

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	clist	20
5.3.8	8-bit encodings	21
5.4	Messages	23
5.5	Escaping definition files	24
5.5.1	Unescape methods	25
5.5.2	Escape methods	30
5.6	Encoding definition files	32
5.6.1	UTF-8 support	32
5.6.2	UTF-16 support	37
5.6.3	UTF-32 support	43
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
utf8	UTF-8
utf16	UTF-16, with byte-order mark
utf16be	UTF-16, big-endian
utf16le	UTF-16, little-endian
utf32	UTF-32, with byte-order mark
utf32be	UTF-32, big-endian
utf32le	UTF-32, little-endian
iso88591, latin1	ISO 8859-1
iso88592, latin2	ISO 8859-2
iso88593, latin3	ISO 8859-3
iso88594, latin4	ISO 8859-4
iso88595	ISO 8859-5
iso88596	ISO 8859-6
iso88597	ISO 8859-7
iso88598	ISO 8859-8
iso88599, latin5	ISO 8859-9
iso885910, latin6	ISO 8859-10
iso885911	ISO 8859-11
iso885913, latin7	ISO 8859-13
iso885914, latin8	ISO 8859-14
iso885915, latin9	ISO 8859-15
iso885916, latin10	ISO 8859-16
clist	comma-list of integers
<i>⟨empty⟩</i>	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
bytes , or empty	arbitrary bytes
hex , hexadecimal	byte = two hexadecimal digits
name	see <code>\pdfescapename</code>
string	see <code>\pdfescapestring</code>
url	encoding used in URLs

2 Conversion functions

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

`\str_set_convert:Nnnn <str var> {<string>} {<name 1>} {<name 2>}`

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 `FEFF00480065006C006F0021`. 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 “FEFF”, 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 <str var> {<string>} {<name 1>} {<name 2>} {<true code>} {<false code>}`

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 <tl var> {<token list>}`

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_twenty_three 18 \int_const:Nn \c_ninety_one { 91 }
\c_one_hundred_twenty_seven 19 \int_const:Nn \c_ninety_seven { 97 }
                     20 \int_const:Nn \c_one_hundred_twenty_three { 123 }
                     21 \int_const:Nn \c_one_hundred_twenty_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 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. 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 \TeX 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:ABCDEFGHIJLPSTUX }  

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.

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

```

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 clist

`__str_convert_decode_clist:` Convert each integer to the internal form. We first turn `\g__str_result_tl` into a clist variable, as this avoids problems with leading or trailing commas.

```

439 \cs_new_protected_nopar:Npn \__str_convert_decode_clist:
440 {
441     \clist_set:No \g__str_result_tl \g__str_result_tl
442     \tl_gset:Nx \g__str_result_tl
443     {
444         \exp_args:No \clist_map_function:nN
445         \g__str_result_tl \__str_decode_clist_char:n
446     }
447 }
448 \cs_new:Npn \__str_decode_clist_char:n #1
449 { #1 \s__tl \int_eval:n {#1} \s__tl }

```

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

`__str_convert_encode_clist:` Convert the internal list of character codes to a comma-list of character codes. The first line produces a comma-list with a leading comma, removed in the next step (this also works in the empty case, since `\tl_tail:N` does not trigger an error in this case).

```

450 \cs_new_protected_nopar:Npn \__str_convert_encode_clist:
451 {
452     \__str_convert_gmap_internal:N \__str_encode_clist_char:n
453     \tl_gset:Nx \g__str_result_tl { \tl_tail:N \g__str_result_tl }
454 }
455 \cs_new:Npn \__str_encode_clist_char:n #1 { , #1 }

```

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

5.3.8 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.

```
456 \cs_new_protected:Npn \_str_declare_eight_bit_encoding:nnn #1#2#3
457 {
458   \tl_set:Nn \l__str_internal_tl {#1}
459   \cs_new_protected_nopar:cpn { __str_convert_decode_#1: }
460   { \_str_convert_decode_eight_bit:n {#1} }
461   \cs_new_protected_nopar:cpn { __str_convert_encode_#1: }
462   { \_str_convert_encode_eight_bit:n {#1} }
463   \tl_const:cn { c__str_encoding_#1_tl } {#2}
464   \tl_const:cn { c__str_encoding_#1_missing_tl } {#3}
465 }
```

(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
466 \cs_new_protected:Npn \_str_convert_decode_eight_bit:n #1
467 {
468   \group_begin:
469   \int_zero:N \l__str_internal_int
470   \exp_last_unbraced:Nx \_str_decode_eight_bit_load:nn
471   { \tl_use:c { c__str_encoding_#1_tl } }
472   { \q_stop \_prg_break: } { }
473   \_prg_break_point:
474   \exp_last_unbraced:Nx \_str_decode_eight_bit_load_missing:n
475   { \tl_use:c { c__str_encoding_#1_missing_tl } }
476   { \q_stop \_prg_break: }
477   \_prg_break_point:
478   \flag_clear:n { str_error }
479   \_str_convert_gmap:N \_str_decode_eight_bit_char:N
```

```

480     \_str_if_flag_error:nmx { str_error } { decode-8-bit } {#1}
481   \group_end:
482 }
483 \cs_new_protected:Npn \_str_decode_eight_bit_load:nn #1#2
484 {
485   \use_none_delimit_by_q_stop:w #1 \q_stop
486   \tex_dimen:D "#1 = \l__str_internal_int sp \scan_stop:
487   \tex_skip:D \l__str_internal_int = "#1 sp \scan_stop:
488   \tex_toks:D \l__str_internal_int \exp_after:wN { \_int_value:w "#2 }
489   \tex_advance:D \l__str_internal_int \c_one
490   \_str_decode_eight_bit_load:nn
491 }
492 \cs_new_protected:Npn \_str_decode_eight_bit_load_missing:n #1
493 {
494   \use_none_delimit_by_q_stop:w #1 \q_stop
495   \tex_dimen:D "#1 = \l__str_internal_int sp \scan_stop:
496   \tex_skip:D \l__str_internal_int = "#1 sp \scan_stop:
497   \tex_toks:D \l__str_internal_int \exp_after:wN
498     { \int_use:N \c__str_replacement_char_int }
499   \tex_advance:D \l__str_internal_int \c_one
500   \_str_decode_eight_bit_load_missing:n
501 }
502 \cs_new:Npn \_str_decode_eight_bit_char:N #1
503 {
504   #1 \s_tl
505   \if_int_compare:w \tex_dimen:D '#1 < \l__str_internal_int
506     \if_int_compare:w \tex_skip:D \tex_dimen:D '#1 = '#1 \exp_stop_f:
507       \tex_the:D \tex_toks:D \tex_dimen:D
508     \fi:
509   \fi:
510   \_int_value:w '#1 \s_tl
511 }

```

(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

```

```

512 \cs_new_protected:Npn \_str_convert_encode_eight_bit:n #1
513 {
514   \group_begin:
515     \int_zero:N \l__str_internal_int
516     \exp_last_unbraced:Nx \_str_encode_eight_bit_load:nn
517       { \tl_use:c { c__str_encoding_#1_tl } }
518     { \q_stop \_prg_break: } { }
519     \_prg_break_point:
520     \flag_clear:n { str_error }
521     \_str_convert_gmap_internal:N \_str_encode_eight_bit_char:n
522     \_str_if_flag_error:nmx { str_error } { encode-8-bit } {#1}
523   \group_end:
524 }
525 \cs_new_protected:Npn \_str_encode_eight_bit_load:nn #1#2
526 {

```

```

527 \use_none_delimit_by_q_stop:w #1 \q_stop
528 \tex_dimen:D "#2 = \l__str_internal_int sp \scan_stop:
529 \tex_skip:D \l__str_internal_int = "#2 sp \scan_stop:
530 \exp_args:NNf \tex_toks:D \l__str_internal_int
531 { \__str_output_byte:n { "#1 } }
532 \tex_advance:D \l__str_internal_int \c_one
533 \__str_encode_eight_bit_load:nn
534 }
535 \cs_new:Npn \__str_encode_eight_bit_char:n #1
536 {
537   \if_int_compare:w #1 > \c_max_register_int
538     \flag_raise:n { str_error }
539   \else:
540     \if_int_compare:w \tex_dimen:D #1 < \l__str_internal_int
541       \if_int_compare:w \tex_skip:D \tex_dimen:D #1 = #1 \exp_stop_f:
542         \tex_the:D \tex_toks:D \tex_dimen:D #1 \exp_stop_f:
543         \exp_after:wN \exp_after:wN \exp_after:wN \use_none:nn
544       \fi:
545     \fi:
546     \__str_encode_eight_bit_char_aux:n {#1}
547   \fi:
548 }
549 \cs_new:Npn \__str_encode_eight_bit_char_aux:n #1
550 {
551   \if_int_compare:w #1 < \c_two_hundred_fifty_six
552     \__str_output_byte:n {#1}
553   \else:
554     \flag_raise:n { str_error }
555   \fi:
556 }

```

(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”).

```

557 \__msg_kernel_new:nnn { str } { unknown-esc }
558 { Escaping~scheme~'#1'~(filtered:~'#2')~unknown. }
559 \__msg_kernel_new:nnn { str } { unknown-enc }
560 { Encoding~scheme~'#1'~(filtered:~'#2')~unknown. }
561 \__msg_kernel_new:nnnn { str } { native-escaping }
562 { The~'native'~encoding~scheme~does~not~support~any~escaping. }
563 {
564   Since~native~strings~do~not~consist~in~bytes,~
565   none~of~the~escaping~methods~make~sense.~
566   The~specified~escaping,~'#1',~will be ignored.
567 }
568 \__msg_kernel_new:nnn { str } { file-not-found }
569 { 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.

```

570 \pdfTeX_if_engine:F
571 {
572   \__msg_kernel_new:nnnn { str } { non-byte }
573   { String~invalid~in~escaping~'#1':~it~may~only~contain~bytes. }
574   {
575     Some~characters~in~the~string~you~asked~to~convert~are~not~
576     8-bit~characters.~Perhaps~the~string~is~a~'native'~Unicode~string?~
577     If~it~is,~try~using\\
578     \\
579     \iow_indent:n
580     {
581       \iow_char:N\\str_set_convert:Nnnn \\
582       \ \ <str~var>~\{-<string>~\}~\{-native~\}~\{-<target~encoding>~\}
583     }
584   }
585 }

```

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

```

586 \__msg_kernel_new:nnnn { str } { decode-8-bit }
587 { Invalid~string~in~encoding~'#1'. }
588 {
589   LaTeX~came~across~a~byte~which~is~not~defined~to~represent~
590   any~character~in~the~encoding~'#1'.
591 }
592 \__msg_kernel_new:nnnn { str } { encode-8-bit }
593 { Unicode~string~cannot~be~converted~to~encoding~'#1'. }
594 {
595   The~encoding~'#1'~only~contains~a~subset~of~all~Unicode~characters.~
596   LaTeX~was~asked~to~convert~a~string~to~that~encoding,~but~that~
597   string~contains~a~character~that~'#1'~does~not~support.
598 }
599 </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.

