# The l3str package: manipulating strings of characters*

The LaTeX3 Project†

Released 2011/11/19

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 "safe" 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 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 slightly faster than those which take care of escaping spaces appropriately;

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

## 0.1 Conversion and input of strings

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

Constant strings, containing a single character, with category code 12.

`\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.

*This file describes v2966, last revised 2011/11/19.

†E-mail: latex-team@latex-project.org

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

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

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

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

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

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

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

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

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and prepends the result to ⟨*str var*⟩, globally. 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_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_gput_right:Nn
\str_gput_right:(Nx|cn|cx)

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

Converts the ⟨*token list*⟩ to a ⟨*string*⟩, and appends the result to ⟨*str var*⟩, globally. 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*⟩, within the current TEX group level for the `input` variant and globally for the `ginput` version. 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 LuaTEX and XƎTEX.
- `\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).

## 0.2 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.

**TEXhackers note:** The `\str_length:n` of a given token list may depend on the category codes in effect when it is measured, and the value of the `\escapechar`: for instance `\str_length:n {\a}` may return 1, 2 or 3 depending on the escape character, and the category code of `a`.

\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` trims the first non-space character.

\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, the it is incremented by the length of the list. Both ⟨*start index*⟩ and ⟨*end index*⟩ count from 0 for the first (left most) character.

## 0.3 String conditionals

\str_if_eq_p:NN ⋆
\str_if_eq_p:(nn|Vn|on|no|nV|VV|xx) ⋆
\str_if_eq:NNTF ⋆
\str_if_eq:(nn|Vn|on|no|nV|VV|xx)TF ⋆

\str_if_eq_p:nn {⟨tl 1⟩} {⟨tl 2⟩}
\str_if_eq:nnTF {⟨tl 1⟩} {⟨tl 2⟩} {⟨true code⟩} {⟨false code⟩}

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).

\str_if_contains_char_p:NN ⋆
\str_if_contains_char_p:nN ⋆
\str_if_contains_char:NNTF ⋆
\str_if_contains_char:nNTF ⋆

\str_if_contains_char:nN {⟨*token list*⟩} ⟨*char*⟩

Converts the ⟨*token list*⟩ to a ⟨*string*⟩ and tests whether the ⟨*char*⟩ is present in the ⟨*string*⟩. The ⟨*char*⟩ can be given either directly, or as a one letter control sequence.

## 0.4 Byte conditionals

\str_if_bytes_p:N ⋆
\str_if_bytes:NTF ⋆

\str_if_bytes:NTF ⟨*str var*⟩ {⟨*true code*⟩} {⟨*false code*⟩}

Tests whether the ⟨*str var*⟩ only contains characters in the range 0 to 255.

## 0.5 Byte conversion

Byte conversions are necessary where strings are to be used in PDF string objects or within URLs. In both cases, only a restricted subset of characters are permitted.

\str_native_from_UTF_viii:NN

\str_native_from_UTF_viii:NN ⟨*str var1*⟩ ⟨*str var2*⟩

Reads the contents of the ⟨*str var2*⟩ as an UTF8-encoded sequence of bytes, and stores the resulting characters (which can now have any character code) into ⟨*str var1*⟩. This function raises an error if the ⟨*str var2*⟩ is not a valid sequence of bytes in the UTF8 encoding. In the pdfTEX engine, this function raises an error if any of the resulting characters is outside the range 0 to 255.

\str_UTF_viii_from_native:NN

\str_UTF_viii_from_native:NN ⟨*str var1*⟩ ⟨*str var2*⟩

Converts each character of the ⟨*str var2*⟩ into a sequence of bytes, as defined by the UTF8 encoding, and stores the result in ⟨*str var1*⟩. In the pdfTEX engine, this function is of course of very little use, but can be used without harm: characters in the range 0 to 127 are left unchanged, and characters in the range 128 to 255 become two-byte sequences, as per the definition of `UTF8`.

\str_bytes_escape_hexadecimal:NN  \str_bytes_escape_hexadecimal:NN ⟨*str var1*⟩ ⟨*str var2*⟩
\str_bytes_unescape_hexadecimal:NN  \str_bytes_unescape_hexadecimal:NN ⟨*str var1*⟩ ⟨*str var2*⟩

The `escape` variant takes each character in ⟨*str var1*⟩ and converts it into the hexadecimal representation. The resulting string is stored in ⟨*str var2*⟩ within the current TEX group level. The `unescape` variant reverses this process. Thus for example

```
\str_set:Nn \l_mya_str { Hello ( ) }
\str_bytes_escape_hexadecimal:NN \l_mya_str \l_myb_str
```

will result in `\l_myb_str` containing the string `48656C6C6F2829`, while

```
\str_set:Nn \l_mya_str { 48656C6C6F2829 }
\str_bytes_unescape_hexadecimal:NN \l_mya_str \l_myb_str
```

will result in `\l_myb_str` containing the string `Hello()`. See also `\pdfescapehex`.

