

# The `l3str` package: manipulating strings of characters<sup>\*</sup>

The L<sup>A</sup>T<sub>E</sub>X3 Project<sup>†</sup>

Released 2011/10/09

L<sup>A</sup>T<sub>E</sub>X3 provides a set of functions to manipulate token lists as strings of characters, ignoring the category codes of those characters.

There is no separate string data type: string variables are simply specialized token lists, conventionally made out only of 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. However, it is often better to name token list variables meant to contain string data with a name such as `\l_..._str`. Those are then manipulated mostly with tools from `l3tl`.

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

---

<code>\c_backslash_str</code>	Constant strings, containing a single character, with category code 12.
<code>\c_lbrace_str</code>	
<code>\c_rbrace_str</code>	
<code>\c_hash_str</code>	
<code>\c_percent_str</code>	

---

<sup>\*</sup>This file describes v2900, last revised 2011/10/09.

<sup>†</sup>E-mail: [latex-team@latex-project.org](mailto:latex-team@latex-project.org)

---

<code>\tl_to_str:N</code>	★	<code>\tl_to_str:N</code>	$\langle \textit{tl var} \rangle$
<code>\tl_to_str:n</code>	★	<code>\tl_to_str:n</code>	$\{ \langle \textit{token list} \rangle \}$

---

Converts the  $\langle \textit{token list} \rangle$  to a  $\langle \textit{string} \rangle$ .

---

<code>\str_set:Nn</code>	<code>\str_set:Nn</code>	$\langle \textit{str var} \rangle$	$\{ \langle \textit{token list} \rangle \}$
<code>\str_set:(Nx cn cx)</code>			
<code>\str_gset:Nn</code>			
<code>\str_put_left:Nn(Nx cn cx)</code>	<code>\str_set:Nn</code>	$\langle \textit{str var} \rangle$	$\{ \langle \textit{token list} \rangle \}$
<code>\str_put_left:(Nx cn cx)</code>			
<code>\str_gput_left:Nn</code>			
<code>\str_put_right:Nn(Nx cn cx)</code>	<code>\str_set:Nn</code>	$\langle \textit{str var} \rangle$	$\{ \langle \textit{token list} \rangle \}$
<code>\str_put_right:(Nx cn cx)</code>			
<code>\str_gput_right:Nn</code>			
<code>\str_gput_right:(Nx cn cx)</code>			

---

Converts the  $\langle \textit{token list} \rangle$  to a  $\langle \textit{string} \rangle$ , and saves the result in  $\langle \textit{str var} \rangle$ .

---

<code>\str_input:Nn</code>	<code>\str_input:Nn</code>	$\langle \textit{str var} \rangle$	$\{ \langle \textit{token list} \rangle \}$
<code>\str_ginput:Nn</code>			

---

Converts the  $\langle \textit{token list} \rangle$  into a  $\langle \textit{string} \rangle$ , and stores it in the  $\langle \textit{str var} \rangle$ . 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 XeTeX.
- `\a`, `\e`, `\f`, `\n`, `\r`, `\t` stand for specific characters:

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

For instance,

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

results in `\l_my_str` containing the characters `\<#abc def^`, since `<` has ascii code 3C (in hexadecimal).

## 0.2 Characters given by their position

---

`\str_length:N` ★ `\str_length:n {⟨token list⟩}`

---



---

`\str_length:n` ★ `\str_length:n {⟨token list⟩}`

---



---

`\str_length_ignore_spaces:n` ★ `\str_length:n {⟨token list⟩}`

---

Leaves the length of the string representation of *⟨token list⟩* in the input stream. 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` returns the number of non-space characters.

**T<sub>E</sub>Xhackers 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:n {⟨token list⟩}`

---



---

`\str_head_ignore_spaces:n` ★ `\str_head:n {⟨token list⟩}`

---

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 returns the first non-space character. 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:n {⟨token list⟩}`

---



---

`\str_tail_ignore_spaces:n` ★ `\str_tail:n {⟨token list⟩}`

---

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:nn {⟨token list⟩} {⟨integer expression⟩}`

---



---

`\str_item_ignore_spaces:nn` ★ `\str_item:nn {⟨token list⟩} {⟨integer expression⟩}`

---

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_from_to:Nnn` ★ `\str_from_to:nnn {⟨token list⟩} {⟨start index⟩} {⟨end index⟩}`

---



---

`\str_from_to:nnn` ★ `\str_from_to:nnn {⟨token list⟩} {⟨start index⟩} {⟨end index⟩}`

---



---

`\str_from_to_ignore_spaces:nnn` ★ `\str_from_to:nnn {⟨token list⟩} {⟨start index⟩} {⟨end index⟩}`

---

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.

### 0.3 String conditionals

---

`\str_if_eq_p:NN` ★ `\str_if_eq_p:nn {⟨t1⟩} {⟨t2⟩}`  
`\str_if_eq:NNTF` ★ `\str_if_eq:nnTF {⟨t1⟩} {⟨t2⟩} {⟨true code⟩} {⟨false code⟩}`

---



---

`\str_if_eq_p:nn` ★ `\str_if_eq_p:nn {⟨t1⟩} {⟨t2⟩}`  
`\str_if_eq_p:(Vn|on|no|nV|VV|xx)` ★ `\str_if_eq:nnTF {⟨t1⟩} {⟨t2⟩} {⟨true code⟩} {⟨false code⟩}`  
`\str_if_eq:nnTF` ★  
`\str_if_eq:(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).

---

<code>\str_if_contains_char_p:NN</code>	★	<code>\str_if_contains_char:nN</code>	{ <i>⟨token list⟩</i> }	<i>⟨char⟩</i>
<code>\str_if_contains_char:NN</code>	<u>NTF</u> ★			

---



---

<code>\str_if_contains_char_p:nN</code>	★	<code>\str_if_contains_char:nN</code>	{ <i>⟨token list⟩</i> }	<i>⟨char⟩</i>
<code>\str_if_contains_char:nN</code>	<u>NTF</u> ★			

---

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 Bytes and encoding

### 0.4.1 Conditionals

---

<code>\str_if_bytes_p:N</code>	★	<code>\str_if_bytes:NTF</code>	<i>⟨str var⟩</i>	{ <i>⟨true code⟩</i> }	{ <i>⟨false code⟩</i> }
<code>\str_if_bytes:NTF</code>	★				

---

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

---

<code>\str_if_UTF_viii_p:N</code>	★	<code>\str_if_UTF_viii:N</code>	<i>⟨str var⟩</i>	{ <i>⟨true code⟩</i> }	{ <i>⟨false code⟩</i> }
<code>\str_if_UTF_viii:NTF</code>	★				

---

Tests whether the *⟨str var⟩* only contains characters in the range 0–255, and forms a valid UTF8 string. **Missing!**

### 0.4.2 Conversion

---

<code>\str_native_from_UTF_viii:NN</code>		<code>\str_native_from_UTF_viii:NN</code>	<i>⟨str var1⟩</i>	<i>⟨str var2⟩</i>
---	--	---	-------------------	-------------------

---

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 0–255 (in other words, don't use this function with pdfTeX).

