii:Nw` auxiliary receives 1 or 2 as its first argument indicating endianness, and the string to convert as its second argument (expanded or not). It sets `_str_decode_utf_xxxii_loop:NNNN` `_str_decode_utf_xxxii_end:w` `_str_decode_utf_xxxii:Nw` auxiliary receives 1 or 2 as its first argument indicating endianness, and the string to convert as its second argument (expanded or not). It sets `_str_tmp:w` to expand to the character code of either of its two arguments depending on endianness, then triggers the `_loop` auxiliary inside an `x`-expanding assignment to `\g__str_result_tl`.

The `_loop` auxiliary first checks for the end-of-string marker `\s__stop`, calling the `_end` auxiliary if appropriate. Otherwise, leave the $\langle 4 \text{ bytes} \rangle$ `\s__tl` behind, then check that the code point is not overflowing: the leading byte must be 0, and the following byte at most 16.

In the ending code, we check that there remains no byte: there should be nothing left until the first `\s__stop`. Break the map.

```

1273 \cs_new_protected_nopar:cpn { __str_convert_decode_utf32be: }
1274 { \_str_decode_utf_xxxii:Nw 1 \g__str_result_tl \s__stop }
1275 \cs_new_protected_nopar:cpn { __str_convert_decode_utf32le: }
1276 { \_str_decode_utf_xxxii:Nw 2 \g__str_result_tl \s__stop }
1277 \cs_new_protected_nopar:cpn { __str_convert_decode_utf32: }
1278 {
1279     \exp_after:wN \_str_decode_utf_xxxii_bom:NNNN \g__str_result_tl
1280     \s__stop \s__stop \s__stop \s__stop \s__stop
1281 }
1282 \cs_new_protected:Npn \_str_decode_utf_xxxii_bom:NNNN #1#2#3#4
1283 {
1284     \str_if_eq_x:nnTF { #1#2#3#4 } { ^^ff ^^fe ^^00 ^^00 }
1285     { \_str_decode_utf_xxxii:Nw 2 }
1286     {
1287         \str_if_eq_x:nnTF { #1#2#3#4 } { ^^00 ^^00 ^^fe ^^ff }
1288         { \_str_decode_utf_xxxii:Nw 1 }
1289         { \_str_decode_utf_xxxii:Nw 1 #1#2#3#4 }
1290     }
1291 }
1292 \cs_new_protected:Npn \_str_decode_utf_xxxii:Nw #1#2 \s__stop
1293 {
1294     \flag_clear:n { str_overflow }
1295     \flag_clear:n { str_end }
1296     \flag_clear:n { str_error }
1297     \cs_set:Npn \_str_tmp:w ##1 ##2 { ' ## #1 }

```

```

1298     \tl_gset:Nx \g__str_result_tl
1299     {
1300         \exp_after:wN \__str_decode_utf_xxxii_loop:NNNN
1301         #2 \s__stop \s__stop \s__stop \s__stop
1302         \__prg_break_point:
1303     }
1304     \__str_if_flag_error:nxx { str_error } { utf32-decode } { }
1305 }
1306 \cs_new:Npn \__str_decode_utf_xxxii_loop:NNNN #1#2#3#4
1307 {
1308     \if_meaning:w \s__stop #4
1309     \exp_after:wN \__str_decode_utf_xxxii_end:w
1310     \fi:
1311     #1#2#3#4 \s__tl
1312     \if_int_compare:w \__str_tmp:w #1#4 > \c_zero
1313         \flag_raise:n { str_overflow }
1314         \flag_raise:n { str_error }
1315         \int_use:N \c__str_replacement_char_int
1316     \else:
1317         \if_int_compare:w \__str_tmp:w #2#3 > \c_sixteen
1318             \flag_raise:n { str_overflow }
1319             \flag_raise:n { str_error }
1320             \int_use:N \c__str_replacement_char_int
1321         \else:
1322             \int_eval:n
1323             { \__str_tmp:w #2#3*"10000 + \__str_tmp:w #3#2*"100 + \__str_tmp:w #4#1 }
1324         \fi:
1325     \fi:
1326     \s__tl
1327     \__str_decode_utf_xxxii_loop:NNNN
1328 }
1329 \cs_new:Npn \__str_decode_utf_xxxii_end:w #1 \s__stop
1330 {
1331     \tl_if_empty:nF {#1}
1332     {
1333         \flag_raise:n { str_end }
1334         \flag_raise:n { str_error }
1335         #1 \s__tl
1336         \int_use:N \c__str_replacement_char_int \s__tl
1337     }
1338     \__prg_break:
1339 }

```

(End definition for __str_convert_decode_utf32:, __str_convert_decode_utf32be:, and __str_convert_decode_utf32l:.)
These functions are documented on page ??.)

Restore the original catcodes of bytes 0, 254 and 255.

```

1340 \group_end:
1341 </utf32>

```

5.6.4 iso 8859 support

The ISO-8859-1 encoding exactly matches with the 256 first Unicode characters. For other 8-bit encodings of the ISO-8859 family, we keep track only of differences, and of unassigned bytes.

```
1342 <*iso88591>
1343 \__str_declare_eight_bit_encoding:nnn { iso88591 }
1344 {
1345 }
1346 {
1347 }
1348 </iso88591>
1349 <*iso88592>
1350 \__str_declare_eight_bit_encoding:nnn { iso88592 }
1351 {
1352   { A1 } { 0104 }
1353   { A2 } { 02D8 }
1354   { A3 } { 0141 }
1355   { A5 } { 013D }
1356   { A6 } { 015A }
1357   { A9 } { 0160 }
1358   { AA } { 015E }
1359   { AB } { 0164 }
1360   { AC } { 0179 }
1361   { AE } { 017D }
1362   { AF } { 017B }
1363   { B1 } { 0105 }
1364   { B2 } { 02DB }
1365   { B3 } { 0142 }
1366   { B5 } { 013E }
1367   { B6 } { 015B }
1368   { B7 } { 02C7 }
1369   { B9 } { 0161 }
1370   { BA } { 015F }
1371   { BB } { 0165 }
1372   { BC } { 017A }
1373   { BD } { 02DD }
1374   { BE } { 017E }
1375   { BF } { 017C }
1376   { C0 } { 0154 }
1377   { C3 } { 0102 }
1378   { C5 } { 0139 }
1379   { C6 } { 0106 }
1380   { C8 } { 010C }
1381   { CA } { 0118 }
1382   { CC } { 011A }
1383   { CF } { 010E }
1384   { D0 } { 0110 }
1385   { D1 } { 0143 }
```

```

1386     { D2 } { 0147 }
1387     { D5 } { 0150 }
1388     { D8 } { 0158 }
1389     { D9 } { 016E }
1390     { DB } { 0170 }
1391     { DE } { 0162 }
1392     { E0 } { 0155 }
1393     { E3 } { 0103 }
1394     { E5 } { 013A }
1395     { E6 } { 0107 }
1396     { E8 } { 010D }
1397     { EA } { 0119 }
1398     { EC } { 011B }
1399     { EF } { 010F }
1400     { F0 } { 0111 }
1401     { F1 } { 0144 }
1402     { F2 } { 0148 }
1403     { F5 } { 0151 }
1404     { F8 } { 0159 }
1405     { F9 } { 016F }
1406     { FB } { 0171 }
1407     { FE } { 0163 }
1408     { FF } { 02D9 }
1409 }
1410 {
1411 }
1412 </iso88592>
1413 <*iso88593>
1414 \__str_declare_eight_bit_encoding:nnn { iso88593 }
1415 {
1416     { A1 } { 0126 }
1417     { A2 } { 02D8 }
1418     { A6 } { 0124 }
1419     { A9 } { 0130 }
1420     { AA } { 015E }
1421     { AB } { 011E }
1422     { AC } { 0134 }
1423     { AF } { 017B }
1424     { B1 } { 0127 }
1425     { B6 } { 0125 }
1426     { B9 } { 0131 }
1427     { BA } { 015F }
1428     { BB } { 011F }
1429     { BC } { 0135 }
1430     { BF } { 017C }
1431     { C5 } { 010A }
1432     { C6 } { 0108 }
1433     { D5 } { 0120 }
1434     { D8 } { 011C }