\str_bytes_escape_name:NN  \str_bytes_escape_name:NN ⟨*str var1*⟩ ⟨*str var2*⟩
\str_bytes_unescape_name:NN  \str_bytes_unescape_name:NN ⟨*str var1*⟩ ⟨*str var2*⟩

The `escape` variant takes each character in ⟨*str var1*⟩ and replaces any which cannot be used directly in a PDF string object with the appropriate hexadecimal representation preceded by the `#` character. The resulting string is stored in ⟨*str var2*⟩ within the current TEX group level. The `unescape` variant reverses this process. Thus for example

```
\str_set:Nn \l_mya_str { Hello ( ) }
\str_bytes_escape_name:NN \l_mya_str \l_myb_str
```

will result in `\l_myb_str` containing the string `Hello#28#29`, while

```
\char_set_catcode_other:N \#
\str_set:Nn \l_mya_str { Hello#28#29  }
\str_bytes_unescape_name:NN \l_mya_str \l_myb_str
```

will result in `\l_myb_str` containing the string `Hello()`. See also `\pdfescapename`.

\str_bytes_escape_string:NN
\str_bytes_unescape_string:NN

\str_bytes_escape_string:NN ⟨*str var1*⟩ ⟨*str var2*⟩
\str_bytes_unescape_string:NN ⟨*str var1*⟩ ⟨*str var2*⟩

The `escape` variant takes each character in ⟨*str var1*⟩ and prepends a `\` to any which cannot be used directly in a PDF string object. The resulting string is stored in ⟨*str var2*⟩ within the current TEX group level. The `unescape` variant reverses this process. Thus for example

```
\str_set:Nn \l_mya_str { Hello [ ] }
\str_bytes_escape_string:NN \l_mya_str \l_myb_str
```

will result in `\l_myb_str` containing the string `Hello\(\)`, while

```
\str_set:Nn \l_mya_str { Hello\(\) }
\str_bytes_unescape_string:NN \l_mya_str \l_myb_str
```

will result in `\l_myb_str` containing the string `Hello()`. Some characters (for example a space) are converted to a three-digit octal representation, so for example a space gives `\040`. See also `\pdfescapename`.

\str_bytes_percent_encode:NN
\str_bytes_percent_decode:NN

\str_bytes_percent_encode:NN metastr var1 ⟨*str var2*⟩
\str_bytes_percent_decode:NN metastr var1 ⟨*str var2*⟩

The `escape` variant takes each character in ⟨*str var1*⟩ and replaces any which cannot be used directly in a URL with the appropriate hexadecimal representation preceded by the `%` character. The resulting string is stored in ⟨*str var2*⟩ within the current TEX group level. The `unescape` variant reverses this process. Thus for example

```
\str_set:Nn \l_mya_str { Hello ( ) }
\str_bytes_precent_encode:NN \l_mya_str \l_myb_str
```

will result in `\l_myb_str` containing the string `Hello%28%29`, while

```
\char_set_catcode_other:N \%
\str_set:Nn \l_mya_str { Hello%28%29  }
\str_bytes_unescape_name:NN \l_mya_str \l_myb_str
```

will result in `\l_myb_str` containing the string `Hello()`.

## 0.6 Internal string functions

\tl_to_other_str:N ⋆
\tl_to_other_str:n ⋆

\tl_to_other_str:n {⟨*token list*⟩}

Converts the ⟨*token list*⟩ to an ⟨*other string*⟩, where spaces have category code "other". These functions create "safe" strings.

**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_sanitize_args:Nn ⋆
\str_sanitize_args:Nnn ⋆

\str_sanitize_args:Nnn ⟨*function*⟩
{⟨*token list*$_1$⟩} {⟨*token list*$_2$⟩}

Converts the ⟨*token lists*⟩ to "safe" strings (where spaces have category code "other"), and hands-in the result as arguments to ⟨*function*⟩.

\str_map_tokens:Nn ☆

\str_map_tokens:Nn ⟨*str var*⟩ ⟨*tokens*⟩

Maps the ⟨*tokens*⟩ over every character in the ⟨*str var*⟩.

\str_aux_escape:NNNn

\str_aux_escape: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 assigned globally to \g_str_result_tl.

### 0.7 Possibilities

- More encodings (see Heiko's stringenc). In particular, what is needed for pdf: UTF-16?
- \str_between:nnn {⟨*str*⟩} {⟨*begin delimiter*⟩} {⟨*end delimiter*⟩} giving the piece of ⟨*str*⟩ between ⟨*begin*⟩ and ⟨*end*⟩; could be used with empty ⟨*begin*⟩ or ⟨*end*⟩ to indicate that we want everything until ⟨*end*⟩ or starting from ⟨*begin*⟩, respectively;
- \str_count_in:nn {⟨*str*⟩} {⟨*substr*⟩} giving the number of occurrences of ⟨*substr*⟩ in ⟨*str*⟩;
- \str_if_head_eq:nN alias of \tl_if_head_eq_charcode:nN
- \str_if_numeric/decimal/integer:n, perhaps in l3fp?

Some functionalities of stringstrings and xstring as well.