---

<code>\str_UTF_viii_from_native:NN</code>		<code>\str_UTF_viii_from_native:NN</code>	<i>⟨str var1⟩</i>	<i>⟨str var2⟩</i>
---	--	---	-------------------	-------------------

---

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 0–127 are left unchanged, and characters in the range 128–255 become two-byte sequences, as per the definition of UTF8.

---

<code>\str_bytes_escape_hexadecimal:NN</code>		<code>\str_bytes_escape_hexadecimal:NN</code>	<i>⟨str var1⟩</i>	<i>⟨str var2⟩</i>
		<code>\str_bytes_unescape_hexadecimal:NN</code>	<i>⟨str var1⟩</i>	<i>⟨str var2⟩</i>

---



---

<code>\str_bytes_unescape_hexadecimal:NN</code>		<code>\str_bytes_escape_hexadecimal:NN</code>	<i>⟨str var1⟩</i>	<i>⟨str var2⟩</i>
		<code>\str_bytes_unescape_hexadecimal:NN</code>	<i>⟨str var1⟩</i>	<i>⟨str var2⟩</i>

---

See `\pdfescapehex`.

---

<code>\str_bytes_escape_name:NN</code>	<code>\str_bytes_escape_name:NN</code> $\langle str\ var1 \rangle$ $\langle str\ var2 \rangle$
	<code>\str_bytes_unescape_name:NN</code> $\langle str\ var1 \rangle$ $\langle str\ var2 \rangle$

---



---

<code>\str_bytes_unescape_name:NN</code>	<code>\str_bytes_escape_name:NN</code> $\langle str\ var1 \rangle$ $\langle str\ var2 \rangle$
	<code>\str_bytes_unescape_name:NN</code> $\langle str\ var1 \rangle$ $\langle str\ var2 \rangle$

---

See `\pdfescapename`.

---

<code>\str_bytes_escape_string:NN</code>	<code>\str_bytes_escape_string:NN</code> $\langle str\ var1 \rangle$ $\langle str\ var2 \rangle$
	<code>\str_bytes_unescape_string:NN</code> $\langle str\ var1 \rangle$ $\langle str\ var2 \rangle$

---



---

<code>\str_bytes_unescape_string:NN</code>	<code>\str_bytes_escape_string:NN</code> $\langle str\ var1 \rangle$ $\langle str\ var2 \rangle$
	<code>\str_bytes_unescape_string:NN</code> $\langle str\ var1 \rangle$ $\langle str\ var2 \rangle$

---

See `\pdfescapestring`.

---

<code>\str_bytes_percent_encode:NN</code>	<code>\str_bytes_percent_encode:NN</code> metastr var1 $\langle str\ var2 \rangle$
	<code>\str_bytes_percent_decode:NN</code> metastr var1 $\langle str\ var2 \rangle$

---



---

<code>\str_bytes_percent_decode:NN</code>	<code>\str_bytes_percent_encode:NN</code> metastr var1 $\langle str\ var2 \rangle$
	<code>\str_bytes_percent_decode:NN</code> metastr var1 $\langle str\ var2 \rangle$

---

Used for urls.

## 0.5 Internal string functions

---

<code>\tl_to_other_str:N</code> ★	<code>\tl_to_other_str:n</code> $\{\langle token\ list \rangle\}$
<code>\tl_to_other_str:n</code> ★	

---

Converts the  $\langle token\ list \rangle$  to an  $\langle other\ string \rangle$ , where spaces have category code “other”. These functions create “safe” strings.

**T<sub>E</sub>Xhackers 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.

---

<code>\str_sanitize_args:Nn</code> ★	<code>\str_sanitize_args:Nnn</code> $\langle function \rangle$
<code>\str_sanitize_args:Nnn</code> ★	$\{\langle token\ list_1 \rangle\}$ $\{\langle token\ list_2 \rangle\}$

---

Converts the  $\langle token\ lists \rangle$  to “safe” strings (where spaces have category code “other”), and hands-in the result as arguments to  $\langle function \rangle$ .

---

<code>\str_map_tokens:Nn</code> ☆	<code>\str_map_tokens:Nn</code> $\langle str\ var \rangle$ $\langle tokens \rangle$
-----------------------------------	---

---

Maps the  $\langle tokens \rangle$  over every character in the  $\langle str\ var \rangle$ .

---

`\str_aux_escape:NNNn` `\str_aux_escape:NNNn`  $\langle fn1 \rangle$   $\langle fn2 \rangle$   $\langle fn3 \rangle$   $\{\langle token\ list \rangle\}$

---

The  $\langle token\ list \rangle$  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  $\langle fn1 \rangle$ , and escaped characters are fed to the function  $\langle fn2 \rangle$  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  $\langle fn3 \rangle$ . The result is assigned globally to `\g_str_result_tl`.

## 0.6 Possibilities

- More encodings (see Heiko’s `stringenc`). In particular, what is needed for pdf: UTF-16?
- `\str_between:nnn`  $\{\langle str \rangle\}$   $\{\langle begin\ delimiter \rangle\}$   $\{\langle end\ delimiter \rangle\}$  giving the piece of  $\langle str \rangle$  between  $\langle begin \rangle$  and  $\langle end \rangle$ ; could be used with empty  $\langle begin \rangle$  or  $\langle end \rangle$  to indicate that we want everything until  $\langle end \rangle$  or starting from  $\langle begin \rangle$ , respectively;
- `\str_count_in:nn`  $\{\langle str \rangle\}$   $\{\langle substr \rangle\}$  giving the number of occurrences of  $\langle substr \rangle$  in  $\langle str \rangle$ ;
- `\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  $\langle *package \rangle$ 
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`, (changed behaviour slightly)
- `\str_tail:n`, `\str_tail_aux:w`, (changed behaviour slightly)
- `\str_length_skip_spaces` (deprecated)
- `\str_length_loop:NNNNNNNNN` (unchanged)

## 1.1 General functions

`\cs_if_exist_use:cF` This function could be moved to a different module in `l3kernel`. If the control sequence exists, use it, otherwise fall back to a default behaviour.

```

4 \cs_if_exist:NF \cs_if_exist_use:cF
5 {
6   \cs_new:Npn \cs_if_exist_use:cF #1
7     { \cs_if_exist:cTF {#1} { \use:c {#1} } }
8 }

```

*(End definition for \cs\_if\_exist\_use:cF. This function is documented on page ??.)*

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

```

9 \cs_if_exist:NF \use_i:nnnnnnnn
10 { \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` Simply convert the token list inputs to *strings*.

