The l3str package: manipulating strings of characters*

The LaTeX3 Project†

Released 2012/01/19

LaTeX3 provides a set of functions to manipulate token lists as strings of characters, ignoring the category codes of those characters.

String variables are simply specialised token lists, but by convention should be named with the suffix `...str`. Such variables should contain characters with category code 12 (other), except spaces, which have category code 10 (blank space). All the functions in this module first convert their argument to a string for internal processing, and will not treat a token list or the corresponding string representation differently.

Most functions in this module come in three flavours:

- `\str_...:N...`, which expect a token list or string variable as their argument;
- `\str_...:n...`, taking any token list (or string) as an argument;
- `\str_..._ignore_spaces:n...`, which ignores any space encountered during the operation: these functions are faster than those which take care of escaping spaces appropriately;

When performance is critical, the internal `\str_..._unsafe:n...` functions, which expect an "other string" in which spaces have category code 12 instead of 10, might be useful.

1 Building strings

`\c_backslash_str`
`\c_lbrace_str`
`\c_rbrace_str`
`\c_hash_str`
`\c_tilde_str`
`\c_percent_str`

Constant strings, containing a single character, with category code 12. Any character can be accessed as `\iow_char:N` \⟨*character*⟩.

*This file describes v3209, last revised 2012/01/19.

†E-mail: latex-team@latex-project.org

\tl_to_str:N ⋆
\tl_to_str:n ⋆

\tl_to_str:N ⟨*tl var*⟩
\tl_to_str:n {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, leaving the resulting tokens in the input stream.

TEXhackers note: The string representation of a token list may depend on the category codes in effect when it is measured, and the value of the `\escapechar`: for instance `\tl_to_str:n {\a}` may produce 1, 2 or 3 characters depending on the escape character, and the category code of `a`. This impacts almost all functions in the module, which use `\tl_to_str:n` internally.

\str_const:Nn
\str_const:(Nx|cn|cx)

\str_const:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Creates a new constant ⟨*str var*⟩ or raises an error if the name is already taken. The value of the ⟨*str var*⟩ will be set globally to the ⟨*token list*⟩, converted to a string.

\str_set:Nn
\str_set:(Nx|cn|cx)
\str_gset:Nn
\str_gset:(Nx|cn|cx)

\str_set:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and stores the result in ⟨*str var*⟩.

\str_put_left:Nn
\str_put_left:(Nx|cn|cx)
\str_gput_left:Nn
\str_gput_left:(Nx|cn|cx)

\str_put_left:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and prepends the result to ⟨*str var*⟩. The current contents of the ⟨*str var*⟩ are not automatically converted to a string.

\str_put_right:Nn
\str_put_right:(Nx|cn|cx)
\str_gput_right:Nn
\str_gput_right:(Nx|cn|cx)

\str_put_right:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and appends the result to ⟨*str var*⟩. The current contents of the ⟨*str var*⟩ are not automatically converted to a string.

\str_input:Nn
\str_ginput:Nn

\str_input:Nn ⟨*str var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ into a ⟨*string*⟩, and stores it in the ⟨*str var*⟩. Special characters can be input by escaping them with a backslash.

- Spaces are ignored unless escaped with a backslash.
- `\xhh` produces the character with code `hh` in hexadecimal: when `\x` is encountered, up to two hexadecimal digits (`0`–`9`, `a`–`f`, `A`–`F`) are read to give a number between 0 and 255.
- `\x{hh...}` produces the character with code `hh...` (an arbitrary number of hexadecimal digits are read): this is mostly useful for Unicode-aware engines.
- `\a`, `\e`, `\f`, `\n`, `\r`, `\t` stand for specific characters:

`\a`	`\^^G`	alarm	hex `07`
`\e`	`\^^[`	escape	hex `1B`
`\f`	`\^^L`	form feed	hex `0C`
`\n`	`\^^J`	new line	hex `0A`
`\r`	`\^^M`	carriage return	hex `0D`
`\t`	`\^^I`	horizontal tab	hex `09`

For instance,

```
\tl_new:N \l_my_str
\str_input:Nn \l_my_str {\x3C \\ \# abc\ def\^\n}
```

results in `\l_my_str` containing the characters `<\#abc def^`, followed by a newline character (hex `0A`) since `<` has ASCII code `3C` (in hexadecimal).

2 Characters given by their position

\str_length:N ⋆
\str_length:n ⋆
\str_length_ignore_spaces:n ⋆

\str_length:n {⟨*token list*⟩}

Leaves the length of the string representation of ⟨*token list*⟩ in the input stream as an integer denotation. The functions differ in their treatment of spaces. In the case of `\str_length:N` and `\str_length:n`, all characters including spaces are counted. The `\str_length_ignore_spaces:n` leaves the number of non-space characters in the input stream.

\str_head:N ⋆ \str_head:n {⟨*token list*⟩}
\str_head:n ⋆
\str_head_ignore_spaces:n ⋆

Converts the ⟨*token list*⟩ into a ⟨*string*⟩. The first character in the ⟨*string*⟩ is then left in the input stream, with category code "other". The functions differ in their treatment of spaces. In the case of `\str_head:N` and `\str_head:n`, a leading space is returned with category code 10 (blank space). The `\str_head_ignore_spaces:n` function leaves the first non-space character in the input stream. If the ⟨*token list*⟩ is empty (or blank in the case of the `_ignore_spaces` variant), then nothing is left on the input stream.

\str_tail:N ⋆ \str_tail:n {⟨*token list*⟩}
\str_tail:n ⋆
\str_tail_ignore_spaces:n ⋆

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, removes the first character, and leaves the remaining characters (if any) in the input stream, with category codes 12 and 10 (for spaces). The functions differ in the case where the first character is a space: `\str_tail:N` and `\str_tail:n` will trim only that space, while `\str_tail_ignore_spaces:n` removes the first non-space character and any space before it. If the ⟨*token list*⟩ is empty (or blank in the case of the `_ignore_spaces` variant), then nothing is left on the input stream.

\str_item:Nn ⋆ \str_item:nn {⟨*token list*⟩} {⟨*integer expression*⟩}
\str_item:nn ⋆
\str_item_ignore_spaces:nn ⋆

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and leaves in the input stream the character in position ⟨*integer expression*⟩ of the ⟨*string*⟩. In the case of `\str_item:Nn` and `\str_item:nn`, all characters including spaces are taken into account. The `\str_item_ignore_spaces:nn` function skips spaces in its argument. If the ⟨*integer expression*⟩ is negative, characters are counted from the end of the ⟨*string*⟩. Hence, −1 is the right-most character, *etc.*, while 0 is the first (left-most) character.

\str_substr:Nnn ⋆ \str_substr:nnn {⟨*token list*⟩} {⟨*start index*⟩} {⟨*end index*⟩}
\str_substr:nnn ⋆
\str_substr_ignore_spaces:nnn ⋆

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and leaves in the input stream the characters between ⟨*start index*⟩ (inclusive) and ⟨*end index*⟩ (exclusive). If either of ⟨*start index*⟩ or ⟨*end index*⟩ is negative, then it is incremented by the length of the list. If either of ⟨*start index*⟩ or ⟨*end index*⟩ is empty, it is replaced by the corresponding end-point of the string. Both ⟨*start index*⟩ and ⟨*end index*⟩ count from 0 for the first (left most) character. For instance,

```
\iow_term:x { \str_substr:nnn { abcdef } { 1 } { 4 } }
\iow_term:x { \str_substr:nnn { abcdef } { -4 } { } }
```

will print `bcd` and `cdef`.

3 String conditionals

`\str_if_eq_p:NN` ⋆	`\str_if_eq_p:nn` {⟨*tl*$_1$⟩} {⟨*tl*$_2$⟩}
`\str_if_eq_p:(nn|Vn|on|no|nV|VV|xx)` ⋆	`\str_if_eq:nnTF` {⟨*tl*$_1$⟩} {⟨*tl*$_2$⟩} {⟨*true code*⟩} {⟨*false code*⟩}
`\str_if_eq:NN`*TF* ⋆	
`\str_if_eq:(nn|Vn|on|no|nV|VV|xx)`*TF* ⋆	

Compares the two ⟨*token lists*⟩ on a character by character basis, and is `true` if the two lists contain the same characters in the same order. Thus for example

```
\str_if_eq_p:xx { abc } { \tl_to_str:n { abc } }
```

is logically `true`. All versions of these functions are fully expandable (including those involving an `x`-type expansion).

4 Encoding functions

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 (PDF string objects, URLs). Hence, storing strings of characters has two components.

- 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 encodings supported.[1]
- Bytes are translated to TEX 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]

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

Table 1: Supported encodings. Non-alphanumeric characters are ignored, and capital letters are lower-cased.

`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
`native`	Native Unicode string.
`internal`	For curious programmers.

Table 2: Supported escapings. Non-alphanumeric characters are ignored, and capital letters are lower-cased.

`bytes`, or empty	arbitrary bytes
`hex`, `hexadecimal`	byte = two hexadecimal digits
`name`	see `\pdfescapename`
`string`	see `\pdfescapestring`
`url`	encoding used in URLs

\str_set_convert:Nnnn

\str_set_convert:Nnnn ⟨*str var*⟩ {⟨*string*⟩}
{⟨*name* 1⟩} {⟨*name* 2⟩}

This function converts the ⟨*string*⟩ from the encoding given by ⟨*name 1*⟩ to the encoding given by ⟨*name 2*⟩, and stores the result in the ⟨*str var*⟩. Each ⟨*name*⟩ can have the form ⟨*encoding*⟩ or ⟨*encoding*⟩/⟨*escaping*⟩, where the possible values of ⟨*encoding*⟩ and ⟨*escaping*⟩ are given in Tables 1 and 2, respectively. The default escaping is to input and output bytes directly. The special encodings `native` and `internal` do not support a ⟨*escaping*⟩, since those formats do not correspond to strings of bytes.

For example,

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

results in the variable `\l_foo_str` holding the string `FEFF00480065006C006C006F0021`. This is obtained by converting each character in the 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`.

Parts of the ⟨*string*⟩ which cannot be converted to bytes using the ⟨*escaping 1*⟩ are silently ignored (including non-byte characters which could be present in the string). Sequences of bytes which are not valid in the ⟨*encoding 1*⟩ are converted to the replacement character "FFFD, and raise an error. Characters which cannot be encoded in the ⟨*encoding 2*⟩ are silently ignored.

5 Internal string functions

\str_escape_use:NNNn

\str_escape_use:NNNn ⟨*fn1*⟩ ⟨*fn2*⟩ ⟨*fn3*⟩ {⟨*token list*⟩}

The ⟨*token list*⟩ is converted to a string, then read from left to right, interpreting backslashes as escaping the next character. Unescaped characters are fed to the function ⟨*fn1*⟩, and escaped characters are fed to the function ⟨*fn2*⟩ within an `x`-expansion context (typically those functions perform some tests on their argument to decide how to output them). The escape sequences `\a`, `\e`, `\f`, `\n`, `\r`, `\t` and `\x` are recognized as described for `\str_input:Nn`, and those are replaced by the corresponding character, then fed to ⟨*fn3*⟩. The result is then left in the input stream.

\str_aux_toks_range:nn ☆

\str_aux_toks_range:nn ⟨*start*⟩ ⟨*end*⟩

Expands to the contents of the `\toks` registers numbered from ⟨*start*⟩ (inclusive) to ⟨*end*⟩ (exclusive). This function is used in l3regex.

\str_aux_hexadecimal_use:NTF \str_aux_hexadecimal_use:NTF ⟨*token*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

If the ⟨*token*⟩ is a hexdecimal digit (upper case or lower case), its upper-case version is left in the input stream, *followed* by the ⟨*true code*⟩. Otherwise, the ⟨*false code*⟩ is left in the input stream.

TEXhackers note: This function will fail if the escape character is a hexadecimal digit, or has a character code in the range $[0, 5]$. We are thus careful to set the escape character to a known value before using it.

\tl_to_other_str:n ⋆ \tl_to_other_str:n {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to a ⟨*other string*⟩, where spaces have category code "other".

TEXhackers note: These functions can be `f`-expanded without fear of losing a leading space, since spaces do not have category code 10 in their result.

\str_gset_other:Nn \str_gset_other:Nn ⟨*tl var*⟩ {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to an ⟨*other string*⟩, where spaces have category code "other", and assigns the result to the ⟨*tl var*⟩, globally.

\str_if_contains_char:NNT ⋆
\str_if_contains_char:(NNTF|nNTF) ⋆ \str_if_contains_char:nNTF {⟨*token list*⟩} ⟨*char*⟩

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ and tests whether the ⟨*char*⟩ is present in the ⟨*string*⟩. Spaces are ignored.

6 Possibilities, and things to do

- Use flags rather than expandable errors.
- Add documentation about each encoding and escaping method, and add examples.
- Perhaps use the pdfTEX primitives for escapings when available. But this changes error detection between engines.
- More encodings (see Heiko's stringenc). CESU.
- More escapings: shell escapes, lua escapes, etc?
- `\str_if_head_eq:nN`
- `\str_if_numeric/decimal/integer:n`, perhaps in l3fp?
- Should `\str_item:Nn` be `\str_char:Nn`?

7 l3str implementation

```
1 ⟨*initex | package⟩
2 \ProvidesExplPackage
3   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
4 \RequirePackage{l3tl-analysis}
```

Those string-related functions are defined in l3kernel.

- `\str_if_eq:nn`[pTF] and variants,
- `\str_if_eq_return:on`,
- `\tl_to_str:n`, `\tl_to_str:N`, `\tl_to_str:c`,
- `\token_to_str:N`, `\cs_to_str:N`
- `\str_head:n`, `\str_head_aux:w`, (copied here)
- `\str_tail:n`, `\str_tail_aux:w`, (copied here)
- `\str_length_skip_spaces` (deprecated)
- `\str_length_loop:NNNNNNNNN` (unchanged)

7.1 Helpers

7.1.1 Assigning strings

```
\str_set:Nn
\str_set:Nx
\str_set:cn
\str_set:cx
\str_gset:Nn
\str_gset:Nx
\str_gset:cn
\str_gset:cx
\str_const:Nn
\str_const:Nx
\str_const:cn
\str_const:cx
\str_put_left:Nn
\str_put_left:Nx
\str_put_left:cn
\str_put_left:cx
\str_gput_left:Nn
\str_gput_left:Nx
\str_gput_left:cn
\str_gput_left:cx
\str_put_right:Nn
\str_put_right:Nx
\str_put_right:cn
\str_put_right:cx
\str_gput_right:Nn
\str_gput_right:Nx
\str_gput_right:cn
\str_gput_right:cx
```

Simply convert the token list inputs to ⟨*strings*⟩.

```
5  \group_begin:
6    \cs_set_protected_nopar:Npn \str_tmp:w #1
7      {
8        \cs_new_protected:cpx { str_ #1 :Nn } ##1##2
9          { \exp_not:c { tl_ #1 :Nx } ##1 { \exp_not:N \tl_to_str:n {##2} } }
10       \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :Nn } { Nx , cn , cx }
11     }
12   \str_tmp:w { set }
13   \str_tmp:w { gset }
14   \str_tmp:w { const }
15   \str_tmp:w { put_left }
16   \str_tmp:w { gput_left }
17   \str_tmp:w { put_right }
18   \str_tmp:w { gput_right }
19 \group_end:
```

(*End definition for* `\str_set:Nn` *and others. These functions are documented on page* **??**.)

7.1.2 Variables and constants

\str_tmp:w
\l_str_tmpa_tl

Internal scratch space for some functions.

```
\cs_new_protected_nopar:Npn \str_tmp:w { }
\tl_new:N \l_str_tmpa_tl
\int_new:N \l_str_tmpa_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 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.

```
\tl_new:N \g_str_result_tl
```

(*End definition for* `\g_str_result_tl`. *This function is documented on page* **??**.)

\l_str_char_int

When converting from various forms to characters, `\l_str_char_int` is the character code of the character which should be created.

```
\int_new:N \l_str_char_int
```

(*End definition for* `\l_str_char_int`. *This function is documented on page* **??**.)

\c_forty_eight
\c_fifty_eight
\c_sixty_five
\c_ninety_one
\c_ninety_seven
\c_one_hundred_twenty_three
\c_one_hundred_twenty_seven

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

```
\int_const:Nn \c_forty_eight  { 48 }
\int_const:Nn \c_fifty_eight  { 58 }
\int_const:Nn \c_sixty_five   { 65 }
\int_const:Nn \c_ninety_one   { 91 }
\int_const:Nn \c_ninety_seven { 97 }
\int_const:Nn \c_one_hundred_twenty_three { 123 }
\int_const:Nn \c_one_hundred_twenty_seven { 127 }
```

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

\c_max_char_int

The maximum character code depends on the engine.