# 1 l3str implementation

```
1 ⟨*initex | package⟩
2 \ProvidesExplPackage
3   {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
```

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)

## 1.1 General functions

`\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_set:Nn` `\str_set:Nx` `\str_set:cn` `\str_set:cx` `\str_gset:Nn` `\str_gset:Nx` `\str_gset:cn` `\str_gset: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_gput_right:Nn` `\str_gput_right:Nx` `\str_gput_right:cn` `\str_gput_right:cx` Simply convert the token list inputs to ⟨*strings*⟩.

```
\group_begin:
  \cs_set_protected_nopar:Npn \str_tmp:w #1
    {
      \cs_new_protected:cpx { str_ #1 :Nn } ##1##2
        { \exp_not:c { tl_ #1 :Nx } ##1 { \exp_not:N \tl_to_str:n {##2} } }
      \exp_args:Nc \cs_generate_variant:Nn { str_ #1 :Nn } { Nx , cn , cx }
    }
  \str_tmp:w { set }
  \str_tmp:w { gset }
  \str_tmp:w { put_left }
  \str_tmp:w { gput_left }
  \str_tmp:w { put_right }
  \str_tmp:w { gput_right }
\group_end:
```

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

## 1.2 Variables and constants

`\str_tmp:w` `\g_str_tmpa_tl` Internal scratch space for some functions.

```
\cs_new_protected_nopar:Npn \str_tmp:w { }
\tl_new:N \g_str_tmpa_tl
```

(*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
\l_str_bytes_int

When converting from various escaped forms to raw characters, we often need to read several digits (hexadecimal or octal depending on the case) and keep track of the corresponding character code in `\l_str_char_int`. For UTF8 support, the number of bytes of for the current character is stored in `\l_str_bytes_int`.

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

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

\c_backslash_str
\c_lbrace_str
\c_rbrace_str
\c_hash_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_percent_str   { \cs_to_str:N \% }
```

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

## 1.3 Escaping spaces

\tl_to_other_str:N
\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`.

```
\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 }
  }
\cs_new_nopar:Npn \tl_to_other_str:N { \exp_args:No \tl_to_other_str:n }
```

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

\str_sanitize_args:Nn
\str_sanitize_args:Nnn

Here, f-expansion does not lose leading spaces, since they have catcode "other" after \str_sanitize:n.

```
\cs_new:Npn \str_sanitize_args:Nn #1#2
  { \exp_args:Nf #1 { \tl_to_other_str:n {#2} } }
\cs_new:Npn \str_sanitize_args:Nnn #1#2#3
  {
    \exp_args:Nff #1
      { \tl_to_other_str:n {#2} }
      { \tl_to_other_str:n {#3} }
  }
```

(*End definition for* \str_sanitize_args:Nn *and* \str_sanitize_args:Nnn*. These functions are documented on page* ??.)

## 1.4 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 { \str_sanitize_args:Nn \str_length_unsafe:n }
\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` a truly "safe" 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
      { \str_sanitize_args:Nn \str_item_unsafe:nn {#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:nnnw`
`\str_substr_aux_ii:nnw`
`\str_aux_eval_args:Nnnn`
`\str_aux_normalize_range:nn`

Sanitize the string, then limit the second and third arguments 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
      { \str_sanitize_args:Nn \str_substr_unsafe:nnn {#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
  {
    \str_aux_eval_args:Nnnn \str_substr_aux:nnnw
      { \str_length_unsafe:n {#1} }
      {#2}
      {#3}
      #1
    \q_stop
  }
\cs_new:Npn \str_substr_aux:nnnw #1#2#3
  {
    \exp_args:Nf \str_substr_aux_ii:nnw
      { \str_aux_normalize_range:nn {#2} {#1} }
      { \str_aux_normalize_range:nn {#3} {#1} }
  }
\cs_new:Npn \str_substr_aux_ii: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_eval_args:Nnnn #1#2#3#4
  {
    \exp_after:wN #1
    \exp_after:wN { \int_value:w \int_eval:w #2 \exp_after:wN }
    \exp_after:wN { \int_value:w \int_eval:w #3 \exp_after:wN }
    \exp_after:wN { \int_value:w \int_eval:w #4 }
  }
\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* **??**.)

## 1.5 Internal mapping function

`\str_map_tokens:nn`
`\str_map_tokens:Nn`
`\str_map_tokens_aux:nn`
`\str_map_tokens_loop:nN`
`\str_map_break_do:n`

We simply need to be careful with spaces. Spaces are fed to the mapped tokens with category code 12, not 10.

```
\cs_new_nopar:Npn \str_map_tokens:Nn
  { \exp_args:No \str_map_tokens:nn }
\cs_new_nopar:Npn \str_map_tokens:nn
  { \str_sanitize_args:Nn \str_map_tokens_aux:nn }
\cs_new:Npn \str_map_tokens_aux:nn #1#2
  {
    \str_map_tokens_loop:nN {#2} #1
      { ? \use_none_delimit_by_q_recursion_stop:w } \q_recursion_stop
  }
\cs_new_nopar:Npn \str_map_tokens_loop:nN #1#2
  {
    \use_none:n #2
```

```
    #1 #2
    \str_map_tokens_loop:nN {#1}
  }
\cs_new_eq:NN \str_map_break_do:n \use_i_delimit_by_q_recursion_stop:nw
```

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

## 1.6 String conditionals

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

The `nn` and `xx` variants are already defined in l3basics.