```

600 <*hex>
601 \cs_new_protected_nopar:Npn \\_str_convert_unescape_hex:
602 {
603   \group_begin:
604     \flag_clear:n { str_error }
605     \int_set:Nn \tex_escapechar:D { 92 }
606     \tl_gset:Nx \g_\\_str_result_tl
607     {
608       \\_str_output_byte:w "
609       \exp_last_unbraced:Nf \\_str_unescape_hex_auxi:N
610       { \tl_to_str:N \g_\\_str_result_tl }
611       0 { ? 0 - \c_one \\_prg_break: }
612       \\_prg_break_point:
613       \\_str_output_end:
614     }
615     \\_str_if_flag_error:nxn { str_error } { unescape-hex } { }
616   \group_end:
617 }
618 \cs_new:Npn \\_str_unescape_hex_auxi:N #1
619 {
620   \use_none:n #1
621   \\_str_hexadecimal_use:NTF #1
622   { \\_str_unescape_hex_auxii:N }
623   {
624     \flag_raise:n { str_error }
625     \\_str_unescape_hex_auxi:N
626   }
627 }
628 \cs_new:Npn \\_str_unescape_hex_auxii:N #1
629 {
630   \use_none:n #1
631   \\_str_hexadecimal_use:NTF #1
632   {
633     \\_str_output_end:
634     \\_str_output_byte:w " \\_str_unescape_hex_auxi:N
635   }
636   {
637     \flag_raise:n { str_error }
638     \\_str_unescape_hex_auxii:N
639   }
640 }
```

```

641 \_msg_kernel_new:nnnn { str } { unescape-hex }
642 { String~invalid~in~escaping~'hex':~only~hexadecimal~digits~allowed. }
643 {
644   Some~characters~in~the~string~you~asked~to~convert~are~not~
645   hexadecimal~digits~(0-9,~A-F,~a-f)~nor~spaces.
646 }
647 </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).

```

648 <*name|url>
649 \cs_set_protected:Npn \_str_tmp:w #1#2#3
650 {
651   \cs_new_protected:cpn { \_str_convert_unescape_#2: }
652   {
653     \group_begin:
654       \flag_clear:n { str_byte }
655       \flag_clear:n { str_error }
656       \int_set:Nn \tex_escapechar:D { 92 }
657       \tl_gset:Nx \g\_str_result_tl
658       {
659         \exp_after:wN #3 \g\_str_result_tl
660         #1 ? { ? \_prg_break: }
661         \_prg_break_point:
662       }
663       \_str_if_flag_error:nnx { str_byte } { non-byte } { #2 }
664       \_str_if_flag_error:nnx { str_error } { unescape-#2 } { }
665     \group_end:
666   }
667   \cs_new:Npn #3 ##1##2##3
668   {
669     \_str_filter_bytes:n {##1}
670     \use_none:n ##3
671     \_str_output_byte:w "
672     \_str_hexadecimal_use:NTF ##2

```

```

673         {
674             \__str_hexadecimal_use:NTF ##3
675             { }
676             {
677                 \flag_raise:n { str_error }
678                 * \c_zero + '#1 \use_i:nn
679             }
680         }
681         {
682             \flag_raise:n { str_error }
683             0 + '#1 \use_i:nn
684         }
685         \__str_output_end:
686         \use_i:nnn #3 ##2##3
687     }
688     \__msg_kernel_new:nnnn { str } { unescape-#2 }
689     { String-invalid-in-escaping~'#2'. }
690     {
691         LaTeX~came~across~the~escape~character~'#1'~not~followed~by~
692         two~hexadecimal~digits.~This~is~invalid~in~the~escaping~'#2'.
693     }
694 }
695 </name|url>
696 <*name>
697 \exp_after:wN \__str_tmp:w \c_hash_str { name }
698 \__str_unescape_name_loop:wNN
699 </name>
700 <*url>
701 \exp_after:wN \__str_tmp:w \c_percent_str { url }
702 \__str_unescape_url_loop:wNN
703 </url>

```

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

__str_convert_unescape_string: 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.

__str_unescape_string_newlines:wN

__str_unescape_string_loop:wNNN

__str_unescape_string_repeat:NNNNNN

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.

```
704 <*string>
705 \group_begin:
706   \char_set_lccode:nn {'\*} {'\\}
707   \char_set_catcode_other:N \^^J
708   \char_set_catcode_other:N \^^M
709   \tl_to_lowercase:n
710   {
711     \cs_new_protected_nopar:Npn \__str_convert_unescape_string:
712     {
713       \group_begin:
714       \flag_clear:n { str_byte }
715       \flag_clear:n { str_error }
716       \int_set:Nn \tex_escapechar:D { 92 }
717       \tl_gset:Nx \g__str_result_tl
718       {
719         \exp_after:wN \__str_unescape_string_newlines:wN
720         \g__str_result_tl \__prg_break: \^^M ?
721         \__prg_break_point:
722       }
723       \tl_gset:Nx \g__str_result_tl
724       {
725         \exp_after:wN \__str_unescape_string_loop:wNNN
726         \g__str_result_tl * ?? { ? \__prg_break: }
727         \__prg_break_point:
728       }
729       \__str_if_flag_error:nmx { str_byte } { non-byte } { string }
730       \__str_if_flag_error:nmx { str_error } { unescape-string } { }
731     \group_end:
732   }
733   \cs_new:Npn \__str_unescape_string_loop:wNNN #1 *#2#3#4
734   {
735     {
736       \__str_filter_bytes:n {#1}
737       \use_none:n #4
738       \__str_output_byte:w '
739       \__str_octal_use:NTF #2
740       {
741         \__str_octal_use:NTF #3
742         {
743           \__str_octal_use:NTF #4
744           {
745             \if_int_compare:w #2 > \c_three
746             - 256
```

```

747         \fi:
748         \_str_unescape_string_repeat:NNNNNN
749     }
750     { \_str_unescape_string_repeat:NNNNNN ? }
751 }
752 { \_str_unescape_string_repeat:NNNNNN ?? }
753 }
754 {
755     \str_case_x:nnn {#2}
756     {
757         { \c_backslash_str } { 134 }
758         { ( } { 50 }
759         { ) } { 51 }
760         { r } { 15 }
761         { f } { 14 }
762         { n } { 12 }
763         { t } { 11 }
764         { b } { 10 }
765         { ^^J } { 0 - \c_one }
766     }
767     {
768         \flag_raise:n { str_error }
769         0 - \c_one \use_i:nn
770     }
771 }
772 \_str_output_end:
773 \use_i:nn \_str_unescape_string_loop:wNNN #2#3#4
774 }
775 \cs_new:Npn \_str_unescape_string_repeat:NNNNNN #1#2#3#4#5#6
776 { \_str_output_end: \_str_unescape_string_loop:wNNN }
777 \cs_new:Npn \_str_unescape_string_newlines:wN #1 ^^M #2
778 {
779     #1
780     \if_charcode:w ^^J #2 \else: ^^J \fi:
781     \_str_unescape_string_newlines:wN #2
782 }
783 \_msg_kernel_new:nnnn { str } { unescape-string }
784 { String~invalid~in~escaping~'string'. }
785 {
786     LaTeX~came~across~an~escape~character~'\c_backslash_str'~
787     not~followed~by~any~of:~'n',~'r',~'t',~'b',~'f',~'(',~')',~
788     '\c_backslash_str',~one~to~three~octal~digits,~or~the~end~
789     of~a~line.
790 }
791 \group_end:
792 \</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.

```

793 <*hex>
794 \cs_new_protected_nopar:Npn \__str_convert_escape_hex:
795   { \__str_convert_gmap:N \__str_escape_hex_char:N }
796 \cs_new:Npn \__str_escape_hex_char:N #1
797   { \__str_output_hexadecimal:n { '#1 } }
798 </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, bytes outside the range [“2A”, “7E”] are hash-encoded. We keep two lists of exceptions: `__str_escape_name_char:N` characters in `\c__str_escape_name_not_str` are not hash-encoded, and characters in the `\c__str_escape_name_str` are encoded.

```

799 <*name>
800 \str_const:Nn \c__str_escape_name_not_str { ! " $ & ' } % $
801 \str_const:Nn \c__str_escape_name_str { {} / < > [ ] }
802 \cs_new_protected_nopar:Npn \__str_convert_escape_name:
803   { \__str_convert_gmap:N \__str_escape_name_char:N }
804 \cs_new:Npn \__str_escape_name_char:N #1
805   {
806     \__str_if_escape_name:NTF #1 {#1}
807     { \c_hash_str \__str_output_hexadecimal:n { '#1 } }
808   }
809 \prg_new_conditional:Npnn \__str_if_escape_name:N #1 { TF }
810   {
811     \if_int_compare:w '#1 < "2A \exp_stop_f:
812       \__str_if_contains_char:NNTF \c__str_escape_name_not_str #1
813       \prg_return_true: \prg_return_false:
814     \else:
815       \if_int_compare:w '#1 > "7E \exp_stop_f:
816       \prg_return_false:
817     \else:
818       \__str_if_contains_char:NNTF \c__str_escape_name_str #1
819       \prg_return_false: \prg_return_true:
820     \fi:
821   \fi:
822   }
823 </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.

```

824 <*string>
825 \str_const:Nx \c__str_escape_string_str

```

```

826 { \c_backslash_str ( ) }
827 \cs_new_protected_nopar:Npn \__str_convert_escape_string:
828 { \__str_convert_gmap:N \__str_escape_string_char:N }
829 \cs_new:Npn \__str_escape_string_char:N #1
830 {
831   \__str_if_escape_string:NTF #1
832   {
833     \__str_if_contains_char:NNT
834     \c_str_escape_string_str #1
835     { \c_backslash_str }
836     #1
837   }
838   {
839     \c_backslash_str
840     \int_div_truncate:nn {'#1} {64}
841     \int_mod:nn { \int_div_truncate:nn {'#1} \c_eight } \c_eight
842     \int_mod:nn {'#1} \c_eight
843   }
844 }
845 \prg_new_conditional:Npnn \__str_if_escape_string:N #1 { TF }
846 {
847   \if_int_compare:w '#1 < "21 \exp_stop_f:
848   \prg_return_false:
849   \else:
850     \if_int_compare:w '#1 > "7E \exp_stop_f:
851     \prg_return_false:
852     \else:
853       \prg_return_true:
854     \fi:
855   \fi:
856 }
857 </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 858 <*url>
\__str_if_escape_url:NTF 859 \cs_new_protected_nopar:Npn \__str_convert_escape_url:
860 { \__str_convert_gmap:N \__str_escape_url_char:N }
861 \cs_new:Npn \__str_escape_url_char:N #1
862 {
863   \__str_if_escape_url:NTF #1 {#1}
864   { \c_percent_str \__str_output_hexadecimal:n { '#1 } }
865 }
866 \prg_new_conditional:Npnn \__str_if_escape_url:N #1 { TF }
867 {
868   \if_int_compare:w '#1 < "41 \exp_stop_f:
869   \__str_if_contains_char:nNTF { "-.<> } #1
870   \prg_return_true: \prg_return_false:
871   \else:
872     \if_int_compare:w '#1 > "7E \exp_stop_f:

```

```

873     \prg_return_false:
874 \else:
875     \__str_if_contains_char:nNTF { [ ] } #1
876     \prg_return_false: \prg_return_true:
877 \fi:
878 \fi:
879 }
880 </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

```

881 <*utf8>