```
\int_const:Nn \c_max_char_int
  { \pdftex_if_engine:TF { 255 } { 1114111 } }
```

(*End definition for* `\c_max_char_int`. *This function is documented on page* **??**.)

\c_str_replacement_char_int

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

```
\int_const:Nn \c_str_replacement_char_int { "FFFD }
```

(*End definition for* `\c_str_replacement_char_int`. *This function is documented on page* **??**.)

\c_backslash_str
\c_lbrace_str
\c_rbrace_str
\c_hash_str
\c_tilde_str
\c_percent_str

For all of those strings, `\cs_to_str:N` produce characters with the correct category code.

```
\tl_const:Nx \c_backslash_str { \cs_to_str:N \\ }
\tl_const:Nx \c_lbrace_str    { \cs_to_str:N \{ }
\tl_const:Nx \c_rbrace_str    { \cs_to_str:N \} }
\tl_const:Nx \c_hash_str      { \cs_to_str:N \# }
\tl_const:Nx \c_tilde_str     { \cs_to_str:N \~ }
\tl_const:Nx \c_percent_str   { \cs_to_str:N \% }
```

(*End definition for* `\c_backslash_str` *and others. These functions are documented on page* 1.)

\g_str_aliases_prop

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

```
\prop_new:N \g_str_aliases_prop
\prop_gput:Nnn \g_str_aliases_prop { latin1 } { iso88591 }
\prop_gput:Nnn \g_str_aliases_prop { latin2 } { iso88592 }
\prop_gput:Nnn \g_str_aliases_prop { latin3 } { iso88593 }
\prop_gput:Nnn \g_str_aliases_prop { latin4 } { iso88594 }
\prop_gput:Nnn \g_str_aliases_prop { latin5 } { iso88599 }
\prop_gput:Nnn \g_str_aliases_prop { latin6 } { iso885910 }
\prop_gput:Nnn \g_str_aliases_prop { latin7 } { iso885913 }
\prop_gput:Nnn \g_str_aliases_prop { latin8 } { iso885914 }
\prop_gput:Nnn \g_str_aliases_prop { latin9 } { iso885915 }
\prop_gput:Nnn \g_str_aliases_prop { latin10 } { iso885916 }
\prop_gput:Nnn \g_str_aliases_prop { hexadecimal } { hex }
```

(*End definition for* `\g_str_aliases_prop`. *This function is documented on page* ??.)

7.1.3 Escaping spaces

\tl_to_other_str:n
\tl_to_other_str_loop:w
\tl_to_other_str_end:w

Replaces all spaces by "other" spaces, after converting the token list to a string via `\tl_to_str:n`. This function is `f`-expandable, storing the part of the string with escaped spaces between the `\q_mark` and `\q_stop` markers.

```
\group_begin:
\char_set_lccode:nn { `\* } { `\  }
\char_set_lccode:nn { `\A } { `\A }
\tl_to_lowercase:n
  {
    \group_end:
    \cs_new:Npn \tl_to_other_str:n #1
      {
        \exp_after:wN \tl_to_other_str_loop:w \tl_to_str:n {#1} ~ %
        A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \q_mark \q_stop
      }
    \cs_new_nopar:Npn \tl_to_other_str_loop:w
      #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 \q_stop
      {
        \if_meaning:w A #8
          \tl_to_other_str_end:w
        \fi:
        \tl_to_other_str_loop:w
        #9 #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * \q_stop
      }
    \cs_new_nopar:Npn \tl_to_other_str_end:w \fi: #1 \q_mark #2 * A #3 \q_stop
      { \fi: #2 }
  }
```

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

\str_gset_other:Nn
\str_gset_other_loop:w
\str_gset_other_end:w

This function could be done by using `\tl_to_other_str: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.

```
\group_begin:
\char_set_lccode:nn { `\* } { `\  }
\char_set_lccode:nn { `\A } { `\A }
\tl_to_lowercase:n
  {
    \group_end:
    \cs_new_protected:Npn \str_gset_other:Nn #1#2
      {
        \tl_gset:Nx #1
          {
            \exp_after:wN \str_gset_other_loop:w \tl_to_str:n {#2} ~ %
            A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \q_mark \q_stop
          }
      }
    \cs_new_nopar:Npn \str_gset_other_loop:w
      #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 ~
      {
        \if_meaning:w A #9
          \str_gset_other_end:w
        \fi:
        #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * #9
        \str_gset_other_loop:w *
      }
    \cs_new_nopar:Npn \str_gset_other_end:w \fi: #1 * A #2 \q_stop
      { \fi: #1 }
  }
```

(*End definition for* `\str_gset_other:Nn`*. This function is documented on page* ??.)

7.1.4 Functions unrelated to strings

`\use_i:nnnnnnnn` A function which may already be defined elsewhere.

```
\cs_if_exist:NF \use_i:nnnnnnnn
  { \cs_new:Npn \use_i:nnnnnnnn #1#2#3#4#5#6#7#8 {#1} }
```

(*End definition for* `\use_i:nnnnnnnn`*. This function is documented on page* ??.)

`\str_aux_toks_range:nn` `\str_aux_toks_range:ww` Some non-expandable functions (in this module, `\str_aux_escape:NNNn`, which is used by `\str_(g)input:Nn`) store pieces of their result in `\toks` registers. Those pieces are concatenated using `\str_aux_toks_range:nn`, which expects two integer registers, ⟨*start*⟩ and ⟨*end*⟩, and expands to the contents of the `\toks` registers from ⟨*start*⟩ (inclusive) to ⟨*end*⟩ (exclusive). This function is also used in the l3regex module.

```
\cs_new:Npn \str_aux_toks_range:nn #1#2
  {
    \exp_after:wN \str_aux_toks_range:ww
    \int_use:N \int_eval:w #1 \exp_after:wN ;
    \int_use:N \int_eval:w #2 ;
    \prg_break_point:n { }
```

```
  }
\cs_new:Npn \str_aux_toks_range:ww #1 ; #2 ;
  {
    \if_num:w #1 = #2 \exp_stop_f:
      \exp_after:wN \prg_map_break:
    \fi:
    \tex_the:D \tex_toks:D #1 \exp_stop_f:
    \exp_after:wN \str_aux_toks_range:ww
      \int_use:N \int_eval:w #1 + \c_one ; #2 ;
  }
```

(*End definition for* `\str_aux_toks_range:nn`*. This function is documented on page* ??.)

7.2 Characters given by their position

`\str_length:N`
`\str_length:n`
`\str_length_ignore_spaces:n`
`\str_length_unsafe:n`
`\str_length_aux:n`
`\str_length_loop:NNNNNNNNN`

The length of a string is found by first changing all spaces to other spaces using `\tl_to_other_str:n`, then counting characters 9 at a time. When the end is reached, `#9` has the form `X`⟨*digit*⟩, the catcode test is true, the digit gets added to the sum, and the loop is terminated by `\use_none_delimit_by_q_stop:w`.

```
\cs_new_nopar:Npn \str_length:N { \exp_args:No \str_length:n }
\cs_new:Npn \str_length:n #1
  { \exp_args:Nf \str_length_unsafe:n { \tl_to_other_str:n {#1} } }
\cs_new_nopar:Npn \str_length_unsafe:n #1
  { \str_length_aux:n { \str_length_loop:NNNNNNNNN #1 } }
\cs_new:Npn \str_length_ignore_spaces:n #1
  {
    \str_length_aux:n
      { \exp_after:wN \str_length_loop:NNNNNNNNN \tl_to_str:n {#1} }
  }
\cs_new:Npn \str_length_aux:n #1
  {
    \int_eval:n
      {
        #1
        { X \c_eight } { X \c_seven } { X \c_six   }
        { X \c_five  } { X \c_four  } { X \c_three }
        { X \c_two   } { X \c_one   } { X \c_zero  }
        \q_stop
      }
  }
\cs_set:Npn \str_length_loop:NNNNNNNNN #1#2#3#4#5#6#7#8#9
  {
    \if_catcode:w X #9
      \exp_after:wN \use_none_delimit_by_q_stop:w
    \fi:
    \c_nine + \str_length_loop:NNNNNNNNN
  }
```

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

\str_head:N
\str_head:n
\str_head_ignore_spaces:n
\str_head_unsafe:n
\str_head_aux:w

The cases of `\str_head_ignore_spaces:n` and `\str_head_unsafe:n` are mostly identical to `\tl_head:n`. As usual, `\str_head:N` expands its argument and hands it to `\str_head:n`. To circumvent the fact that TEX skips spaces when grabbing undelimited macro parameters, `\str_head_aux:w` takes an argument delimited by a space. If `#1` starts with a non-space character, `\use_i_delimit_by_q_stop:nw` leaves that in the input stream. On the other hand, if `#1` starts with a space, the `\str_head_aux:w` takes an empty argument, and the single (braced) space in the definition of `\str_head_aux:w` makes its way to the output. Finally, for an empty argument, the (braced) empty brace group in the definition of `\str_head:n` gives an empty result after passing through `\use_i_delimit_by_q_stop:nw`.

```
\cs_new_nopar:Npn \str_head:N { \exp_args:No \str_head:n }
\cs_set:Npn \str_head:n #1
  {
    \exp_after:wN \str_head_aux:w
    \tl_to_str:n {#1}
    { { } } ~ \q_stop
  }
\cs_set_nopar:Npn \str_head_aux:w #1 ~ %
  { \use_i_delimit_by_q_stop:nw #1 { ~ } }
\cs_new:Npn \str_head_ignore_spaces:n #1
  { \exp_after:wN \use_i_delimit_by_q_stop:nw \tl_to_str:n {#1} { } \q_stop }
\cs_new_nopar:Npn \str_head_unsafe:n #1
  { \use_i_delimit_by_q_stop:nw #1 { } \q_stop }
```

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

\str_tail:N
\str_tail:n
\str_tail_ignore_spaces:n
\str_tail_unsafe:n
\str_tail_aux:w
\str_tail_aux_ii:w

As when fetching the head of a string, the cases "`ignore_spaces:n`" and "`unsafe:n`" are similar to `\tl_tail:n`. The more commonly used `\str_tail:n` function is a little bit more convoluted: hitting the front of the string with `\reverse_if:N \if_charcode:w \scan_stop:` removes the first character (which necessarily makes the test true, since it cannot match `\scan_stop:`). The auxiliary function inserts the required `\fi:` to close the conditional, and leaves the tail of the string in the input string. The details are such that an empty string has an empty tail.

```
\cs_new_nopar:Npn \str_tail:N { \exp_args:No \str_tail:n }
\cs_set:Npn \str_tail:n #1
  {
    \exp_after:wN \str_tail_aux:w
    \reverse_if:N \if_charcode:w
        \scan_stop: \tl_to_str:n {#1} X X \q_stop
  }
\cs_set_nopar:Npn \str_tail_aux:w #1 X #2 \q_stop { \fi: #1 }
\cs_new:Npn \str_tail_ignore_spaces:n #1
  {
    \exp_after:wN \str_tail_aux_ii:w
    \tl_to_str:n {#1} X { } X \q_stop
  }
\cs_new_nopar:Npn \str_tail_unsafe:n #1
  { \str_tail_aux_ii:w #1 X { } X \q_stop }
\cs_new_nopar:Npn \str_tail_aux_ii:w #1 #2 X #3 \q_stop { #2 }
```

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

`\str_skip_do:nn`
`\str_skip_aux:nnnnnnnnn`
`\str_skip_end:nn`
`\str_skip_end_ii:nwn`

Removes `max(#1,0)` characters then leaves `#2` in the input stream. We remove characters 7 at a time. When the number of characters to remove is not a multiple of 7, we need to remove less than 7 characters in the last step. This is done by inserting a number of `X`, which are discarded as if they were part of the string.

```
\cs_new:Npn \str_skip_do:nn #1
  {
    \if_num:w \int_eval:w #1 > \c_seven
      \exp_after:wN \str_skip_aux:nnnnnnnnn
    \else:
      \exp_after:wN \str_skip_end:n
    \fi:
      {#1}
  }
\cs_new:Npn \str_skip_aux:nnnnnnnnn #1#2#3#4#5#6#7#8#9
  { \exp_args:Nf \str_skip_do:nn { \int_eval:n { #1 - \c_seven } } {#2} }
\cs_new:Npn \str_skip_end:n #1
  {
    \if_case:w \int_eval:w #1 \int_eval_end:
         \str_skip_end_ii:nwn { XXXXXXX }
    \or: \str_skip_end_ii:nwn { XXXXXX }
    \or: \str_skip_end_ii:nwn { XXXXX }
    \or: \str_skip_end_ii:nwn { XXXX }
    \or: \str_skip_end_ii:nwn { XXX }
    \or: \str_skip_end_ii:nwn { XX }
    \or: \str_skip_end_ii:nwn { X }
    \or: \str_skip_end_ii:nwn {  }
    \else: \str_skip_end_ii:nwn { XXXXXXX }
    \fi:
  }
\cs_new:Npn \str_skip_end_ii:nwn #1#2 \fi: #3
  { \fi: \use_i:nnnnnnnn {#3} #1 }
```

(*End definition for* `\str_skip_do:nn`. *This function is documented on page* ??.)

`\str_collect_do:nn`
`\str_collect_aux:n`
`\str_collect_aux:nnNNNNNNN`
`\str_collect_end:nn`
`\str_collect_end_ii:nwn`
`\str_collect_end_iii:nwNNNNNNN`

Collects `max(#1,0)` characters, and feeds them as an argument to `#2`. Again, we grab 7 characters at a time. Instead of inserting a string of `X` to fill in to a multiple of 7, we insert empty groups, so that they disappear in this context where arguments are accumulated.

```
\cs_new:Npn \str_collect_do:nn #1#2
  { \str_collect_aux:n {#1} { \str_collect_end_iii:nwNNNNNNN {#2} } }
\cs_new:Npn \str_collect_aux:n #1
  {
    \int_compare:nNnTF {#1} > \c_seven
      { \str_collect_aux:nnNNNNNNN }
      { \str_collect_end:n }
      {#1}
  }
\cs_new:Npn \str_collect_aux:nnNNNNNNN #1#2 #3#4#5#6#7#8#9
  {
```

```
    \exp_args:Nf \str_collect_aux:n
      { \int_eval:n { #1 - \c_seven } }
      { #2 #3#4#5#6#7#8#9 }
  }
\cs_new:Npn \str_collect_end:n #1
  {
    \if_case:w \int_eval:w #1 \int_eval_end:
         \str_collect_end_ii:nwn { { } { } { } { } { } { } { } { } }
    \or: \str_collect_end_ii:nwn { { } { } { } { } { } { } { } }
    \or: \str_collect_end_ii:nwn { { } { } { } { } { } { } }
    \or: \str_collect_end_ii:nwn { { } { } { } { } { } }
    \or: \str_collect_end_ii:nwn { { } { } { } { } }
    \or: \str_collect_end_ii:nwn { { } { } { } }
    \or: \str_collect_end_ii:nwn { { } { } }
    \or: \str_collect_end_ii:nwn {  }
    \else: \str_collect_end_ii:nwn { { } { } { } { } { } { } { } { } }
    \fi:
  }
\cs_new:Npn \str_collect_end_ii:nwn #1#2 \fi: #3
  { \fi: #3 \q_stop #1 }
\cs_new:Npn \str_collect_end_iii:nwNNNNNNN #1 #2 \q_stop #3#4#5#6#7#8#9
  { #1 {#2#3#4#5#6#7#8#9} }
```

(*End definition for* `\str_collect_do:nn`*. This function is documented on page* ??.)

`\str_item:Nn` `\str_item:nn` `\str_item_ignore_spaces:nn` `\str_item_unsafe:nn` `\str_item_aux:nn`

This is mostly shuffling arguments around to avoid measuring the length of the string more than once, and make sure that the parameters given to `\str_skip_do:nn` are necessarily within the bounds of the length of the string. The `\str_item_ignore_spaces:nn` function cheats a little bit in that it doesn't hand to `\str_item_unsafe:nn` an "other string". This is alright, as everything else is done with undelimited arguments.

```
\cs_new_nopar:Npn \str_item:Nn { \exp_args:No \str_item:nn }
\cs_new:Npn \str_item:nn #1#2
  {
    \exp_args:Nf \tl_to_str:n
      { \exp_args:Nf \str_item_unsafe:nn { \tl_to_other_str:n {#1} } {#2} }
  }
\cs_new:Npn \str_item_ignore_spaces:nn #1
  { \exp_args:No \str_item_unsafe:nn { \tl_to_str:n {#1} } }
\cs_new_nopar:Npn \str_item_unsafe:nn #1#2
  {
    \exp_args:Nff \str_item_aux:nn
      { \int_eval:n {#2} }
      { \str_length_unsafe:n {#1} }
      #1
    \q_stop
  }
\cs_new_nopar:Npn \str_item_aux:nn #1#2
  {
    \int_compare:nNnTF {#1} < \c_zero
      {
```

```
        \int_compare:nNnTF {#1} < {-#2}
          { \use_none_delimit_by_q_stop:w }
          {
            \str_skip_do:nn { #1 + #2 }
              { \use_i_delimit_by_q_stop:nw }
          }
      }
      {
        \int_compare:nNnTF {#1} < {#2}
          {
            \str_skip_do:nn {#1}
              { \use_i_delimit_by_q_stop:nw }
          }
          { \use_none_delimit_by_q_stop:w }
      }
  }
```

(*End definition for* `\str_item:Nn`*. This function is documented on page* **??**.)

`\str_substr:Nnn`
`\str_substr:nnn`
`\str_substr_ignore_spaces:nnn`
`\str_substr_unsafe:nnn`
`\str_substr_aux:www`
`\str_substr_aux:nnw`
`\str_aux_normalize_range:nn`

Sanitize the string. Then evaluate the arguments, replacing them by `\c_zero` or `\c_max_int` if they are empty. This is done by using the construction

```
\int_use:N \int_eval:w #2 \c_zero \c_zero ;
```

which expands to the value of `#2`, followed by `\c_zero \c_zero ;` if `#2` is an expression, and expands to `0\c_zero ;` otherwise. The same is done to the end-point of the range. Then limit the range to be at most the length of the string (this avoids needing to check for the end of the string when grabbing characters). Afterwards, skip characters, then keep some more, and finally drop the end of the string.