```
\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:NN` `\str_if_contains_char:nN` `\str_if_contains_char_aux:NN` `\str_if_contains_char_end:w`

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 { p , T , F , TF }
  {
    \str_map_tokens:Nn #1 { \str_if_contains_char_aux:NN #2 }
    \prg_return_false:
  }
\prg_new_conditional:Npnn \str_if_contains_char:nN #1#2 { p , T , F , TF }
  {
    \str_map_tokens:nn {#1} { \str_if_contains_char_aux:NN #2 }
    \prg_return_false:
  }
\cs_new_nopar:Npn \str_if_contains_char_aux:NN #1#2
  {
    \if_num:w `#1 = `#2 \exp_stop_f:
      \str_if_contains_char_end:w
    \fi:
  }
\cs_new_nopar:Npn \str_if_contains_char_end:w \fi: #1 \prg_return_false:
  { \fi: \prg_return_true: }
```

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

`\str_if_bytes:N` `\str_if_bytes_aux:N`

Loop over the string, checking if every character code is less than 256.

```
\prg_new_conditional:Npnn \str_if_bytes:N #1 { p , T , F , TF }
  {
    \str_map_tokens:Nn #1 { \str_if_bytes_aux:N }
    \prg_return_true:
  }
\cs_new_nopar:Npn \str_if_bytes_aux:N #1
  {
```

```
    \int_compare:nNnF {‘#1} < \c_two_hundred_fifty_six
      {
        \use_i_delimit_by_q_recursion_stop:nw
          { \prg_return_false: \use_none:n }
      }
  }
```

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

## 1.7 Internal conditionals

`\str_aux_hexadecimal_test:NTF` This test is used when reading hexadecimal digits, for the `\x` escape sequence, and some conversion functions. It returns ⟨*true*⟩ if the token is a hexadecimal digit, and ⟨*false*⟩ otherwise. It has the additional side-effect of updating the value of `\l_str_char_int` (number formed in base 16 from the digits read so far).

```
\prg_new_protected_conditional:Npnn \str_aux_hexadecimal_test:N #1 { TF }
  {
    \tl_if_in:onTF { \tl_to_str:n { abcdef } } {#1}
      {
        \int_set:Nn \l_str_char_int
          { \c_sixteen * \l_str_char_int + ‘#1 - 87 }
        \prg_return_true:
      }
      {
        \if_num:w \c_fifteen < "1 \exp_not:N #1 \exp_stop_f:
          \int_set:Nn \l_str_char_int
            { \c_sixteen * \l_str_char_int + "#1 }
          \prg_return_true:
        \else:
          \prg_return_false:
        \fi:
      }
  }
```

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

`\str_aux_octal_test:NTF` This test is used when reading octal digits, for some conversion functions. It returns ⟨*true*⟩ if the token is an octal digit, and ⟨*false*⟩ otherwise. It has the additional side-effect of updating the value of `\l_str_char_int` (number formed in base 8 from the digits read so far).

```
\prg_new_protected_conditional:Npnn \str_aux_octal_test:N #1 { TF }
  {
    \if_num:w \c_seven < ’1 \exp_not:N #1 \exp_stop_f:
      \int_set:Nn \l_str_char_int
        { \c_eight * \l_str_char_int + ’#1 }
      \prg_return_true:
    \else:
      \prg_return_false:
    \fi:
  }
```

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

`\str_aux_char_if_octal_digit:NTF`

```
\cs_new_protected_nopar:Npn \str_aux_char_if_octal_digit:NTF #1
  { \tl_if_in:nnTF { 01234567 } {#1} }
```

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

## 1.8 Conversions

### 1.8.1 Simple unescaping

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_aux_escape:NNNn`
`\str_aux_escape_unescaped:N`
`\str_aux_escape_escaped:N`
`\str_aux_escape_raw:N`
`\str_aux_escape_loop:N`
`\str_aux_escape_\:w`

Treat the argument as an ⟨*escaped string*⟩, and store the corresponding ⟨*string*⟩ globally in `\g_str_result_tl`.

```
\cs_new_eq:NN \str_aux_escape_unescaped:N \use:n
\cs_new_eq:NN \str_aux_escape_escaped:N   \use:n
\cs_new_eq:NN \str_aux_escape_raw:N       \use:n
\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 }
      \tl_gset:Nx \g_str_result_tl { \tl_to_other_str:n {#4} }
      \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
        }
    \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_\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 (this is a side effect of \str_aux_hexadecimal_test:NTF), and check that the run of digits was interrupted 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 a \if_false: trick to restart the assignment.

```
\cs_new_nopar:cpn { str_aux_escape_/x:w } \str_aux_escape_loop:N
  {
    \if_false: { \fi: }
    \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 }
      {
        \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_protected_nopar:Npn \str_aux_escape_x_unbraced_i:N #1
  {
    \str_aux_hexadecimal_test:NTF #1
      { \str_aux_escape_x_unbraced_ii:N }
      { \str_aux_escape_x_end: #1 }
  }
\cs_new_protected_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_test:NTF #1
          { \str_aux_escape_x_end: }
          { \str_aux_escape_x_end: #1 }
      }
  }
\cs_new_protected_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_test:NTF #1
          { \str_aux_escape_x_braced_loop:N }
          { \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_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:
    {
      \group_begin:
      \char_set_lccode:nn { \c_zero } { \l_str_char_int }
      \tl_to_lowercase:n
        {
          \group_end:
          \tl_gput_right: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* **??**.)

### 1.8.2 Escape and unescape strings for PDF use

`\str_aux_convert_store:`
`\str_aux_convert_store:NNn`