```

```

1435     { DD } { 016C }
1436     { DE } { 015C }
1437     { E5 } { 010B }
1438     { E6 } { 0109 }
1439     { F5 } { 0121 }
1440     { F8 } { 011D }
1441     { FD } { 016D }
1442     { FE } { 015D }
1443     { FF } { 02D9 }
1444 }
1445 {
1446     { A5 }
1447     { AE }
1448     { BE }
1449     { C3 }
1450     { D0 }
1451     { E3 }
1452     { F0 }
1453 }
1454 </iso88593>
1455 <*iso88594>
1456 \__str_declare_eight_bit_encoding:nnn { iso88594 }
1457 {
1458     { A1 } { 0104 }
1459     { A2 } { 0138 }
1460     { A3 } { 0156 }
1461     { A5 } { 0128 }
1462     { A6 } { 013B }
1463     { A9 } { 0160 }
1464     { AA } { 0112 }
1465     { AB } { 0122 }
1466     { AC } { 0166 }
1467     { AE } { 017D }
1468     { B1 } { 0105 }
1469     { B2 } { 02DB }
1470     { B3 } { 0157 }
1471     { B5 } { 0129 }
1472     { B6 } { 013C }
1473     { B7 } { 02C7 }
1474     { B9 } { 0161 }
1475     { BA } { 0113 }
1476     { BB } { 0123 }
1477     { BC } { 0167 }
1478     { BD } { 014A }
1479     { BE } { 017E }
1480     { BF } { 014B }
1481     { C0 } { 0100 }
1482     { C7 } { 012E }
1483     { C8 } { 010C }

```



```

1484 { CA } { 0118 }
1485 { CC } { 0116 }
1486 { CF } { 012A }
1487 { D0 } { 0110 }
1488 { D1 } { 0145 }
1489 { D2 } { 014C }
1490 { D3 } { 0136 }
1491 { D9 } { 0172 }
1492 { DD } { 0168 }
1493 { DE } { 016A }
1494 { E0 } { 0101 }
1495 { E7 } { 012F }
1496 { E8 } { 010D }
1497 { EA } { 0119 }
1498 { EC } { 0117 }
1499 { EF } { 012B }
1500 { F0 } { 0111 }
1501 { F1 } { 0146 }
1502 { F2 } { 014D }
1503 { F3 } { 0137 }
1504 { F9 } { 0173 }
1505 { FD } { 0169 }
1506 { FE } { 016B }
1507 { FF } { 02D9 }
1508 }
1509 {
1510 }
1511 </iso88594>
1512 <*iso88595>
1513 \__str_declare_eight_bit_encoding:nnn { iso88595 }
1514 {
1515 { A1 } { 0401 }
1516 { A2 } { 0402 }
1517 { A3 } { 0403 }
1518 { A4 } { 0404 }
1519 { A5 } { 0405 }
1520 { A6 } { 0406 }
1521 { A7 } { 0407 }
1522 { A8 } { 0408 }
1523 { A9 } { 0409 }
1524 { AA } { 040A }
1525 { AB } { 040B }
1526 { AC } { 040C }
1527 { AE } { 040E }
1528 { AF } { 040F }
1529 { B0 } { 0410 }
1530 { B1 } { 0411 }
1531 { B2 } { 0412 }
1532 { B3 } { 0413 }

```

1533	{ B4 }	{ 0414 }
1534	{ B5 }	{ 0415 }
1535	{ B6 }	{ 0416 }
1536	{ B7 }	{ 0417 }
1537	{ B8 }	{ 0418 }
1538	{ B9 }	{ 0419 }
1539	{ BA }	{ 041A }
1540	{ BB }	{ 041B }
1541	{ BC }	{ 041C }
1542	{ BD }	{ 041D }
1543	{ BE }	{ 041E }
1544	{ BF }	{ 041F }
1545	{ C0 }	{ 0420 }
1546	{ C1 }	{ 0421 }
1547	{ C2 }	{ 0422 }
1548	{ C3 }	{ 0423 }
1549	{ C4 }	{ 0424 }
1550	{ C5 }	{ 0425 }
1551	{ C6 }	{ 0426 }
1552	{ C7 }	{ 0427 }
1553	{ C8 }	{ 0428 }
1554	{ C9 }	{ 0429 }
1555	{ CA }	{ 042A }
1556	{ CB }	{ 042B }
1557	{ CC }	{ 042C }
1558	{ CD }	{ 042D }
1559	{ CE }	{ 042E }
1560	{ CF }	{ 042F }
1561	{ D0 }	{ 0430 }
1562	{ D1 }	{ 0431 }
1563	{ D2 }	{ 0432 }
1564	{ D3 }	{ 0433 }
1565	{ D4 }	{ 0434 }
1566	{ D5 }	{ 0435 }
1567	{ D6 }	{ 0436 }
1568	{ D7 }	{ 0437 }
1569	{ D8 }	{ 0438 }
1570	{ D9 }	{ 0439 }
1571	{ DA }	{ 043A }
1572	{ DB }	{ 043B }
1573	{ DC }	{ 043C }
1574	{ DD }	{ 043D }
1575	{ DE }	{ 043E }
1576	{ DF }	{ 043F }
1577	{ E0 }	{ 0440 }
1578	{ E1 }	{ 0441 }
1579	{ E2 }	{ 0442 }
1580	{ E3 }	{ 0443 }
1581	{ E4 }	{ 0444 }
1582	{ E5 }	{ 0445 }

```

1583     { E6 } { 0446 }
1584     { E7 } { 0447 }
1585     { E8 } { 0448 }
1586     { E9 } { 0449 }
1587     { EA } { 044A }
1588     { EB } { 044B }
1589     { EC } { 044C }
1590     { ED } { 044D }
1591     { EE } { 044E }
1592     { EF } { 044F }
1593     { F0 } { 2116 }
1594     { F1 } { 0451 }
1595     { F2 } { 0452 }
1596     { F3 } { 0453 }
1597     { F4 } { 0454 }
1598     { F5 } { 0455 }
1599     { F6 } { 0456 }
1600     { F7 } { 0457 }
1601     { F8 } { 0458 }
1602     { F9 } { 0459 }
1603     { FA } { 045A }
1604     { FB } { 045B }
1605     { FC } { 045C }
1606     { FD } { 00A7 }
1607     { FE } { 045E }
1608     { FF } { 045F }
1609 }
1610 {
1611 }
1612 </iso88595>
1613 <*iso88596>
1614 \__str_declare_eight_bit_encoding:nnn { iso88596 }
1615 {
1616     { AC } { 060C }
1617     { BB } { 061B }
1618     { BF } { 061F }
1619     { C1 } { 0621 }
1620     { C2 } { 0622 }
1621     { C3 } { 0623 }
1622     { C4 } { 0624 }
1623     { C5 } { 0625 }
1624     { C6 } { 0626 }
1625     { C7 } { 0627 }
1626     { C8 } { 0628 }
1627     { C9 } { 0629 }
1628     { CA } { 062A }
1629     { CB } { 062B }
1630     { CC } { 062C }
1631     { CD } { 062D }