```
\cs_new_nopar:Npn \str_substr:Nnn { \exp_args:No \str_substr:nnn }
\cs_new:Npn \str_substr:nnn #1#2#3
  {
    \exp_args:Nf \tl_to_str:n
      {
        \exp_args:Nf \str_substr_unsafe:nnn
          { \tl_to_other_str:n {#1} } {#2} {#3}
      }
  }
\cs_new:Npn \str_substr_ignore_spaces:nnn #1
  { \exp_args:No \str_substr_unsafe:nnn { \tl_to_str:n {#1} } }
\cs_new:Npn \str_substr_unsafe:nnn #1#2#3
  {
    \exp_after:wN \str_substr_aux:www
    \int_value:w \str_length_unsafe:n {#1} \exp_after:wN ;
    \int_use:N \int_eval:w #2 + \c_zero \exp_after:wN ;
    \int_use:N \int_eval:w \exp_args:Nf \str_substr_unsafe:nN {#3} \c_max_int ;
      #1
    \q_stop
  }
\cs_new:Npn \str_substr_unsafe:nN #1 #2
  { \tl_if_empty:nTF {#1} {#2} {#1} }
```

```
\cs_new:Npn \str_substr_aux:www #1; #2; #3;
  {
    \exp_args:Nf \str_substr_aux:nnw
      { \str_aux_normalize_range:nn {#2} {#1} }
      { \str_aux_normalize_range:nn {#3} {#1} }
  }
\cs_new:Npn \str_substr_aux:nnw #1#2
  {
    \str_skip_do:nn {#1}
      {
        \exp_args:Nf \str_collect_do:nn
          { \int_eval:n { #2 - #1 } }
          { \use_i_delimit_by_q_stop:nw }
      }
  }
\cs_new:Npn \str_aux_normalize_range:nn #1#2
  {
    \int_eval:n
      {
        \if_num:w #1 < \c_zero
          \if_num:w #1 < - #2 \exp_stop_f:
            \c_zero
          \else:
            #1 + #2
          \fi:
        \else:
          \if_num:w #1 < #2 \exp_stop_f:
            #1
          \else:
            #2
          \fi:
        \fi:
      }
  }
```

(*End definition for* `\str_substr:Nnn`*. This function is documented on page* **??**.)

7.3 String conditionals

`\str_if_eq:NN`
`\str_if_eq:nn`
`\str_if_eq:xx`

The `nn` and `xx` variants are already defined in l3basics. Note that `\str_if_eq:NN` is different from `\tl_if_eq:NN` because it needs to ignore category codes.

```
\prg_new_conditional:Npnn \str_if_eq:NN #1#2 { p , TF , T , F }
  {
    \if_int_compare:w \pdftex_strcmp:D { \tl_to_str:N #1 } { \tl_to_str:N #2 }
      = \c_zero \prg_return_true: \else: \prg_return_false: \fi:
  }
```

(*End definition for* `\str_if_eq:NN`*. This function is documented on page* **??**.)

`\str_if_contains_char:NNT`
`\str_if_contains_char:NNTF`
`\str_if_contains_char:nNTF`
`\str_if_contains_char_aux:NN`
`\str_if_contains_char_end:`

Loop over the characters of the string, comparing character codes. We allow `#2` to be a single-character control sequence, hence the use of `\int_compare:nNnT` rather than

`\token_if_eq_charcode:NNT`. The loop is broken if character codes match. Otherwise we return "false".

```
\prg_new_conditional:Npnn \str_if_contains_char:NN #1#2 { T , TF }
  {
    \exp_after:wN \str_if_contains_char_aux:NN \exp_after:wN #2
      #1 { \prg_map_break:n { ? \fi: } }
    \prg_break_point:n { }
    \prg_return_false:
  }
\prg_new_conditional:Npnn \str_if_contains_char:nN #1#2 { TF }
  {
    \exp_after:wN \str_if_contains_char_aux:NN \exp_after:wN #2
      \tl_to_str:n {#1} { \prg_map_break:n { ? \fi: } }
    \prg_break_point:n { }
    \prg_return_false:
  }
\cs_new_nopar:Npn \str_if_contains_char_aux:NN #1#2
  {
    \if_charcode:w #1 #2
      \exp_after:wN \str_if_contains_char_end:
    \fi:
    \str_if_contains_char_aux:NN #1
  }
\cs_new_nopar:Npn \str_if_contains_char_end:
  { \prg_map_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_aux_octal_use:NTF` TEX 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.

```
\prg_new_conditional:Npnn \str_aux_octal_use:N #1 { TF }
  {
    \if_num:w \c_one < '1 \token_to_str:N #1 \exp_stop_f:
      #1 \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

(*End definition for* `\str_aux_octal_use:NTF`*. This function is documented on page* **??**.)

`\str_aux_hexadecimal_use:NTF` TEX detects uppercase hexadecimal digits for us (see `\str_aux_octal_use:NTF`, but not the lowercase letters, which we need to detect and replace by their uppercase counterpart.

```
\prg_new_conditional:Npnn \str_aux_hexadecimal_use:N #1 { TF }
  {
    \if_num:w \c_two < "1 \token_to_str:N #1 \exp_stop_f:
      #1 \prg_return_true:
    \else:
      \if_case:w \int_eval:w
```

```
          \exp_after:wN ‘ \token_to_str:N #1 - ‘a
        \int_eval_end:
          A
      \or: B
      \or: C
      \or: D
      \or: E
      \or: F
      \else:
        \prg_return_false:
        \exp_after:wN \use_none:n
      \fi:
      \prg_return_true:
    \fi:
  }
```

(*End definition for* `\str_aux_hexadecimal_use:NTF`*. This function is documented on page* 8.)

7.4 Conversions

7.4.1 Producing one byte or character

`\c_str_byte_0_tl`
`\c_str_byte_-1_tl`
`\c_str_positive_bytes_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. Simultaneously, we build a list of all bytes (from which we remove the null byte) in `\c_str_positive_bytes_tl`.

```
\group_begin:
  \tl_clear:N \l_str_tmpa_tl
  \char_set_catcode_other:n { \c_zero }
  \tl_gset:Nx \g_str_result_tl { \tl_to_str:n { 0123456789ABCDEF } }
  \tl_map_inline:Nn \g_str_result_tl
    { \char_set_lccode:nn {‘#1} {‘#1} }
  \tl_map_inline:Nn \g_str_result_tl
    {
      \tl_map_inline:Nn \g_str_result_tl
        {
          \char_set_lccode:nn { \c_zero } {"#1##1}
          \tl_to_lowercase:n
            {
              \tl_const:cx
                { c_str_byte_ \int_eval:n {"#1##1} _tl }
                { ^^@ #1 ##1 }
              \tl_put_right:Nn \l_str_tmpa_tl { ^^@ }
            }
        }
    }
  \tl_const:Nx \c_str_positive_bytes_tl
    { \exp_after:wN \use_none:n \l_str_tmpa_tl }
```

```
\group_end:
\tl_const:cn { c_str_byte_-1_tl } { { } \use_none:n { } }
```
(*End definition for* `\c_str_byte_0_tl`*. This function is documented on page* **??**.)

`\str_output_byte:n`
`\str_output_byte:w`
`\str_output_hexadecimal:n`
`\str_output_hexadecimal:w`
`\str_output_end:`

For now, no error detection, we simply pick either the byte or the hexadecimal representation of it from the three-character token list corresponding to the argument. The value -1 produces an empty result.

```
\cs_new:Npn \str_output_byte:n #1
  { \str_output_byte:w #1 \str_output_end: }
\cs_new_nopar:Npn \str_output_byte:w
  {
    \exp_after:wN \exp_after:wN
    \exp_after:wN \use_i:nnn
    \cs:w c_str_byte_ \int_use:N \int_eval:w
  }
\cs_new:Npn \str_output_hexadecimal:n #1
  { \str_output_hexadecimal:w #1 \str_output_end: }
\cs_new_nopar:Npn \str_output_hexadecimal:w
  {
    \exp_after:wN \exp_after:wN
    \exp_after:wN \use_none:n
    \cs:w c_str_byte_ \int_use:N \int_eval:w
  }
\cs_new_nopar:Npn \str_output_end:
  { \int_eval_end: _tl \cs_end: }
```
(*End definition for* `\str_output_byte:n`*. This function is documented on page* **??**.)

7.4.2 Mapping functions for encoding conversion

`\str_aux_gmap_result:N`
`\str_aux_gmap_result_loop:NN`

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.

```
\cs_new_protected_nopar:Npn \str_aux_gmap_result:N #1
  {
    \tl_gset:Nx \g_str_result_tl
      {
        \exp_after:wN \str_aux_gmap_result_loop:NN
        \exp_after:wN #1
          \g_str_result_tl { ? \prg_map_break: }
        \prg_break_point:n { }
      }
  }
\cs_new_nopar:Npn \str_aux_gmap_result_loop:NN #1#2
  {
    \use_none:n #2
    #1#2
    \str_aux_gmap_result_loop:NN #1
  }
```
(*End definition for* `\str_aux_gmap_result:N`*. This function is documented on page* **??**.)

\str_aux_gmap_internal_result:N
\str_aux_gmap_internal_result_loop:Nw

This maps the function #1 over all character codes in \g_str_result_tl, which must be in the internal representation.

```
\cs_new_protected_nopar:Npn \str_aux_gmap_internal_result:N #1
  {
    \tl_gset:Nx \g_str_result_tl
      {
        \exp_after:wN \str_aux_gmap_internal_result_loop:Nww
        \exp_after:wN #1
          \g_str_result_tl \s_tl \q_stop \prg_map_break: \s_tl
        \prg_break_point:n { }
      }
  }
\cs_new_nopar:Npn \str_aux_gmap_internal_result_loop:Nww #1 #2 \s_tl #3 \s_tl
  {
    \use_none_delimit_by_q_stop:w #3 \q_stop
    #1 {#3}
    \str_aux_gmap_internal_result_loop:Nww #1
  }
```

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

7.4.3 Unescape user input

The code of this section is used both here for `\str_(g)input:Nn`, and in the regular expression module to go through the regular expression once before actually parsing it. The goal in that case is to turn any character with a meaning in regular expressions (`*`, `?`, `{`, etc.) into a marker indicating that this was a special character, and replace any escaped character by the corresponding unescaped character, so that the l3regex code can avoid caring about escaping issues (those can become quite complex to handle in combination with ranges in character classes).

The idea is to feed unescaped characters to one function, escaped characters to another, and feed `\a`, `\e`, `\f`, `\n`, `\r`, `\t` and `\x` converted to the appropriate character to a third function. Spaces are ignored unless escaped. For the `\str_(g)input:Nn` application, all the functions are simply `\token_to_str:N` (this ensures that spaces correctly get assigned category code 10). For the l3regex applications, the functions do some further tests on the character they receive.

\str_input:Nn
\str_ginput:Nn

Simple wrappers around the internal `\str_aux_escape:NNNn`.

```
\cs_new_protected:Npn \str_input:Nn #1#2
  {
    \str_aux_escape:NNNn \token_to_str:N \token_to_str:N \token_to_str:N {#2}
    \tl_set_eq:NN #1 \g_str_result_tl
  }
\cs_new_protected:Npn \str_ginput:Nn #1#2
  {
    \str_aux_escape:NNNn \token_to_str:N \token_to_str:N \token_to_str:N {#2}
    \tl_gset_eq:NN #1 \g_str_result_tl
  }
```

(*End definition for* `\str_input:Nn` *and* `\str_ginput:Nn`. *These functions are documented on page* 3.)

`\str_escape_use:NNNn` Use the internal `\str_aux_escape:NNNn`, then leave the result in the input stream. This is used in l3regex.

```
\cs_new_protected:Npn \str_escape_use:NNNn #1#2#3#4
  {
    \str_aux_escape:NNNn #1#2#3 {#4}
    \g_str_result_tl
  }
```

(*End definition for* `\str_escape_use:NNNn`. *This function is documented on page* 7.)

`\str_aux_escape:NNNn` `\str_aux_escape_loop:N` `\str_aux_escape_\:w` Most of the work is done within an x-expanding assignment, but the `\x` escape sequence cannot be done in that way. Therefore, we interrupt the assignment at each `\x` escape sequence, store the partial result in a `\toks` register, and at the end unpack all of the `\toks` registers to the left of the last chunk of `\g_str_result_tl`.

```
\cs_new_protected:Npn \str_aux_escape:NNNn #1#2#3#4
  {
    \group_begin:
      \cs_set_nopar:Npn \str_aux_escape_unescaped:N { #1 }
      \cs_set_nopar:Npn \str_aux_escape_escaped:N { #2 }
      \cs_set_nopar:Npn \str_aux_escape_raw:N { #3 }
      \int_set:Nn \tex_escapechar:D { 92 }
      \str_gset_other:Nn \g_str_result_tl {#4}
      \int_zero:N \l_str_tmpa_int
      \tl_gset:Nx \g_str_result_tl
        {
          \exp_after:wN \str_aux_escape_loop:N \g_str_result_tl
          \q_recursion_tail \q_recursion_stop
        }
      \tl_gput_left:Nx \g_str_result_tl
        { \str_aux_toks_range:nn \c_zero \l_str_tmpa_int }
    \group_end:
  }
\cs_new_nopar:Npn \str_aux_escape_loop:N #1
  {
    \cs_if_exist_use:cF { str_aux_escape_\token_to_str:N #1:w }
      { \str_aux_escape_unescaped:N #1 }
    \str_aux_escape_loop:N
  }
\cs_new_nopar:cpn { str_aux_escape_ \c_backslash_str :w }
    \str_aux_escape_loop:N #1
  {
    \cs_if_exist_use:cF { str_aux_escape_/\token_to_str:N #1:w }
      { \str_aux_escape_escaped:N #1 }
    \str_aux_escape_loop:N
  }
```

(*End definition for* `\str_aux_escape:NNNn`. *This function is documented on page* ??.)

\str_aux_escape_unescaped:N
\str_aux_escape_escaped:N
\str_aux_escape_raw:N

Those functions are never called before being given a new meaning, so their definitions here don't matter.

```
\cs_new_eq:NN \str_aux_escape_unescaped:N \prg_do_nothing:
\cs_new_eq:NN \str_aux_escape_escaped:N   \prg_do_nothing:
\cs_new_eq:NN \str_aux_escape_raw:N       \prg_do_nothing:
```

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

\str_aux_escape_\q_recursion_tail:w
\str_aux_escape_/\q_recursion_tail:w
\str_aux_escape_ :w
\str_aux_escape_/a:w
\str_aux_escape_/e:w
\str_aux_escape_/f:w
\str_aux_escape_/n:w
\str_aux_escape_/r:w
\str_aux_escape_/t:w

The loop is ended upon seeing `\q_recursion_tail`. Spaces are ignored, and `\a`, `\e`, `\f`, `\n`, `\r`, `\t` take their meaning here.

```
\cs_new_eq:cN
  { str_aux_escape_ \c_backslash_str q_recursion_tail :w }
  \use_none_delimit_by_q_recursion_stop:w
\cs_new_eq:cN
  { str_aux_escape_ / \c_backslash_str q_recursion_tail :w }
  \use_none_delimit_by_q_recursion_stop:w
\cs_new_nopar:cpn { str_aux_escape_~:w } { }
\cs_new_nopar:cpx { str_aux_escape_/a:w }
  { \exp_not:N \str_aux_escape_raw:N \iow_char:N \^^G }
\cs_new_nopar:cpx { str_aux_escape_/t:w }
  { \exp_not:N \str_aux_escape_raw:N \iow_char:N \^^I }
\cs_new_nopar:cpx { str_aux_escape_/n:w }
  { \exp_not:N \str_aux_escape_raw:N \iow_char:N \^^J }
\cs_new_nopar:cpx { str_aux_escape_/f:w }
  { \exp_not:N \str_aux_escape_raw:N \iow_char:N \^^L }
\cs_new_nopar:cpx { str_aux_escape_/r:w }
  { \exp_not:N \str_aux_escape_raw:N \iow_char:N \^^M }
\cs_new_nopar:cpx { str_aux_escape_/e:w }
  { \exp_not:N \str_aux_escape_raw:N \iow_char:N \^^[ }
```

(*End definition for* `\str_aux_escape_\q_recursion_tail:w`*. This function is documented on page* ??.)

\str_aux_escape_/x:w
\str_aux_escape_x_test:N
\str_aux_escape_x_unbraced_i:N
\str_aux_escape_x_unbraced_ii:N
\str_aux_escape_x_braced_loop:N
\str_aux_escape_x_braced_end:N
\str_aux_escape_x_end:

When `\x` is encountered, interrupt the assignment, and distinguish the cases of a braced or unbraced syntax. In the braced case, collect arbitrarily many hexadecimal digits, building the number in `\l_str_char_int` (using `\str_aux_hexadecimal_use:NTF`), and check that the run of digits was stopped by a closing brace. In the unbraced case, collect up to two hexadecimal digits, possibly less, building the character number in `\l_str_char_int`. In both cases, once all digits have been collected, use the TEX primitive `\lowercase` to produce that character, and use an `\if_false:` trick to restart the assignment.