```

\str_set:Nx 11 \cs_set:Npn \str_tmp:w #1
\str_set:cn 12 {
\str_set:cx 13   \cs_new_protected:cpx { str_#1:Nn } ##1##2
\str_gset:Nn 14   { \exp_not:c { tl_#1:Nx } ##1 { \exp_not:N \tl_to_str:n {##2} } }
\str_gset:Nx 15   \exp_args:Nc \cs_generate_variant:Nn { str_#1:Nn } { Nx , cn , cx }
\str_gset:cn 16 }
\str_gset:cx 17 \str_tmp:w {set}
\str_put_left:Nn 18 \str_tmp:w {gset}
\str_put_left:Nx 19 \str_tmp:w {put_left}
\str_put_left:cn 20 \str_tmp:w {gput_left}
\str_put_left:cx 21 \str_tmp:w {put_right}
\str_gput_left:Nn 22 \str_tmp:w {gput_right}

```

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

## 1.2 Variables and constants

Internal scratch space for some functions.

```

23 \cs_set_protected_nopar:Npn \str_tmp:w { }
24 \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.

```

25 \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 UTF-8 support, the number of bytes of for the current character is stored in `\l_str_bytes_int`.

```

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

```

\c_lbrace_str 28 \tl_const:Nx \c_backslash_str { \cs_to_str:N \ }
\c_rbrace_str 29 \tl_const:Nx \c_lbrace_str { \cs_to_str:N \{ }
\c_hash_str 30 \tl_const:Nx \c_rbrace_str { \cs_to_str:N \} }
\c_percent_str 31 \tl_const:Nx \c_hash_str { \cs_to_str:N \% }
32 \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` Replaces all spaces by “other” spaces, after converting the token list to a string via `\tl_to_str:n`.

```

\tl_to_other_str_loop:w 33 \group_begin:
\tl_to_other_str_end:w 34 \char_set_lccode:nn {'\*}{'\ }
35 \char_set_lccode:nn {'\A}{'\A}
36 \tl_to_lowercase:n {
37 \group_end:
38 \cs_new:Npn \tl_to_other_str:n #1
39 {
40 \exp_after:wN \tl_to_other_str_loop:w \tl_to_str:n {#1} ~ %
41 A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \q_mark \q_stop
42 }
43 \cs_new_nopar:Npn \tl_to_other_str_loop:w
44 #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 \q_stop
45 {
46 \if_meaning:w A #8
47 \tl_to_other_str_end:w
48 \fi:
49 \tl_to_other_str_loop:w
50 #9 #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * \q_stop
51 }
52 \cs_new_nopar:Npn \tl_to_other_str_end:w \fi: #1 \q_mark #2 * A #3 \q_stop
53 { \fi: #2 }
54 }
55 \cs_new_nopar:Npn \tl_to_other_str:N
56 { \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` Here, f-expansion does not lose leading spaces, since they have catcode “other” after  
`\str_sanitize_args:Nnn` `\str_sanitize:n`.

```

57 \cs_new:Npn \str_sanitize_args:Nn #1 #2
58 {
59   \exp_args:Nf #1
60   { \tl_to_other_str:n {#2} }
61 }
62 \cs_new:Npn \str_sanitize_args:Nnn #1#2#3
63 {
64   \exp_args:Nff #1
65   { \tl_to_other_str:n {#2} }
66   { \tl_to_other_str:n {#3} }
67 }

```

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

```

68 \cs_new_nopar:Npn \str_length:N { \exp_args:No \str_length:n }
69 \cs_new:Npn \str_length:n { \str_sanitize_args:Nn \str_length_unsafe:n }
70 \cs_new_nopar:Npn \str_length_unsafe:n #1
71 { \str_length_aux:n { \str_length_loop:NNNNNNNN #1 } }
72 \cs_new:Npn \str_length_ignore_spaces:n #1
73 {
74   \str_length_aux:n
75   { \exp_after:wN \str_length_loop:NNNNNNNN \tl_to_str:n {#1} }
76 }
77 \cs_new:Npn \str_length_aux:n #1
78 {
79   \int_eval:n
80   {
81     #1
82     { X \c_eight } { X \c_seven } { X \c_six }
83     { X \c_five } { X \c_four } { X \c_three }
84     { X \c_two } { X \c_one } { X \c_zero }
85     \q_stop
86   }
87 }
88 \cs_set:Npn \str_length_loop:NNNNNNNN #1#2#3#4#5#6#7#8#9
89 {
90   \if_catcode:w X #9
91   \exp_after:wN \use_none_delimit_by_q_stop:w
92   \fi:
93   \c_nine + \str_length_loop:NNNNNNNN
94 }

```

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

`\str_head:N` 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 `\str_head_ignore_spaces:n` `\str_head_unsafe:n` `\str_head_aux:w` `\str_head_aux:w` `#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`.

```

95 \cs_set_nopar:Npn \str_head:N { \exp_args:No \str_head:n }
96 \cs_set:Npn \str_head:n #1
97   {
98     \exp_after:wN \str_head_aux:w
99     \tl_to_str:n {#1}
100    { { } } ~ \q_stop
101  }
102 \cs_set_nopar:Npx \str_head_aux:w #1 ~ %
103   { \exp_not:N \use_i_delimit_by_q_stop:nw #1 { ~ } }
104 \cs_new:Npn \str_head_ignore_spaces:n #1
105   { \exp_after:wN \use_i_delimit_by_q_stop:nw \tl_to_str:n {#1} { } \q_stop }
106 \cs_new_nopar:Npn \str_head_unsafe:n #1
107   { \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` 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.