```

`__str_convert_encode_utf8:` Loop through the internal string, and convert each character to its UTF-8 representation.
`_str_encode_utf_viii_char:n` 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).

```

882 \cs_new_protected_nopar:cpn { __str_convert_encode_utf8: }
883 { \__str_convert_gmap_internal:N \__str_encode_utf_viii_char:n }
884 \cs_new:Npn \__str_encode_utf_viii_char:n #1
885 {
886   \__str_encode_utf_viii_loop:wwnnw #1 ; \c_minus_one + \c_zero * ;
887   { 128 } { \c_zero }
888   { 32 } { 192 }
889   { 16 } { 224 }
890   { 8 } { 240 }
891   \q_stop
892 }
893 \cs_new:Npn \__str_encode_utf_viii_loop:wwnnw #1; #2; #3#4 #5 \q_stop
894 {
895   \if_int_compare:w #1 < #3 \exp_stop_f:
896     \__str_output_byte:n { #1 + #4 }
897     \exp_after:wN \use_none_delimit_by_q_stop:w
898   \fi:
899   \exp_after:wN \__str_encode_utf_viii_loop:wwnnw
900     \__int_value:w \int_div_truncate:nn {#1} {64} ; #1 ;
901     #5 \q_stop
902   \__str_output_byte:n { #2 - 64 * ( #1 - \c_two ) }
903 }

```

(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.

```

904 \flag_clear_new:n { str_missing }
905 \flag_clear_new:n { str_extra }
906 \flag_clear_new:n { str_overlong }
907 \flag_clear_new:n { str_overflow }
908 \__msg_kernel_new:nnnn { str } { utf8-decode }
909 {
910   Invalid-UTF-8-string: \exp_last_unbraced:Nf \use_none:n
911   \__str_if_flag_times:nT { str_missing } { ,~missing-continuation-byte }
912   \__str_if_flag_times:nT { str_extra } { ,~extra-continuation-byte }
913   \__str_if_flag_times:nT { str_overlong } { ,~overlong-form }
914   \__str_if_flag_times:nT { str_overflow } { ,~code-point-too-large }
915   .
916 }
917 {
918   In-the-UTF-8-encoding,~each~Unicode~character~consists~in~
919   1-to~4~bytes,~with~the~following~bit~pattern: \\
920   \iow_indent:n
921   {
922     Code~point~\ \ \ \ <~128:-0xxxxxxx \\
923     Code~point~\ \ \ \ <~2048:-110xxxxx-10xxxxxx \\
924     Code~point~\ \ \ \ <~65536:-1110xxxx-10xxxxxx-10xxxxxx \\
925     Code~point~\ \ \ \ <~1114112:-11110xxx-10xxxxxx-10xxxxxx-10xxxxxx \\
926   }
927   Bytes-of~the~form~10xxxxxx~are~called~continuation~bytes.
928   \flag_if_raised:nT { str_missing }
929   {
930     \\\
931     A~leading~byte~(in~the~range~[192,255])~was~not~followed~by~
932     the~appropriate~number~of~continuation~bytes.
933   }
934   \flag_if_raised:nT { str_extra }
935   {
936     \\\
937     LaTeX~came~across~a~continuation~byte~when~it~was~not~expected.
938   }
939   \flag_if_raised:nT { str_overlong }
940   {
941     \\\
942     Every~Unicode~code~point~must~be~expressed~in~the~shortest~
943     possible~form.~For~instance,~'0xC0'~'0x83'~is~not~a~valid~
944     representation~for~the~code~point~3.
945   }
946   \flag_if_raised:nT { str_overflow }
947   {
948     \\\
949     Unicode~limits~code~points~to~the~range~[0,1114111].
950   }
951 }

```

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

`__str_convert_decode_utf8:` 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 $\langle start\ byte \rangle \langle continuation\ bytes \rangle$. The `_start` auxiliary tests the first byte:

`_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:`

- `[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.

```

952 \cs_new_protected_nopar:cpn { __str_convert_decode_utf8: }
953 {
954   \flag_clear:n { str_error }
955   \flag_clear:n { str_missing }
956   \flag_clear:n { str_extra }
957   \flag_clear:n { str_overlong }
958   \flag_clear:n { str_overflow }
959   \tl_gset:Nx \g__str_result_tl
960   {
961     \exp_after:wN \__str_decode_utf_viii_start:N \g__str_result_tl

```

```

962         { \__prg_break: \__str_decode_utf_viii_end: }
963         \__prg_break_point:
964     }
965     \__str_if_flag_error:nmx { str_error } { utf8-decode } { }
966 }
967 \cs_new:Npn \__str_decode_utf_viii_start:N #1
968 {
969     #1
970     \if_int_compare:w '#1 < "C0 \exp_stop_f:
971         \s__tl
972         \if_int_compare:w '#1 < "80 \exp_stop_f:
973             \__int_value:w '#1
974         \else:
975             \flag_raise:n { str_extra }
976             \flag_raise:n { str_error }
977             \int_use:N \c__str_replacement_char_int
978         \fi:
979     \else:
980         \exp_after:wN \__str_decode_utf_viii_continuation:wwN
981         \int_use:N \__int_eval:w '#1 - "C0 \exp_after:wN \__int_eval_end:
982     \fi:
983     \s__tl
984     \use_none_delimit_by_q_stop:w {"80} {"800} {"10000} {"110000} \q_stop
985     \__str_decode_utf_viii_start:N
986 }
987 \cs_new:Npn \__str_decode_utf_viii_continuation:wwN
988     #1 \s__tl #2 \__str_decode_utf_viii_start:N #3
989 {
990     \use_none:n #3
991     \if_int_compare:w '#3 <
992         \if_int_compare:w '#3 < "80 \exp_stop_f: - \fi:
993         "C0 \exp_stop_f:
994     #3
995     \exp_after:wN \__str_decode_utf_viii_aux:wNnnwN
996     \int_use:N \__int_eval:w
997         #1 * "40 + '#3 - "80
998     \exp_after:wN \__int_eval_end:
999     \else:
1000         \s__tl
1001         \flag_raise:n { str_missing }
1002         \flag_raise:n { str_error }
1003         \int_use:N \c__str_replacement_char_int
1004     \fi:
1005     \s__tl
1006     #2
1007     \__str_decode_utf_viii_start:N #3
1008 }
1009 \cs_new:Npn \__str_decode_utf_viii_aux:wNnnwN
1010     #1 \s__tl #2#3#4 #5 \__str_decode_utf_viii_start:N #6
1011 {

```

```

1012 \if_int_compare:w #1 < #4 \exp_stop_f:
1013   \s__tl
1014   \if_int_compare:w #1 < #3 \exp_stop_f:
1015     \flag_raise:n { str_overlong }
1016     \flag_raise:n { str_error }
1017     \int_use:N \c__str_replacement_char_int
1018   \else:
1019     #1
1020   \fi:
1021 \else:
1022   \if_meaning:w \q_stop #5
1023     \__str_decode_utf_viii_overflow:w #1
1024   \fi:
1025   \exp_after:wN \__str_decode_utf_viii_continuation:wwN
1026   \int_use:N \__int_eval:w #1 - #4 \exp_after:wN \__int_eval_end:
1027 \fi:
1028 \s__tl
1029 #2 {#4} #5
1030 \__str_decode_utf_viii_start:N
1031 }
1032 \cs_new:Npn \__str_decode_utf_viii_overflow:w #1 \fi: #2 \fi:
1033 {
1034   \fi: \fi:
1035   \flag_raise:n { str_overflow }
1036   \flag_raise:n { str_error }
1037   \int_use:N \c__str_replacement_char_int
1038 }
1039 \cs_new_nopar:Npn \__str_decode_utf_viii_end:
1040 {
1041   \s__tl
1042   \flag_raise:n { str_missing }
1043   \flag_raise:n { str_error }
1044   \int_use:N \c__str_replacement_char_int \s__tl
1045   \__prg_break:
1046 }

```

(End definition for __str_convert_decode_utf8:. This function is documented on page ??.)

1047 </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.

```

1048 <*utf16>
1049 \group_begin:
1050   \char_set_catcode_other:N \^^fe
1051   \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

```

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

```

the character code.

- [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.

```

1052 \cs_new_protected_nopar:cpn { __str_convert_encode_utf16: }
1053 {
1054   \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n
1055   \tl_gput_left:Nx \g__str_result_tl { ^^fe ^^ff }
1056 }
1057 \cs_new_protected_nopar:cpn { __str_convert_encode_utf16be: }
1058 { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n }
1059 \cs_new_protected_nopar:cpn { __str_convert_encode_utf16le: }
1060 { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_le:n }
1061 \cs_new_protected:Npn \__str_encode_utf_xvi_aux:N #1
1062 {
1063   \flag_clear:n { str_error }
1064   \cs_set_eq:NN \__str_tmp:w #1
1065   \__str_convert_gmap_internal:N \__str_encode_utf_xvi_char:n
1066   \__str_if_flag_error:nnx { str_error } { utf16-encode } { }
1067 }
1068 \cs_new:Npn \__str_encode_utf_xvi_char:n #1
1069 {
1070   \if_int_compare:w #1 < "D800 \exp_stop_f:
1071     \__str_tmp:w {#1}
1072   \else:
1073     \if_int_compare:w #1 < "10000 \exp_stop_f:
1074       \if_int_compare:w #1 < "E000 \exp_stop_f:
1075         \flag_raise:n { str_error }
1076         \__str_tmp:w { \c__str_replacement_char_int }
1077       \else:
1078         \__str_tmp:w {#1}
1079       \fi:
1080     \else:
1081       \exp_args:Nf \__str_tmp:w { \int_div_truncate:nn {#1} {"400} + "D7C0 }
1082       \exp_args:Nf \__str_tmp:w { \int_mod:nn {#1} {"400} + "DC00 }
1083     \fi:
1084   \fi:
1085 }

```

(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
`\l__str_extra_flag` the range ["D800,"DFFF], corresponding to surrogates, cannot be encoded. We use the
`\l__str_end_flag` all-purpose flag `@@_error` to signal that error.

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.

```

1086 \flag_clear_new:n { str_missing }
1087 \flag_clear_new:n { str_extra }
1088 \flag_clear_new:n { str_end }
1089 \__msg_kernel_new:nnnn { str } { utf16-encode }
1090 { Unicode~string~cannot~be~expressed~in~UTF-16:~surrogate. }
1091 {
1092   Surrogate~code~points~(in~the~range~[U+D800,~U+DFFF])~
1093   can~be~expressed~in~the~UTF-8~and~UTF-32~encodings,~
1094   but~not~in~the~UTF-16~encoding.
1095 }
1096 \__msg_kernel_new:nnnn { str } { utf16-decode }
1097 {
1098   Invalid~UTF-16~string: \exp_last_unbraced:Nf \use_none:n
1099   \__str_if_flag_times:nT { str_missing } { ,~missing~trail~surrogate }
1100   \__str_if_flag_times:nT { str_extra } { ,~extra~trail~surrogate }
1101   \__str_if_flag_times:nT { str_end } { ,~odd~number~of~bytes }
1102   .
1103 }
1104 {
1105   In~the~UTF-16~encoding,~each~Unicode~character~is~encoded~as~
1106   2~or~4~bytes: \\
1107   \iow_indent:n
1108   {
1109     Code~point~in~[U+0000,~U+D7FF]:~two~bytes \\
1110     Code~point~in~[U+D800,~U+DFFF]:~illegal \\
1111     Code~point~in~[U+E000,~U+FFFF]:~two~bytes \\
1112     Code~point~in~[U+10000,~U+10FFFF]:~
1113     a~lead~surrogate~and~a~trail~surrogate \\
1114   }
1115   Lead~surrogates~are~pairs~of~bytes~in~the~range~[0xD800,~0xDBFF],~
1116   and~trail~surrogates~are~in~the~range~[0xDC00,~0xDFFF].
1117   \flag_if_raised:nT { str_missing }
1118   {
1119     \\ \\
1120     A~lead~surrogate~was~not~followed~by~a~trail~surrogate.
1121   }
1122   \flag_if_raised:nT { str_extra }
1123   {
1124     \\ \\
1125     LaTeX~came~across~a~trail~surrogate~when~it~was~not~expected.

```

```

1126     }
1127     \flag_if_raised:nT { str_end }
1128     {
1129         \\\
1130         The~string~contained~an~odd~number~of~bytes.~This~is~invalid:~
1131         the~basic~code~unit~for~UTF-16~is~16~bits~(2~bytes).
1132     }
1133 }

```

(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.