```
\cs_new_nopar:cpn { str_aux_escape_/x:w } \str_aux_escape_loop:N
  {
    \if_false: { \fi: }
    \tex_toks:D \l_str_tmpa_int \exp_after:wN { \g_str_result_tl }
    \int_incr:N \l_str_tmpa_int
    \int_zero:N \l_str_char_int
    \str_aux_escape_x_test:N
  }
\cs_new_protected_nopar:Npx \str_aux_escape_x_test:N #1
  {
```

```
    \exp_not:N \token_if_eq_charcode:NNTF \c_space_token #1
      { \exp_not:N \str_aux_escape_x_test:N }
      {
        \l_str_char_int = "0
          \exp_not:N \token_if_eq_charcode:NNTF \c_lbrace_str #1
            { \exp_not:N \str_aux_escape_x_braced_loop:N }
            { \exp_not:N \str_aux_escape_x_unbraced_i:N #1 }
      }
  }
\cs_new_nopar:Npn \str_aux_escape_x_unbraced_i:N #1
  {
    \str_aux_hexadecimal_use:NTF #1
      { \str_aux_escape_x_unbraced_ii:N }
      { \exp_stop_f: \str_aux_escape_x_end: #1 }
  }
\cs_new_nopar:Npn \str_aux_escape_x_unbraced_ii:N #1
  {
    \token_if_eq_charcode:NNTF \c_space_token #1
      { \str_aux_escape_x_unbraced_ii:N }
      {
        \str_aux_hexadecimal_use:NTF #1
          { \exp_stop_f: \str_aux_escape_x_end: }
          { \exp_stop_f: \str_aux_escape_x_end: #1 }
      }
  }
\cs_new_nopar:Npn \str_aux_escape_x_braced_loop:N #1
  {
    \token_if_eq_charcode:NNTF \c_space_token #1
      { \str_aux_escape_x_braced_loop:N }
      {
        \str_aux_hexadecimal_use:NTF #1
          { \str_aux_escape_x_braced_loop:N }
          { \exp_stop_f: \str_aux_escape_x_braced_end:N #1 }
      }
  }
\cs_new_protected_nopar:Npx \str_aux_escape_x_braced_end:N #1
  {
    \exp_not:N \token_if_eq_charcode:NNTF \c_rbrace_str #1
      { \exp_not:N \str_aux_escape_x_end: }
      {
        \msg_kernel_error:nn { str } { x-missing-brace }
        \exp_not:N \str_aux_escape_x_end: #1
      }
  }
\group_begin:
  \char_set_catcode_other:N \^^@
  \cs_new_protected_nopar:Npn \str_aux_escape_x_end:
    {
      \tex_lccode:D \c_zero \l_str_char_int
      \tl_to_lowercase:n
```

```
        {
          \tl_gset:Nx \g_str_result_tl
            { \if_false: } \fi:
            \str_aux_escape_raw:N ^^@
            \str_aux_escape_loop:N
        }
    }
\group_end:
```

(*End definition for* `\str_aux_escape_/x:w`. *This function is documented on page* ??.)

7.4.4 Framework for encoding 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 as character tokens with the relevant character code, or as pairs of hexadecimal digits, *etc.* The questions of going from arbitrary characters to bytes, and from bytes to TEX tokens are mostly independent.

Conversions are done in four steps:

- "input" produces a string of bytes;
- "from" interprets the byte string in the old encoding, and converts it to a comma-list, each character becoming "⟨*character code*⟩`,`";
- "to" encodes the internal comma list as a byte string in the new encoding;
- "output" escapes bytes as requested.

The process is modified in case one of the encoding is `internal` or `native`: then the input or output step is ignored.

`\str_set_convert:Nnnn`
`\str_gset_convert:Nnnn`
`\str_convert_aux_i:NNnnn`
`\str_convert_aux_ii:wwwnn`
`\str_convert_aux_iii:nnF`
`\str_convert_aux_iv:nnnF`
`\str_convert_aux_v:NNnNN`

The input string is stored in `\g_str_result_tl`, then we call the various conversion functions, which all act on `\g_str_result_tl`. The arguments `#3` and `#4` of the `aux_i` function are split into their ⟨*encoding*⟩ and ⟨*escaping*⟩ parts by `aux_ii`. The conversion function names are built by `aux_iii`, which first tries to use the name as supplied by the user, then filters it with `\str_aux_lowercase_alphanum:n`, and calls `aux_iv` to load the relevant definition file. In all cases, `aux_iii` ensures that the two required conversion functions are defined, with a fall-back to the native encoding with no escaping. Once this is done, `aux_v` does the main work:

- `#1` is the encoding conversion function;
- `#2` is the escaping function;
- `#3` is the escaping name;
- `#4` is the "native" encoding function (either "from" or "to");
- `#5` is `\use_i:nn` or `\use_ii:nn`.

When converting to the internal format, the escaping must be done first, then the encoding (this is controlled by #5). If the encoding is native, then the escaping is ignored and must be empty. Otherwise, the escaping #2 is performed.

```
\cs_new_protected:Npn \str_set_convert:Nnnn
  { \str_convert_aux_i:NNnnn \tl_set_eq:NN }
\cs_new_protected:Npn \str_gset_convert:Nnnn
  { \str_convert_aux_i:NNnnn \tl_gset_eq:NN }
\cs_new_protected:Npn \str_convert_aux_i:NNnnn #1#2#3#4#5
  {
    \group_begin:
      \str_gset_other:Nn \g_str_result_tl {#5}

      \exp_after:wN \str_convert_aux_ii:wwwnn
        \tl_to_str:n {#3} /// \q_stop
        { from } { input } \str_convert_from_native: \use_i:nn

      \exp_after:wN \str_convert_aux_ii:wwwnn
        \tl_to_str:n {#4} /// \q_stop
        { to } { output } \str_convert_to_native: \use_ii:nn

    \group_end:
    #1 #2 \g_str_result_tl
  }
\cs_new_protected:Npn \str_convert_aux_ii:wwwnn #1 / #2 // #3 \q_stop #4#5
  {
    \str_convert_aux_iii:nnF {#4} {#1} {native}
    \str_convert_aux_iii:nnF {#5} {#2} { }
    \exp_args:Ncc \str_convert_aux_v:NNnNN
      { str_convert_#4_#1: } { str_convert_#5_#2: } {#2}
  }
\cs_new_protected:Npn \str_convert_aux_iii:nnF #1#2#3
  {
    \cs_if_exist:cF { str_convert_#1_#2: }
      {
        \exp_args:Nx \str_convert_aux_iv:nnnF
          { \str_aux_lowercase_alphanum:n {#2} }
          {#1} {#2} {#3}
      }
  }
\cs_new_protected:Npn \str_convert_aux_iv:nnnF #1#2#3#4
  {
    \cs_if_exist:cF { str_convert_#2_#1: }
      {
        \group_begin:
          \str_load_catcodes:
          \str_load_one:n {#1}
        \group_end:
      }
    \cs_if_exist:cTF { str_convert_#2_#1: }
```

```
      { \cs_gset_eq:cc { str_convert_#2_#3: } { str_convert_#2_#1: } }
      {
        \msg_kernel_error:nnx { str } { unknown-#2 } {#3}
        \cs_gset_eq:cc { str_convert_#2_#3: } { str_convert_#2_#4: }
      }
  }
\cs_new_protected:Npn \str_convert_aux_v:NNnNN #1#2#3#4#5
  {
    #5 { } {#1}
    \cs_if_eq:NNTF #1 #4
      {
        \tl_if_empty:nF {#3}
          { \msg_kernel_error:nnx { str } { native-ignore-escaping } {#3} }
      }
      { #2 }
    #5 {#1} { }
  }
```

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

`\str_aux_lowercase_alphanum:n`
`\str_aux_lowercase_alphanum_loop:N`

This function keeps only letters and digits, with upper case letters converted to lower case.

```
\cs_new:Npn \str_aux_lowercase_alphanum:n #1
  {
    \exp_after:wN \str_aux_lowercase_alphanum_loop:N
      \tl_to_str:n {#1} { ? \prg_map_break: }
    \prg_break_point:n { }
  }
\cs_new:Npn \str_aux_lowercase_alphanum_loop:N #1
  {
    \use_none:n #1
    \if_num:w `#1 < \c_ninety_one
      \if_num:w `#1 < \c_sixty_five
        \if_num:w \c_one < 1#1 \exp_stop_f:
          #1
        \fi:
      \else:
        \str_output_byte:n { `#1 + \c_thirty_two }
      \fi:
    \else:
      \if_num:w `#1 < \c_one_hundred_twenty_three
        \if_num:w `#1 < \c_ninety_seven
        \else:
          #1
        \fi:
      \fi:
    \fi:
    \str_aux_lowercase_alphanum_loop:N
  }
```

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

`\str_convert_from_internal:`
`\str_convert_to_internal:`

Converting from the internal format to itself is of course trivial.

```
\cs_new_protected_nopar:Npn \str_convert_from_internal: { }
\cs_new_protected_nopar:Npn \str_convert_to_internal:   { }
```

(*End definition for* `\str_convert_from_internal:`. *This function is documented on page* **??**.)

7.4.5 Loading encoding files

`\str_load:n`
`\str_load_one:n`
`\str_load_alias:n`
`\str_load_alias_aux:nnn`

Loading encoding or escaping files can be done in bulk by specifying a comma-list of encodings and escapings. Each name is normalized: only alphanumeric characters are kept, lowercased. The relevant file is loaded if it was not loaded already.

```
\cs_new_protected:Npn \str_load:n #1
  {
    \group_begin:
      \str_load_catcodes:
      \clist_map_inline:nn {#1}
        {
          \exp_args:Nx \str_load_one:n
            { \str_aux_lowercase_alphanum:n {#1} }
        }
    \group_end:
  }
\cs_new_protected:Npn \str_load_one:n #1
  {
    \cs_if_exist:cF { str_convert_from_ #1 : }
      {
        \cs_if_exist:cF { str_convert_input_#1: }
          {
            \file_if_exist:nTF { l3str- #1 .def }
              { \file_input:n { l3str- #1 .def } }
              { \str_load_alias:n {#1} }
          }
      }
  }
\cs_new_protected:Npn \str_load_alias:n #1
  {
    \exp_args:NNx \prop_get:NnNTF \g_str_aliases_prop {#1} \l_str_tmpa_tl
      {
        \str_load_one:n { \l_str_tmpa_tl }
        \cs_if_exist:cTF
          { str_convert_from_ \l_str_tmpa_tl : }
          {
            \str_load_alias_aux:nnn {#1} { \l_str_tmpa_tl } { from }
            \str_load_alias_aux:nnn {#1} { \l_str_tmpa_tl } { to }
          }
          {
            \str_load_alias_aux:nnn {#1} { \l_str_tmpa_tl } { input }
            \str_load_alias_aux:nnn {#1} { \l_str_tmpa_tl } { output }
```

```
          }
      }
      { \msg_kernel_error:nnx { str } { file-not-found } {#1} }
  }
\cs_new_protected:Npn \str_load_alias_aux:nnn #1#2#3
  { \cs_new_eq:cc { str_convert_#3_#1: } { str_convert_#3_#2: } }
```

(*End definition for* `\str_load: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.

```
\cs_new_protected:Npn \str_load_catcodes:
  {
    \char_set_catcode_escape:N \\
    \char_set_catcode_group_begin:N \{
    \char_set_catcode_group_end:N \}
    \char_set_catcode_math_superscript:N \^
    \char_set_catcode_ignore:N \ %
    \tl_map_function:nN { abcdefghijklmnopqrstuvwxyz_:ABCDEFN }
      \char_set_catcode_letter:N
    \tl_map_function:nN { 0123456789"'?*+ }
      \char_set_catcode_other:N
    \char_set_catcode_comment:N \%
    \int_set:Nn \tex_endlinechar:D {32}
  }
```

(*End definition for* `\str_load_catcodes:`. *This function is documented on page* ??.)

`\str_convert_from_eight_bit:n`
`\str_convert_from_eight_bit_load:nn`
`\str_convert_from_eight_bit_load_missing:n`
`\str_convert_from_eight_bit_aux:N`

```
\cs_new_protected:Npn \str_convert_from_eight_bit:n #1
  {
    \group_begin:
      \int_zero:N \l_str_tmpa_int
      \exp_last_unbraced:Nx \str_convert_from_eight_bit_load:nn
        { \tl_use:c { c_str_encoding_#1_tl } }
        { \q_stop \prg_map_break: } { }
      \prg_break_point:n { }
      \exp_last_unbraced:Nx \str_convert_from_eight_bit_load_missing:n
        { \tl_use:c { c_str_encoding_#1_missing_tl } }
        { \q_stop \prg_map_break: }
      \prg_break_point:n { }
      \str_aux_gmap_result:N \str_convert_from_eight_bit_aux:N
    \group_end:
  }
\cs_new_protected_nopar:Npn \str_convert_from_eight_bit_load:nn #1#2
  {
    \use_none_delimit_by_q_stop:w #1 \q_stop
    \tex_dimen:D "#1 = \l_str_tmpa_int sp \scan_stop:
    \tex_skip:D \l_str_tmpa_int = "#1 sp \scan_stop:
    \tex_toks:D \l_str_tmpa_int \exp_after:wN { \int_value:w "#2 }
```

```
    \tex_advance:D \l_str_tmpa_int \c_one
    \str_convert_from_eight_bit_load:nn
  }
\cs_new_protected_nopar:Npn \str_convert_from_eight_bit_load_missing:n #1
  {
    \use_none_delimit_by_q_stop:w #1 \q_stop
    \tex_dimen:D "#1 = \l_str_tmpa_int sp \scan_stop:
    \tex_skip:D \l_str_tmpa_int = "#1 sp \scan_stop:
    \tex_toks:D \l_str_tmpa_int \exp_after:wN
      { \int_use:N \c_str_replacement_char_int }
    \tex_advance:D \l_str_tmpa_int \c_one
    \str_convert_from_eight_bit_load_missing:n
  }
\cs_new:Npn \str_convert_from_eight_bit_aux:N #1
  {
    #1 \s_tl
    \if_num:w \tex_dimen:D `#1 < \l_str_tmpa_int
      \if_num:w \tex_skip:D \tex_dimen:D `#1 = `#1 \exp_stop_f:
        \tex_the:D \tex_toks:D \tex_dimen:D
      \fi:
    \fi:
    \int_value:w `#1 \s_tl
  }
```

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

`\str_convert_to_eight_bit:n`
`\str_convert_to_eight_bit_load:nn`
`\str_convert_to_eight_bit_aux:n`
`\str_convert_to_eight_bit_aux_ii:n`

```
\cs_new_protected:Npn \str_convert_to_eight_bit:n #1
  {
    \group_begin:
      \int_zero:N \l_str_tmpa_int
      \exp_last_unbraced:Nx \str_convert_to_eight_bit_load:nn
        { \tl_use:c { c_str_encoding_#1_tl } }
        { \q_stop \prg_map_break: } { }
      \prg_break_point:n { }
      \str_aux_gmap_internal_result:N \str_convert_to_eight_bit_aux:n
    \group_end:
  }
\cs_new_protected_nopar:Npn \str_convert_to_eight_bit_load:nn #1#2
  {
    \use_none_delimit_by_q_stop:w #1 \q_stop
    \tex_dimen:D "#2 = \l_str_tmpa_int sp \scan_stop:
    \tex_skip:D \l_str_tmpa_int = "#2 sp \scan_stop:
    \exp_args:NNf \tex_toks:D \l_str_tmpa_int
      { \str_output_byte:n { "#1 } }
    \tex_advance:D \l_str_tmpa_int \c_one
    \str_convert_to_eight_bit_load:nn
  }
\cs_new:Npn \str_convert_to_eight_bit_aux:n #1
  {
    \if_num:w #1 > \c_max_register_int
```

```
      \msg_expandable_kernel_error:nnn
        { str } { eight-bit-to-byte-overflow } {#1}
    \else:
      \if_num:w \tex_dimen:D #1 < \l_str_tmpa_int
        \if_num:w \tex_skip:D \tex_dimen:D #1 = #1 \exp_stop_f:
          \tex_the:D \tex_toks:D \tex_dimen:D #1 \exp_stop_f:
          \exp_after:wN \exp_after:wN \exp_after:wN \use_none:nn
        \fi:
      \fi:
      \str_convert_to_eight_bit_aux_ii:n {#1}
    \fi:
  }
\cs_new:Npn \str_convert_to_eight_bit_aux_ii:n #1
  {
    \if_num:w #1 < \c_two_hundred_fifty_six
      \str_output_byte:n {#1}
    \else:
      \msg_expandable_kernel_error:nn { str } { eight-bit-to-byte }
    \fi:
  }
```

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

`\str_encoding_eight_bit:n`

```
\cs_new_protected:Npn \str_encoding_eight_bit:n #1
  {
    \cs_new_protected:cpn { str_convert_from_#1: }
      { \str_convert_from_eight_bit:n {#1} }
    \cs_new_protected:cpn { str_convert_to_#1: }
      { \str_convert_to_eight_bit:n {#1} }
  }
```

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

7.4.6 Byte input and output

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

`\str_filter_bytes:n`
`\str_filter_bytes_aux:N`

In the case of pdfTEX, every character is a byte. For Unicode-aware engines, test the character code. Spaces have already been given the correct category code when this function is called.