```

```

1632 { CE } { 062E }
1633 { CF } { 062F }
1634 { D0 } { 0630 }
1635 { D1 } { 0631 }
1636 { D2 } { 0632 }
1637 { D3 } { 0633 }
1638 { D4 } { 0634 }
1639 { D5 } { 0635 }
1640 { D6 } { 0636 }
1641 { D7 } { 0637 }
1642 { D8 } { 0638 }
1643 { D9 } { 0639 }
1644 { DA } { 063A }
1645 { E0 } { 0640 }
1646 { E1 } { 0641 }
1647 { E2 } { 0642 }
1648 { E3 } { 0643 }
1649 { E4 } { 0644 }
1650 { E5 } { 0645 }
1651 { E6 } { 0646 }
1652 { E7 } { 0647 }
1653 { E8 } { 0648 }
1654 { E9 } { 0649 }
1655 { EA } { 064A }
1656 { EB } { 064B }
1657 { EC } { 064C }
1658 { ED } { 064D }
1659 { EE } { 064E }
1660 { EF } { 064F }
1661 { F0 } { 0650 }
1662 { F1 } { 0651 }
1663 { F2 } { 0652 }
1664 }
1665 {
1666 { A1 }
1667 { A2 }
1668 { A3 }
1669 { A5 }
1670 { A6 }
1671 { A7 }
1672 { A8 }
1673 { A9 }
1674 { AA }
1675 { AB }
1676 { AE }
1677 { AF }
1678 { B0 }
1679 { B1 }
1680 { B2 }
1681 { B3 }

```

```

1682     { B4 }
1683     { B5 }
1684     { B6 }
1685     { B7 }
1686     { B8 }
1687     { B9 }
1688     { BA }
1689     { BC }
1690     { BD }
1691     { BE }
1692     { C0 }
1693     { DB }
1694     { DC }
1695     { DD }
1696     { DE }
1697     { DF }
1698   }
1699   </iso88596>
1700   <*iso88597>
1701   \_str_declare_eight_bit_encoding:nmm { iso88597 }
1702   {
1703     { A1 } { 2018 }
1704     { A2 } { 2019 }
1705     { A4 } { 20AC }
1706     { A5 } { 20AF }
1707     { AA } { 037A }
1708     { AF } { 2015 }
1709     { B4 } { 0384 }
1710     { B5 } { 0385 }
1711     { B6 } { 0386 }
1712     { B8 } { 0388 }
1713     { B9 } { 0389 }
1714     { BA } { 038A }
1715     { BC } { 038C }
1716     { BE } { 038E }
1717     { BF } { 038F }
1718     { C0 } { 0390 }
1719     { C1 } { 0391 }
1720     { C2 } { 0392 }
1721     { C3 } { 0393 }
1722     { C4 } { 0394 }
1723     { C5 } { 0395 }
1724     { C6 } { 0396 }
1725     { C7 } { 0397 }
1726     { C8 } { 0398 }
1727     { C9 } { 0399 }
1728     { CA } { 039A }
1729     { CB } { 039B }
1730     { CC } { 039C }

```

```

1731 { CD } { 039D }
1732 { CE } { 039E }
1733 { CF } { 039F }
1734 { D0 } { 03A0 }
1735 { D1 } { 03A1 }
1736 { D3 } { 03A3 }
1737 { D4 } { 03A4 }
1738 { D5 } { 03A5 }
1739 { D6 } { 03A6 }
1740 { D7 } { 03A7 }
1741 { D8 } { 03A8 }
1742 { D9 } { 03A9 }
1743 { DA } { 03AA }
1744 { DB } { 03AB }
1745 { DC } { 03AC }
1746 { DD } { 03AD }
1747 { DE } { 03AE }
1748 { DF } { 03AF }
1749 { E0 } { 03B0 }
1750 { E1 } { 03B1 }
1751 { E2 } { 03B2 }
1752 { E3 } { 03B3 }
1753 { E4 } { 03B4 }
1754 { E5 } { 03B5 }
1755 { E6 } { 03B6 }
1756 { E7 } { 03B7 }
1757 { E8 } { 03B8 }
1758 { E9 } { 03B9 }
1759 { EA } { 03BA }
1760 { EB } { 03BB }
1761 { EC } { 03BC }
1762 { ED } { 03BD }
1763 { EE } { 03BE }
1764 { EF } { 03BF }
1765 { F0 } { 03C0 }
1766 { F1 } { 03C1 }
1767 { F2 } { 03C2 }
1768 { F3 } { 03C3 }
1769 { F4 } { 03C4 }
1770 { F5 } { 03C5 }
1771 { F6 } { 03C6 }
1772 { F7 } { 03C7 }
1773 { F8 } { 03C8 }
1774 { F9 } { 03C9 }
1775 { FA } { 03CA }
1776 { FB } { 03CB }
1777 { FC } { 03CC }
1778 { FD } { 03CD }
1779 { FE } { 03CE }
1780 }

```

```

1781     {
1782         { AE }
1783         { D2 }
1784     }
1785 </iso88597>
1786 <*iso88598>
1787 \_str_declare_eight_bit_encoding:nnn { iso88598 }
1788     {
1789         { AA } { 00D7 }
1790         { BA } { 00F7 }
1791         { DF } { 2017 }
1792         { E0 } { 05D0 }
1793         { E1 } { 05D1 }
1794         { E2 } { 05D2 }
1795         { E3 } { 05D3 }
1796         { E4 } { 05D4 }
1797         { E5 } { 05D5 }
1798         { E6 } { 05D6 }
1799         { E7 } { 05D7 }
1800         { E8 } { 05D8 }
1801         { E9 } { 05D9 }
1802         { EA } { 05DA }
1803         { EB } { 05DB }
1804         { EC } { 05DC }
1805         { ED } { 05DD }
1806         { EE } { 05DE }
1807         { EF } { 05DF }
1808         { F0 } { 05E0 }
1809         { F1 } { 05E1 }
1810         { F2 } { 05E2 }
1811         { F3 } { 05E3 }
1812         { F4 } { 05E4 }
1813         { F5 } { 05E5 }
1814         { F6 } { 05E6 }
1815         { F7 } { 05E7 }
1816         { F8 } { 05E8 }
1817         { F9 } { 05E9 }
1818         { FA } { 05EA }
1819         { FD } { 200E }
1820         { FE } { 200F }
1821     }
1822     {
1823         { A1 }
1824         { BF }
1825         { C0 }
1826         { C1 }
1827         { C2 }
1828         { C3 }
1829         { C4 }