```
\group_begin:
  \char_set_catcode_other:n { `\^^@ }
  \cs_new_protected_nopar:Npn \str_aux_convert_store:
    {
      \char_set_lccode:nn { \c_zero } { \l_str_char_int }
      \tl_to_lowercase:n { \tl_gput_right:Nx \g_str_result_tl {^^@} }
    }
  \cs_new_protected_nopar:Npn \str_aux_convert_store:NNn #1#2#3
    {
      \char_set_lccode:nn { \c_zero } {#3}
      \tl_to_lowercase:n { #1 #2 {^^@} }
    }
\group_end:
```

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

`\str_aux_byte_to_hexadecimal:N`
`\str_aux_byte_to_octal:N`

```
\cs_new_nopar:Npn \str_aux_byte_to_hexadecimal:N #1
  {
```

```
    \int_compare:nNnTF {‘#1} < { 256 }
      {
        \int_to_letter:n { \int_div_truncate:nn {‘#1} \c_sixteen }
        \int_to_letter:n { \int_mod:nn {‘#1} \c_sixteen }
      }
      { \msg_expandable_error:n { Invalid~byte~‘#1’. } }
  }
\cs_new_nopar:Npn \str_aux_byte_to_octal:N #1
  {
    \int_compare:nNnTF {‘#1} < { 64 }
      {
        0
        \int_to_letter:n { \int_div_truncate:nn {‘#1} \c_eight }
        \int_to_letter:n { \int_mod:nn {‘#1} \c_eight }
      }
      {
        \int_compare:nNnTF {‘#1} < { 256 }
          { \int_to_octal:n {‘#1} }
          { \msg_expandable_error:n { Invalid~byte~‘#1’. } }
      }
  }
```

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

`\str_bytes_escape_hexadecimal:NN`

```
\cs_new_protected_nopar:Npn \str_bytes_escape_hexadecimal:NN #1#2
  {
    \str_set:Nx #1
      { \str_map_tokens:Nn #2 \str_aux_byte_to_hexadecimal:N }
  }
```

(*End definition for* `\str_bytes_escape_hexadecimal:NN`. *This function is documented on page* 6.)

`\str_bytes_escape_name:NN`

```
\tl_const:Nx \c_str_bytes_escape_name_str
  { \c_hash_str \c_percent_str \c_lbrace_str \c_rbrace_str ()/<>[] }
\cs_new_protected_nopar:Npn \str_bytes_escape_name:NN #1#2
  {
    \str_set:Nx #1
      { \str_map_tokens:Nn #2 \str_bytes_escape_name_aux:N }
  }
\cs_new_nopar:Npn \str_bytes_escape_name_aux:N #1
  {
    \int_compare:nNnTF {‘#1} < { "21 }
      { \c_hash_str \str_aux_byte_to_hexadecimal:N #1 }
      {
        \int_compare:nNnTF {‘#1} > { "7E }
          { \c_hash_str \str_aux_byte_to_hexadecimal:N #1 }
          {
            \str_if_contains_char:NNTF \c_str_bytes_escape_name_str #1
              { \c_hash_str \str_aux_byte_to_hexadecimal:N #1 }
```

```
            {#1}
        }
    }
}
```

(*End definition for* `\str_bytes_escape_name:NN`*. This function is documented on page 6.*)

`\str_bytes_escape_string:NN` 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.

```
\tl_const:Nx \c_str_bytes_escape_string_str { \c_backslash_str ( ) }
\cs_new_protected_nopar:Npn \str_bytes_escape_string:NN #1#2
  {
    \str_set:Nx #1
      { \str_map_tokens:Nn #2 \str_bytes_escape_string_aux:N }
  }
\cs_new_nopar:Npn \str_bytes_escape_string_aux:N #1
  {
    \int_compare:nNnTF {‘#1} < { "21 }
      { \c_backslash_str \str_aux_byte_to_octal:N #1 }
      {
        \int_compare:nNnTF {‘#1} > { "7E }
          { \c_backslash_str \str_aux_byte_to_octal:N #1 }
          {
            \str_if_contains_char:NNT \c_str_bytes_escape_string_str #1
              { \c_backslash_str }
            #1
          }
      }
  }
```

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

`\str_bytes_unescape_hexadecimal:NN` `\str_bytes_unescape_hexadecimal_aux:N` `\str_bytes_unescape_hexadecimal_aux_ii:N` Takes chars two by two, and interprets each pair as a hexadecimal code for a character. Any non-hexadecimal-digit is ignored. An odd-length string gets a `0` appended to it (this is equivalent to appending a `0` in all cases, and dropping it if it is alone).