```

1134 \cs_new_protected_nopar:cpn { __str_convert_decode_utf16be: }
1135 { \__str_decode_utf_xvi:Nw 1 \g__str_result_tl \s__stop }
1136 \cs_new_protected_nopar:cpn { __str_convert_decode_utf16le: }
1137 { \__str_decode_utf_xvi:Nw 2 \g__str_result_tl \s__stop }
1138 \cs_new_protected_nopar:cpn { __str_convert_decode_utf16: }
1139 {
1140     \exp_after:wN \__str_decode_utf_xvi_bom:NN
1141     \g__str_result_tl \s__stop \s__stop \s__stop
1142 }
1143 \cs_new_protected:Npn \__str_decode_utf_xvi_bom:NN #1#2
1144 {
1145     \str_if_eq_x:nnTF { #1#2 } { ^^ff ^^fe }
1146     { \__str_decode_utf_xvi:Nw 2 }
1147     {
1148         \str_if_eq_x:nnTF { #1#2 } { ^^fe ^^ff }
1149         { \__str_decode_utf_xvi:Nw 1 }
1150         { \__str_decode_utf_xvi:Nw 1 #1#2 }
1151     }
1152 }
1153 \cs_new_protected:Npn \__str_decode_utf_xvi:Nw #1#2 \s__stop
1154 {
1155     \flag_clear:n { str_error }
1156     \flag_clear:n { str_missing }
1157     \flag_clear:n { str_extra }
1158     \flag_clear:n { str_end }
1159     \cs_set:Npn \__str_tmp:w ##1 ##2 { ' ## #1 }
1160     \tl_gset:Nx \g__str_result_tl

```



```

1161     {
1162         \exp_after:wN \_str_decode_utf_xvi_pair:NN
1163         #2 \q_nil \q_nil
1164         \_prg_break_point:
1165     }
1166     \_str_if_flag_error:nxx { str_error } { utf16-decode } { }
1167 }

```

(End definition for `_str_convert_decode_utf16:`, `_str_convert_decode_utf16be:`, and `_str_convert_decode_utf16l:`.
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.

```

1168 \cs_new:Npn \_str_decode_utf_xvi_pair:NN #1#2
1169 {
1170     \if_meaning:w \q_nil #2
1171     \_str_decode_utf_xvi_pair_end:Nw #1
1172     \fi:
1173     \if_case:w
1174         \_int_eval:w ( \_str_tmp:w #1#2 - "D6 ) / \c_four \_int_eval_end:
1175     \or: \exp_after:wN \_str_decode_utf_xvi_quad:NNwNN
1176     \or: \exp_after:wN \_str_decode_utf_xvi_extra:NNw
1177     \fi:
1178     #1#2 \s__tl
1179     \int_eval:n { "100 * \_str_tmp:w #1#2 + \_str_tmp:w #2#1 } \s__tl
1180     \_str_decode_utf_xvi_pair:NN

```

```

1181     }
1182     \cs_new:Npn \__str_decode_utf_xvi_quad:NNwNN
1183       #1#2 #3 \__str_decode_utf_xvi_pair:NN #4#5
1184     {
1185       \if_meaning:w \q_nil #5
1186         \__str_decode_utf_xvi_error:nNN { missing } #1#2
1187         \__str_decode_utf_xvi_pair_end:Nw #4
1188       \fi:
1189       \if_int_compare:w
1190         \if_int_compare:w \__str_tmp:w #4#5 < "DC \exp_stop_f:
1191           \c_zero = \c_one
1192         \else:
1193           \__str_tmp:w #4#5 < "E0 \exp_stop_f:
1194         \fi:
1195         #1 #2 #4 #5 \s__tl
1196         \int_eval:n
1197         {
1198           ( "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 - "D7F7 ) * "400
1199           + "100 * \__str_tmp:w #4#5 + \__str_tmp:w #5#4
1200         }
1201         \s__tl
1202         \exp_after:wN \use_i:nnn
1203       \else:
1204         \__str_decode_utf_xvi_error:nNN { missing } #1#2
1205       \fi:
1206       \__str_decode_utf_xvi_pair:NN #4#5
1207     }
1208     \cs_new:Npn \__str_decode_utf_xvi_pair_end:Nw #1 \fi:
1209     {
1210       \fi:
1211       \if_meaning:w \q_nil #1
1212       \else:
1213         \__str_decode_utf_xvi_error:nNN { end } #1 \prg_do_nothing:
1214       \fi:
1215       \__prg_break:
1216     }
1217     \cs_new:Npn \__str_decode_utf_xvi_extra:NNw #1#2 \s__tl #3 \s__tl
1218     { \__str_decode_utf_xvi_error:nNN { extra } #1#2 }
1219     \cs_new:Npn \__str_decode_utf_xvi_error:nNN #1#2#3
1220     {
1221       \flag_raise:n { str_error }
1222       \flag_raise:n { str_#1 }
1223       #2 #3 \s__tl
1224       \int_use:N \c__str_replacement_char_int \s__tl
1225     }

```

(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.

```

1226 \group_end:

```

1227 </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”.

```
1228 <*utf32>
1229 \group_begin:
1230   \char_set_catcode_other:N \^^00
1231   \char_set_catcode_other:N \^^fe
1232   \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.
`__str_convert_encode_utf32be:` The functions for big-endian and little-endian encodings are very similar, but the `__-`
`__str_convert_encode_utf32le:` `str_output_byte:n` instructions are reversed.

```
\__str_encode_utf_XXXii_be:n 1233   \cs_new_protected_nopar:cpn { __str_convert_encode_utf32: }
    \__str_encode_utf_XXXii_be_aux:nn 1234   {
\__str_encode_utf_XXXii_le:n 1235     \__str_convert_gmap_internal:N \__str_encode_utf_XXXii_be:n
    \__str_encode_utf_XXXii_le_aux:nn 1236     \tl_gput_left:Nx \g__str_result_tl { ^^00 ^^00 ^^fe ^^ff }
1237   }
1238   \cs_new_protected_nopar:cpn { __str_convert_encode_utf32be: }
1239   { \__str_convert_gmap_internal:N \__str_encode_utf_XXXii_be:n }
1240   \cs_new_protected_nopar:cpn { __str_convert_encode_utf32le: }
1241   { \__str_convert_gmap_internal:N \__str_encode_utf_XXXii_le:n }
1242   \cs_new:Npn \__str_encode_utf_XXXii_be:n #1
1243   {
1244     \exp_args:Nf \__str_encode_utf_XXXii_be_aux:nn
1245     { \int_div_truncate:nn {#1} { "100 } } {#1}
1246   }
1247   \cs_new:Npn \__str_encode_utf_XXXii_be_aux:nn #1#2
1248   {
1249     ^^00
1250     \__str_output_byte_pair_be:n {#1}
1251     \__str_output_byte:n { #2 - #1 * "100 }
1252   }
1253   \cs_new:Npn \__str_encode_utf_XXXii_le:n #1
1254   {
1255     \exp_args:Nf \__str_encode_utf_XXXii_le_aux:nn
1256     { \int_div_truncate:nn {#1} { "100 } } {#1}
1257   }
1258   \cs_new:Npn \__str_encode_utf_XXXii_le_aux:nn #1#2
1259   {
1260     \__str_output_byte:n { #2 - #1 * "100 }
1261     \__str_output_byte_pair_le:n {#1}
1262     ^^00
1263   }
```

(End definition for `__str_convert_encode_utf32:`, `__str_convert_encode_utf32be:`, and `__str_convert_encode_utf32le:`.
These functions are documented on page ??.)

str_overflow There can be no error when encoding in UTF-32. When decoding, the string may not 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.

```

1264 \flag_clear_new:n { str_overflow }
1265 \flag_clear_new:n { str_end }
1266 \__msg_kernel_new:nnnn { str } { utf32-decode }
1267 {
1268   Invalid~UTF-32~string: \exp_last_unbraced:Nf \use_none:n
1269   \__str_if_flag_times:nT { str_overflow } { ,~code~point~too~large }
1270   \__str_if_flag_times:nT { str_end } { ,~truncated~string }
1271   .
1272 }
1273 {
1274   In~the~UTF-32~encoding,~every~Unicode~character~
1275   (in~the~range~[U+0000,~U+10FFFF])~is~encoded~as~4~bytes.
1276   \flag_if_raised:nT { str_overflow }
1277   {
1278     \\\
1279     LaTeX~came~across~a~code~point~larger~than~1114111,~
1280     the~maximum~code~point~defined~by~Unicode.~
1281     Perhaps~the~string~was~not~encoded~in~the~UTF-32~encoding?
1282   }
1283   \flag_if_raised:nT { str_end }
1284   {
1285     \\\
1286     The~length~of~the~string~is~not~a~multiple~of~4.~
1287     Perhaps~the~string~was~truncated?
1288   }
1289 }

```

(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, 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_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.

```

1290 \cs_new_protected_nopar:cpn { __str_convert_decode_utf32be: }

```

```

1291     { \__str_decode_utf_xxxii:Nw 1 \g__str_result_tl \s__stop }
1292 \cs_new_protected_nopar:cpn { __str_convert_decode_utf32le: }
1293     { \__str_decode_utf_xxxii:Nw 2 \g__str_result_tl \s__stop }
1294 \cs_new_protected_nopar:cpn { __str_convert_decode_utf32: }
1295     {
1296     \exp_after:wN \__str_decode_utf_xxxii_bom:NNNN \g__str_result_tl
1297     \s__stop \s__stop \s__stop \s__stop \s__stop
1298     }
1299 \cs_new_protected:Npn \__str_decode_utf_xxxii_bom:NNNN #1#2#3#4
1300     {
1301     \str_if_eq_x:nnTF { #1#2#3#4 } { ^^ff ^^fe ^^00 ^^00 }
1302     { \__str_decode_utf_xxxii:Nw 2 }
1303     {
1304     \str_if_eq_x:nnTF { #1#2#3#4 } { ^^00 ^^00 ^^fe ^^ff }
1305     { \__str_decode_utf_xxxii:Nw 1 }
1306     { \__str_decode_utf_xxxii:Nw 1 #1#2#3#4 }
1307     }
1308     }
1309 \cs_new_protected:Npn \__str_decode_utf_xxxii:Nw #1#2 \s__stop
1310     {
1311     \flag_clear:n { str_overflow }
1312     \flag_clear:n { str_end }
1313     \flag_clear:n { str_error }
1314     \cs_set:Npn \__str_tmp:w ##1 ##2 { ' ## #1 }
1315     \tl_gset:Nx \g__str_result_tl
1316     {
1317     \exp_after:wN \__str_decode_utf_xxxii_loop:NNNN
1318     #2 \s__stop \s__stop \s__stop \s__stop
1319     \__prg_break_point:
1320     }
1321     \__str_if_flag_error:nnx { str_error } { utf32-decode } { }
1322     }
1323 \cs_new:Npn \__str_decode_utf_xxxii_loop:NNNN #1#2#3#4
1324     {
1325     \if_meaning:w \s__stop #4
1326     \exp_after:wN \__str_decode_utf_xxxii_end:w
1327     \fi:
1328     #1#2#3#4 \s__tl
1329     \if_int_compare:w \__str_tmp:w #1#4 > \c_zero
1330     \flag_raise:n { str_overflow }
1331     \flag_raise:n { str_error }
1332     \int_use:N \c__str_replacement_char_int
1333     \else:
1334     \if_int_compare:w \__str_tmp:w #2#3 > \c_sixteen
1335     \flag_raise:n { str_overflow }
1336     \flag_raise:n { str_error }
1337     \int_use:N \c__str_replacement_char_int
1338     \else:
1339     \int_eval:n
1340     { \__str_tmp:w #2#3*"10000 + \__str_tmp:w #3#2*"100 + \__str_tmp:w #4#1 }

```

```

1341     \fi:
1342     \fi:
1343     \s__tl
1344     \__str_decode_utf_xxxii_loop:NNNN
1345   }
1346   \cs_new:Npn \__str_decode_utf_xxxii_end:w #1 \s__stop
1347   {
1348     \tl_if_empty:nF {#1}
1349     {
1350       \flag_raise:n { str_end }
1351       \flag_raise:n { str_error }
1352       #1 \s__tl
1353       \int_use:N \c__str_replacement_char_int \s__tl
1354     }
1355     \__prg_break:
1356   }

```

(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.

```

1357 \group_end:
1358 </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.