```
\pdftex_if_engine:TF
  { \cs_new_eq:NN \str_filter_bytes:n \use:n }
  {
    \cs_new_nopar:Npn \str_filter_bytes:n #1
      {
        \str_filter_bytes_aux:N #1
          { ? \prg_map_break: }
        \prg_break_point:n { }
```

```
      }
    \cs_new_nopar:Npn \str_filter_bytes_aux:N #1
      {
        \use_none:n #1
        \if_num:w `#1 < 256 \exp_stop_f: #1 \fi:
        \str_filter_bytes_aux:N
      }
  }
```

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

`\str_convert_input_:` `\str_convert_input_bytes:`

The simplest input method removes non-bytes from `\g_str_result_tl`.

```
\pdftex_if_engine:TF
  { \cs_new_protected_nopar:Npn \str_convert_input_: { } }
  {
    \cs_new_protected_nopar:Npn \str_convert_input_:
      {
        \tl_gset:Nx \g_str_result_tl
          { \exp_args:No \str_filter_bytes:n \g_str_result_tl }
      }
  }
\cs_new_eq:NN \str_convert_input_bytes: \str_convert_input_:
```

(*End definition for* `\str_convert_input_:`. *This function is documented on page* **??**.)

`\str_convert_output_:` `\str_convert_output_bytes:`

The simplest form of output leaves the bytes from the previous step of the conversion unchanged.

```
\cs_new_protected_nopar:Npn \str_convert_output_: { }
\cs_new_eq:NN \str_convert_output_bytes: \str_convert_output_:
```

(*End definition for* `\str_convert_output_:`. *This function is documented on page* **??**.)

7.4.7 Convert to and from native strings

`\str_convert_from_native:` `\str_convert_from_native_aux:N`

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

```
\cs_new_protected_nopar:Npn \str_convert_from_native:
  { \str_aux_gmap_result:N \str_convert_from_native_aux:N }
\cs_new:Npn \str_convert_from_native_aux:N #1
  { #1 \s_tl \int_value:w `#1 \s_tl }
```

(*End definition for* `\str_convert_from_native:`. *This function is documented on page* **??**.)

`\str_convert_to_native:` `\str_convert_to_native_aux:w`

This function is based on the internal functions `\str_aux_convert_to_native:n` and `\str_aux_convert_to_native_step:n`. Loop through the comma-list, and call `\str_aux_convert_to_native_step:n` with the relevant integer argument. We need to store `\g_str_result_tl` in a temporary token list, because that global "result" is cleared silently by `\str_aux_convert_to_native:n`.

```
\cs_new_protected_nopar:Npn \str_convert_to_native:
  {
    \tl_set_eq:NN \l_str_tmpa_tl \g_str_result_tl
    \str_aux_convert_to_native:n
      {
```

```
        \exp_after:wN \str_convert_to_native_aux:w
        \l_str_tmpa_tl \s_tl { \q_stop \prg_map_break: } \s_tl
      }
  }
\cs_new_protected_nopar:Npn \str_convert_to_native_aux:w #1 \s_tl #2 \s_tl
  {
    \use_none_delimit_by_q_stop:w #2 \q_stop
    \str_aux_convert_to_native_step:n {#2}
    \str_convert_to_native_aux:w
  }
```

(*End definition for* `\str_convert_to_native:`. *This function is documented on page* ??.)

`\str_aux_convert_to_native:n`
`\str_aux_convert_to_native_step:n`
`\str_aux_convert_to_native_flush:`
`\str_aux_convert_to_native_filter:N`

```
\cs_new_protected_nopar:Npn \str_aux_convert_to_native:n #1
  {
    \tex_lccode:D \c_zero \c_zero
    \int_zero:N \l_str_tmpa_int
    \tl_gclear:N \g_str_result_tl
    #1
    \prg_break_point:n { \str_aux_convert_to_native_flush: }
  }
\cs_new_protected_nopar:Npn \str_aux_convert_to_native_step:n #1
  {
    \if_num:w \l_str_tmpa_int = \c_two_hundred_fifty_five
      \str_aux_convert_to_native_flush:
      \l_str_tmpa_int \c_zero
    \fi:
    \tex_advance:D \l_str_tmpa_int \c_one
    \if_int_compare:w #1 > \c_max_char_int
      \msg_kernel_error:nnx { str } { overflow } {#1}
      \tex_lccode:D \l_str_tmpa_int \c_str_replacement_char_int
    \else:
      \tex_lccode:D \l_str_tmpa_int \int_eval:w #1 \int_eval_end:
    \fi:
  }
\cs_new_protected_nopar:Npn \str_aux_convert_to_native_flush:
  {
    \use:x
      {
        \tl_to_lowercase:n
          {
            \tl_gput_right:Nx \exp_not:N \g_str_result_tl
              {
                \exp_after:wN \str_aux_convert_to_native_filter:N
                  \c_str_positive_bytes_tl
                  { ? = \c_zero \fi: \prg_map_break: }
                \prg_break_point:n { }
              }
          }
      }
```

```
  }
\group_begin:
  \char_set_catcode_other:n { 0 }
  \cs_new:Npn \str_aux_convert_to_native_filter:N #1
    {
      \if_num:w `#1 > \l_str_tmpa_int
        \exp_after:wN \prg_map_break:
      \fi:
      \if_num:w \tex_lccode:D `#1 = \c_zero
        ^^@
      \else:
        #1
      \fi:
      \str_aux_convert_to_native_filter:N
    }
\group_end:
```

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

7.5 Messages

```
\msg_kernel_new:nnnn { str } { x-missing-brace }
  { Missing~closing~brace~in~ \token_to_str:N \x ~byte~sequence. }
  {
    You~wrote~something~like~
    `\iow_char:N\\x\{ \int_to_hexadecimal:n { \l_str_char_int }'.~
    The~closing~brace~is~missing.
  }
\msg_kernel_new:nnnn { str } { overflow }
  { Character~code~#1~too~big. }
  {
    \int_compare:nNnTF {#1} > { 1114111 }
      { The~Unicode~standard~limits~code~points~to~1114111. }
      {
        The~pdfTeX~engine~only~supports~8-bit~characters:~
        valid~character~codes~are~in~the~range~[0,255].~
        To~manipulate~arbitrary~Unicode,~use~LuaTeX~or~XeTeX.
      }
  }
\msg_kernel_new:nnn { str } { convert-input }
  { Input~scheme~`#1'~unknown. }
\msg_kernel_new:nnn { str } { convert-from }
  { Encoding~`#1'~unknown. }
\msg_kernel_new:nnn { str } { convert-to }
  { Encoding~`#1'~unknown.~Using~UTF-8. }
\msg_kernel_new:nnn { str } { convert-output }
  { Output~scheme~`#1~unknown. }
\msg_kernel_new:nnn { str } { to-native }
  { Use~\str_set_convert:Nnn to~convert~to~native~strings. }
\msg_kernel_new:nnn { str } { unicode-surrogate }
```

```
  { Code~point~#1~is~an~unpaired~surrogate. }
\msg_kernel_new:nnn { str } { utf16-surrogate }
  { UTF-16~unpaired~surrogate~#1. }
\msg_kernel_new:nnn { str } { utf16-odd }
  { UTF-16~dangling~byte~#1. }
\msg_kernel_new:nnn { str } { utf8-extra-conti }
  { UTF-8~extra~continuation~byte~#1. }
\msg_kernel_new:nnn { str } { utf8-missing-conti }
  { UTF-8~missing~continuation~byte~#1. }
\msg_kernel_new:nnn { str } { utf8-invalid-byte }
  { Byte~#1~cannot~appear~in~UTF-8. }
\msg_kernel_new:nnn { str } { utf8-overlong }
  { Overlong~UTF-8~byte~sequence~for~code~point~#1. }
\msg_kernel_new:nnn { str } { utf8-premature-end }
  { Incomplete~last~UTF-8~character. }
\msg_kernel_new:nnn { str } { utf8-overflow }
  { Code~point~#1~too~large~(UTF-8). }
\msg_kernel_new:nnn { str } { utf32-overflow }
  { Code~point~too~large~(UTF-32~'#1'). }
\msg_kernel_new:nnn { str } { utf32-truncated }
  { Truncated~UTF-32~string~'...#1'. }
```

7.6 Deprecated string functions

`\str_length_skip_spaces:N`
`\str_length_skip_spaces:n`

The naming scheme is a little bit more consistent with "ignore_spaces" instead of "skip_spaces".

```
\cs_set:Npn \str_length_skip_spaces:N
  { \exp_args:No \str_length_skip_spaces:n }
\cs_set_eq:NN \str_length_skip_spaces:n \str_length_ignore_spaces:n
```

(*End definition for* `\str_length_skip_spaces:N` *and* `\str_length_skip_spaces:n`*. These functions are documented on page* **??**.)

⟨/initex | package⟩

7.7 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;
- `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.

7.7.1 Input methods

\str_convert_input_hex:
\str_convert_input_hexadecimal:
\str_convert_input_hex_aux:N
\str_convert_input_hex_aux_ii:N

Take chars two by two, and interpret each pair as the hexadecimal code for a byte. Anything else than hexadecimal digits is ignored. A string which contains 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.

```
⟨*hex⟩
\cs_new_protected_nopar:Npn \str_convert_input_hex:
  {
    \group_begin:
      \int_set:Nn \tex_escapechar:D { 92 }
      \tl_gset:Nx \g_str_result_tl
        {
          \str_output_byte:w "
            \exp_after:wN \str_convert_input_hex_aux:N
            \g_str_result_tl 0 { ? 0 - \c_one \prg_map_break: }
          \prg_break_point:n { \str_output_end: }
        }
    \group_end:
  }
\cs_new_nopar:Npn \str_convert_input_hex_aux:N #1
  {
    \use_none:n #1
    \str_aux_hexadecimal_use:NTF #1
      \str_convert_input_hex_aux_ii:N
      \str_convert_input_hex_aux:N
  }
\cs_new_nopar:Npn \str_convert_input_hex_aux_ii:N #1
  {
    \use_none:n #1
    \str_aux_hexadecimal_use:NTF #1
      {
        \str_output_end:
        \str_output_byte:w " \str_convert_input_hex_aux:N
      }
      \str_convert_input_hex_aux_ii:N
  }
⟨/hex⟩
```

(*End definition for* `\str_convert_input_hex:`*. This function is documented on page* ??.)

\str_convert_input_name:
\str_convert_input_name_aux:wNN
\str_convert_input_url:
\str_convert_input_url_aux:wNN

The `\str_convert_input_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_aux_hexadecimal_use:NTF` leaves the uppercase 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, and call the looping function followed by the two characters (remove \use_i:nnn).

```
⟨*name | url⟩
\cs_set_protected:Npn \str_tmp:w #1#2#3
  {
    \cs_new_protected_nopar:Npn #2
      {
        \group_begin:
          \int_set:Nn \tex_escapechar:D { 92 }
          \tl_gset:Nx \g_str_result_tl
            {
              \exp_after:wN #3 \g_str_result_tl
                #1 ? { ? \prg_map_break: }
              \prg_break_point:n { }
            }
        \group_end:
      }
    \cs_new_nopar:Npn #3 ##1#1##2##3
      {
        \str_filter_bytes:n {##1}
        \use_none:n ##3
        \str_output_byte:w "
          \str_aux_hexadecimal_use:NTF ##2
            {
              \str_aux_hexadecimal_use:NTF ##3
                { }
                { * \c_zero + ‘#1 \use_i:nn }
            }
            { 0 + ‘#1 \use_i:nn }
        \str_output_end:
        \use_i:nnn #3 ##2##3
      }
  }
⟨/name | url⟩
⟨name⟩\exp_after:wN \str_tmp:w \c_hash_str
⟨name⟩  \str_convert_input_name: \str_convert_input_name_aux:wNN
⟨url⟩\exp_after:wN \str_tmp:w \c_percent_str
⟨url⟩  \str_convert_input_url: \str_convert_input_url_aux:wNN
```

(*End definition for* \str_convert_input_name:. *This function is documented on page* ??.)

\str_convert_input_string:
\str_convert_input_string_aux:wN
\str_convert_input_string_aux:wNNN
\str_convert_input_string_aux:NNNNNN

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. 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.

```
⟨*string⟩
\group_begin:
  \char_set_lccode:nn {`\*} {`\\}
  \char_set_catcode_other:N \^^J
  \char_set_catcode_other:N \^^M
  \tl_to_lowercase:n
    {
      \cs_new_protected_nopar:Npn \str_convert_input_string:
        {
          \group_begin:
            \int_set:Nn \tex_escapechar:D { 92 }
            \tl_gset:Nx \g_str_result_tl
              {
                \exp_after:wN \str_convert_input_string_aux:wN
                  \g_str_result_tl \prg_map_break: ^^M ?
                \prg_break_point:n { }
              }
            \tl_gset:Nx \g_str_result_tl
              {
                \exp_after:wN \str_convert_input_string_aux:wNNN
                  \g_str_result_tl * ?? { ? \prg_map_break: }
                \prg_break_point:n { }
              }
          \group_end:
        }
      \cs_new_nopar:Npn \str_convert_input_string_aux:wNNN #1 *#2#3#4
    }
        {
          \str_filter_bytes:n {#1}
          \use_none:n #4
          \str_output_byte:w ’
            \str_aux_octal_use:NTF #2
              {
                \str_aux_octal_use:NTF #3
```

```
                {
                  \str_aux_octal_use:NTF #4
                    {
                      \if_int_compare:w #2 > \c_three
                        - 256
                      \fi:
                      \str_convert_input_string_aux:NNNNNN
                    }
                    { \str_convert_input_string_aux:NNNNNN ? }
                }
                { \str_convert_input_string_aux:NNNNNN ?? }
            }
            {
              \prg_case_str:xxn {#2}
                {
                  { \c_backslash_str } { 134 }
                  { ( } { 50 }
                  { ) } { 51 }
                  { r } { 15 }
                  { f } { 14 }
                  { n } { 12 }
                  { t } { 11 }
                  { b } { 10 }
                  { ^^J } { 0 - \c_one }
                }
                { 0 - \c_one \use_i:nn }
            }
          \str_output_end:
          \use_i:nn \str_convert_input_string_aux:wNNN #2#3#4
        }
    \cs_new_nopar:Npn \str_convert_input_string_aux:NNNNNN #1#2#3#4#5#6
      { \str_output_end: \str_convert_input_string_aux:wNNN }
    \cs_new_nopar:Npn \str_convert_input_string_aux:wN #1 ^^M #2
      {
        #1 ^^J
        \if_charcode:w ^^J #2
          \exp_after:wN \use_i:nn
        \fi:
        \str_convert_input_string_aux:wN #2
      }
  \group_end:
⟨/string⟩
```

(*End definition for* `\str_convert_input_string:`*. This function is documented on page* **??**.)

7.7.2 Output methods

`\str_convert_output_hex:` Loop and convert each byte to hexadecimal.

`\str_convert_output_hexadecimal:`

`\str_convert_output_hex_aux:N`

```
⟨*hex⟩
\cs_new_protected_nopar:Npn \str_convert_output_hex:
```

```
  { \str_aux_gmap_result:N \str_convert_output_hex_aux:N }
\cs_new_nopar:Npn \str_convert_output_hex_aux:N #1
  { \str_output_hexadecimal:n { `#1 } }
⟨/hex⟩
```

(*End definition for* `\str_convert_output_hex:`*. This function is documented on page* **??**.)

`\str_convert_output_name:`
`\str_convert_output_name_aux:N`
`\str_convert_output_name_aux:NTF`
`\c_str_convert_output_name_str`
`\c_str_convert_output_name_not_str`

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: characters in `\c_str_convert_output_name_not_str` are not hash-encoded, and characters in the `\c_str_convert_output_name_str` are encoded.