```

```

1830     { C5 }
1831     { C6 }
1832     { C7 }
1833     { C8 }
1834     { C9 }
1835     { CA }
1836     { CB }
1837     { CC }
1838     { CD }
1839     { CE }
1840     { CF }
1841     { D0 }
1842     { D1 }
1843     { D2 }
1844     { D3 }
1845     { D4 }
1846     { D5 }
1847     { D6 }
1848     { D7 }
1849     { D8 }
1850     { D9 }
1851     { DA }
1852     { DB }
1853     { DC }
1854     { DD }
1855     { DE }
1856     { FB }
1857     { FC }
1858 }
1859 </iso88598>
1860 <*iso88599>
1861 \_str_declare\_eight\_bit\_encoding:nnn { iso88599 }
1862 {
1863     { D0 } { 011E }
1864     { DD } { 0130 }
1865     { DE } { 015E }
1866     { F0 } { 011F }
1867     { FD } { 0131 }
1868     { FE } { 015F }
1869 }
1870 {
1871 }
1872 </iso88599>
1873 <*iso885910>
1874 \_str_declare\_eight\_bit\_encoding:nnn { iso885910 }
1875 {
1876     { A1 } { 0104 }
1877     { A2 } { 0112 }
1878     { A3 } { 0122 }

```



```

1879     { A4 } { 012A }
1880     { A5 } { 0128 }
1881     { A6 } { 0136 }
1882     { A8 } { 013B }
1883     { A9 } { 0110 }
1884     { AA } { 0160 }
1885     { AB } { 0166 }
1886     { AC } { 017D }
1887     { AE } { 016A }
1888     { AF } { 014A }
1889     { B1 } { 0105 }
1890     { B2 } { 0113 }
1891     { B3 } { 0123 }
1892     { B4 } { 012B }
1893     { B5 } { 0129 }
1894     { B6 } { 0137 }
1895     { B8 } { 013C }
1896     { B9 } { 0111 }
1897     { BA } { 0161 }
1898     { BB } { 0167 }
1899     { BC } { 017E }
1900     { BD } { 2015 }
1901     { BE } { 016B }
1902     { BF } { 014B }
1903     { C0 } { 0100 }
1904     { C7 } { 012E }
1905     { C8 } { 010C }
1906     { CA } { 0118 }
1907     { CC } { 0116 }
1908     { D1 } { 0145 }
1909     { D2 } { 014C }
1910     { D7 } { 0168 }
1911     { D9 } { 0172 }
1912     { E0 } { 0101 }
1913     { E7 } { 012F }
1914     { E8 } { 010D }
1915     { EA } { 0119 }
1916     { EC } { 0117 }
1917     { F1 } { 0146 }
1918     { F2 } { 014D }
1919     { F7 } { 0169 }
1920     { F9 } { 0173 }
1921     { FF } { 0138 }
1922 }
1923 {
1924 }
1925 </iso885910>
1926 <*iso885911>
1927 \__str_declare_eight_bit_encoding:nnn { iso885911 }

```

```

1928 {
1929 { A1 } { 0E01 }
1930 { A2 } { 0E02 }
1931 { A3 } { 0E03 }
1932 { A4 } { 0E04 }
1933 { A5 } { 0E05 }
1934 { A6 } { 0E06 }
1935 { A7 } { 0E07 }
1936 { A8 } { 0E08 }
1937 { A9 } { 0E09 }
1938 { AA } { 0E0A }
1939 { AB } { 0E0B }
1940 { AC } { 0E0C }
1941 { AD } { 0E0D }
1942 { AE } { 0E0E }
1943 { AF } { 0E0F }
1944 { B0 } { 0E10 }
1945 { B1 } { 0E11 }
1946 { B2 } { 0E12 }
1947 { B3 } { 0E13 }
1948 { B4 } { 0E14 }
1949 { B5 } { 0E15 }
1950 { B6 } { 0E16 }
1951 { B7 } { 0E17 }
1952 { B8 } { 0E18 }
1953 { B9 } { 0E19 }
1954 { BA } { 0E1A }
1955 { BB } { 0E1B }
1956 { BC } { 0E1C }
1957 { BD } { 0E1D }
1958 { BE } { 0E1E }
1959 { BF } { 0E1F }
1960 { C0 } { 0E20 }
1961 { C1 } { 0E21 }
1962 { C2 } { 0E22 }
1963 { C3 } { 0E23 }
1964 { C4 } { 0E24 }
1965 { C5 } { 0E25 }
1966 { C6 } { 0E26 }
1967 { C7 } { 0E27 }
1968 { C8 } { 0E28 }
1969 { C9 } { 0E29 }
1970 { CA } { 0E2A }
1971 { CB } { 0E2B }
1972 { CC } { 0E2C }
1973 { CD } { 0E2D }
1974 { CE } { 0E2E }
1975 { CF } { 0E2F }
1976 { D0 } { 0E30 }
1977 { D1 } { 0E31 }

```

```

1978     { D2 } { 0E32 }
1979     { D3 } { 0E33 }
1980     { D4 } { 0E34 }
1981     { D5 } { 0E35 }
1982     { D6 } { 0E36 }
1983     { D7 } { 0E37 }
1984     { D8 } { 0E38 }
1985     { D9 } { 0E39 }
1986     { DA } { 0E3A }
1987     { DF } { 0E3F }
1988     { E0 } { 0E40 }
1989     { E1 } { 0E41 }
1990     { E2 } { 0E42 }
1991     { E3 } { 0E43 }
1992     { E4 } { 0E44 }
1993     { E5 } { 0E45 }
1994     { E6 } { 0E46 }
1995     { E7 } { 0E47 }
1996     { E8 } { 0E48 }
1997     { E9 } { 0E49 }
1998     { EA } { 0E4A }
1999     { EB } { 0E4B }
2000     { EC } { 0E4C }
2001     { ED } { 0E4D }
2002     { EE } { 0E4E }
2003     { EF } { 0E4F }
2004     { F0 } { 0E50 }
2005     { F1 } { 0E51 }
2006     { F2 } { 0E52 }
2007     { F3 } { 0E53 }
2008     { F4 } { 0E54 }
2009     { F5 } { 0E55 }
2010     { F6 } { 0E56 }
2011     { F7 } { 0E57 }
2012     { F8 } { 0E58 }
2013     { F9 } { 0E59 }
2014     { FA } { 0E5A }
2015     { FB } { 0E5B }
2016     }
2017     {
2018         { DB }
2019         { DC }
2020         { DD }
2021         { DE }
2022     }
2023     </iso885911>
2024     <*iso885913>
2025     \__str_declare_eight_bit_encoding:nmm { iso885913 }
2026     {