```

108 \cs_new_nopar:Npn \str_tail:N { \exp_args:No \str_tail:n }
109 \cs_set:Npn \str_tail:n #1
110   {
111     \exp_after:wN \str_tail_aux:w
112     \reverse_if:N \if_charcode:w
113     \scan_stop: \tl_to_str:n {#1} X X \q_stop
114   }
115 \cs_set_nopar:Npn \str_tail_aux:w #1 X #2 \q_stop { \fi: #1 }
116 \cs_new:Npn \str_tail_ignore_spaces:n #1
117   {
118     \exp_after:wN \str_tail_aux_ii:w
119     \tl_to_str:n {#1} X { } X \q_stop
120   }
121 \cs_new_nopar:Npn \str_tail_unsafe:n #1

```

```

122 { \str_tail_aux_ii:w #1 X { } X \q_stop }
123 \cs_set_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` 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.

```

124 \cs_new:Npn \str_skip_do:nn #1
125 {
126   \int_compare:nNnTF {#1} > \c_seven
127   { \str_skip_aux:nnnnnnnnn }
128   { \str_skip_end:n }
129   {#1}
130 }
131 \cs_new:Npn \str_skip_aux:nnnnnnnnn #1#2 #3#4#5#6#7#8#9
132 { \exp_args:Nf \str_skip_do:nn { \int_eval:n { #1 - \c_seven } } {#2} }
133 \cs_new:Npn \str_skip_end:n #1
134 {
135   \if_case:w \int_eval:w #1 \int_eval_end:
136     \str_skip_end_ii:nwn { XXXXXXX }
137   \or: \str_skip_end_ii:nwn { XXXXXX }
138   \or: \str_skip_end_ii:nwn { XXXXX }
139   \or: \str_skip_end_ii:nwn { XXXX }
140   \or: \str_skip_end_ii:nwn { XXX }
141   \or: \str_skip_end_ii:nwn { XX }
142   \or: \str_skip_end_ii:nwn { X }
143   \or: \str_skip_end_ii:nwn { }
144   \else: \str_skip_end_ii:nwn { XXXXXXX }
145   \fi:
146 }
147 \cs_new:Npn \str_skip_end_ii:nwn #1 #2 \fi: #3
148 { \fi: \use_i:nnnnnnnnn {#3} #1 }
      (End definition for \str_skip_do:nn. This function is documented on page ??.)

```

`\str_collect_do:nn` 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.

```

149 \cs_new:Npn \str_collect_do:nn #1#2
150 { \str_collect_aux:n {#1} { \str_collect_end_iii:nwnnnnnnnn {#2} } }
151 \cs_new:Npn \str_collect_aux:n #1
152 {
153   \int_compare:nNnTF {#1} > \c_seven
154   { \str_collect_aux:nnnnnnnnn }
155   { \str_collect_end:n }
156   {#1}
157 }
158 \cs_new:Npn \str_collect_aux:nnnnnnnnn #1#2 #3#4#5#6#7#8#9
159 {

```

```

160     \exp_args:Nf \str_collect_aux:n
161     { \int_eval:n { #1 - \c_seven } }
162     { #2 #3#4#5#6#7#8#9 }
163 }
164 \cs_new:Npn \str_collect_end:n #1
165 {
166     \if_case:w \int_eval:w #1 \int_eval_end:
167         \str_collect_end_ii:nwn { {}{}{}{}{}{} }
168     \or: \str_collect_end_ii:nwn { {}{}{}{}{}{} }
169     \or: \str_collect_end_ii:nwn { {}{}{}{}{}{} }
170     \or: \str_collect_end_ii:nwn { {}{}{}{}{}{} }
171     \or: \str_collect_end_ii:nwn { {}{}{}{}{}{} }
172     \or: \str_collect_end_ii:nwn { {}{}{}{}{}{} }
173     \or: \str_collect_end_ii:nwn { {}{}{}{}{}{} }
174     \or: \str_collect_end_ii:nwn { {}{}{}{}{}{} }
175     \else: \str_collect_end_ii:nwn { {}{}{}{}{}{}{} }
176     \fi:
177 }
178 \cs_new:Npn \str_collect_end_ii:nwn #1 #2 \fi: #3
179 { \fi: #3 \q_stop #1 }
180 \cs_new:Npn \str_collect_end_iii:nwnNNNNNNN #1 #2 \q_stop #3#4#5#6#7#8#9
181 { #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 This is mostly shuffling arguments around to avoid measuring the length of the string  
 \str\_item:nn more than once, and make sure that the parameters given to \str\_skip\_do:nn are  
 \str\_item\_ignore\_spaces:nn necessarily within the bounds of the length of the string. The \str\_item\_ignore\_  
 \str\_item\_unsafe:nn spaces:nn function cheats a little bit in that it doesn't hand to \str\_item\_unsafe:nn a  
 \str\_item\_aux:nn truly "safe" string. This is alright, as everything else is done with undelimited arguments.

```

182 \cs_new_nopar:Npn \str_item:Nn { \exp_args:No \str_item:nn }
183 \cs_new:Npn \str_item:nn #1#2
184 {
185     \exp_args:Nf \tl_to_str:n
186     { \str_sanitize_args:Nn \str_item_unsafe:nn {#1} {#2} }
187 }
188 \cs_new:Npn \str_item_ignore_spaces:nn #1
189 { \exp_args:No \str_item_unsafe:nn { \tl_to_str:n {#1} } }
190 \cs_new_nopar:Npn \str_item_unsafe:nn #1#2
191 {
192     \exp_args:Nff \str_item_aux:nn
193     { \int_eval:n {#2} }
194     { \str_length_unsafe:n {#1} }
195     #1
196     \q_stop
197 }
198 \cs_new_nopar:Npn \str_item_aux:nn #1#2
199 {
200     \int_compare:nNnTF {#1} < \c_zero
201     {

```