```
⟨*name⟩
\str_const:Nn \c_str_convert_output_name_not_str { ! " $ & ' } %$
\str_const:Nn \c_str_convert_output_name_str { {}/<>[] }
\cs_new_protected_nopar:Npn \str_convert_output_name:
  { \str_aux_gmap_result:N \str_convert_output_name_aux:N }
\cs_new_nopar:Npn \str_convert_output_name_aux:N #1
  {
    \str_convert_output_name_aux:NTF #1 {#1}
      { \c_hash_str \str_output_hexadecimal:n {`#1} }
  }
\prg_new_conditional:Npnn \str_convert_output_name_aux:N #1 { TF }
  {
    \if_num:w `#1 < "2A \exp_stop_f:
      \str_if_contains_char:NNTF \c_str_convert_output_name_not_str #1
        \prg_return_true: \prg_return_false:
    \else:
      \if_num:w `#1 > "7E \exp_stop_f:
        \prg_return_false:
      \else:
        \str_if_contains_char:NNTF \c_str_convert_output_name_str #1
          \prg_return_false: \prg_return_true:
      \fi:
    \fi:
  }
⟨/name⟩
```

(*End definition for* `\str_convert_output_name:`*. This function is documented on page* **??**.)

`\str_convert_output_string:`
`\str_convert_output_string_aux:N`
`\str_convert_output_string_aux:NTF`
`\c_str_convert_output_string_str`

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.

```
⟨*string⟩
\str_const:Nx \c_str_convert_output_string_str
  { \c_backslash_str ( ) }
\cs_new_protected_nopar:Npn \str_convert_output_string:
  { \str_aux_gmap_result:N \str_convert_output_string_aux:N }
\cs_new_nopar:Npn \str_convert_output_string_aux:N #1
  {
    \str_convert_output_string_aux:NTF #1
      {
        \str_if_contains_char:NNT
```

```
          \c_str_convert_output_string_str #1
          { \c_backslash_str }
        #1
      }
      {
        \c_backslash_str
        \int_div_truncate:nn {‘#1} {64}
        \int_mod:nn { \int_div_truncate:nn {‘#1} \c_eight } \c_eight
        \int_mod:nn {‘#1} \c_eight
      }
  }
\prg_new_conditional:Npnn \str_convert_output_string_aux:N #1 { TF }
  {
    \if_num:w ‘#1 < "21 \exp_stop_f:
      \prg_return_false:
    \else:
      \if_num:w ‘#1 > "7E \exp_stop_f:
        \prg_return_false:
      \else:
        \prg_return_true:
      \fi:
    \fi:
  }
⟨/string⟩
```

(*End definition for* `\str_convert_output_string:`*. This function is documented on page* **??**.)

`\str_convert_output_url:`
`\str_convert_output_url_aux:N`
`\str_convert_output_url_aux:NTF`

This function is similar to `\str_convert_output_name:`, escaping different characters.

```
⟨*url⟩
\cs_new_protected_nopar:Npn \str_convert_output_url:
  { \str_aux_gmap_result:N \str_convert_output_url_aux:N }
\cs_new_nopar:Npn \str_convert_output_url_aux:N #1
  {
    \str_convert_output_url_aux:NTF #1 {#1}
      { \c_percent_str \str_output_hexadecimal:n { ‘#1 } }
  }
\prg_new_conditional:Npnn \str_convert_output_url_aux:N #1 { TF }
  {
    \if_num:w ‘#1 < "41 \exp_stop_f:
      \str_if_contains_char:nNTF { "-.<> } #1
        \prg_return_true: \prg_return_false:
    \else:
      \if_num:w ‘#1 > "7E \exp_stop_f:
        \prg_return_false:
      \else:
        \str_if_contains_char:nNTF { [ ] } #1
          \prg_return_false: \prg_return_true:
      \fi:
    \fi:
  }
⟨/url⟩
```

(*End definition for* `\str_convert_output_url:`. *This function is documented on page* **??**.)

7.8 Encoding definition files

The `native` and `internal` encodings are automatically defined. Other encodings are loaded as needed. The following encodings are supported:

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

7.8.1 utf-8 support

```
⟨*utf8⟩
```

`\str_convert_to_utf8:`
`\str_aux_to_utf_viii:n`
`\str_aux_to_utf_viii:nnnnw`

Loop through the internal string, and convert each character to the UTF-8 representation. The details are messy.

```
\cs_new_protected_nopar:cpn { str_convert_to_utf8: }
  { \str_aux_gmap_internal_result:N \str_aux_to_utf_viii:n }
\cs_new:Npn \str_aux_to_utf_viii:n #1
  {
    \str_aux_to_utf_viii:nnnnw
      {#1} { \c_minus_one + \c_zero * \use_none:n }
      { 128 } { \c_zero }
      {  32 } {     192 }
      {  16 } {     224 }
      {   8 } {     240 }
    \q_stop
  }
\cs_new_nopar:Npn \str_aux_to_utf_viii:nnnnw #1#2#3#4 #5 \q_stop
  {
    \if_num:w #1 < #3 \exp_stop_f:
      \str_output_byte:n { #1 + #4 }
    \else:
      \exp_args:Nf \str_aux_to_utf_viii:nnnnw
        { \int_div_truncate:nn {#1} {64} }
        {#1}
        #5 \q_stop
    \fi:
    \str_output_byte:n { #2 - 64 * ( #1 - \c_two ) }
  }
```

(*End definition for* `\str_convert_to_utf8:`. *This function is documented on page* **??**.)

\str_convert_from_utf8:
\str_aux_from_utf_viii:N
\str_aux_from_utf_viii:wwN
\str_aux_from_utf_viii:wNnnwN
\str_aux_from_utf_viii_overflow:w
\str_aux_from_utf_viii_error:

In the UTF-8 encoding, bytes in the range [0, "7F] stand for themselves, bytes in the range ["80, "BF] are continuation bytes, and larger bytes must be followed by a number of continuation bytes. Documentation to come.

```
\cs_new_protected_nopar:cpn { str_convert_from_utf8: }
  {
    \tl_gset:Nx \g_str_result_tl
      {
        \exp_after:wN \str_aux_from_utf_viii:N \g_str_result_tl
          { \prg_map_break: \str_aux_from_utf_viii_error: }
        \prg_break_point:n { }
      }
  }
\cs_new_nopar:Npn \str_aux_from_utf_viii:N #1
  {
    #1
    \if_num:w `#1 < "C0 \exp_stop_f:
      \s_tl
      \if_num:w `#1 < "80 \exp_stop_f:
        \int_value:w `#1
      \else:
        \msg_expandable_kernel_error:nnn
          { str } { utf8-extra-conti } {#1}
        \int_use:N \c_str_replacement_char_int
      \fi:
    \else:
      \if_num:w `#1 < "F5 \exp_stop_f:
        \exp_after:wN \str_aux_from_utf_viii:wwN
          \int_value:w \int_eval:n { `#1 - "C0 }
      \else:
        \s_tl
        \msg_expandable_kernel_error:nnn
          { str } { utf8-invalid-byte } {#1}
        \int_use:N \c_str_replacement_char_int
      \fi:
    \fi:
    \s_tl
    \use_none_delimit_by_q_stop:w {"80} {"800} {"10000} {"110000} \q_stop
    \str_aux_from_utf_viii:N
  }
\cs_new_nopar:Npn \str_aux_from_utf_viii:wwN
    #1 \s_tl #2 \str_aux_from_utf_viii:N #3
  {
    \use_none:n #3
    \if_num:w \int_eval:w ( `#3 + "20 ) / "40 = \c_three
      #3
      \exp_after:wN \str_aux_from_utf_viii:wNnnwN
      \int_value:w \int_eval:n { #1 * "40 + `#3 - "80 }
    \else:
      \s_tl
```

```
      \msg_expandable_kernel_error:nnn
        { str } { utf8-missing-conti } {#3}
      \int_use:N \c_str_replacement_char_int
    \fi:
    \s_tl
    #2
    \str_aux_from_utf_viii:N #3
  }
\cs_new_nopar:Npn \str_aux_from_utf_viii:wNnnwN
    #1 \s_tl #2#3#4 #5 \str_aux_from_utf_viii:N #6
  {
    \if_num:w #1 < #4
      \s_tl
      \if_num:w #1 < #3 \exp_stop_f:
        \msg_expandable_kernel_error:nnn
          { str } { utf8-overlong } {#1}
        \int_use:N \c_str_replacement_char_int
      \else:
        #1
      \fi:
    \else:
      \if_meaning:w \q_stop #5
        \str_aux_from_utf_viii_overflow:w #1
      \fi:
      \exp_after:wN \str_aux_from_utf_viii:wwN
      \int_value:w \int_eval:n { #1 - #4 }
    \fi:
    \s_tl
    #2 {#4} #5
    \str_aux_from_utf_viii:N
  }
\cs_new_nopar:Npn \str_aux_from_utf_viii_overflow:w #1 \fi: #2 \fi:
  {
    \fi: \fi:
    \msg_expandable_kernel_error:nnn
      { str } { utf8-overflow } {#1}
    \int_use:N \c_str_replacement_char_int
  }
\cs_new_nopar:Npn \str_aux_from_utf_viii_error:
  {
    \s_tl
    \msg_expandable_kernel_error:nn
      { str } { utf8-premature-end }
    \int_use:N \c_str_replacement_char_int \s_tl
    \prg_map_break:
  }
```

(*End definition for* `\str_convert_from_utf8:`. *This function is documented on page* ??.)

```
⟨/utf8⟩
```

7.8.2 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 "other".

```
⟨*utf16⟩
\group_begin:
  \char_set_catcode_other:N \^^fe
  \char_set_catcode_other:N \^^ff
```

\str_convert_to_utf16:
\str_convert_to_utf16be:
\str_convert_to_utf16le:
\str_aux_to_utf_xvi:n
\str_aux_to_utf_xvi_be:n
\str_aux_to_utf_xvi_le:n

Convert characters one by one in a loop, with different behaviours depending on the character code:

- $[0, 55295]$ converted to two bytes;
- $[55296, 57343]$ cannot be converted, replaced by the replacement character;
- $[57344, 65535]$ converted to two bytes;
- $[65536, 1114111]$ converted to a pair of surrogates, each two bytes.

For the duration of this operation, `\str_tmp:w` is defined as a function to convert a number in the range $[0, 65536]$ to a pair of bytes (either big endian or little endian).

```
\cs_new_protected_nopar:cpx { str_convert_to_utf16: }
  {
    \exp_not:c { str_convert_to_utf16be: }
    \exp_not:n { \tl_gput_left:Nx \g_str_result_tl { ^^fe ^^ff } }
  }
\cs_new_protected_nopar:cpn { str_convert_to_utf16be: }
  {
    \cs_set_eq:NN \str_tmp:w \str_aux_to_utf_xvi_be:n
    \str_aux_gmap_internal_result:N \str_aux_to_utf_xvi:n
  }
\cs_new_protected_nopar:cpn { str_convert_to_utf16le: }
  {
    \cs_set_eq:NN \str_tmp:w \str_aux_to_utf_xvi_le:n
    \str_aux_gmap_internal_result:N \str_aux_to_utf_xvi:n
  }
\cs_new_nopar:Npn \str_aux_to_utf_xvi:n #1
  {
    \if_int_compare:w #1 < "D800 \exp_stop_f:
      \str_tmp:w {#1}
    \else:
      \if_int_compare:w #1 < "10000 \exp_stop_f:
        \if_int_compare:w #1 < "E000 \exp_stop_f:
          \msg_expandable_kernel_error:nnn
            { str } { unicode-surrogate } {#1}
          \str_tmp:w { \c_str_replacement_char_int }
        \else:
          \str_tmp:w {#1}
        \fi:
```

```
        \else:
          \exp_args:Nf \str_tmp:w { \int_div_truncate:nn {#1} {"400} + "D800 }
          \exp_args:Nf \str_tmp:w { \int_mod:nn {#1} {"400} + "DC00 }
        \fi:
      \fi:
    }
  \cs_new_nopar:Npn \str_aux_to_utf_xvi_be:n #1
    {
      \str_output_byte:n { \int_div_truncate:nn {#1} {"100} }
      \str_output_byte:n { \int_mod:nn {#1} {"100} }
    }
  \cs_new_nopar:Npn \str_aux_to_utf_xvi_le:n #1
    {
      \str_output_byte:n { \int_mod:nn {#1} {"100} }
      \str_output_byte:n { \int_div_truncate:nn {#1} {"100} }
    }
```

(*End definition for* `\str_convert_to_utf16:`*. This function is documented on page* **??**.)

`\str_convert_from_utf16:`
`\str_convert_from_utf16be:`
`\str_convert_from_utf16le:`
`\str_aux_from_utf_xvi_bom:NNw`
`\str_aux_from_utf_xvi:No`

Define `\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. If the endianness is not known, look for a byte order mark to decide.

```
  \cs_new_protected_nopar:cpn { str_convert_from_utf16be: }
    { \str_aux_from_utf_xvi:No 1 { \g_str_result_tl } }
  \cs_new_protected_nopar:cpn { str_convert_from_utf16le: }
    { \str_aux_from_utf_xvi:No 2 { \g_str_result_tl } }
  \cs_new_protected_nopar:cpn { str_convert_from_utf16: }
    {
      \exp_after:wN \str_aux_from_utf_xvi_bom:NNw
        \g_str_result_tl \scan_stop: \scan_stop: \scan_stop:
    }
  \cs_new_protected_nopar:Npn
      \str_aux_from_utf_xvi_bom:NNw #1#2#3\scan_stop:
    {
      \str_if_eq:nnTF { #1#2 } { ^^ff ^^fe }
        { \str_aux_from_utf_xvi:No 2 {#3} }
        {
          \str_if_eq:nnTF { #1#2 } { ^^fe ^^ff }
            { \str_aux_from_utf_xvi:No 1 {#3} }
            { \str_aux_from_utf_xvi:No 1 {#1#2#3} }
        }
    }
  \cs_new_protected_nopar:Npn \str_aux_from_utf_xvi:No #1#2
    {
      \cs_set_nopar:Npn \str_tmp:w ##1 ##2 { ‘ ## #1 }
      \tl_gset:Nx \g_str_result_tl
        {
          \exp_after:wN \str_aux_from_utf_xvi:NN
            #2 \scan_stop: \scan_stop:
          \prg_break_point:n { }
```

```
      }
    }
```

(*End definition for* `\str_convert_from_utf16:`, `\str_convert_from_utf16be:`, *and* `\str_convert_from_utf16le:`. *These functions are documented on page* ??.)

`\str_aux_from_utf_xvi:NN`
`\str_aux_from_utf_xvi:NNwNNN`
`\str_aux_from_utf_xvi_end:Nw`
`\str_aux_from_utf_xvi_error:NNw`
`\str_aux_from_utf_xvi_error:NNN`

When `\str_aux_from_utf_xvi:NN` is called, `\str_tmp:w` is such that `\str_tmp:w #1#2` expands to `‘#1` if the string is big-endian, and `‘#2` if the string is little-endian. If the code unit (a pair of characters) does not represent a surrogate, the `\if_case:w` construction expands to nothing, and we leave in the input stream ⟨*characters*⟩ `\s_tl` ⟨*char code*⟩ `\s_tl`, and move on to the next code unit. When the end is found, we check for the case of an odd-length string (an error) and end the mapping.

Surrogates are characterized by a most significant byte in the ranges ["D8, "DB] and ["DC, "DF], detected by dividing the character code by 4 (ε-TEX rounds to nearest, ties away from zero). Here, a trail surrogate is invalid, replaced by the replacement character. Lead surrogates are treated by `\str_aux_from_utf_xvi:NNwNNN`, which grabs the next two characters, checks for a premature end, and checks whether the code unit is a trailing surrogate. If it is, leave ⟨*characters*⟩ `\s_tl` ⟨*char code*⟩ `\s_tl` in the input stream and move on to the next code unit. Otherwise, raise an error, and continue parsing, replacing only the first code unit (the leading surrogate) by the replacement character.