```

2027	{ A1 }	{ 201D }
2028	{ A5 }	{ 201E }
2029	{ A8 }	{ 00D8 }
2030	{ AA }	{ 0156 }
2031	{ AF }	{ 00C6 }
2032	{ B4 }	{ 201C }
2033	{ B8 }	{ 00F8 }
2034	{ BA }	{ 0157 }
2035	{ BF }	{ 00E6 }
2036	{ C0 }	{ 0104 }
2037	{ C1 }	{ 012E }
2038	{ C2 }	{ 0100 }
2039	{ C3 }	{ 0106 }
2040	{ C6 }	{ 0118 }
2041	{ C7 }	{ 0112 }
2042	{ C8 }	{ 010C }
2043	{ CA }	{ 0179 }
2044	{ CB }	{ 0116 }
2045	{ CC }	{ 0122 }
2046	{ CD }	{ 0136 }
2047	{ CE }	{ 012A }
2048	{ CF }	{ 013B }
2049	{ D0 }	{ 0160 }
2050	{ D1 }	{ 0143 }
2051	{ D2 }	{ 0145 }
2052	{ D4 }	{ 014C }
2053	{ D8 }	{ 0172 }
2054	{ D9 }	{ 0141 }
2055	{ DA }	{ 015A }
2056	{ DB }	{ 016A }
2057	{ DD }	{ 017B }
2058	{ DE }	{ 017D }
2059	{ E0 }	{ 0105 }
2060	{ E1 }	{ 012F }
2061	{ E2 }	{ 0101 }
2062	{ E3 }	{ 0107 }
2063	{ E6 }	{ 0119 }
2064	{ E7 }	{ 0113 }
2065	{ E8 }	{ 010D }
2066	{ EA }	{ 017A }
2067	{ EB }	{ 0117 }
2068	{ EC }	{ 0123 }
2069	{ ED }	{ 0137 }
2070	{ EE }	{ 012B }
2071	{ EF }	{ 013C }
2072	{ F0 }	{ 0161 }
2073	{ F1 }	{ 0144 }
2074	{ F2 }	{ 0146 }
2075	{ F4 }	{ 014D }
2076	{ F8 }	{ 0173 }

```

2077     { F9 } { 0142 }
2078     { FA } { 015B }
2079     { FB } { 016B }
2080     { FD } { 017C }
2081     { FE } { 017E }
2082     { FF } { 2019 }
2083 }
2084 {
2085 }
2086 </iso885913>
2087 <*iso885914>
2088 \__str_declare_eight_bit_encoding:nnn { iso885914 }
2089 {
2090     { A1 } { 1E02 }
2091     { A2 } { 1E03 }
2092     { A4 } { 010A }
2093     { A5 } { 010B }
2094     { A6 } { 1E0A }
2095     { A8 } { 1E80 }
2096     { AA } { 1E82 }
2097     { AB } { 1E0B }
2098     { AC } { 1EF2 }
2099     { AF } { 0178 }
2100     { B0 } { 1E1E }
2101     { B1 } { 1E1F }
2102     { B2 } { 0120 }
2103     { B3 } { 0121 }
2104     { B4 } { 1E40 }
2105     { B5 } { 1E41 }
2106     { B7 } { 1E56 }
2107     { B8 } { 1E81 }
2108     { B9 } { 1E57 }
2109     { BA } { 1E83 }
2110     { BB } { 1E60 }
2111     { BC } { 1EF3 }
2112     { BD } { 1E84 }
2113     { BE } { 1E85 }
2114     { BF } { 1E61 }
2115     { D0 } { 0174 }
2116     { D7 } { 1E6A }
2117     { DE } { 0176 }
2118     { F0 } { 0175 }
2119     { F7 } { 1E6B }
2120     { FE } { 0177 }
2121 }
2122 {
2123 }
2124 </iso885914>
2125 <*iso885915>

```

```

2126 \_str_declare_eight_bit_encoding:nnn { iso885915 }
2127 {
2128     { A4 } { 20AC }
2129     { A6 } { 0160 }
2130     { A8 } { 0161 }
2131     { B4 } { 017D }
2132     { B8 } { 017E }
2133     { BC } { 0152 }
2134     { BD } { 0153 }
2135     { BE } { 0178 }
2136 }
2137 {
2138 }
2139 </iso885915>
2140 <*iso885916>
2141 \_str_declare_eight_bit_encoding:nnn { iso885916 }
2142 {
2143     { A1 } { 0104 }
2144     { A2 } { 0105 }
2145     { A3 } { 0141 }
2146     { A4 } { 20AC }
2147     { A5 } { 201E }
2148     { A6 } { 0160 }
2149     { A8 } { 0161 }
2150     { AA } { 0218 }
2151     { AC } { 0179 }
2152     { AE } { 017A }
2153     { AF } { 017B }
2154     { B2 } { 010C }
2155     { B3 } { 0142 }
2156     { B4 } { 017D }
2157     { B5 } { 201D }
2158     { B8 } { 017E }
2159     { B9 } { 010D }
2160     { BA } { 0219 }
2161     { BC } { 0152 }
2162     { BD } { 0153 }
2163     { BE } { 0178 }
2164     { BF } { 017C }
2165     { C3 } { 0102 }
2166     { C5 } { 0106 }
2167     { D0 } { 0110 }
2168     { D1 } { 0143 }
2169     { D5 } { 0150 }
2170     { D7 } { 015A }
2171     { D8 } { 0170 }
2172     { DD } { 0118 }
2173     { DE } { 021A }
2174     { E3 } { 0103 }

```

```

2175     { E5 } { 0107 }
2176     { F0 } { 0111 }
2177     { F1 } { 0144 }
2178     { F5 } { 0151 }
2179     { F7 } { 015B }
2180     { F8 } { 0171 }
2181     { FD } { 0119 }
2182     { FE } { 021B }
2183 }
2184 {
2185 }
2186 </iso885916>

```

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.