```

202     \int_compare:nNnTF {#1} < {-#2}
203     { \use_none_delimit_by_q_stop:w }
204     {
205         \str_skip_do:nn { #1 + #2 }
206         { \use_i_delimit_by_q_stop:nw }
207     }
208 }
209 {
210     \int_compare:nNnTF {#1} < {#2}
211     {
212         \str_skip_do:nn {#1}
213         { \use_i_delimit_by_q_stop:nw }
214     }
215     { \use_none_delimit_by_q_stop:w }
216 }
217 }

```

(End definition for \str\_item:Nn. This function is documented on page ??.)

<pre> \str_from_to:Nnn \str_from_to:nnn \str_from_to_ignore_spaces:nnn \str_from_to_unsafe:nnn \str_from_to_aux:nnnw \str_from_to_aux_ii:nnw \str_aux_eval_args:Nnnn \str_aux_normalize_range:nn </pre>	<p>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.</p> <pre> 218 \cs_new_nopar:Npn \str_from_to:Nnn { \exp_args:No \str_from_to:nnn } 219 \cs_new:Npn \str_from_to:nnn #1#2#3 220 { 221     \exp_args:Nf \tl_to_str:n 222     { \str_sanitize_args:Nn \str_from_to_unsafe:nnn {#1}{#2}{#3} } 223 } 224 \cs_new:Npn \str_from_to_ignore_spaces:nnn #1 225 { \exp_args:No \str_from_to_unsafe:nnn { \tl_to_str:n {#1} } } 226 \cs_new:Npn \str_from_to_unsafe:nnn #1#2#3 % &lt;string&gt; &lt;from&gt; &lt;to&gt; 227 { 228     \str_aux_eval_args:Nnnn \str_from_to_aux:nnnw 229     { \str_length_unsafe:n {#1} } 230     {#2} 231     {#3} 232     #1 233     \q_stop 234 } 235 \cs_new:Npn \str_from_to_aux:nnnw #1#2#3 % &lt;len&gt; &lt;from&gt; &lt;to&gt; 236 { 237     \exp_args:Nf \str_from_to_aux_ii:nnw 238     { \str_aux_normalize_range:nn {#2} {#1} } 239     { \str_aux_normalize_range:nn {#3} {#1} } 240 } 241 \cs_new:Npn \str_from_to_aux_ii:nnw #1#2 242 { 243     \str_skip_do:nn {#1} 244     { </pre>
---	--

```

245     \exp_args:Nf \str_collect_do:nn
246     { \int_eval:n { #2 - #1 } }
247     { \use_i_delimit_by_q_stop:nw }
248   }
249 }
250 \cs_new:Npn \str_aux_eval_args:Nnnn #1#2#3#4
251 {
252   \exp_after:wN #1
253   \exp_after:wN { \int_value:w \int_eval:w #2 \exp_after:wN }
254   \exp_after:wN { \int_value:w \int_eval:w #3 \exp_after:wN }
255   \exp_after:wN { \int_value:w \int_eval:w #4 }
256 }
257 \cs_new:Npn \str_aux_normalize_range:nn #1#2
258 {
259   \int_eval:n
260   {
261     \if_num:w #1 < \c_zero
262     \if_num:w #1 < - #2 \exp_stop_f:
263       \c_zero
264     \else:
265       #1 + #2
266     \fi:
267   \else:
268     \if_num:w #1 < #2 \exp_stop_f:
269       #1
270     \else:
271       #2
272     \fi:
273   \fi:
274 }
275 }

```

(End definition for \str\_from\_to:Nnn. This function is documented on page ??.)

## 1.5 Internal mapping function

\str\_map\_tokens:nn We simply need to be careful with spaces. Spaces are fed to the mapped tokens with category code 12, not 10.

```

\str_map_tokens:Nn
\str_map_tokens_aux:nn
\str_map_tokens_loop:nN
\str_map_break_do:n
276 \cs_new_nopar:Npn \str_map_tokens:Nn
277 { \exp_args:No \str_map_tokens:nn }
278 \cs_new_nopar:Npn \str_map_tokens:nn
279 { \str_sanitiz_args:Nn \str_map_tokens_aux:nn }
280 \cs_new:Npn \str_map_tokens_aux:nn #1#2
281 {
282   \str_map_tokens_loop:nN {#2} #1
283   { ? \use_none_delimit_by_q_recursion_stop:w } \q_recursion_stop
284 }
285 \cs_new_nopar:Npn \str_map_tokens_loop:nN #1#2
286 {
287   \use_none:n #2

```

```

288     #1 #2
289     \str_map_tokens_loop:nN {#1}
290   }
291 \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 The nn and xx variants are already defined in l3basics.

```

\str_if_eq:nn 292 \prg_new_conditional:Npnn \str_if_eq:NN #1#2 { p , TF , T , F }
\str_if_eq:xx 293 {
294     \if_int_compare:w \pdfTeX_strcmp:D { \tl_to_str:N #1 } { \tl_to_str:N #2 }
295     = \c_zero \prg_return_true: \else: \prg_return_false: \fi:
296   }
      (End definition for \str_if_eq:NN. This function is documented on page ??.)

```

\str\_if\_contains\_char:NN Loop over the characters of the string, comparing character codes. We allow #2 to be  
\str\_if\_contains\_char:nN a single-character control sequence, hence the use of \int\_compare:nNt rather than  
\str\_if\_contains\_char\_aux:NN \token\_if\_eq\_charcode:NNT. The loop is broken if character codes match. Otherwise  
\str\_if\_contains\_char\_end:w we return “false”.

```

297 \prg_new_conditional:Npnn \str_if_contains_char:NN #1#2 { p , T , F , TF }
298 {
299     \str_map_tokens:Nn #1 { \str_if_contains_char_aux:NN #2 }
300     \prg_return_false:
301   }
302 \prg_new_conditional:Npnn \str_if_contains_char:nN #1#2 { p , T , F , TF }
303 {
304     \str_map_tokens:nn {#1} { \str_if_contains_char_aux:NN #2 }
305     \prg_return_false:
306   }
307 \cs_new_nopar:Npn \str_if_contains_char_aux:NN #1#2
308 {
309     \if_num:w '#1 = '#2 \exp_stop_f:
310     \str_if_contains_char_end:w
311     \fi:
312   }
313 \cs_new_nopar:Npn \str_if_contains_char_end:w \fi: #1 \prg_return_false:
314 { \fi: \prg_return_true: }
      (End definition for \str_if_contains_char:NN. This function is documented on page ??.)

```

\str\_if\_bytes:N Loop over the string, checking if every character code is less than 256.

```

\str_if_bytes_aux:N 315 \prg_new_conditional:Npnn \str_if_bytes:N #1 { p , T , F , TF }
316 {
317     \str_map_tokens:Nn #1 { \str_if_bytes_aux:N }
318     \prg_return_true:
319   }
320 \cs_new_nopar:Npn \str_if_bytes_aux:N #1
321 {

```



```

322 \int_compare:nNnF {'#1} < \c_two_hundred_fifty_six
323 {
324   \use_i_delimit_by_q_recursion_stop:nw
325   { \prg_return_false: \use_none:n }
326 }
327 }

```

(End definition for \str\_if\_bytes:N. This function is documented on page ??.)