```
  \cs_new:Npn \str_aux_from_utf_xvi:NN #1#2
    {
      \if_meaning:w \scan_stop: #2
        \str_aux_from_utf_xvi_end:Nw #1
      \fi:
      \if_case:w
        \int_eval:w ( \str_tmp:w #1#2 - "D6 ) / \c_four \int_eval_end:
      \or: \exp_after:wN \str_aux_from_utf_xvi:NNwNNN
      \or: \exp_after:wN \str_aux_from_utf_xvi_error:NNw
      \fi:
      #1#2 \s_tl
      \int_eval:n { "100 * \str_tmp:w #1#2 + \str_tmp:w #2#1 } \s_tl
      \str_aux_from_utf_xvi:NN
    }
  \cs_new:Npn \str_aux_from_utf_xvi:NNwNNN #1#2 \s_tl #3 \s_tl #4#5#6
    {
      \if_meaning:w \scan_stop: #6
        \str_aux_from_utf_xvi_error:NNN #1#2#5
      \fi:
      \if_num:w \int_eval:w ( \str_tmp:w #5#6 - "DA ) / \c_four = \c_one
        #1 #2 #5 #6 \s_tl
        \int_eval:n
          {
            ( "100 * \str_tmp:w #1#2 + \str_tmp:w #2#1 - "D800 ) * "400
            + "100 * \str_tmp:w #4#5 + \str_tmp:w #5#4 - "DC00
          }
        \s_tl
        \exp_after:wN \use_i:nnn
      \else:
```

```
        \str_aux_from_utf_xvi_error:NNw #1#2 \s_tl \s_tl
      \fi:
      #4#5#6
    }
  \cs_new:Npn \str_aux_from_utf_xvi_end:Nw #1 \fi:
    {
      \fi:
      \if_meaning:w \scan_stop: #1
      \else:
        #1 \s_tl
        \msg_expandable_kernel_error:nnn
          { str } { utf16-odd } {#1}
        \int_use:N \c_str_replacement_char_int \s_tl
      \fi:
      \prg_map_break:
    }
  \cs_new:Npn \str_aux_from_utf_xvi_error:NNw #1#2 \s_tl #3 \s_tl
    {
      #1 #2 \s_tl
      \msg_expandable_kernel_error:nnn
        { str } { utf16-surrogate } { #1#2 }
      \int_use:N \c_str_replacement_char_int \s_tl
    }
  \cs_new:Npn \str_aux_from_utf_xvi_error:NNN #1#2#3
    {
      \if_meaning:w \scan_stop: #3
        #1#2 \s_tl
        \msg_expandable_kernel_error:nnn
          { str } { utf16-surrogate } { #1#2 }
      \else:
        #1#2#3 \s_tl
        \msg_expandable_kernel_error:nnn
          { str } { utf16-odd } { #1#2#3 }
      \fi:
      \int_use:N \c_str_replacement_char_int \s_tl
      \prg_map_break:
    }
```

(*End definition for* `\str_aux_from_utf_xvi:NN`*. This function is documented on page* **??**.)

Restore the original catcodes of bytes 254 and 255.

```
\group_end:
```

⟨/utf16⟩

7.8.3 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".

```
⟨*utf32⟩
\group_begin:
  \char_set_catcode_other:N \^^00
```

```
\char_set_catcode_other:N \^^fe
\char_set_catcode_other:N \^^ff
```

\str_convert_to_utf32:
\str_convert_to_utf32be:
\str_convert_to_utf32le:
\str_aux_to_utf_xxxii_be:n
\str_aux_to_utf_xxxii_le:n

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

```
\cs_new_protected_nopar:cpx { str_convert_to_utf32: }
  {
    \exp_not:c { str_convert_to_utf32be: }
    \exp_not:n { \tl_gput_left:Nx \g_str_result_tl { ^^00 ^^00 ^^fe ^^ff } }
  }
\cs_new_protected_nopar:cpn { str_convert_to_utf32be: }
  { \str_aux_gmap_internal_result:N \str_aux_to_utf_xxxii_be:n }
\cs_new_protected_nopar:cpn { str_convert_to_utf32le: }
  { \str_aux_gmap_internal_result:N \str_aux_to_utf_xxxii_le:n }
\cs_new:Npn \str_aux_to_utf_xxxii_be:n #1
  {
    ^^00
    \str_output_byte:n { \int_div_truncate:nn {#1} { "10000 } }
    \str_output_byte:n
      { \int_mod:nn { \int_div_truncate:nn {#1} {"100} } {"100} }
    \str_output_byte:n { \int_mod:nn {#1} {"100} }
  }
\cs_new:Npn \str_aux_to_utf_xxxii_le:n #1
  {
    \str_output_byte:n { \int_mod:nn {#1} {"100} }
    \str_output_byte:n
      { \int_mod:nn { \int_div_truncate:nn {#1} {"100} } {"100} }
    \str_output_byte:n { \int_div_truncate:nn {#1} { "10000 } }
    ^^00
  }
```

(*End definition for* \str_convert_to_utf32: , \str_convert_to_utf32be: , *and* \str_convert_to_utf32le: . *These functions are documented on page* **??**.)

\str_convert_from_utf32:
\str_convert_from_utf32be:
\str_convert_from_utf32le:
\str_aux_from_utf_xxxii_bom:NNNNw
\str_aux_from_utf_xxxii:No
\str_aux_from_utf_xxxii:NNNN
\str_aux_from_utf_xxxii_end:w

See the conversion functions from UTF-16 encodings. The \str_aux_from_utf_xxxii:No auxiliary defines \str_tmp:w to take two arguments and give the character code of either the first argument (big-endian) or the second argument (little-endian). Then we loop over the string, four bytes at a time, check for overflow, and otherwise produce the number corresponding to the four bytes with the given endianness.

```
\cs_new_protected_nopar:cpn { str_convert_from_utf32be: }
  { \str_aux_from_utf_xxxii:No 1 { \g_str_result_tl } }
\cs_new_protected_nopar:cpn { str_convert_from_utf32le: }
  { \str_aux_from_utf_xxxii:No 2 { \g_str_result_tl } }
\cs_new_protected_nopar:cpn { str_convert_from_utf32: }
  {
    \exp_after:wN \str_aux_from_utf_xxxii_bom:NNNNw \g_str_result_tl
      \scan_stop: \scan_stop: \scan_stop: \scan_stop: \scan_stop:
  }
\cs_new_protected_nopar:Npn
```

```
      \str_aux_from_utf_xxxii_bom:NNNNw #1#2#3#4#5\scan_stop:
    {
      \str_if_eq:nnTF { #1#2#3#4 } { ^^ff ^^fe ^^00 ^^00 }
        { \str_aux_from_utf_xxxii:No 2 {#5} }
        {
          \str_if_eq:nnTF { #1#2#3#4 } { ^^00 ^^00 ^^fe ^^ff }
            { \str_aux_from_utf_xxxii:No 1 {#5} }
            { \str_aux_from_utf_xxxii:No 1 {#1#2#3#4#5} }
        }
    }
  \cs_new_protected_nopar:Npn \str_aux_from_utf_xxxii:No #1#2
    {
      \cs_set_nopar:Npn \str_tmp:w ##1 ##2 { ‘ ## #1 }
      \tl_gset:Nx \g_str_result_tl
        {
          \exp_after:wN \str_aux_from_utf_xxxii:NNNN
            #2 \scan_stop: \scan_stop: \scan_stop: \scan_stop:
          \prg_break_point:n { }
        }
    }
  \cs_new:Npn \str_aux_from_utf_xxxii:NNNN #1#2#3#4
    {
      \if_meaning:w \scan_stop: #4
        \exp_after:wN \str_aux_from_utf_xxxii_end:w
      \fi:
      #1#2#3#4 \s_tl
      \if_num:w \int_eval:w
                  \str_tmp:w #1#4 * \c_two_hundred_fifty_six
                  + \str_tmp:w #2#3
                > \c_sixteen
        \msg_expandable_kernel_error:nnn
          { str } { utf32-overflow } { #1#2#3#4 }
        \int_use:N \c_str_replacement_char_int
      \else:
        \int_eval:n
          { \str_tmp:w #2#3*"10000 + \str_tmp:w #3#2*"100 + \str_tmp:w #4#1 }
      \fi:
      \s_tl
      \str_aux_from_utf_xxxii:NNNN
    }
  \cs_new:Npn \str_aux_from_utf_xxxii_end:w #1 \scan_stop:
    {
      \tl_if_empty:nF {#1}
        {
          \msg_expandable_kernel_error:nnn
            { str } { utf32-truncated } { #1 }
          #1 \s_tl
          \int_use:N \c_str_replacement_char_int \s_tl
        }
      \prg_map_break:
```

```
    }
```

(*End definition for* `\str_convert_from_utf32:` , `\str_convert_from_utf32be:` , *and* `\str_convert_from_utf32le:` . *These functions are documented on page* ??.)

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

```
\group_end:
```
⟨/utf32⟩

7.8.4 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.

⟨*iso88591⟩
```
\str_encoding_eight_bit:n { iso88591 }
\tl_const:cn { c_str_encoding_iso88591_tl }
  {
  }
\tl_const:cn { c_str_encoding_iso88591_missing_tl }
  {
  }
```
⟨/iso88591⟩

⟨*iso88592⟩
```
\str_encoding_eight_bit:n { iso88592 }
\tl_const:cn { c_str_encoding_iso88592_tl }
  {
    { A1 } { 0104 }
    { A2 } { 02D8 }
    { A3 } { 0141 }
    { A5 } { 013D }
    { A6 } { 015A }
    { A9 } { 0160 }
    { AA } { 015E }
    { AB } { 0164 }
    { AC } { 0179 }
    { AE } { 017D }
    { AF } { 017B }
    { B1 } { 0105 }
    { B2 } { 02DB }
    { B3 } { 0142 }
    { B5 } { 013E }
    { B6 } { 015B }
    { B7 } { 02C7 }
    { B9 } { 0161 }
    { BA } { 015F }
    { BB } { 0165 }
    { BC } { 017A }
    { BD } { 02DD }
    { BE } { 017E }
```

```
    { BF } { 017C }
    { C0 } { 0154 }
    { C3 } { 0102 }
    { C5 } { 0139 }
    { C6 } { 0106 }
    { C8 } { 010C }
    { CA } { 0118 }
    { CC } { 011A }
    { CF } { 010E }
    { D0 } { 0110 }
    { D1 } { 0143 }
    { D2 } { 0147 }
    { D5 } { 0150 }
    { D8 } { 0158 }
    { D9 } { 016E }
    { DB } { 0170 }
    { DE } { 0162 }
    { E0 } { 0155 }
    { E3 } { 0103 }
    { E5 } { 013A }
    { E6 } { 0107 }
    { E8 } { 010D }
    { EA } { 0119 }
    { EC } { 011B }
    { EF } { 010F }
    { F0 } { 0111 }
    { F1 } { 0144 }
    { F2 } { 0148 }
    { F5 } { 0151 }
    { F8 } { 0159 }
    { F9 } { 016F }
    { FB } { 0171 }
    { FE } { 0163 }
    { FF } { 02D9 }
  }
\tl_const:cn { c_str_encoding_iso88592_missing_tl }
  {
  }
```
⟨/iso88592⟩

⟨*iso88593⟩

```
\str_encoding_eight_bit:n { iso88593 }
\tl_const:cn { c_str_encoding_iso88593_tl }
  {
    { A1 } { 0126 }
    { A2 } { 02D8 }
    { A6 } { 0124 }
    { A9 } { 0130 }
    { AA } { 015E }
    { AB } { 011E }
```

```
    { AC } { 0134 }
    { AF } { 017B }
    { B1 } { 0127 }
    { B6 } { 0125 }
    { B9 } { 0131 }
    { BA } { 015F }
    { BB } { 011F }
    { BC } { 0135 }
    { BF } { 017C }
    { C5 } { 010A }
    { C6 } { 0108 }
    { D5 } { 0120 }
    { D8 } { 011C }
    { DD } { 016C }
    { DE } { 015C }
    { E5 } { 010B }
    { E6 } { 0109 }
    { F5 } { 0121 }
    { F8 } { 011D }
    { FD } { 016D }
    { FE } { 015D }
    { FF } { 02D9 }
  }
\tl_const:cn { c_str_encoding_iso88593_missing_tl }
  {
    { A5 }
    { AE }
    { BE }
    { C3 }
    { D0 }
    { E3 }
    { F0 }
  }
```

⟨/iso88593⟩

⟨*iso88594⟩

```
\str_encoding_eight_bit:n { iso88594 }
\tl_const:cn { c_str_encoding_iso88594_tl }
  {
    { A1 } { 0104 }
    { A2 } { 0138 }
    { A3 } { 0156 }
    { A5 } { 0128 }
    { A6 } { 013B }
    { A9 } { 0160 }
    { AA } { 0112 }
    { AB } { 0122 }
    { AC } { 0166 }
    { AE } { 017D }
    { B1 } { 0105 }
```

```
    { B2 } { 02DB }
    { B3 } { 0157 }
    { B5 } { 0129 }
    { B6 } { 013C }
    { B7 } { 02C7 }
    { B9 } { 0161 }
    { BA } { 0113 }
    { BB } { 0123 }
    { BC } { 0167 }
    { BD } { 014A }
    { BE } { 017E }
    { BF } { 014B }
    { C0 } { 0100 }
    { C7 } { 012E }
    { C8 } { 010C }
    { CA } { 0118 }
    { CC } { 0116 }
    { CF } { 012A }
    { D0 } { 0110 }
    { D1 } { 0145 }
    { D2 } { 014C }
    { D3 } { 0136 }
    { D9 } { 0172 }
    { DD } { 0168 }
    { DE } { 016A }
    { E0 } { 0101 }
    { E7 } { 012F }
    { E8 } { 010D }
    { EA } { 0119 }
    { EC } { 0117 }
    { EF } { 012B }
    { F0 } { 0111 }
    { F1 } { 0146 }
    { F2 } { 014D }
    { F3 } { 0137 }
    { F9 } { 0173 }
    { FD } { 0169 }
    { FE } { 016B }
    { FF } { 02D9 }
  }
\tl_const:cn { c_str_encoding_iso88594_missing_tl }
  {
  }
⟨/iso88594⟩
⟨*iso88595⟩
\str_encoding_eight_bit:n { iso88595 }
\tl_const:cn { c_str_encoding_iso88595_tl }
  {
    { A1 } { 0401 }
```

```
    { A2 } { 0402 }
    { A3 } { 0403 }
    { A4 } { 0404 }
    { A5 } { 0405 }
    { A6 } { 0406 }
    { A7 } { 0407 }
    { A8 } { 0408 }
    { A9 } { 0409 }
    { AA } { 040A }
    { AB } { 040B }
    { AC } { 040C }
    { AE } { 040E }
    { AF } { 040F }
    { B0 } { 0410 }
    { B1 } { 0411 }
    { B2 } { 0412 }
    { B3 } { 0413 }
    { B4 } { 0414 }
    { B5 } { 0415 }
    { B6 } { 0416 }
    { B7 } { 0417 }
    { B8 } { 0418 }
    { B9 } { 0419 }
    { BA } { 041A }
    { BB } { 041B }
    { BC } { 041C }
    { BD } { 041D }
    { BE } { 041E }
    { BF } { 041F }
    { C0 } { 0420 }
    { C1 } { 0421 }
    { C2 } { 0422 }
    { C3 } { 0423 }
    { C4 } { 0424 }
    { C5 } { 0425 }
    { C6 } { 0426 }
    { C7 } { 0427 }
    { C8 } { 0428 }
    { C9 } { 0429 }
    { CA } { 042A }
    { CB } { 042B }
    { CC } { 042C }
    { CD } { 042D }
    { CE } { 042E }
    { CF } { 042F }
    { D0 } { 0430 }
    { D1 } { 0431 }
    { D2 } { 0432 }
    { D3 } { 0433 }
    { D4 } { 0434 }
```

```
    { D5 } { 0435 }
    { D6 } { 0436 }
    { D7 } { 0437 }
    { D8 } { 0438 }
    { D9 } { 0439 }
    { DA } { 043A }
    { DB } { 043B }
    { DC } { 043C }
    { DD } { 043D }
    { DE } { 043E }
    { DF } { 043F }
    { E0 } { 0440 }
    { E1 } { 0441 }
    { E2 } { 0442 }
    { E3 } { 0443 }
    { E4 } { 0444 }
    { E5 } { 0445 }
    { E6 } { 0446 }
    { E7 } { 0447 }
    { E8 } { 0448 }
    { E9 } { 0449 }
    { EA } { 044A }
    { EB } { 044B }
    { EC } { 044C }
    { ED } { 044D }
    { EE } { 044E }
    { EF } { 044F }
    { F0 } { 2116 }
    { F1 } { 0451 }
    { F2 } { 0452 }
    { F3 } { 0453 }
    { F4 } { 0454 }
    { F5 } { 0455 }
    { F6 } { 0456 }
    { F7 } { 0457 }
    { F8 } { 0458 }
    { F9 } { 0459 }
    { FA } { 045A }
    { FB } { 045B }
    { FC } { 045C }
    { FD } { 00A7 }
    { FE } { 045E }
    { FF } { 045F }
  }
\tl_const:cn { c_str_encoding_iso88595_missing_tl }
  {
  }
```

⟨/iso88595⟩

⟨*iso88596⟩