Symbols		
\#	343	_prg_break_point: 71, 77, 179, 195, 313, 361, 427, 456, 460, 502, 595, 644, 704, 710, 946, 1147, 1302
\\$	341	_str_convert:NNnNN 254, 259, 262
\%	351	_str_convert:nNNnnn 214, 215, 217, 222, 231, 236
\	342	_str_convert:nnn 257, 258, 272, 272
*	42, 689	_str_convert:nnnn 272, 276, 281
\\	338, 560, 561, 564, 689, 902, 905, 906, 907, 908, 913, 919, 924, 931, 1089, 1092, 1093, 1094, 1096, 1102, 1107, 1112, 1261, 1268	_str_convert:wwwnn .. 241, 246, 254, 254
\{	339, 565	_str_convert_decode_: .. 245, 388, 388
\}	340, 565	_str_convert_decode_eight_bit:n 443, 449, 449
\^	344, 690, 691, 1033, 1034, 1213, 1214, 1215	_str_convert_decode_utf16: 1117
_int_eval:w	102, 144, 152, 433, 964, 979, 1009, 1157	_str_convert_decode_utf16be: .. 1117
_int_eval_end:	104, 155, 433, 964, 981, 1009, 1157	_str_convert_decode_utf16le: .. 1117
_int_value:w	391, 471, 493, 883, 956	_str_convert_decode_utf32: 1273
_msg_kernel_error:nnx	207, 266	_str_convert_decode_utf32be: .. 1273
_msg_kernel_error:nnxx	298	_str_convert_decode_utf32le: .. 1273
_msg_kernel_new:nnn	540, 542, 551	_str_convert_decode_utf8: 935
_msg_kernel_new:nnnn	412, 544, 555, 569, 575, 624, 671, 766, 891, 1072, 1079, 1249	_str_convert_encode_: 250, 392, 396, 421
_prg_break:	178, 194, 312, 360, 426, 455, 459, 501, 594, 643, 703, 709, 945, 1028, 1198, 1338	_str_convert_encode_eight_bit:n 445, 495, 495
_prg_break:n	70, 76, 88	_str_convert_encode_utf16: 1035
		_str_convert_encode_utf16be: .. 1035
		_str_convert_encode_utf16le: .. 1035
		_str_convert_encode_utf32: 1216
		_str_convert_encode_utf32be: .. 1216
		_str_convert_encode_utf32le: .. 1216
		_str_convert_encode_utf8: 865

```

\\_str_convert_escape_: .. 386, 386, 387
\\_str_convert_escape_bytes: .. 386, 387
\\_str_convert_escape_hex: .... 776, 777
\\_str_convert_escape_name: .... 782, 785
\\_str_convert_escape_string: . 807, 810
\\_str_convert_escape_url: .... 841, 842
\\_str_convert_gmap:N .....
    172, 172, 389, 462, 778, 786, 811, 843
\\_str_convert_gmap_internal:N ....
    ..... 188, 188,
    399, 504, 866, 1048, 1218, 1222, 1224
\\_str_convert_gmap_internal_loop:Nw
    ..... 188
\\_str_convert_gmap_internal_loop:Nww
    ..... 192, 198, 202
\\_str_convert_gmap_loop:NN .....
    ..... 172, 176, 182, 186
\\_str_convert_lowercase_alphanum:n
    ..... 277, 309, 309
\\_str_convert_lowercase_alphanum_loop:N
    ..... 309, 311, 315, 334
\\_str_convert_unescape_: .....
    ..... 374, 375, 377, 385
\\_str_convert_unescape_bytes: 374, 385
\\_str_convert_unescape_hex: .. 583, 584
\\_str_convert_unescape_name: .... 631
\\_str_convert_unescape_string: 687, 694
\\_str_convert_unescape_url: .... 631
\\_str_declare_eight_bit_encoding:nmn
    ..... 439, 439, 1343, 1350, 1414,
    1456, 1513, 1614, 1701, 1787, 1861,
    1874, 1927, 2025, 2088, 2126, 2141
\\_str_decode_eight_bit_char:N ....
    ..... 449, 462, 485
\\_str_decode_eight_bit_load:nn ....
    ..... 449, 453, 466, 473
\\_str_decode_eight_bit_load_missing:n
    ..... 449, 457, 475, 483
\\_str_decode_native_char:N 388, 389, 390
\\_str_decode_utf_viii_aux:wNnnwN ..
    ..... 935, 978, 992
\\_str_decode_utf_viii_continuation:wwN
    ..... 935, 963, 970, 1008
\\_str_decode_utf_viii_end: .....
    ..... 935, 945, 1022
\\_str_decode_utf_viii_overflow:w ..
    ..... 935, 1006, 1015
\\_str_decode_utf_viii_start:N ....
    935, 944, 950, 968, 971, 990, 993, 1013
\\_str_decode_utf_xvi:Nw ..... 1117,
    1118, 1120, 1129, 1132, 1133, 1136
\\_str_decode_utf_xvi_bom:NN .....
    ..... 1117, 1123, 1126
\\_str_decode_utf_xvi_error:nNN ....
    .. 1151, 1169, 1187, 1196, 1201, 1202
\\_str_decode_utf_xvi_extra:NNw ....
    ..... 1151, 1159, 1200
\\_str_decode_utf_xvi_pair:NN .....
    .. 1145, 1151, 1151, 1163, 1166, 1189
\\_str_decode_utf_xvi_pair_end:Nw ..
    ..... 1151, 1154, 1170, 1191
\\_str_decode_utf_xvi_quad:NNwNN ..
    ..... 1151, 1158, 1165
\\_str_decode_utf_xxxii:Nw ... 1273,
    1274, 1276, 1285, 1288, 1289, 1292
\\_str_decode_utf_xxxii_bom:NNNN ...
    ..... 1273, 1279, 1282
\\_str_decode_utf_xxxii_end:w .....
    ..... 1273, 1309, 1329
\\_str_decode_utf_xxxii_loop:NNNN ..
    ..... 1273, 1300, 1306, 1327
\\_str_encode_eight_bit_char:n ....
    ..... 495, 504, 518
\\_str_encode_eight_bit_char_aux:n .
    ..... 495, 529, 532
\\_str_encode_eight_bit_load:nn ....
    ..... 495, 499, 508, 516
\\_str_encode_native_char:n 395, 399, 403
\\_str_encode_native_filter:N .... 420
\\_str_encode_native_flush: ..... 420
\\_str_encode_native_loop:w .....
    ..... 420, 425, 430, 435
\\_str_encode_utf_viii_char:n .....
    ..... 865, 866, 867
\\_str_encode_utf_viii_loop:wwnnw ..
    ..... 865, 869, 876, 882
\\_str_encode_utf_xvi_aux:N .....
    ..... 1035, 1037, 1041, 1043, 1044
\\_str_encode_utf_xvi_char:n .....
    ..... 1035, 1048, 1051
\\_str_encode_utf_xxxii_be:n .....
    ..... 1216, 1218, 1222, 1225
\\_str_encode_utf_xxxii_be_aux:nn ..
    ..... 1216, 1227, 1230
\\_str_encode_utf_xxxii_le:n .....
    ..... 1216, 1224, 1236
\\_str_encode_utf_xxxii_le_aux:nn ..
    ..... 1216, 1238, 1241
\\_str_escape_hex_char:N .. 776, 778, 779