```

1359 <*iso88591>
1360 \__str_declare_eight_bit_encoding:nnn { iso88591 }
1361 {
1362 }
1363 {
1364 }
1365 </iso88591>
1366 <*iso88592>
1367 \__str_declare_eight_bit_encoding:nnn { iso88592 }
1368 {
1369   { A1 } { 0104 }
1370   { A2 } { 02D8 }
1371   { A3 } { 0141 }
1372   { A5 } { 013D }
1373   { A6 } { 015A }
1374   { A9 } { 0160 }
1375   { AA } { 015E }
1376   { AB } { 0164 }
1377   { AC } { 0179 }
1378   { AE } { 017D }
1379   { AF } { 017B }

```

```

1380 { B1 } { 0105 }
1381 { B2 } { 02DB }
1382 { B3 } { 0142 }
1383 { B5 } { 013E }
1384 { B6 } { 015B }
1385 { B7 } { 02C7 }
1386 { B9 } { 0161 }
1387 { BA } { 015F }
1388 { BB } { 0165 }
1389 { BC } { 017A }
1390 { BD } { 02DD }
1391 { BE } { 017E }
1392 { BF } { 017C }
1393 { C0 } { 0154 }
1394 { C3 } { 0102 }
1395 { C5 } { 0139 }
1396 { C6 } { 0106 }
1397 { C8 } { 010C }
1398 { CA } { 0118 }
1399 { CC } { 011A }
1400 { CF } { 010E }
1401 { D0 } { 0110 }
1402 { D1 } { 0143 }
1403 { D2 } { 0147 }
1404 { D5 } { 0150 }
1405 { D8 } { 0158 }
1406 { D9 } { 016E }
1407 { DB } { 0170 }
1408 { DE } { 0162 }
1409 { E0 } { 0155 }
1410 { E3 } { 0103 }
1411 { E5 } { 013A }
1412 { E6 } { 0107 }
1413 { E8 } { 010D }
1414 { EA } { 0119 }
1415 { EC } { 011B }
1416 { EF } { 010F }
1417 { F0 } { 0111 }
1418 { F1 } { 0144 }
1419 { F2 } { 0148 }
1420 { F5 } { 0151 }
1421 { F8 } { 0159 }
1422 { F9 } { 016F }
1423 { FB } { 0171 }
1424 { FE } { 0163 }
1425 { FF } { 02D9 }
1426 }
1427 {
1428 }
1429 </iso88592>

```

```

1430 <iso88593>
1431 \_str_declare_eight_bit_encoding:nnn { iso88593 }
1432 {
1433     { A1 } { 0126 }
1434     { A2 } { 02D8 }
1435     { A6 } { 0124 }
1436     { A9 } { 0130 }
1437     { AA } { 015E }
1438     { AB } { 011E }
1439     { AC } { 0134 }
1440     { AF } { 017B }
1441     { B1 } { 0127 }
1442     { B6 } { 0125 }
1443     { B9 } { 0131 }
1444     { BA } { 015F }
1445     { BB } { 011F }
1446     { BC } { 0135 }
1447     { BF } { 017C }
1448     { C5 } { 010A }
1449     { C6 } { 0108 }
1450     { D5 } { 0120 }
1451     { D8 } { 011C }
1452     { DD } { 016C }
1453     { DE } { 015C }
1454     { E5 } { 010B }
1455     { E6 } { 0109 }
1456     { F5 } { 0121 }
1457     { F8 } { 011D }
1458     { FD } { 016D }
1459     { FE } { 015D }
1460     { FF } { 02D9 }
1461 }
1462 {
1463     { A5 }
1464     { AE }
1465     { BE }
1466     { C3 }
1467     { D0 }
1468     { E3 }
1469     { F0 }
1470 }
1471 </iso88593>
1472 <iso88594>
1473 \_str_declare_eight_bit_encoding:nnn { iso88594 }
1474 {
1475     { A1 } { 0104 }
1476     { A2 } { 0138 }
1477     { A3 } { 0156 }
1478     { A5 } { 0128 }

```



```

1479 { A6 } { 013B }
1480 { A9 } { 0160 }
1481 { AA } { 0112 }
1482 { AB } { 0122 }
1483 { AC } { 0166 }
1484 { AE } { 017D }
1485 { B1 } { 0105 }
1486 { B2 } { 02DB }
1487 { B3 } { 0157 }
1488 { B5 } { 0129 }
1489 { B6 } { 013C }
1490 { B7 } { 02C7 }
1491 { B9 } { 0161 }
1492 { BA } { 0113 }
1493 { BB } { 0123 }
1494 { BC } { 0167 }
1495 { BD } { 014A }
1496 { BE } { 017E }
1497 { BF } { 014B }
1498 { C0 } { 0100 }
1499 { C7 } { 012E }
1500 { C8 } { 010C }
1501 { CA } { 0118 }
1502 { CC } { 0116 }
1503 { CF } { 012A }
1504 { D0 } { 0110 }
1505 { D1 } { 0145 }
1506 { D2 } { 014C }
1507 { D3 } { 0136 }
1508 { D9 } { 0172 }
1509 { DD } { 0168 }
1510 { DE } { 016A }
1511 { E0 } { 0101 }
1512 { E7 } { 012F }
1513 { E8 } { 010D }
1514 { EA } { 0119 }
1515 { EC } { 0117 }
1516 { EF } { 012B }
1517 { F0 } { 0111 }
1518 { F1 } { 0146 }
1519 { F2 } { 014D }
1520 { F3 } { 0137 }
1521 { F9 } { 0173 }
1522 { FD } { 0169 }
1523 { FE } { 016B }
1524 { FF } { 02D9 }
1525 }
1526 {
1527 }
1528 </iso88594>

```

```

1529 <*iso88595>
1530 \_str_declare_eight_bit_encoding:nnn { iso88595 }
1531 {
1532     { A1 } { 0401 }
1533     { A2 } { 0402 }
1534     { A3 } { 0403 }
1535     { A4 } { 0404 }
1536     { A5 } { 0405 }
1537     { A6 } { 0406 }
1538     { A7 } { 0407 }
1539     { A8 } { 0408 }
1540     { A9 } { 0409 }
1541     { AA } { 040A }
1542     { AB } { 040B }
1543     { AC } { 040C }
1544     { AE } { 040E }
1545     { AF } { 040F }
1546     { B0 } { 0410 }
1547     { B1 } { 0411 }
1548     { B2 } { 0412 }
1549     { B3 } { 0413 }
1550     { B4 } { 0414 }
1551     { B5 } { 0415 }
1552     { B6 } { 0416 }
1553     { B7 } { 0417 }
1554     { B8 } { 0418 }
1555     { B9 } { 0419 }
1556     { BA } { 041A }
1557     { BB } { 041B }
1558     { BC } { 041C }
1559     { BD } { 041D }
1560     { BE } { 041E }
1561     { BF } { 041F }
1562     { C0 } { 0420 }
1563     { C1 } { 0421 }
1564     { C2 } { 0422 }
1565     { C3 } { 0423 }
1566     { C4 } { 0424 }
1567     { C5 } { 0425 }
1568     { C6 } { 0426 }
1569     { C7 } { 0427 }
1570     { C8 } { 0428 }
1571     { C9 } { 0429 }
1572     { CA } { 042A }
1573     { CB } { 042B }
1574     { CC } { 042C }
1575     { CD } { 042D }
1576     { CE } { 042E }
1577     { CF } { 042F }
1578     { D0 } { 0430 }

```

```

1579 { D1 } { 0431 }
1580 { D2 } { 0432 }
1581 { D3 } { 0433 }
1582 { D4 } { 0434 }
1583 { D5 } { 0435 }
1584 { D6 } { 0436 }
1585 { D7 } { 0437 }
1586 { D8 } { 0438 }
1587 { D9 } { 0439 }
1588 { DA } { 043A }
1589 { DB } { 043B }
1590 { DC } { 043C }
1591 { DD } { 043D }
1592 { DE } { 043E }
1593 { DF } { 043F }
1594 { E0 } { 0440 }
1595 { E1 } { 0441 }
1596 { E2 } { 0442 }
1597 { E3 } { 0443 }
1598 { E4 } { 0444 }
1599 { E5 } { 0445 }
1600 { E6 } { 0446 }
1601 { E7 } { 0447 }
1602 { E8 } { 0448 }
1603 { E9 } { 0449 }
1604 { EA } { 044A }
1605 { EB } { 044B }
1606 { EC } { 044C }
1607 { ED } { 044D }
1608 { EE } { 044E }
1609 { EF } { 044F }
1610 { F0 } { 2116 }
1611 { F1 } { 0451 }
1612 { F2 } { 0452 }
1613 { F3 } { 0453 }
1614 { F4 } { 0454 }
1615 { F5 } { 0455 }
1616 { F6 } { 0456 }
1617 { F7 } { 0457 }
1618 { F8 } { 0458 }
1619 { F9 } { 0459 }
1620 { FA } { 045A }
1621 { FB } { 045B }
1622 { FC } { 045C }
1623 { FD } { 00A7 }
1624 { FE } { 045E }
1625 { FF } { 045F }
1626 }
1627 {
1628 }

```

```

1629 </iso88595>
1630 <*iso88596>
1631 \__str_declare_eight_bit_encoding:nnn { iso88596 }
1632 {
1633     { AC } { 060C }
1634     { BB } { 061B }
1635     { BF } { 061F }
1636     { C1 } { 0621 }
1637     { C2 } { 0622 }
1638     { C3 } { 0623 }
1639     { C4 } { 0624 }
1640     { C5 } { 0625 }
1641     { C6 } { 0626 }
1642     { C7 } { 0627 }
1643     { C8 } { 0628 }
1644     { C9 } { 0629 }
1645     { CA } { 062A }
1646     { CB } { 062B }
1647     { CC } { 062C }
1648     { CD } { 062D }
1649     { CE } { 062E }
1650     { CF } { 062F }
1651     { D0 } { 0630 }
1652     { D1 } { 0631 }
1653     { D2 } { 0632 }
1654     { D3 } { 0633 }
1655     { D4 } { 0634 }
1656     { D5 } { 0635 }
1657     { D6 } { 0636 }
1658     { D7 } { 0637 }
1659     { D8 } { 0638 }
1660     { D9 } { 0639 }
1661     { DA } { 063A }
1662     { E0 } { 0640 }
1663     { E1 } { 0641 }
1664     { E2 } { 0642 }
1665     { E3 } { 0643 }
1666     { E4 } { 0644 }
1667     { E5 } { 0645 }
1668     { E6 } { 0646 }
1669     { E7 } { 0647 }
1670     { E8 } { 0648 }
1671     { E9 } { 0649 }
1672     { EA } { 064A }
1673     { EB } { 064B }
1674     { EC } { 064C }
1675     { ED } { 064D }
1676     { EE } { 064E }
1677     { EF } { 064F }

```