```
\cs_new_protected_nopar:Npn \str_bytes_unescape_hexadecimal:NN #1#2
  {
    \group_begin:
      \tl_gclear:N \g_str_result_tl
      \int_zero:N \l_str_char_int
      \exp_last_unbraced:Nf \str_bytes_unescape_hexadecimal_aux:N
        { \tl_to_other_str:N #2 } 0
      \q_recursion_tail \q_recursion_stop
    \group_end:
    \tl_set_eq:NN #1 \g_str_result_tl
  }
\cs_new_protected_nopar:Npn \str_bytes_unescape_hexadecimal_aux:N #1
  {
    \quark_if_recursion_tail_stop:N #1
    \str_aux_hexadecimal_test:NTF #1
      { \str_bytes_unescape_hexadecimal_aux_ii:N }
```

```
      { \str_bytes_unescape_hexadecimal_aux:N }
  }
\cs_new_protected_nopar:Npn \str_bytes_unescape_hexadecimal_aux_ii:N #1
  {
    \quark_if_recursion_tail_stop:N #1
    \str_aux_hexadecimal_test:NTF #1
      {
        \str_aux_convert_store:
        \int_zero:N \l_str_char_int
        \str_bytes_unescape_hexadecimal_aux:N
      }
      { \str_bytes_unescape_hexadecimal_aux_ii:N }
  }
```

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

`\str_bytes_unescape_name:NN`
`\str_bytes_unescape_name_aux:wN`
`\str_bytes_unescape_name_aux_ii:N`

This changes all occurrences of `#` followed by two upper- or lowercase hexadecimal digits by the corresponding unescaped character.

```
\group_begin:
  \char_set_lccode:nn {`\*} {`\#}
  \tl_to_lowercase:n
    {
      \group_end:
      \cs_new_protected_nopar:Npn \str_bytes_unescape_name:NN #1#2
        {
          \group_begin:
            \tl_gclear:N \g_str_result_tl
            \int_zero:N \l_str_char_int
            \exp_last_unbraced:Nf \str_bytes_unescape_name_aux:wN
              { \tl_to_other_str:N #2 }
            * \q_recursion_tail \q_recursion_stop
          \group_end:
          \tl_set_eq:NN #1 \g_str_result_tl
        }
      \cs_new_protected_nopar:Npn \str_bytes_unescape_name_aux:wN #1 * #2
        {
          \tl_gput_right:Nx \g_str_result_tl {#1}
          \quark_if_recursion_tail_stop:N #2
          \str_aux_hexadecimal_test:NTF #2
            { \str_bytes_unescape_name_aux_ii:NN #2 }
            {
              \tl_gput_right:Nx \g_str_result_tl { \c_hash_str }
              \str_bytes_unescape_name_aux:wN #2
            }
        }
    }
\cs_new_protected_nopar:Npn \str_bytes_unescape_name_aux_ii:NN #1#2
  {
    \str_aux_hexadecimal_test:NTF #2
      {
```

```
        \str_aux_convert_store:
        \int_zero:N \l_str_char_int
        \str_bytes_unescape_name_aux:wN
      }
      {
        \tl_gput_right:Nx \g_str_result_tl { \c_hash_str #1 }
        \int_zero:N \l_str_char_int
        \str_bytes_unescape_name_aux:wN #2
      }
  }
```

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

`\str_bytes_unescape_string:NN` Here, we need to detect backslashes, which escape characters as follows.

`\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` Character code `ddd` (octal)

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. The PDF specification indicates that LF, CR, and CRLF should be converted to LF: *this is not implemented here*.

```
\group_begin:
  \char_set_lccode:nn {`\*} {`\\}
  \char_set_catcode_other:N \^^J
  \char_set_catcode_other:N \^^M
  \tl_to_lowercase:n
    {
      \group_end:
      \cs_new_protected_nopar:Npn \str_bytes_unescape_string:NN #1#2
        {
          \group_begin:
            \tl_gclear:N \g_str_result_tl
            \exp_last_unbraced:Nf \str_bytes_unescape_string_aux:wN
              { \tl_to_other_str:N #2 }
            * \q_recursion_tail \q_recursion_stop
          \group_end:
          \tl_set_eq:NN #1 \g_str_result_tl
```

```
        }
      \cs_new_protected_nopar:Npn \str_bytes_unescape_string_aux:wN #1 * #2
        {
          \tl_gput_right:Nx \g_str_result_tl {#1}
          \quark_if_recursion_tail_stop:N #2
          \int_zero:N \l_str_char_int
          \str_aux_octal_test:NTF #2
            { \str_bytes_unescape_string_aux_d:N }
            {
              \int_set:Nn \l_str_char_int
                {
                  \prg_case_str:xxn {#2}
                    {
                      { n } { 10 }
                      { r } { 13 }
                      { t } { 9 }
                      { b } { 8 }
                      { f } { 12 }
                      { ( } { 40 }
                      { ) } { 41 }
                      { \c_backslash_str } { 92 }
                      { ^^J } { -1 }
                      { ^^M } { -1 }
                    }
                    {‘#2}
                }
              \int_compare:nNnF \l_str_char_int = \c_minus_one
                { \str_aux_convert_store: }
              \str_bytes_unescape_string_aux:wN
            }
        }
    }
\cs_new_protected_nopar:Npn \str_bytes_unescape_string_aux_d:N #1
  {
    \str_aux_octal_test:NTF #1
      { \str_bytes_unescape_string_aux_dd:N }
      {
        \str_aux_convert_store:
        \str_bytes_unescape_string_aux:wN #1
      }
  }
\cs_new_protected_nopar:Npn \str_bytes_unescape_string_aux_dd:N #1
  {
    \str_aux_octal_test:NTF #1
      {
        \str_aux_convert_store:
        \str_bytes_unescape_string_aux:wN
      }
      {
        \str_aux_convert_store:
```