```



```

__str_escape_name_char:N . 782, 786, 787
__str_escape_string_char:N 807, 811, 812
__str_escape_url_char:N . 841, 843, 844
__str_filter_bytes:n . . . . . 354, 355, 357, 381, 652, 719
__str_filter_bytes_aux:N . . . . . 354, 359, 363, 371
__str_gset_other:Nn . . . . 4, 41, 47, 240
__str_gset_other_end:w . . . . 41, 59, 64
__str_gset_other_loop:w . . 41, 51, 55, 62
__str_hexadecimal_use:N . . . . . 97
__str_hexadecimal_use:NTF . . . . . 5, 97, 604, 614, 655, 657
__str_if_contains_char:NN . . . . . 67
__str_if_contains_char:NNT . . . . 67, 816
__str_if_contains_char:NNTF 67, 795, 801
__str_if_contains_char:nN . . . . . 74
__str_if_contains_char:nNTF 67, 852, 858
__str_if_contains_char_aux:NN . . . . 67, 69, 76, 80, 85
__str_if_contains_char_true: 67, 83, 87
__str_if_escape_name:N . . . . . 792
__str_if_escape_name:NTF . . . . 782, 789
__str_if_escape_string:N . . . . . 828
__str_if_escape_string:NTF . . . 807, 814
__str_if_escape_url:N . . . . . 849
__str_if_escape_url:NTF . . . . 841, 846
__str_if_flag_error:nmx 204, 204, 223,
232, 382, 400, 463, 505, 598, 646,
647, 712, 713, 948, 1049, 1149, 1304
__str_if_flag_no_error:nmx . . . . . 204, 210, 223, 232
__str_if_flag_times:nT . . . . . 212, 212, 894, 895,
896, 897, 1082, 1083, 1084, 1252, 1253
__str_load_catcodes: . . . . 292, 336, 336
__str_octal_use:N . . . . . 89
__str_octal_use:NTF . . . 89, 722, 724, 726
__str_output_byte:n . . . . . 5, 138, 138, 169, 170,
324, 406, 514, 535, 879, 885, 1234, 1243
__str_output_byte:w . . . . . 138, 139, 140, 591, 617, 654, 721
__str_output_byte_pair:nnN . . . . . 156, 158, 163, 166
__str_output_byte_pair_be:n . . . . . 156, 156, 1037, 1041, 1233
__str_output_byte_pair_le:n . . . . . 156, 161, 1043, 1244
__str_output_name:N . . . . . 138,
139, 147, 154, 596, 616, 668, 755, 759
__str_output_hexadecimal:n . . . . . 138, 146, 780, 790, 847
__str_output_hexadecimal:w 138, 147, 148
__str_tmp:w . . . . . 10, 10, 632, 680,
684, 1047, 1054, 1059, 1061, 1064,
1065, 1142, 1157, 1162, 1173, 1176,
1181, 1182, 1297, 1312, 1317, 1323
__str_unescape_hex_auxi:N . . . . . 583, 592, 601, 608, 617
__str_unescape_hex_auxii:N . . . . . 583, 605, 611, 621
__str_unescape_name_loop:wNN . 631, 681
__str_unescape_string_loop:wNNN . . . . . 687, 708, 716, 756, 759
__str_unescape_string_newlines:wN . . . . . 687, 702, 760, 764
__str_unescape_string_repeat:NNNNN . . . . . 687, 731, 733, 735, 758
__str_unescape_url_loop:wNN . . 631, 685
__tl_build_end: . . . . . 428
__tl_build_one:n . . . . . 434
__tl_gbuild_x:Nw . . . . . 424
~ . . . . . 346

□ . . . . . 42, 345, 565, 905, 906, 907

A
A . . . . . 43

B
bool_gset_false:N . . . . . 221, 230
bool_gset_true:N . . . . . 211
bool_if:NTF . . . . . 225, 234
bool_new:N . . . . . 38

C
c_str_byte_-1_tl . . . . . 118
c_str_byte_0_tl . . . . . 118
c_str_byte_1_tl . . . . . 118
c_str_byte_255_tl . . . . . 118
c_str_escape_name_not_str 782, 783, 795
c_str_escape_name_str . . 782, 784, 801
c_str_escape_string_str . 807, 808, 817
c_str_replacement_char_int . . . . . 14, 14, 481, 960, 986, 1000, 1020,
1027, 1059, 1207, 1315, 1320, 1336
c_backslash_str 740, 769, 771, 809, 818, 822

```

```

\c_eight ..... 824, 825
\c_fifty_eight ..... 15, 16
\c_forty_eight ..... 15, 15
\c_four ..... 1157
\c_hash_str ..... 680, 790
\c_max_char_int ..... 4, 8, 8
\c_max_register_int ..... 520
\c_minus_one ..... 869
\c_ninety_one ..... 15, 18, 318
\c_ninety_seven ..... 15, 19, 328
\c_one ..... 91,
320, 472, 482, 515, 594, 748, 752, 1174
\c_one_hundred_twenty_seven .... 15, 21
\c_one_hundred_twenty_three . 15, 20, 327
\c_percent_str ..... 684, 847
\c_sixteen ..... 1317
\c_sixty_five ..... 15, 17, 319
\c_thirty_two ..... 324
\c_three ..... 728
\c_two ..... 99, 885
\c_two_hundred_fifty_six ..... 405, 534
\c_zero ..... 119,
122, 127, 661, 869, 870, 1174, 1312
\char_set_catcode_alignment:N .... 342
\char_set_catcode_comment:N ..... 351
\char_set_catcode_escape:N ..... 338
\char_set_catcode_group_begin:N ... 339
\char_set_catcode_group_end:N .... 340
\char_set_catcode_ignore:N ..... 345
\char_set_catcode_letter:N ..... 348
\char_set_catcode_math_superscript:N
..... 344
\char_set_catcode_math_toggle:N ... 341
\char_set_catcode_other:N ..... 350,
690, 691, 1033, 1034, 1213, 1214, 1215
\char_set_catcode_other:n ..... 119, 393
\char_set_catcode_parameter:N .... 343
\char_set_catcode_space:N ..... 346
\char_set_lccode:nn . 42, 43, 122, 127, 689
\cs:w ..... 144, 152
\cs_end: ..... 155
\cs_gset_eq:cc ..... 303, 307
\cs_if_exist:cF ..... 274, 283, 287, 301
\cs_if_exist:NF ..... 6
\cs_new:Npn .....
7, 55, 64, 80, 138, 146, 156, 161,
166, 182, 198, 309, 315, 357, 363,
390, 403, 485, 518, 532, 601, 611,
650, 716, 758, 760, 779, 787, 812,
844, 867, 876, 950, 970, 992, 1015,
1051, 1151, 1165, 1191, 1200, 1202,
1225, 1230, 1236, 1241, 1306, 1329
\cs_new_eq:NN ..... 355, 385, 387
\cs_new_nopar:Npn 87, 140, 148, 154, 1022
\cs_new_protected:cpn ..... 634
\cs_new_protected:Npn .. 47, 172, 188,
204, 210, 212, 236, 254, 262, 272,
281, 336, 430, 439, 449, 466, 475,
495, 508, 1044, 1126, 1136, 1282, 1292
\cs_new_protected_nopar:cpn .....
442, 444, 865, 935,
1035, 1040, 1042, 1117, 1119, 1121,
1216, 1221, 1223, 1273, 1275, 1277
\cs_new_protected_nopar:Npn .....
10, 214, 216, 375, 377, 386, 388,
396, 421, 584, 694, 777, 785, 810, 842
\cs_set:Npn ..... 1142, 1297
\cs_set_eq:NN ..... 223, 232, 1047
\cs_set_protected:Npn ..... 632

E
\else: ..... 93, 101, 111, 268, 323,
326, 329, 368, 407, 522, 536, 763,
797, 800, 832, 835, 854, 857, 957,
962, 982, 1001, 1004, 1055, 1060,
1063, 1175, 1186, 1195, 1316, 1321
\exp_after:wN ..... 51,
69, 83, 103, 113, 142, 143, 150, 151,
176, 177, 192, 193, 241, 246, 311,
425, 471, 480, 526, 642, 680, 684,
702, 708, 880, 882, 944, 963, 964,
978, 981, 1008, 1009, 1123, 1145,
1158, 1159, 1185, 1279, 1300, 1309
\exp_args:Ncc ..... 259
\exp_args:Nf 158, 163, 1064, 1065, 1227, 1238
\exp_args:NNf ..... 513
\exp_args:No ..... 121, 381
\exp_args:Nx ..... 276
\exp_last_unbraced:Nf 592, 893, 1081, 1251
\exp_last_unbraced:Nx .... 453, 457, 499
\exp_stop_f: ..... 91, 99, 320, 366,
489, 524, 525, 794, 798, 830, 833,
851, 855, 878, 953, 955, 975, 976,
995, 997, 1053, 1056, 1057, 1173, 1176
\ExplFileDate ..... 4
\ExplFileDescription ..... 4
\ExplFileName ..... 4
\ExplFileVersion ..... 4