```

1678     { F0 } { 0650 }
1679     { F1 } { 0651 }
1680     { F2 } { 0652 }
1681 }
1682 {
1683     { A1 }
1684     { A2 }
1685     { A3 }
1686     { A5 }
1687     { A6 }
1688     { A7 }
1689     { A8 }
1690     { A9 }
1691     { AA }
1692     { AB }
1693     { AE }
1694     { AF }
1695     { B0 }
1696     { B1 }
1697     { B2 }
1698     { B3 }
1699     { B4 }
1700     { B5 }
1701     { B6 }
1702     { B7 }
1703     { B8 }
1704     { B9 }
1705     { BA }
1706     { BC }
1707     { BD }
1708     { BE }
1709     { C0 }
1710     { DB }
1711     { DC }
1712     { DD }
1713     { DE }
1714     { DF }
1715 }
1716 </iso88596>
1717 <*iso88597>
1718 \__str_declare_eight_bit_encoding:nnn { iso88597 }
1719 {
1720     { A1 } { 2018 }
1721     { A2 } { 2019 }
1722     { A4 } { 20AC }
1723     { A5 } { 20AF }
1724     { AA } { 037A }
1725     { AF } { 2015 }
1726     { B4 } { 0384 }

```

1727	{ B5 }	{ 0385 }
1728	{ B6 }	{ 0386 }
1729	{ B8 }	{ 0388 }
1730	{ B9 }	{ 0389 }
1731	{ BA }	{ 038A }
1732	{ BC }	{ 038C }
1733	{ BE }	{ 038E }
1734	{ BF }	{ 038F }
1735	{ C0 }	{ 0390 }
1736	{ C1 }	{ 0391 }
1737	{ C2 }	{ 0392 }
1738	{ C3 }	{ 0393 }
1739	{ C4 }	{ 0394 }
1740	{ C5 }	{ 0395 }
1741	{ C6 }	{ 0396 }
1742	{ C7 }	{ 0397 }
1743	{ C8 }	{ 0398 }
1744	{ C9 }	{ 0399 }
1745	{ CA }	{ 039A }
1746	{ CB }	{ 039B }
1747	{ CC }	{ 039C }
1748	{ CD }	{ 039D }
1749	{ CE }	{ 039E }
1750	{ CF }	{ 039F }
1751	{ D0 }	{ 03A0 }
1752	{ D1 }	{ 03A1 }
1753	{ D3 }	{ 03A3 }
1754	{ D4 }	{ 03A4 }
1755	{ D5 }	{ 03A5 }
1756	{ D6 }	{ 03A6 }
1757	{ D7 }	{ 03A7 }
1758	{ D8 }	{ 03A8 }
1759	{ D9 }	{ 03A9 }
1760	{ DA }	{ 03AA }
1761	{ DB }	{ 03AB }
1762	{ DC }	{ 03AC }
1763	{ DD }	{ 03AD }
1764	{ DE }	{ 03AE }
1765	{ DF }	{ 03AF }
1766	{ E0 }	{ 03B0 }
1767	{ E1 }	{ 03B1 }
1768	{ E2 }	{ 03B2 }
1769	{ E3 }	{ 03B3 }
1770	{ E4 }	{ 03B4 }
1771	{ E5 }	{ 03B5 }
1772	{ E6 }	{ 03B6 }
1773	{ E7 }	{ 03B7 }
1774	{ E8 }	{ 03B8 }
1775	{ E9 }	{ 03B9 }
1776	{ EA }	{ 03BA }

```

1777     { EB } { 03BB }
1778     { EC } { 03BC }
1779     { ED } { 03BD }
1780     { EE } { 03BE }
1781     { EF } { 03BF }
1782     { F0 } { 03C0 }
1783     { F1 } { 03C1 }
1784     { F2 } { 03C2 }
1785     { F3 } { 03C3 }
1786     { F4 } { 03C4 }
1787     { F5 } { 03C5 }
1788     { F6 } { 03C6 }
1789     { F7 } { 03C7 }
1790     { F8 } { 03C8 }
1791     { F9 } { 03C9 }
1792     { FA } { 03CA }
1793     { FB } { 03CB }
1794     { FC } { 03CC }
1795     { FD } { 03CD }
1796     { FE } { 03CE }
1797 }
1798 {
1799     { AE }
1800     { D2 }
1801 }
1802 </iso88597>
1803 <*iso88598>
1804 \__str_declare_eight_bit_encoding:nnn { iso88598 }
1805 {
1806     { AA } { 00D7 }
1807     { BA } { 00F7 }
1808     { DF } { 2017 }
1809     { E0 } { 05D0 }
1810     { E1 } { 05D1 }
1811     { E2 } { 05D2 }
1812     { E3 } { 05D3 }
1813     { E4 } { 05D4 }
1814     { E5 } { 05D5 }
1815     { E6 } { 05D6 }
1816     { E7 } { 05D7 }
1817     { E8 } { 05D8 }
1818     { E9 } { 05D9 }
1819     { EA } { 05DA }
1820     { EB } { 05DB }
1821     { EC } { 05DC }
1822     { ED } { 05DD }
1823     { EE } { 05DE }
1824     { EF } { 05DF }
1825     { F0 } { 05E0 }

```

```

1826 { F1 } { 05E1 }
1827 { F2 } { 05E2 }
1828 { F3 } { 05E3 }
1829 { F4 } { 05E4 }
1830 { F5 } { 05E5 }
1831 { F6 } { 05E6 }
1832 { F7 } { 05E7 }
1833 { F8 } { 05E8 }
1834 { F9 } { 05E9 }
1835 { FA } { 05EA }
1836 { FD } { 200E }
1837 { FE } { 200F }
1838 }
1839 {
1840 { A1 }
1841 { BF }
1842 { C0 }
1843 { C1 }
1844 { C2 }
1845 { C3 }
1846 { C4 }
1847 { C5 }
1848 { C6 }
1849 { C7 }
1850 { C8 }
1851 { C9 }
1852 { CA }
1853 { CB }
1854 { CC }
1855 { CD }
1856 { CE }
1857 { CF }
1858 { D0 }
1859 { D1 }
1860 { D2 }
1861 { D3 }
1862 { D4 }
1863 { D5 }
1864 { D6 }
1865 { D7 }
1866 { D8 }
1867 { D9 }
1868 { DA }
1869 { DB }
1870 { DC }
1871 { DD }
1872 { DE }
1873 { FB }
1874 { FC }
1875 }

```



```

1876 </iso88598>
1877 <*iso88599>
1878 \_str_declare_eight_bit_encoding:nmn { iso88599 }
1879 {
1880     { D0 } { 011E }
1881     { DD } { 0130 }
1882     { DE } { 015E }
1883     { F0 } { 011F }
1884     { FD } { 0131 }
1885     { FE } { 015F }
1886 }
1887 {
1888 }
1889 </iso88599>
1890 <*iso885910>
1891 \_str_declare_eight_bit_encoding:nmn { iso885910 }
1892 {
1893     { A1 } { 0104 }
1894     { A2 } { 0112 }
1895     { A3 } { 0122 }
1896     { A4 } { 012A }
1897     { A5 } { 0128 }
1898     { A6 } { 0136 }
1899     { A8 } { 013B }
1900     { A9 } { 0110 }
1901     { AA } { 0160 }
1902     { AB } { 0166 }
1903     { AC } { 017D }
1904     { AE } { 016A }
1905     { AF } { 014A }
1906     { B1 } { 0105 }
1907     { B2 } { 0113 }
1908     { B3 } { 0123 }
1909     { B4 } { 012B }
1910     { B5 } { 0129 }
1911     { B6 } { 0137 }
1912     { B8 } { 013C }
1913     { B9 } { 0111 }
1914     { BA } { 0161 }
1915     { BB } { 0167 }
1916     { BC } { 017E }
1917     { BD } { 2015 }
1918     { BE } { 016B }
1919     { BF } { 014B }
1920     { C0 } { 0100 }
1921     { C7 } { 012E }
1922     { C8 } { 010C }
1923     { CA } { 0118 }
1924     { CC } { 0116 }

```

```

1925     { D1 } { 0145 }
1926     { D2 } { 014C }
1927     { D7 } { 0168 }
1928     { D9 } { 0172 }
1929     { E0 } { 0101 }
1930     { E7 } { 012F }
1931     { E8 } { 010D }
1932     { EA } { 0119 }
1933     { EC } { 0117 }
1934     { F1 } { 0146 }
1935     { F2 } { 014D }
1936     { F7 } { 0169 }
1937     { F9 } { 0173 }
1938     { FF } { 0138 }
1939 }
1940 {
1941 }
1942 </iso885910>
1943 <*iso885911>
1944 \_str_declare_eight_bit_encoding:nmm { iso885911 }
1945 {
1946     { A1 } { 0E01 }
1947     { A2 } { 0E02 }
1948     { A3 } { 0E03 }
1949     { A4 } { 0E04 }
1950     { A5 } { 0E05 }
1951     { A6 } { 0E06 }
1952     { A7 } { 0E07 }
1953     { A8 } { 0E08 }
1954     { A9 } { 0E09 }
1955     { AA } { 0E0A }
1956     { AB } { 0E0B }
1957     { AC } { 0E0C }
1958     { AD } { 0E0D }
1959     { AE } { 0E0E }
1960     { AF } { 0E0F }
1961     { B0 } { 0E10 }
1962     { B1 } { 0E11 }
1963     { B2 } { 0E12 }
1964     { B3 } { 0E13 }
1965     { B4 } { 0E14 }
1966     { B5 } { 0E15 }
1967     { B6 } { 0E16 }
1968     { B7 } { 0E17 }
1969     { B8 } { 0E18 }
1970     { B9 } { 0E19 }
1971     { BA } { 0E1A }
1972     { BB } { 0E1B }
1973     { BC } { 0E1C }

```

1974	{ BD }	{ 0E1D }
1975	{ BE }	{ 0E1E }
1976	{ BF }	{ 0E1F }
1977	{ C0 }	{ 0E20 }
1978	{ C1 }	{ 0E21 }
1979	{ C2 }	{ 0E22 }
1980	{ C3 }	{ 0E23 }
1981	{ C4 }	{ 0E24 }
1982	{ C5 }	{ 0E25 }
1983	{ C6 }	{ 0E26 }
1984	{ C7 }	{ 0E27 }
1985	{ C8 }	{ 0E28 }
1986	{ C9 }	{ 0E29 }
1987	{ CA }	{ 0E2A }
1988	{ CB }	{ 0E2B }
1989	{ CC }	{ 0E2C }
1990	{ CD }	{ 0E2D }
1991	{ CE }	{ 0E2E }
1992	{ CF }	{ 0E2F }
1993	{ D0 }	{ 0E30 }
1994	{ D1 }	{ 0E31 }
1995	{ D2 }	{ 0E32 }
1996	{ D3 }	{ 0E33 }
1997	{ D4 }	{ 0E34 }
1998	{ D5 }	{ 0E35 }
1999	{ D6 }	{ 0E36 }
2000	{ D7 }	{ 0E37 }
2001	{ D8 }	{ 0E38 }
2002	{ D9 }	{ 0E39 }
2003	{ DA }	{ 0E3A }
2004	{ DF }	{ 0E3F }
2005	{ E0 }	{ 0E40 }
2006	{ E1 }	{ 0E41 }
2007	{ E2 }	{ 0E42 }
2008	{ E3 }	{ 0E43 }
2009	{ E4 }	{ 0E44 }
2010	{ E5 }	{ 0E45 }
2011	{ E6 }	{ 0E46 }
2012	{ E7 }	{ 0E47 }
2013	{ E8 }	{ 0E48 }
2014	{ E9 }	{ 0E49 }
2015	{ EA }	{ 0E4A }
2016	{ EB }	{ 0E4B }
2017	{ EC }	{ 0E4C }
2018	{ ED }	{ 0E4D }
2019	{ EE }	{ 0E4E }
2020	{ EF }	{ 0E4F }
2021	{ F0 }	{ 0E50 }
2022	{ F1 }	{ 0E51 }
2023	{ F2 }	{ 0E52 }