```
\str_encoding_eight_bit:n { iso88596 }
\tl_const:cn { c_str_encoding_iso88596_tl }
  {
    { AC } { 060C }
    { BB } { 061B }
    { BF } { 061F }
    { C1 } { 0621 }
    { C2 } { 0622 }
    { C3 } { 0623 }
    { C4 } { 0624 }
    { C5 } { 0625 }
    { C6 } { 0626 }
    { C7 } { 0627 }
    { C8 } { 0628 }
    { C9 } { 0629 }
    { CA } { 062A }
    { CB } { 062B }
    { CC } { 062C }
    { CD } { 062D }
    { CE } { 062E }
    { CF } { 062F }
    { D0 } { 0630 }
    { D1 } { 0631 }
    { D2 } { 0632 }
    { D3 } { 0633 }
    { D4 } { 0634 }
    { D5 } { 0635 }
    { D6 } { 0636 }
    { D7 } { 0637 }
    { D8 } { 0638 }
    { D9 } { 0639 }
    { DA } { 063A }
    { E0 } { 0640 }
    { E1 } { 0641 }
    { E2 } { 0642 }
    { E3 } { 0643 }
    { E4 } { 0644 }
    { E5 } { 0645 }
    { E6 } { 0646 }
    { E7 } { 0647 }
    { E8 } { 0648 }
    { E9 } { 0649 }
    { EA } { 064A }
    { EB } { 064B }
    { EC } { 064C }
    { ED } { 064D }
    { EE } { 064E }
    { EF } { 064F }
    { F0 } { 0650 }
    { F1 } { 0651 }
```

```
    { F2 } { 0652 }
  }
\tl_const:cn { c_str_encoding_iso88596_missing_tl }
  {
    { A1 }
    { A2 }
    { A3 }
    { A5 }
    { A6 }
    { A7 }
    { A8 }
    { A9 }
    { AA }
    { AB }
    { AE }
    { AF }
    { B0 }
    { B1 }
    { B2 }
    { B3 }
    { B4 }
    { B5 }
    { B6 }
    { B7 }
    { B8 }
    { B9 }
    { BA }
    { BC }
    { BD }
    { BE }
    { C0 }
    { DB }
    { DC }
    { DD }
    { DE }
    { DF }
  }
⟨/iso88596⟩
⟨*iso88597⟩
\str_encoding_eight_bit:n { iso88597 }
\tl_const:cn { c_str_encoding_iso88597_tl }
  {
    { A1 } { 2018 }
    { A2 } { 2019 }
    { A4 } { 20AC }
    { A5 } { 20AF }
    { AA } { 037A }
    { AF } { 2015 }
    { B4 } { 0384 }
```

```
{ B5 } { 0385 }
{ B6 } { 0386 }
{ B8 } { 0388 }
{ B9 } { 0389 }
{ BA } { 038A }
{ BC } { 038C }
{ BE } { 038E }
{ BF } { 038F }
{ C0 } { 0390 }
{ C1 } { 0391 }
{ C2 } { 0392 }
{ C3 } { 0393 }
{ C4 } { 0394 }
{ C5 } { 0395 }
{ C6 } { 0396 }
{ C7 } { 0397 }
{ C8 } { 0398 }
{ C9 } { 0399 }
{ CA } { 039A }
{ CB } { 039B }
{ CC } { 039C }
{ CD } { 039D }
{ CE } { 039E }
{ CF } { 039F }
{ D0 } { 03A0 }
{ D1 } { 03A1 }
{ D3 } { 03A3 }
{ D4 } { 03A4 }
{ D5 } { 03A5 }
{ D6 } { 03A6 }
{ D7 } { 03A7 }
{ D8 } { 03A8 }
{ D9 } { 03A9 }
{ DA } { 03AA }
{ DB } { 03AB }
{ DC } { 03AC }
{ DD } { 03AD }
{ DE } { 03AE }
{ DF } { 03AF }
{ E0 } { 03B0 }
{ E1 } { 03B1 }
{ E2 } { 03B2 }
{ E3 } { 03B3 }
{ E4 } { 03B4 }
{ E5 } { 03B5 }
{ E6 } { 03B6 }
{ E7 } { 03B7 }
{ E8 } { 03B8 }
{ E9 } { 03B9 }
{ EA } { 03BA }
```

```
    { EB } { 03BB }
    { EC } { 03BC }
    { ED } { 03BD }
    { EE } { 03BE }
    { EF } { 03BF }
    { F0 } { 03C0 }
    { F1 } { 03C1 }
    { F2 } { 03C2 }
    { F3 } { 03C3 }
    { F4 } { 03C4 }
    { F5 } { 03C5 }
    { F6 } { 03C6 }
    { F7 } { 03C7 }
    { F8 } { 03C8 }
    { F9 } { 03C9 }
    { FA } { 03CA }
    { FB } { 03CB }
    { FC } { 03CC }
    { FD } { 03CD }
    { FE } { 03CE }
  }
\tl_const:cn { c_str_encoding_iso88597_missing_tl }
  {
    { AE }
    { D2 }
  }
⟨/iso88597⟩
⟨*iso88598⟩
\str_encoding_eight_bit:n { iso88598 }
\tl_const:cn { c_str_encoding_iso88598_tl }
  {
    { AA } { 00D7 }
    { BA } { 00F7 }
    { DF } { 2017 }
    { E0 } { 05D0 }
    { E1 } { 05D1 }
    { E2 } { 05D2 }
    { E3 } { 05D3 }
    { E4 } { 05D4 }
    { E5 } { 05D5 }
    { E6 } { 05D6 }
    { E7 } { 05D7 }
    { E8 } { 05D8 }
    { E9 } { 05D9 }
    { EA } { 05DA }
    { EB } { 05DB }
    { EC } { 05DC }
    { ED } { 05DD }
    { EE } { 05DE }
```

```
    { EF } { 05DF }
    { F0 } { 05E0 }
    { F1 } { 05E1 }
    { F2 } { 05E2 }
    { F3 } { 05E3 }
    { F4 } { 05E4 }
    { F5 } { 05E5 }
    { F6 } { 05E6 }
    { F7 } { 05E7 }
    { F8 } { 05E8 }
    { F9 } { 05E9 }
    { FA } { 05EA }
    { FD } { 200E }
    { FE } { 200F }
  }
\tl_const:cn { c_str_encoding_iso88598_missing_tl }
  {
    { A1 }
    { BF }
    { C0 }
    { C1 }
    { C2 }
    { C3 }
    { C4 }
    { C5 }
    { C6 }
    { C7 }
    { C8 }
    { C9 }
    { CA }
    { CB }
    { CC }
    { CD }
    { CE }
    { CF }
    { D0 }
    { D1 }
    { D2 }
    { D3 }
    { D4 }
    { D5 }
    { D6 }
    { D7 }
    { D8 }
    { D9 }
    { DA }
    { DB }
    { DC }
    { DD }
    { DE }
```

```
    { FB }
    { FC }
  }
⟨/iso88598⟩
⟨*iso88599⟩
\str_encoding_eight_bit:n { iso88599 }
\tl_const:cn { c_str_encoding_iso88599_tl }
  {
    { D0 } { 011E }
    { DD } { 0130 }
    { DE } { 015E }
    { F0 } { 011F }
    { FD } { 0131 }
    { FE } { 015F }
  }
\tl_const:cn { c_str_encoding_iso88599_missing_tl }
  {
  }
⟨/iso88599⟩
⟨*iso885910⟩
\str_encoding_eight_bit:n { iso885910 }
\tl_const:cn { c_str_encoding_iso885910_tl }
  {
    { A1 } { 0104 }
    { A2 } { 0112 }
    { A3 } { 0122 }
    { A4 } { 012A }
    { A5 } { 0128 }
    { A6 } { 0136 }
    { A8 } { 013B }
    { A9 } { 0110 }
    { AA } { 0160 }
    { AB } { 0166 }
    { AC } { 017D }
    { AE } { 016A }
    { AF } { 014A }
    { B1 } { 0105 }
    { B2 } { 0113 }
    { B3 } { 0123 }
    { B4 } { 012B }
    { B5 } { 0129 }
    { B6 } { 0137 }
    { B8 } { 013C }
    { B9 } { 0111 }
    { BA } { 0161 }
    { BB } { 0167 }
    { BC } { 017E }
    { BD } { 2015 }
    { BE } { 016B }
```

```
    { BF } { 014B }
    { C0 } { 0100 }
    { C7 } { 012E }
    { C8 } { 010C }
    { CA } { 0118 }
    { CC } { 0116 }
    { D1 } { 0145 }
    { D2 } { 014C }
    { D7 } { 0168 }
    { D9 } { 0172 }
    { E0 } { 0101 }
    { E7 } { 012F }
    { E8 } { 010D }
    { EA } { 0119 }
    { EC } { 0117 }
    { F1 } { 0146 }
    { F2 } { 014D }
    { F7 } { 0169 }
    { F9 } { 0173 }
    { FF } { 0138 }
  }
\tl_const:cn { c_str_encoding_iso885910_missing_tl }
  {
  }
```
⟨/iso885910⟩

⟨*iso885911⟩

```
\str_encoding_eight_bit:n { iso885911 }
\tl_const:cn { c_str_encoding_iso885911_tl }
  {
    { A1 } { 0E01 }
    { A2 } { 0E02 }
    { A3 } { 0E03 }
    { A4 } { 0E04 }
    { A5 } { 0E05 }
    { A6 } { 0E06 }
    { A7 } { 0E07 }
    { A8 } { 0E08 }
    { A9 } { 0E09 }
    { AA } { 0E0A }
    { AB } { 0E0B }
    { AC } { 0E0C }
    { AD } { 0E0D }
    { AE } { 0E0E }
    { AF } { 0E0F }
    { B0 } { 0E10 }
    { B1 } { 0E11 }
    { B2 } { 0E12 }
    { B3 } { 0E13 }
    { B4 } { 0E14 }
```

```
{ B5 } { 0E15 }
{ B6 } { 0E16 }
{ B7 } { 0E17 }
{ B8 } { 0E18 }
{ B9 } { 0E19 }
{ BA } { 0E1A }
{ BB } { 0E1B }
{ BC } { 0E1C }
{ BD } { 0E1D }
{ BE } { 0E1E }
{ BF } { 0E1F }
{ C0 } { 0E20 }
{ C1 } { 0E21 }
{ C2 } { 0E22 }
{ C3 } { 0E23 }
{ C4 } { 0E24 }
{ C5 } { 0E25 }
{ C6 } { 0E26 }
{ C7 } { 0E27 }
{ C8 } { 0E28 }
{ C9 } { 0E29 }
{ CA } { 0E2A }
{ CB } { 0E2B }
{ CC } { 0E2C }
{ CD } { 0E2D }
{ CE } { 0E2E }
{ CF } { 0E2F }
{ D0 } { 0E30 }
{ D1 } { 0E31 }
{ D2 } { 0E32 }
{ D3 } { 0E33 }
{ D4 } { 0E34 }
{ D5 } { 0E35 }
{ D6 } { 0E36 }
{ D7 } { 0E37 }
{ D8 } { 0E38 }
{ D9 } { 0E39 }
{ DA } { 0E3A }
{ DF } { 0E3F }
{ E0 } { 0E40 }
{ E1 } { 0E41 }
{ E2 } { 0E42 }
{ E3 } { 0E43 }
{ E4 } { 0E44 }
{ E5 } { 0E45 }
{ E6 } { 0E46 }
{ E7 } { 0E47 }
{ E8 } { 0E48 }
{ E9 } { 0E49 }
{ EA } { 0E4A }
```

```
    { EB } { 0E4B }
    { EC } { 0E4C }
    { ED } { 0E4D }
    { EE } { 0E4E }
    { EF } { 0E4F }
    { F0 } { 0E50 }
    { F1 } { 0E51 }
    { F2 } { 0E52 }
    { F3 } { 0E53 }
    { F4 } { 0E54 }
    { F5 } { 0E55 }
    { F6 } { 0E56 }
    { F7 } { 0E57 }
    { F8 } { 0E58 }
    { F9 } { 0E59 }
    { FA } { 0E5A }
    { FB } { 0E5B }
  }
\tl_const:cn { c_str_encoding_iso885911_missing_tl }
  {
    { DB }
    { DC }
    { DD }
    { DE }
  }
⟨/iso885911⟩
⟨*iso885913⟩
\str_encoding_eight_bit:n { iso885913 }
\tl_const:cn { c_str_encoding_iso885913_tl }
  {
    { A1 } { 201D }
    { A5 } { 201E }
    { A8 } { 00D8 }
    { AA } { 0156 }
    { AF } { 00C6 }
    { B4 } { 201C }
    { B8 } { 00F8 }
    { BA } { 0157 }
    { BF } { 00E6 }
    { C0 } { 0104 }
    { C1 } { 012E }
    { C2 } { 0100 }
    { C3 } { 0106 }
    { C6 } { 0118 }
    { C7 } { 0112 }
    { C8 } { 010C }
    { CA } { 0179 }
    { CB } { 0116 }
    { CC } { 0122 }
```

```
    { CD } { 0136 }
    { CE } { 012A }
    { CF } { 013B }
    { D0 } { 0160 }
    { D1 } { 0143 }
    { D2 } { 0145 }
    { D4 } { 014C }
    { D8 } { 0172 }
    { D9 } { 0141 }
    { DA } { 015A }
    { DB } { 016A }
    { DD } { 017B }
    { DE } { 017D }
    { E0 } { 0105 }
    { E1 } { 012F }
    { E2 } { 0101 }
    { E3 } { 0107 }
    { E6 } { 0119 }
    { E7 } { 0113 }
    { E8 } { 010D }
    { EA } { 017A }
    { EB } { 0117 }
    { EC } { 0123 }
    { ED } { 0137 }
    { EE } { 012B }
    { EF } { 013C }
    { F0 } { 0161 }
    { F1 } { 0144 }
    { F2 } { 0146 }
    { F4 } { 014D }
    { F8 } { 0173 }
    { F9 } { 0142 }
    { FA } { 015B }
    { FB } { 016B }
    { FD } { 017C }
    { FE } { 017E }
    { FF } { 2019 }
  }
\tl_const:cn { c_str_encoding_iso885913_missing_tl }
  {
  }
⟨/iso885913⟩
⟨*iso885914⟩
\str_encoding_eight_bit:n { iso885914 }
\tl_const:cn { c_str_encoding_iso885914_tl }
  {
    { A1 } { 1E02 }
    { A2 } { 1E03 }
    { A4 } { 010A }
```

```
    { A5 } { 010B }
    { A6 } { 1E0A }
    { A8 } { 1E80 }
    { AA } { 1E82 }
    { AB } { 1E0B }
    { AC } { 1EF2 }
    { AF } { 0178 }
    { B0 } { 1E1E }
    { B1 } { 1E1F }
    { B2 } { 0120 }
    { B3 } { 0121 }
    { B4 } { 1E40 }
    { B5 } { 1E41 }
    { B7 } { 1E56 }
    { B8 } { 1E81 }
    { B9 } { 1E57 }
    { BA } { 1E83 }
    { BB } { 1E60 }
    { BC } { 1EF3 }
    { BD } { 1E84 }
    { BE } { 1E85 }
    { BF } { 1E61 }
    { D0 } { 0174 }
    { D7 } { 1E6A }
    { DE } { 0176 }
    { F0 } { 0175 }
    { F7 } { 1E6B }
    { FE } { 0177 }
  }
\tl_const:cn { c_str_encoding_iso885914_missing_tl }
  {
  }
```
⟨/iso885914⟩

⟨*iso885915⟩

```
\str_encoding_eight_bit:n { iso885915 }
\tl_const:cn { c_str_encoding_iso885915_tl }
  {
    { A4 } { 20AC }
    { A6 } { 0160 }
    { A8 } { 0161 }
    { B4 } { 017D }
    { B8 } { 017E }
    { BC } { 0152 }
    { BD } { 0153 }
    { BE } { 0178 }
  }
\tl_const:cn { c_str_encoding_iso885915_missing_tl }
  {
  }
```

```
⟨/iso885915⟩
⟨*iso885916⟩
\str_encoding_eight_bit:n { iso885916 }
\tl_const:cn { c_str_encoding_iso885916_tl }
  {
    { A1 } { 0104 }
    { A2 } { 0105 }
    { A3 } { 0141 }
    { A4 } { 20AC }
    { A5 } { 201E }
    { A6 } { 0160 }
    { A8 } { 0161 }
    { AA } { 0218 }
    { AC } { 0179 }
    { AE } { 017A }
    { AF } { 017B }
    { B2 } { 010C }
    { B3 } { 0142 }
    { B4 } { 017D }
    { B5 } { 201D }
    { B8 } { 017E }
    { B9 } { 010D }
    { BA } { 0219 }
    { BC } { 0152 }
    { BD } { 0153 }
    { BE } { 0178 }
    { BF } { 017C }
    { C3 } { 0102 }
    { C5 } { 0106 }
    { D0 } { 0110 }
    { D1 } { 0143 }
    { D5 } { 0150 }
    { D7 } { 015A }
    { D8 } { 0170 }
    { DD } { 0118 }
    { DE } { 021A }
    { E3 } { 0103 }
    { E5 } { 0107 }
    { F0 } { 0111 }
    { F1 } { 0144 }
    { F5 } { 0151 }
    { F7 } { 015B }
    { F8 } { 0171 }
    { FD } { 0119 }
    { FE } { 021B }
  }
\tl_const:cn { c_str_encoding_iso885916_missing_tl }
  {
  }
```

2457 ⟨/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.

E

T

U

X