```
        \str_bytes_unescape_string_aux:wN #1
      }
  }
```

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

### 1.8.3 Percent encoding

`\str_bytes_percent_encode:NN`

```
\tl_const:Nx \c_str_bytes_percent_encode_str { \tl_to_str:n { [] } }
\tl_const:Nx \c_str_bytes_percent_encode_not_str { \tl_to_str:n { "-.<> } }
\cs_new_protected_nopar:Npn \str_bytes_percent_encode:NN #1#2
  {
    \str_set:Nx #1
      { \str_map_tokens:Nn #2 \str_bytes_percent_encode_aux:N }
  }
\cs_new_nopar:Npn \str_bytes_percent_encode_aux:N #1
  {
    \int_compare:nNnTF {‘#1} < { "41 }
      {
        \str_if_contains_char:NNTF \c_str_bytes_percent_encode_not_str #1
          { #1 }
          { \c_percent_str \str_aux_byte_to_hexadecimal:N #1 }
      }
      {
        \int_compare:nNnTF {‘#1} > { "7E }
          { \c_percent_str \str_aux_byte_to_hexadecimal:N #1 }
          {
            \str_if_contains_char:NNTF \c_str_bytes_percent_encode_str #1
              { \c_percent_str \str_aux_byte_to_hexadecimal:N #1 }
              { #1 }
          }
      }
  }
```

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

`\str_bytes_percent_decode:NN`
`\str_bytes_percent_decode_aux:wN`
`\str_bytes_percent_decode_aux_ii:NN`

```
\group_begin:
  \char_set_lccode:nn {‘\*} {‘\%}
  \tl_to_lowercase:n
    {
      \group_end:
      \cs_new_protected_nopar:Npn \str_bytes_percent_decode:NN #1#2
        {
          \group_begin:
            \tl_gclear:N \g_str_result_tl
            \exp_last_unbraced:Nf \str_bytes_percent_decode_aux:wN
              { \tl_to_other_str:N #2 }
            * \q_recursion_tail \q_recursion_stop
          \group_end:
```

```
          \tl_set_eq:NN #1 \g_str_result_tl
        }
      \cs_new_protected_nopar:Npn \str_bytes_percent_decode_aux:wN #1 * #2
        {
          \tl_gput_right:Nx \g_str_result_tl {#1}
          \quark_if_recursion_tail_stop:N #2
          \int_zero:N \l_str_char_int
          \str_aux_hexadecimal_test:NTF #2
            { \str_bytes_percent_decode_aux_ii:NN #2 }
            {
              \tl_gput_right:Nx \g_str_result_tl { \c_percent_str }
              \str_bytes_percent_decode_aux:wN #2
            }
        }
    }
\cs_new_protected_nopar:Npn \str_bytes_percent_decode_aux_ii:NN #1#2
  {
    \str_aux_hexadecimal_test:NTF #2
      {
        \str_aux_convert_store:
        \str_bytes_percent_decode_aux:wN
      }
      {
        \tl_gput_right:Nx \g_str_result_tl {#1}
        \str_bytes_percent_decode_aux:wN #2
      }
  }
```

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

### 1.8.4 UTF-8 support

`\str_native_from_UTF_viii:NN`
`\str_native_from_UTF_viii_aux_i:N`
`\str_native_from_UTF_viii_aux_ii:nN`
`\str_native_from_UTF_viii_aux_iii:N`
`\str_native_from_UTF_viii_aux_iv:`
`\str_native_from_UTF_viii_aux_v:`

```
\cs_new_protected_nopar:Npn \str_native_from_UTF_viii:NN #1#2
  {
    \group_begin:
      \tl_gclear:N \g_str_result_tl
      \exp_last_unbraced:Nf \str_native_from_UTF_viii_aux_i:N
        { \tl_to_other_str:N #2 }
      \q_recursion_tail \q_recursion_stop
    \group_end:
    \tl_set_eq:NN #1 \g_str_result_tl
  }
\cs_new_protected_nopar:Npn \str_native_from_UTF_viii_aux_i:N #1
  {
    \quark_if_recursion_tail_stop:N #1
    \int_set:Nn \l_str_char_int {`#1}
    \str_native_from_UTF_viii_aux_ii:nN { 128 } \c_one
    \str_native_from_UTF_viii_aux_ii:nN { 64 }  \c_zero
    \str_native_from_UTF_viii_aux_ii:nN { 32 }  \c_two
```