```

2024     { F3 } { 0E53 }
2025     { F4 } { 0E54 }
2026     { F5 } { 0E55 }
2027     { F6 } { 0E56 }
2028     { F7 } { 0E57 }
2029     { F8 } { 0E58 }
2030     { F9 } { 0E59 }
2031     { FA } { 0E5A }
2032     { FB } { 0E5B }
2033 }
2034 {
2035     { DB }
2036     { DC }
2037     { DD }
2038     { DE }
2039 }
2040 </iso885911>
2041 <*iso885913>
2042 \__str_declare_eight_bit_encoding:nnn { iso885913 }
2043 {
2044     { A1 } { 201D }
2045     { A5 } { 201E }
2046     { A8 } { 00D8 }
2047     { AA } { 0156 }
2048     { AF } { 00C6 }
2049     { B4 } { 201C }
2050     { B8 } { 00F8 }
2051     { BA } { 0157 }
2052     { BF } { 00E6 }
2053     { C0 } { 0104 }
2054     { C1 } { 012E }
2055     { C2 } { 0100 }
2056     { C3 } { 0106 }
2057     { C6 } { 0118 }
2058     { C7 } { 0112 }
2059     { C8 } { 010C }
2060     { CA } { 0179 }
2061     { CB } { 0116 }
2062     { CC } { 0122 }
2063     { CD } { 0136 }
2064     { CE } { 012A }
2065     { CF } { 013B }
2066     { D0 } { 0160 }
2067     { D1 } { 0143 }
2068     { D2 } { 0145 }
2069     { D4 } { 014C }
2070     { D8 } { 0172 }
2071     { D9 } { 0141 }
2072     { DA } { 015A }

```

```

2073     { DB } { 016A }
2074     { DD } { 017B }
2075     { DE } { 017D }
2076     { E0 } { 0105 }
2077     { E1 } { 012F }
2078     { E2 } { 0101 }
2079     { E3 } { 0107 }
2080     { E6 } { 0119 }
2081     { E7 } { 0113 }
2082     { E8 } { 010D }
2083     { EA } { 017A }
2084     { EB } { 0117 }
2085     { EC } { 0123 }
2086     { ED } { 0137 }
2087     { EE } { 012B }
2088     { EF } { 013C }
2089     { F0 } { 0161 }
2090     { F1 } { 0144 }
2091     { F2 } { 0146 }
2092     { F4 } { 014D }
2093     { F8 } { 0173 }
2094     { F9 } { 0142 }
2095     { FA } { 015B }
2096     { FB } { 016B }
2097     { FD } { 017C }
2098     { FE } { 017E }
2099     { FF } { 2019 }
2100 }
2101 {
2102 }
2103 </iso885913>
2104 <*iso885914>
2105 \__str_declare_eight_bit_encoding:nnn { iso885914 }
2106 {
2107     { A1 } { 1E02 }
2108     { A2 } { 1E03 }
2109     { A4 } { 010A }
2110     { A5 } { 010B }
2111     { A6 } { 1E0A }
2112     { A8 } { 1E80 }
2113     { AA } { 1E82 }
2114     { AB } { 1E0B }
2115     { AC } { 1EF2 }
2116     { AF } { 0178 }
2117     { B0 } { 1E1E }
2118     { B1 } { 1E1F }
2119     { B2 } { 0120 }
2120     { B3 } { 0121 }
2121     { B4 } { 1E40 }

```

```

2122     { B5 } { 1E41 }
2123     { B7 } { 1E56 }
2124     { B8 } { 1E81 }
2125     { B9 } { 1E57 }
2126     { BA } { 1E83 }
2127     { BB } { 1E60 }
2128     { BC } { 1EF3 }
2129     { BD } { 1E84 }
2130     { BE } { 1E85 }
2131     { BF } { 1E61 }
2132     { D0 } { 0174 }
2133     { D7 } { 1E6A }
2134     { DE } { 0176 }
2135     { F0 } { 0175 }
2136     { F7 } { 1E6B }
2137     { FE } { 0177 }
2138 }
2139 {
2140 }
2141 </iso885914>
2142 <*iso885915>
2143 \__str_declare_eight_bit_encoding:nnn { iso885915 }
2144 {
2145     { A4 } { 20AC }
2146     { A6 } { 0160 }
2147     { A8 } { 0161 }
2148     { B4 } { 017D }
2149     { B8 } { 017E }
2150     { BC } { 0152 }
2151     { BD } { 0153 }
2152     { BE } { 0178 }
2153 }
2154 {
2155 }
2156 </iso885915>
2157 <*iso885916>
2158 \__str_declare_eight_bit_encoding:nnn { iso885916 }
2159 {
2160     { A1 } { 0104 }
2161     { A2 } { 0105 }
2162     { A3 } { 0141 }
2163     { A4 } { 20AC }
2164     { A5 } { 201E }
2165     { A6 } { 0160 }
2166     { A8 } { 0161 }
2167     { AA } { 0218 }
2168     { AC } { 0179 }
2169     { AE } { 017A }
2170     { AF } { 017B }

```

```

2171 { B2 } { 010C }
2172 { B3 } { 0142 }
2173 { B4 } { 017D }
2174 { B5 } { 201D }
2175 { B8 } { 017E }
2176 { B9 } { 010D }
2177 { BA } { 0219 }
2178 { BC } { 0152 }
2179 { BD } { 0153 }
2180 { BE } { 0178 }
2181 { BF } { 017C }
2182 { C3 } { 0102 }
2183 { C5 } { 0106 }
2184 { D0 } { 0110 }
2185 { D1 } { 0143 }
2186 { D5 } { 0150 }
2187 { D7 } { 015A }
2188 { D8 } { 0170 }
2189 { DD } { 0118 }
2190 { DE } { 021A }
2191 { E3 } { 0103 }
2192 { E5 } { 0107 }
2193 { F0 } { 0111 }
2194 { F1 } { 0144 }
2195 { F5 } { 0151 }
2196 { F7 } { 015B }
2197 { F8 } { 0171 }
2198 { FD } { 0119 }
2199 { FE } { 021B }
2200 }
2201 {
2202 }
2203 </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		941, 948, 1106, 1109, 1110, 1111, 1113, 1119, 1124, 1129, 1278, 1285
\#	343	\{ 339, 582
\\$	341	\} 340, 582
\%	351	\^ 344, 707, 708, 1050, 1051, 1230, 1231, 1232
\	342	_int_eval:w 102,
*	42, 706	144, 152, 433, 981, 996, 1026, 1174
\\	338, 577, 578, 581, 706, 919, 922, 923, 924, 925, 930, 936,	_int_eval_end:

..	104, 155, 433, 981, 998, 1026, 1174	_str_convert_gmap:N
_int_value:w	... 391, 488, 510, 900, 973		172, 172, 389, 479, 795, 803, 828, 860
_msg_kernel_error:nxx 207, 266	_str_convert_gmap_internal:N
_msg_kernel_error:nxxx 298	 188, 188, 399,
_msg_kernel_new:nnn	... 557, 559, 568		452, 521, 883, 1065, 1235, 1239, 1241
_msg_kernel_new:nnnn	_str_convert_gmap_internal_loop:Nw
 412, 561, 572, 586, 592,		188
	641, 688, 783, 908, 1089, 1096, 1266	_str_convert_gmap_internal_loop:Nww
_prg_break: 178, 194,		192, 198, 202
	312, 360, 426, 472, 476, 518, 611,	_str_convert_gmap_loop:NN
	660, 720, 726, 962, 1045, 1215, 1355		172, 176, 182, 186
_prg_break:n 70, 76, 88	_str_convert_lowercase_alphanum:n
_prg_break_point: 71, 77,		277, 309, 309
	179, 195, 313, 361, 427, 473, 477,	_str_convert_lowercase_alphanum_loop:N
	519, 612, 661, 721, 727, 963, 1164, 1319		309, 311, 315, 334
_str_convert:NNnNN 254, 259, 262	_str_convert_unescape:
_str_convert:nNNnnn		374, 375, 377, 385
 214, 215, 217, 222, 231, 236	_str_convert_unescape_bytes:	374, 385
_str_convert:nnn	... 257, 258, 272, 272	_str_convert_unescape_hex:	.. 600, 601
_str_convert:nnnn 272, 276, 281	_str_convert_unescape_name:	... 648
_str_convert:wwnn	.. 241, 246, 254, 254	_str_convert_unescape_string:	704, 711
_str_convert_decode:	.. 245, 388, 388	_str_convert_unescape_url: 648
_str_convert_decode_clist:	.. 439, 439	_str_declare_eight_bit_encoding:nnn
_str_convert_decode_eight_bit:n	..		456, 456, 1360, 1367, 1431,
 460, 466, 466		1473, 1530, 1631, 1718, 1804, 1878,
_str_convert_decode_utf16:	... 1134		1891, 1944, 2042, 2105, 2143, 2158
_str_convert_decode_utf16be:	.. 1134	_str_decode_clist_char:n	439, 445, 448
_str_convert_decode_utf16le:	.. 1134	_str_decode_eight_bit_char:N
_str_convert_decode_utf32:	... 1290		466, 479, 502
_str_convert_decode_utf32be:	.. 1290	_str_decode_eight_bit_load:nn
_str_convert_decode_utf32le:	.. 1290		466, 470, 483, 490
_str_convert_decode_utf8: 952	_str_decode_eight_bit_load_missing:n
_str_convert_encode:	250, 392, 396, 421		466, 474, 492, 500
_str_convert_encode_clist:	.. 450, 450	_str_decode_native_char:N	388, 389, 390
_str_convert_encode_eight_bit:n	..	_str_decode_utf_viii_aux:wNnnwN	..
 462, 512, 512		952, 995, 1009
_str_convert_encode_utf16:	... 1052	_str_decode_utf_viii_continuation:wwN
_str_convert_encode_utf16be:	.. 1052		952, 980, 987, 1025
_str_convert_encode_utf16le:	.. 1052	_str_decode_utf_viii_end:
_str_convert_encode_utf32:	... 1233		952, 962, 1039
_str_convert_encode_utf32be:	.. 1233	_str_decode_utf_viii_overflow:w	..
_str_convert_encode_utf32le:	.. 1233		952, 1023, 1032
_str_convert_encode_utf8: 882	_str_decode_utf_viii_start:N	952,
_str_convert_escape:	.. 386, 386, 387		961, 967, 985, 988, 1007, 1010, 1030
_str_convert_escape_bytes:	.. 386, 387	_str_decode_utf_xvi:Nw	... 1134,
_str_convert_escape_hex:	... 793, 794		1135, 1137, 1146, 1149, 1150, 1153
_str_convert_escape_name:	... 799, 802	_str_decode_utf_xvi_bom:NN
_str_convert_escape_string:	.. 824, 827		1134, 1140, 1143
_str_convert_escape_url:	... 858, 859	_str_decode_utf_xvi_error:nNN
			.. 1168, 1186, 1204, 1213, 1218, 1219