## 1.7 Internal conditionals

**\str\_aux\_hexadecimal\_test:N** 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).

```

328 \prg_new_protected_conditional:Npnn \str_aux_hexadecimal_test:N #1 { TF }
329 {
330   \tl_if_in:onTF { \tl_to_str:n {abcdef} } {'#1}
331   {
332     \int_set:Nn \l_str_char_int
333     { \c_sixteen * \l_str_char_int + '#1 - 87 }
334     \prg_return_true:
335   }
336   {
337     \if_num:w \c_fifteen < "1 \exp_not:N #1 \exp_stop_f:
338     \int_set:Nn \l_str_char_int
339     { \c_sixteen * \l_str_char_int + "#1 }
340     \prg_return_true:
341     \else:
342     \prg_return_false:
343     \fi:
344   }
345 }

```

(End definition for \str\_aux\_hexadecimal\_test:N. This function is documented on page ??.)

**\str\_aux\_octal\_test:N** 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).

```

346 \prg_new_protected_conditional:Npnn \str_aux_octal_test:N #1 { TF }
347 {
348   \if_num:w \c_seven < '1 \exp_not:N #1 \exp_stop_f:
349   \int_set:Nn \l_str_char_int
350   { \c_eight * \l_str_char_int + '#1 }
351   \prg_return_true:
352   \else:
353   \prg_return_false:
354   \fi:
355 }

```

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

`\str_aux_char_if_octal_digit:N`

```

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

```

(End definition for `\str_aux_char_if_octal_digit:N`. 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` Simple wrappers around the internal `\str_aux_escape:NNNn`.

```

\str_ginput:Nn
358 \cs_new_protected:Npn \str_input:Nn #1#2
359   {
360     \str_aux_escape:NNNn \token_to_str:N \token_to_str:N \token_to_str:N {#2}
361     \tl_set_eq:NN #1 \g_str_result_tl
362   }
363 \cs_new_protected:Npn \str_ginput:Nn #1#2
364   {
365     \str_aux_escape:NNNn \token_to_str:N \token_to_str:N \token_to_str:N {#2}
366     \tl_gset_eq:NN #1 \g_str_result_tl
367   }

```

(End definition for `\str_input:Nn` and `\str_ginput:Nn`. These functions are documented on page 2.)

`\str_aux_escape:NNNn` Treat the argument as an *escaped string*, and store the corresponding *string* globally in `\g_str_result_tl`.

```

\str_aux_escape_unescaped:N
\str_aux_escape_escaped:N
\str_aux_escape_raw:N
\str_aux_escape_loop:N
\str_aux_escape_\:w
368 \cs_new_eq:NN \str_aux_escape_unescaped:N \use:n
369 \cs_new_eq:NN \str_aux_escape_escaped:N \use:n
370 \cs_new_eq:NN \str_aux_escape_raw:N \use:n
371 \cs_new_protected:Npn \str_aux_escape:NNNn #1#2#3#4
372   {
373     \group_begin:
374     \cs_set_nopar:Npn \str_aux_escape_unescaped:N { #1 }

```

```

375 \cs_set_nopar:Npn \str_aux_escape_escaped:N { #2 }
376 \cs_set_nopar:Npn \str_aux_escape_raw:N { #3 }
377 \int_set:Nn \tex_escapechar:D {92}
378 \tl_gset:Nx \g_str_result_tl { \tl_to_other_str:n {#4} }
379 \tl_gset:Nx \g_str_result_tl
380 {
381 \exp_after:wN \str_aux_escape_loop:N \g_str_result_tl
382 \q_recursion_tail \q_recursion_stop
383 }
384 \group_end:
385 }
386 \cs_new_nopar:Npn \str_aux_escape_loop:N #1
387 {
388 \cs_if_exist_use:cF { str_aux_escape_\token_to_str:N #1:w }
389 { \str_aux_escape_unescaped:N #1 }
390 \str_aux_escape_loop:N
391 }
392 \cs_new_nopar:cpn { str_aux_escape_ \c_backslash_str :w }
393 \str_aux_escape_loop:N #1
394 {
395 \cs_if_exist_use:cF { str_aux_escape_/ \token_to_str:N #1:w }
396 { \str_aux_escape_escaped:N #1 }
397 \str_aux_escape_loop:N
398 }

```

*(End definition for \str\_aux\_escape:NNNn. This function is documented on page ??.)*

\str\_aux\_escape\_\q\_recursion\_tail:w The loop is ended upon seeing \q\_recursion\_tail. Spaces are ignored, and \a, \e, \f, \n, \r, \t take their meaning here.

```

\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
399 \cs_new_eq:cN
400 { str_aux_escape_ \c_backslash_str q_recursion_tail :w }
401 \use_none_delimit_by_q_recursion_stop:w
402 \cs_new_eq:cN
403 { str_aux_escape_ / \c_backslash_str q_recursion_tail :w }
404 \use_none_delimit_by_q_recursion_stop:w
405 \cs_new_nopar:cpn { str_aux_escape_~:w } { }
406 \cs_new_nopar:cpx { str_aux_escape_/a:w }
407 { \exp_not:N \str_aux_escape_raw:N \iow_char:N \^^G }
408 \cs_new_nopar:cpx { str_aux_escape_/t:w }
409 { \exp_not:N \str_aux_escape_raw:N \iow_char:N \^^I }
410 \cs_new_nopar:cpx { str_aux_escape_/n:w }
411 { \exp_not:N \str_aux_escape_raw:N \iow_char:N \^^J }
412 \cs_new_nopar:cpx { str_aux_escape_/f:w }
413 { \exp_not:N \str_aux_escape_raw:N \iow_char:N \^^L }
414 \cs_new_nopar:cpx { str_aux_escape_/r:w }
415 { \exp_not:N \str_aux_escape_raw:N \iow_char:N \^^M }
416 \cs_new_nopar:cpx { str_aux_escape_/e:w }
417 { \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 ??.)*

<code>\str_aux_escape_/x:w</code> <code>\str_aux_escape_x_test:N</code> <code>\str_aux_escape_x_unbraced_i:N</code> <code>\str_aux_escape_x_unbraced_ii:N</code> <code>\str_aux_escape_x_braced_loop:N</code> <code>\str_aux_escape_x_braced_end:N</code> <code>\str_aux_escape_x_end:</code>	<p>When <code>\x</code> 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 <code>\l_str_char_int</code> (this is a side effect of <code>\str_aux_hexadecimal_test:N</code>), 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 <code>\l_str_char_int</code>. In both cases, once all digits have been collected, use the TeX primitive <code>\lowercase</code> to produce that character, and use a <code>\if_false:</code> trick to restart the assignment.</p>
---	--