```
    \str_native_from_UTF_viii_aux_ii:nN { 16 }  \c_three
    \str_native_from_UTF_viii_aux_ii:nN { 8 }   \c_four
    \msg_error:nnx { str } { utf8-invalid-byte } {#1}
    \use_i:nnn \str_native_from_UTF_viii_aux_i:N
    \q_stop
    \str_native_from_UTF_viii_aux_iii:N
  }
\cs_new_protected_nopar:Npn \str_native_from_UTF_viii_aux_ii:nN #1#2
  {
    \int_compare:nNnTF \l_str_char_int < {#1}
      {
        \int_set_eq:NN \l_str_bytes_int #2
        \use_none_delimit_by_q_stop:w
      }
      { \int_sub:Nn \l_str_char_int {#1} }
  }
\cs_new_protected_nopar:Npn \str_native_from_UTF_viii_aux_iii:N #1
  {
    \int_compare:nNnTF \l_str_bytes_int < \c_two
      { \str_native_from_UTF_viii_aux_iv: #1 }
      {
        \if_meaning:w \q_recursion_tail #1
          \msg_error:nn { str } { utf8-premature-end }
          \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
        \fi:
        \quark_if_recursion_tail_stop:N #1
        \tex_multiply:D \l_str_char_int by 64 \scan_stop:
        \int_decr:N \l_str_bytes_int
        \int_compare:nNnTF {`#1} < { 128 }
          { \use_ii:nn }
          { \int_compare:nNnTF {`#1} < { 192 } }
          {
            \int_add:Nn \l_str_char_int { `#1 - 128 }
            \str_native_from_UTF_viii_aux_iii:N
          }
          {
            \msg_error:nnx { str } { utf8-missing-byte } {#1}
            \str_native_from_UTF_viii_aux_i:N #1
          }
      }
  }
\cs_new_protected_nopar:Npn \str_native_from_UTF_viii_aux_iv:
  {
    \int_compare:nNnTF \l_str_bytes_int = \c_zero
      {
        \msg_error:nnx { str } { utf8-extra-byte }
          { \int_eval:n { \l_str_char_int + 128 } }
      }
      { \str_native_from_UTF_viii_aux_v: }
    \str_native_from_UTF_viii_aux_i:N
```

```
  }
\pdftex_if_engine:TF
  {
    \cs_new_protected_nopar:Npn \str_native_from_UTF_viii_aux_v:
      {
        \int_compare:nNnTF { \l_str_char_int } < { 256 }
          { \str_aux_convert_store: }
          {
            \msg_error:nnx { str } { utf8-pdftex-overflow }
              { \int_use:N \l_str_char_int }
          }
      }
  }
  {
    \cs_new_protected_nopar:Npn \str_native_from_UTF_viii_aux_v:
      { \str_aux_convert_store: }
  }
```

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

`\str_UTF_viii_from_native:NN`

```
\cs_new_protected_nopar:Npn \str_UTF_viii_from_native:NN #1#2
  {
    \group_begin:
      \tl_gclear:N \g_str_result_tl
      \str_map_tokens:Nn #2 \str_UTF_viii_from_native_aux_i:N
    \group_end:
    \tl_set_eq:NN #1 \g_str_result_tl
  }
\cs_new_protected_nopar:Npn \str_UTF_viii_from_native_aux_i:N #1
  {
    \int_set:Nn \l_str_char_int {`#1}
    \int_compare:nNnTF \l_str_char_int < { 128 }
      { \tl_gput_right:Nx \g_str_result_tl {#1} }
      {
        \tl_gclear:N \g_str_tmpa_tl
        \int_set:Nn \l_str_bytes_int { 64 }
        \str_UTF_viii_from_native_aux_ii:n { 32 }
        \str_UTF_viii_from_native_aux_ii:n { 16 }
        \str_UTF_viii_from_native_aux_ii:n { 8 }
        \ERROR % somehow the unicode char was > "1FFFFF > "10FFFF
        \tl_gclear:N \g_str_tmpa_tl
        \use_none:n \q_stop
        \int_add:Nn \l_str_char_int { 2 * \l_str_bytes_int }
        \str_aux_convert_store:
        \tl_gput_right:Nx \g_str_result_tl { \g_str_tmpa_tl }
      }
  }
\cs_new_protected_nopar:Npn \str_UTF_viii_from_native_aux_ii:n #1
  {
    \str_aux_convert_store:NNn
```

```
      \tl_gput_left:Nx \g_str_tmpa_tl
      { 128 + \int_mod:nn { \l_str_char_int } { 64 } }
    \tex_divide:D \l_str_char_int by 64 \scan_stop:
    \int_add:Nn \l_str_bytes_int {#1}
    \int_compare:nNnT \l_str_char_int < {#1}
      { \use_none_delimit_by_q_stop:w }
  }
```

(*End definition for* `\str_UTF_viii_from_native:NN`. *This function is documented on page* 5.)

## 1.9 Messages

```
\msg_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_new:nnn { str } { utf8-invalid-byte }
  {
    \int_compare:nNnTF {`#1} < { 256 }
      { Byte~number~ \int_eval:n {`#1} ~invalid~in~utf-8~encoding. }
      { The~character~number~ \int_eval:n {`#1} ~is~not~a~byte. }
  }
\msg_new:nnn { str } { utf8-missing-byte }
  { The~byte~number~ \int_eval:n {`#1} ~is~not~a~valid~continuation~byte. }
\msg_new:nnn { str } { utf8-extra-byte }
  { The~byte~number~ \int_eval:n {`#1} ~is~only~valid~as~a~continuation~byte. }
\msg_new:nnnn { str } { utf8-premature-end }
  { Incomplete~last~UTF8~character. }
  {
    The~sequence~of~byte~that~to~be~converted~to~UTF8~
    ended~before~the~last~character~was~complete.~Perhaps~
    it~was~mistakenly~truncated?
  }
\msg_new:nnn { str } { utf8-pdftex-overflow }
  { The~character~number~#1~is~too~big~for~pdfTeX. }
```

## 1.10 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⟩
```

# 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.

### F



### G



### I



### L



### M



### O



### P

## Q



## R



## S

## T



## U



## X