```

F		488, 489, 520, 523, 524, 534, 728,
<code>\fi:</code> . 60, 64, 65, 70, 76, 84, 95, 114, 116,		794, 798, 830, 833, 851, 855, 878,
270, 322, 325, 331, 332, 333, 370,		953, 955, 974, 975, 995, 997, 1053,
410, 491, 492, 527, 528, 530, 538,		1056, 1057, 1172, 1173, 1312, 1317
730, 763, 803, 804, 837, 838, 860,	<code>\if_meaning:w</code>	58, 264, 1005, 1153, 1168, 1194, 1308
861, 881, 961, 965, 975, 987, 1003,	<code>\int_const:Nn</code> 8, 14, 15, 16, 17, 18, 19, 20, 21	
1007, 1010, 1015, 1017, 1062, 1066,	<code>\int_div_truncate:nn</code> 159,	164, 823, 824, 883, 1064, 1228, 1239
1067, 1155, 1160, 1171, 1177, 1188,	<code>\int_eval:n</code> 131, 1162, 1179, 1322	
1191, 1193, 1197, 1310, 1324, 1325	<code>\int_mod:nn</code> 824, 825, 1065	
<code>\file_if_exist:nTF</code> 289	<code>\int_new:N</code> 12	
<code>\file_input:n</code> 293	<code>\int_set:Nn</code> 352, 588, 639, 699	
<code>\flag_clear:n</code> 379, 398,	<code>\int_use:N</code> 144, 152,	481, 960, 964, 979, 986, 1000, 1009,
461, 503, 587, 637, 638, 697, 698,		1020, 1027, 1207, 1315, 1320, 1336
937, 938, 939, 940, 941, 1046, 1138,	<code>\int_zero:N</code> 423, 452, 498	
1139, 1140, 1141, 1294, 1295, 1296	<code>\iow_char:N</code> 564	
<code>\flag_clear_new:n</code> 887, 888,	<code>\iow_indent:n</code> 562, 903, 1090	
889, 890, 1069, 1070, 1071, 1247, 1248		
<code>\flag_height:n</code> 213		
<code>\flag_if_raised:nT</code> . 211, 213, 911, 917,		
922, 929, 1100, 1105, 1110, 1259, 1266		
<code>\flag_if_raised:nTF</code> 206		
<code>\flag_new:n</code> 39, 40		
<code>\flag_raise:n</code> 369, 408,		
521, 537, 607, 620, 660, 665, 751,		
958, 959, 984, 985, 998, 999, 1018,		
1019, 1025, 1026, 1058, 1204, 1205,		
1313, 1314, 1318, 1319, 1333, 1334		
G		
<code>\g_str_alias_prop</code>		
. 22, 22, 23, 24, 25, 26, 27, 28,		
29, 30, 31, 32, 33, 34, 35, 36, 37, 285		
<code>\g_str_error_bool</code>		
. 38, 38, 211, 221, 225, 230, 234		
<code>\g_str_result_tl</code>		
. 13, 13, 174, 178, 190, 194,		
240, 252, 380, 381, 424, 426, 589,		
593, 640, 642, 700, 703, 706, 709,		
942, 944, 1038, 1118, 1120, 1124,		
1143, 1219, 1274, 1276, 1279, 1298		
<code>\group_begin:</code> . . 41, 118, 238, 291, 392,		
451, 497, 586, 636, 688, 696, 1032, 1212		
<code>\group_end:</code> . . . 46, 136, 251, 294, 438,		
464, 506, 599, 648, 714, 774, 1209, 1340		
I		
<code>\if_case:w</code> 102, 1156		
<code>\if_charcode:w</code> 82, 763		
<code>\if_int_compare:w</code> 91, 99,		
318, 319, 320, 327, 328, 366, 405,		

<code>\prop_get:NnNF</code>	285	<code>\tex_endlinechar:D</code>	352
<code>\prop_gput:Nnn</code>	23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37	<code>\tex_escapechar:D</code>	588, 639, 699
<code>\prop_new:N</code>	22	<code>\tex_lccode:D</code>	433
<code>\ProvidesExplPackage</code>	3	<code>\tex_skip:D</code>	470, 479, 489, 512, 524
Q		<code>\tex_the:D</code>	490, 525
<code>\q_nil</code>	1146, 1153, 1168, 1194	<code>\tex_toks:D</code>	471, 480, 490, 513, 525
<code>\q_stop</code>	52, 64, 194, 200, 242, 247, 255, 426, 432, 455, 459, 468, 477, 501, 510, 874, 876, 884, 967, 1005	<code>\tl_clear:N</code>	297
R		<code>\tl_const:cn</code>	137, 446, 447
<code>\RequirePackage</code>	5	<code>\tl_const:cx</code>	130
S		<code>\tl_gput_left:Nx</code>	1038, 1219
<code>\s_stop</code>	1118, 1120, 1124, 1136, 1274, 1276, 1280, 1292, 1301, 1308, 1329	<code>\tl_gset:Nx</code>	49, 174, 190, 380, 589, 640, 700, 706, 942, 1143, 1298
<code>\s_tl</code>	194, 198, 391, 426, 430, 487, 493, 954, 966, 971, 983, 988, 993, 996, 1011, 1024, 1027, 1161, 1162, 1178, 1184, 1200, 1206, 1207, 1311, 1326, 1335, 1336	<code>\tl_gset_eq:NN</code>	217, 233
<code>\scan_stop:</code> ...	469, 470, 478, 479, 511, 512	<code>\tl_if_empty:nF</code>	265, 1331
<code>\str_byte</code>	39	<code>\tl_map_function:nN</code>	347, 349
<code>\str_case_x:nnn</code>	738	<code>\tl_map_inline:Nn</code>	123, 125
<code>\str_const:Nn</code>	783, 784	<code>\tl_map_inline:nn</code>	121
<code>\str_const:Nx</code>	808	<code>\tl_new:N</code>	11, 13
<code>\str_end</code>	1247	<code>\tl_set:Nn</code>	286, 441
<code>\str_error</code>	39	<code>\tl_set:Nx</code>	120
<code>\str_gset_convert:Nnnn</code>	4, 214, 216, 228	<code>\tl_set_eq:NN</code>	215, 224
<code>\str_gset_convert:NnnnTF</code>	4, 214	<code>\tl_to_lowercase:n</code>	44, 128, 434, 692
<code>\str_if_eq_x:nnTF</code>	1128, 1131, 1284, 1287	<code>\tl_to_str:N</code>	593
<code>\str_overflow</code>	1247	<code>\tl_to_str:n</code>	51, 120, 242, 247, 312
<code>\str_set_convert:Nnnn</code>	4, 214, 214, 219	<code>\tl_use:c</code>	454, 458, 500
<code>\str_set_convert:NnnnTF</code>	4, 214	<code>\token_to_str:N</code>	91, 99, 103
T		U	
<code>\tex_advance:D</code>	472, 482, 515	<code>\use:n</code>	355
<code>\tex_dimen:D</code>	469, 478, 488, 489, 490, 511, 523, 524, 525	<code>\use:nn</code>	159
		<code>\use_i:nn</code>	661, 666, 752, 756
		<code>\use_i:nnn</code>	143, 669, 1185
		<code>\use_ii_i:nn</code>	6, 6, 7, 164, 249
		<code>\use_none:n</code>	88, 113, 137, 151, 184, 317, 365, 603, 613, 653, 720, 893, 973, 1081, 1251
		<code>\use_none:nn</code>	208, 526
		<code>\use_none_delimit_by_q_stop:w</code>	200, 432, 468, 477, 510, 880, 967