```

418 \cs_new_nopar:cpn { str_aux_escape_/x:w } \str_aux_escape_loop:N
419 {
420   \if_false: { \fi: }
421   \int_zero:N \l_str_char_int
422   \str_aux_escape_x_test:N
423 }
424 \cs_new_protected_nopar:Npx \str_aux_escape_x_test:N #1
425 {
426   \exp_not:N \token_if_eq_charcode:NNTF \c_space_token #1
427   { \exp_not:N \str_aux_escape_x_test:N }
428   {
429     \exp_not:N \token_if_eq_charcode:NNTF \c_lbrace_str #1
430     { \exp_not:N \str_aux_escape_x_braced_loop:N }
431     { \exp_not:N \str_aux_escape_x_unbraced_i:N #1 }
432   }
433 }
434 \cs_new_protected_nopar:Npn \str_aux_escape_x_unbraced_i:N #1
435 {
436   \str_aux_hexadecimal_test:NNTF #1
437   { \str_aux_escape_x_unbraced_ii:N }
438   { \str_aux_escape_x_end: #1 }
439 }
440 \cs_new_protected_nopar:Npn \str_aux_escape_x_unbraced_ii:N #1
441 {
442   \token_if_eq_charcode:NNTF \c_space_token #1
443   { \str_aux_escape_x_unbraced_ii:N }
444   {
445     \str_aux_hexadecimal_test:NNTF #1
446     { \str_aux_escape_x_end: }
447     { \str_aux_escape_x_end: #1 }
448   }
449 }
450 \cs_new_protected_nopar:Npn \str_aux_escape_x_braced_loop:N #1
451 {
452   \token_if_eq_charcode:NNTF \c_space_token #1
453   { \str_aux_escape_x_braced_loop:N }
454   {
455     \str_aux_hexadecimal_test:NNTF #1
456     { \str_aux_escape_x_braced_loop:N }
457     { \str_aux_escape_x_braced_end:N #1 }
458   }

```

```

459 }
460 \cs_new_protected_nopar:Npx \str_aux_escape_x_braced_end:N #1
461 {
462   \exp_not:N \token_if_eq_charcode:NNTF \c_rbrace_str #1
463   { \exp_not:N \str_aux_escape_x_end: }
464   {
465     \msg_error:nn { str } { x-missing-brace }
466     \exp_not:N \str_aux_escape_x_end: #1
467   }
468 }
469 \group_begin:
470   \char_set_catcode_other:N ^^@
471   \cs_new_protected_nopar:Npn \str_aux_escape_x_end:
472   {
473     \group_begin:
474     \char_set_lccode:nn { \c_zero } { \l_str_char_int }
475     \tl_to_lowercase:n
476     {
477       \group_end:
478       \tl_gput_right:Nx \g_str_result_tl
479       { \if_false: } \fi:
480       \str_aux_escape_raw:N ^^@
481       \str_aux_escape_loop:N
482     }
483   }
484 \group_end:
485 \msg_new:nnn { str } { x-missing-brace }
486 {
487   You-wrote-something-like~
488   ‘\iow_char:N\x\{ \int_to_hexadecimal:n { \l_str_char_int }’.-
489   The-closing-brace-is-missing.
490 }

```

*(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
491 \group_begin:
492   \char_set_catcode_other:n {‘^^@}
493   \cs_new_protected_nopar:Npn \str_aux_convert_store:
494   {
495     \char_set_lccode:nn { \c_zero } { \l_str_char_int }
496     \tl_to_lowercase:n { \tl_gput_right:Nx \g_str_result_tl {^^@} }
497   }
498   \cs_new_protected_nopar:Npn \str_aux_convert_store:NNn #1#2#3
499   {
500     \char_set_lccode:nn { \c_zero } { #3 }
501     \tl_to_lowercase:n { #1 #2 {^^@} }
502   }

```

```

503 \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
504 \cs_new_nopar:Npn \str_aux_byte_to_hexadecimal:N #1
505 {
506     \int_compare:nNnTF {'#1} < {256}
507     {
508         \int_to_letter:n { \int_div_truncate:nn {'#1} \c_sixteen }
509         \int_to_letter:n { \int_mod:nn {'#1} \c_sixteen }
510     }
511     { \msg_expandable_error:n { Invalid~byte~'#1'. } }
512 }
513 \cs_new_nopar:Npn \str_aux_byte_to_octal:N #1
514 {
515     \int_compare:nNnTF {'#1} < {64}
516     {
517         0
518         \int_to_letter:n { \int_div_truncate:nn {'#1} \c_eight }
519         \int_to_letter:n { \int_mod:nn {'#1} \c_eight }
520     }
521     {
522         \int_compare:nNnTF {'#1} < {256}
523         { \int_to_octal:n {'#1} }
524         { \msg_expandable_error:n { Invalid~byte~'#1'. } }
525     }
526 }
    (End definition for \str_aux_byte_to_hexadecimal:N. This function is documented on page ??.)

```

\str\_bytes\_escape\_hexadecimal:NN

```

527 \cs_new_protected_nopar:Npn \str_bytes_escape_hexadecimal:NN #1#2
528 {
529     \str_set:Nx #1
530     { \str_map_tokens:Nn #2 \str_aux_byte_to_hexadecimal:N }
531 }
    (End definition for \str_bytes_escape_hexadecimal:NN. This function is documented on page 5.)