__str_decode_utf_xvi_extra:NNw	__str_filter_bytes_aux:N
1168, 1176, 1217	354, 359, 363, 371
__str_decode_utf_xvi_pair:NN	__str_gset_other:Nn 4, 41, 47, 240
. . 1162, 1168, 1168, 1180, 1183, 1206	__str_gset_other_end:w 41, 59, 64
__str_decode_utf_xvi_pair_end:Nw . .	__str_gset_other_loop:w . . 41, 51, 55, 62
1168, 1171, 1187, 1208	__str_hexadecimal_use:N 97
__str_decode_utf_xvi_quad:NNwNN . . .	__str_hexadecimal_use:NNTF
1168, 1175, 1182	5, 97, 621, 631, 672, 674
__str_decode_utf_xxxii:Nw . . . 1290,	__str_if_contains_char:NN 67
1291, 1293, 1302, 1305, 1306, 1309	__str_if_contains_char:NNT 67, 833
__str_decode_utf_xxxii_bom:NNNN . . .	__str_if_contains_char:NNTF 67, 812, 818
1290, 1296, 1299	__str_if_contains_char:nN 74
__str_decode_utf_xxxii_end:w	__str_if_contains_char:nNTF 67, 869, 875
1290, 1326, 1346	__str_if_contains_char_aux:NN
__str_decode_utf_xxxii_loop:NNNN . .	67, 69, 76, 80, 85
1290, 1317, 1323, 1344	__str_if_contains_char_true: 67, 83, 87
__str_encode_clist_char:n 450, 452, 455	__str_if_escape_name:N 809
__str_encode_eight_bit_char:n	__str_if_escape_name:NNTF 799, 806
512, 521, 535	__str_if_escape_string:N 845
__str_encode_eight_bit_char_aux:n .	__str_if_escape_string:NNTF . . . 824, 831
512, 546, 549	__str_if_escape_url:N 866
__str_encode_eight_bit_load:nn	__str_if_escape_url:NNTF 858, 863
512, 516, 525, 533	__str_if_flag_error:nnx 204, 204, 223,
__str_encode_native_char:n 395, 399, 403	232, 382, 400, 480, 522, 615, 663,
__str_encode_native_filter:N 420	664, 729, 730, 965, 1066, 1166, 1321
__str_encode_native_flush: 420	__str_if_flag_no_error:nnx
__str_encode_native_loop:w	204, 210, 223, 232
420, 425, 430, 435	__str_if_flag_times:nT
__str_encode_utf_viii_char:n	212, 212, 911, 912,
882, 883, 884	913, 914, 1099, 1100, 1101, 1269, 1270
__str_encode_utf_viii_loop:wwnw . .	__str_load_catcodes: 292, 336, 336
882, 886, 893, 899	__str_octal_use:N 89
__str_encode_utf_xvi_aux:N	__str_octal_use:NNTF . . . 89, 739, 741, 743
1052, 1054, 1058, 1060, 1061	__str_output_byte:n
__str_encode_utf_xvi_char:n	5, 138, 138, 169, 170,
1052, 1065, 1068	324, 406, 531, 552, 896, 902, 1251, 1260
__str_encode_utf_xxxii_be:n	__str_output_byte:w
1233, 1235, 1239, 1242	138, 139, 140, 608, 634, 671, 738
__str_encode_utf_xxxii_be_aux:nn . .	__str_output_byte_pair:nnN
1233, 1244, 1247	156, 158, 163, 166
__str_encode_utf_xxxii_le:n	__str_output_byte_pair_be:n
1233, 1241, 1253	156, 156, 1054, 1058, 1250
__str_encode_utf_xxxii_le_aux:nn . .	__str_output_byte_pair_le:n
1233, 1255, 1258	156, 161, 1060, 1261
__str_escape_hex_char:N . . 793, 795, 796	__str_output_end: 138,
__str_escape_name_char:N . 799, 803, 804	139, 147, 154, 613, 633, 685, 772, 776
__str_escape_string_char:N 824, 828, 829	__str_output_hexadecimal:n
__str_escape_url_char:N . . 858, 860, 861	138, 146, 797, 807, 864
__str_filter_bytes:n	__str_output_hexadecimal:w 138, 147, 148
354, 355, 357, 381, 669, 736	

_str_tmp:w	10, 10, 649, 697, 701, 1064, 1071, 1076, 1078, 1081, 1082, 1159, 1174, 1179, 1190, 1193, 1198, 1199, 1314, 1329, 1334, 1340	\c_max_char_int	4, 8, 8
_str_unescape_hex_auxi:N	600, 609, 618, 625, 634	\c_max_register_int	537
_str_unescape_hex_auxii:N	600, 622, 628, 638	\c_minus_one	886
_str_unescape_name_loop:wNN	648, 698	\c_ninety_one	15, 18, 318
_str_unescape_string_loop:wNNN	704, 725, 733, 773, 776	\c_ninety_seven	15, 19, 328
_str_unescape_string_newlines:wN	704, 719, 777, 781	\c_one	91, 320, 489, 499, 532, 611, 765, 769, 1191
_str_unescape_string_repeat:NNNNNN	704, 748, 750, 752, 775	\c_one_hundred_twenty_seven	15, 21
_str_unescape_url_loop:wNN	648, 702	\c_one_hundred_twenty_three	15, 20, 327
_tl_build_end:	428	\c_percent_str	701, 864
_tl_build_one:n	434	\c_sixteen	1334
_tl_gbuild_x:Nw	424	\c_sixty_five	15, 17, 319
\~	346	\c_thirty_two	324
		\c_three	745
		\c_two	99, 902
		\c_two_hundred_fifty_six	405, 551
		\c_zero	119, 122, 127, 678, 886, 887, 1191, 1329
_	42, 345, 582, 922, 923, 924	\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, 707, 708, 1050, 1051, 1230, 1231, 1232
		\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, 706
		\clist_map_function:nN	444
		\clist_set:No	441
		\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, 448, 455, 502, 535, 549, 618, 628, 667, 733, 775, 777, 796, 804, 829, 861, 884, 893, 967, 987, 1009, 1032, 1068, 1168, 1182, 1208, 1217, 1219, 1242, 1247, 1253, 1258, 1323, 1346
		\cs_new_eq:NN	355, 385, 387
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	799, 800, 812		
\c_str_escape_name_str	799, 801, 818		
\c_str_escape_string_str	824, 825, 834		
\c_str_replacement_char_int	14, 14, 498, 977, 1003, 1017, 1037, 1044, 1076, 1224, 1332, 1337, 1353		
\c_backslash_str	757, 786, 788, 826, 835, 839		
\c_eight	841, 842		
\c_fifty_eight	15, 16		
\c_forty_eight	15, 15		
\c_four	1174		
\c_hash_str	697, 807		

\if_meaning:w	58, 264, 1022, 1170, 1185, 1211, 1325	\ProvidesExplPackage	3
\int_const:Nn	8, 14, 15, 16, 17, 18, 19, 20, 21	Q	
\int_div_truncate:nn	159, 164, 840, 841, 900, 1081, 1245, 1256	\q_nil	1163, 1170, 1185, 1211
\int_eval:n	131, 449, 1179, 1196, 1339	\q_stop	52, 64, 194, 200, 242, 247, 255, 426, 432, 472, 476, 485, 494, 518, 527, 891, 893, 901, 984, 1022
\int_mod:nn	841, 842, 1082	R	
\int_new:N	12	\RequirePackage	5
\int_set:Nn	352, 605, 656, 716	S	
\int_use:N	144, 152, 498, 977, 981, 996, 1003, 1017, 1026, 1037, 1044, 1224, 1332, 1337, 1353	\s__stop	1135, 1137, 1141, 1153, 1291, 1293, 1297, 1309, 1318, 1325, 1346
\int_zero:N	423, 469, 515	\s__tl	194, 198, 391, 426, 430, 449, 504, 510, 971, 983, 988, 1000, 1005, 1010, 1013, 1028, 1041, 1044, 1178, 1179, 1195, 1201, 1217, 1223, 1224, 1328, 1343, 1352, 1353
\iow_char:N	581	\scan_stop:	486, 487, 495, 496, 528, 529
\iow_indent:n	579, 920, 1107	\str_byte	39
L		\str_case:x:nnn	755
\l__str_end_flag	1086	\str_const:Nn	800, 801
\l__str_extra_flag	904, 1086	\str_const:Nx	825
\l__str_internal_int	10, 12, 433, 469, 486, 487, 488, 489, 495, 496, 497, 499, 505, 515, 528, 529, 530, 532, 540	\str_end	1264
\l__str_internal_tl	10, 11, 120, 121, 123, 125, 285, 286, 287, 289, 293, 297, 304, 458	\str_error	39
\l__str_missing_flag	904, 1086	\str_gset_convert:Nnnn	4, 214, 216, 228
\l__str_overflow_flag	904	\str_gset_convert:NnnnTF	4, 214
\l__str_overlong_flag	904	\str_if_eq_x:nnTF	1145, 1148, 1301, 1304
\l__tl_build_offset_int	423	\str_overflow	1264
O		\str_set_convert:Nnnn	4, 214, 214, 219
\or:	106, 107, 108, 109, 110, 1175, 1176	\str_set_convert:NnnnTF	4, 214
P		T	
\pdfTeX_if_engine:F	570	\tex_advance:D	489, 499, 532
\pdfTeX_if_engine:TF	9, 354, 374, 394	\tex_dimen:D	486, 495, 505, 506, 507, 528, 540, 541, 542
\prg_do_nothing:	244, 1213	\tex_endlinechar:D	352
\prg_new_conditional:Npnn	67, 74, 89, 97, 809, 845, 866	\tex_escapechar:D	605, 656, 716
\prg_new_protected_conditional:Npnn	218, 227	\tex_lccode:D	433
\prg_return_false:	72, 78, 94, 112, 225, 234, 813, 816, 819, 848, 851, 870, 873, 876	\tex_skip:D	487, 496, 506, 529, 541
\prg_return_true:	88, 92, 100, 115, 225, 234, 813, 819, 853, 870, 876	\tex_the:D	507, 542
\prop_get:NnNF	285	\tex_toks:D	488, 497, 507, 530, 542
\prop_gput:Nnn	23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37	\tl_clear:N	297
\prop_new:N	22	\tl_const:cn	137, 463, 464
		\tl_const:cx	130
		\tl_gput_left:Nx	1055, 1236
		\tl_gset:Nx	49, 174, 190, 380, 442, 453, 606, 657, 717, 723, 959, 1160, 1315
		\tl_gset_eq:NN	217, 233