```

\str\_bytes\_escape\_name:NN

```

532 \tl_const:Nx \c_str_bytes_escape_name_str
533 { \c_hash_str \c_percent_str \c_lbrace_str \c_rbrace_str () /<>[] }
534 \cs_new_protected_nopar:Npn \str_bytes_escape_name:NN #1#2
535 {
536     \str_set:Nx #1
537     { \str_map_tokens:Nn #2 \str_bytes_escape_name_aux:N }
538 }
539 \cs_new_nopar:Npn \str_bytes_escape_name_aux:N #1
540 {
541     \int_compare:nNnTF {'#1} < {"21}
542     { \c_hash_str \str_aux_byte_to_hexadecimal:N #1 }

```

```

543     {
544         \int_compare:nNnTF {'#1} > {"7E}
545         { \c_hash_str \str_aux_byte_to_hexadecimal:N #1 }
546         {
547             \str_if_contains_char:NNTF \c_str_bytes_escape_name_str #1
548             { \c_hash_str \str_aux_byte_to_hexadecimal:N #1 }
549             {#1}
550         }
551     }
552 }

```

(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.

```

553 \tl_const:Nx \c_str_bytes_escape_string_str { \c_backslash_str ( ) }
554 \cs_new_protected_nopar:Npn \str_bytes_escape_string:NN #1#2
555 {
556     \str_set:Nx #1
557     { \str_map_tokens:Nn #2 \str_bytes_escape_string_aux:N }
558 }
559 \cs_new_nopar:Npn \str_bytes_escape_string_aux:N #1
560 {
561     \int_compare:nNnTF {'#1} < {"21}
562     { \c_backslash_str \str_aux_byte_to_octal:N #1 }
563     {
564         \int_compare:nNnTF {'#1} > {"7E}
565         { \c_backslash_str \str_aux_byte_to_octal:N #1 }
566         {
567             \str_if_contains_char:NNTF \c_str_bytes_escape_string_str #1
568             { \c_backslash_str }
569             #1
570         }
571     }
572 }

```

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

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

```

573 \cs_new_protected_nopar:Npn \str_bytes_unescape_hexadecimal:NN #1#2
574 {
575     \group_begin:
576     \tl_gclear:N \g_str_result_tl
577     \int_zero:N \l_str_char_int
578     \exp_last_unbraced:Nf \str_bytes_unescape_hexadecimal_aux:N
579     { \tl_to_other_str:N #2 } 0
580     \q_recursion_tail \q_recursion_stop
581     \group_end:
582     \tl_set_eq:NN #1 \g_str_result_tl

```

```

583     }
584     \cs_new_protected_nopar:Npn \str_bytes_unescape_hexadecimal_aux:N #1
585     {
586         \quark_if_recursion_tail_stop:N #1
587         \str_aux_hexadecimal_test:NTF #1
588         { \str_bytes_unescape_hexadecimal_aux_ii:N }
589         { \str_bytes_unescape_hexadecimal_aux:N }
590     }
591     \cs_new_protected_nopar:Npn \str_bytes_unescape_hexadecimal_aux_ii:N #1
592     {
593         \quark_if_recursion_tail_stop:N #1
594         \str_aux_hexadecimal_test:NTF #1
595         {
596             \str_aux_convert_store:
597             \int_zero:N \l_str_char_int
598             \str_bytes_unescape_hexadecimal_aux:N
599         }
600         { \str_bytes_unescape_hexadecimal_aux_ii:N }
601     }

```

(End definition for \str\_bytes\_unescape\_hexadecimal:NN. This function is documented on page ??.)

\str\_bytes\_unescape\_name:NN This changes all occurrences of # followed by two upper- or lowercase hexadecimal digits by the corresponding unescaped character.

```

\str_bytes_unescape_name_aux:wN
\str_bytes_unescape_name_aux_ii:N
602 \group_begin:
603   \char_set_lccode:nn {'\*} {'\#}
604   \tl_to_lowercase:n
605   {
606     \group_end:
607     \cs_new_protected_nopar:Npn \str_bytes_unescape_name:NN #1#2
608     {
609       \group_begin:
610         \tl_gclear:N \g_str_result_tl
611         \int_zero:N \l_str_char_int
612         \exp_last_unbraced:Nf \str_bytes_unescape_name_aux:wN
613         { \tl_to_other_str:N #2 }
614         * \q_recursion_tail \q_recursion_stop
615       \group_end:
616       \tl_set_eq:NN #1 \g_str_result_tl
617     }
618     \cs_new_protected_nopar:Npn \str_bytes_unescape_name_aux:wN #1 * #2
619     {
620       \tl_gput_right:Nx \g_str_result_tl {#1}
621       \quark_if_recursion_tail_stop:N #2
622       \str_aux_hexadecimal_test:NTF #2
623       { \str_bytes_unescape_name_aux_ii:NN #2 }
624       {
625         \tl_gput_right:Nx \g_str_result_tl { \c_hash_str }
626         \str_bytes_unescape_name_aux:wN #2
627       }

```



```

628     }
629 }
630 \cs_new_protected_nopar:Npn \str_bytes_unescape_name_aux:ii:NN #1#2
631 {
632   \str_aux_hexadecimal_test:NTF #2
633   {
634     \str_aux_convert_store:
635     \int_zero:N \l_str_char_int
636     \str_bytes_unescape_name_aux:wN
637   }
638   {
639     \tl_gput_right:Nx \g_str_result_tl { \c_hash_str #1 }
640     \int_zero:N \l_str_char_int
641     \str_bytes_unescape_name_aux:wN #2
642   }
643 }

```

(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.

```

\l 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.*

```

644 \group_begin:
645   \char_set_lccode:nn {'\*} {'\\}
646   \char_set_catcode_other:N ^^J
647   \char_set_catcode_other:N ^^M
648   \tl_to_lowercase:n
649   {
650     \group_end:
651     \cs_new_protected_nopar:Npn \str_bytes_unescape_string:NN #1#2
652     {
653       \group_begin:

```

```

654         \tl_gclear:N \g_str_result_tl
655         \exp_last_unbraced:Nf \str_bytes_unescape_string_aux:wN
656         { \tl_to_other_str:N #2 }
657         * \q_recursion_tail \q_recursion_stop
658     \group_end:
659     \tl_set_eq:NN #1 \g_str_result_tl
660 }
661 \cs_new_protected_nopar:Npn \str_bytes_unescape_string_aux:wN #1 * #2
662 {
663     \tl_gput_right:Nx \g_str_result_tl {#1}
664     \quark_if_recursion_tail_stop:N #2
665     \int_zero:N \l_str_char_int
666     \str_aux_octal_test:NTF #2
667     { \str_bytes_unescape_string_aux_d:N }
668     {
669         \int_set:Nn \l_str_char_int
670         {
671             \prg_case_str:xxn {#2}
672             {
673                 {n} {10}
674                 {r} {13}
675                 {t} {9}
676                 {b} {8}
677                 {f} {12}
678                 {(} {40}
679                 {)} {41}
680                 {\c_backslash_str} {92}
681                 {\c_caron} {-1}
682                 {\c_tilde} {-1}
683             }
684             {'#2}
685         }
686         \int_compare:nNnF \l_str_char_int = \c_minus_one
687         { \str_aux_convert_store: }
688         \str_bytes_unescape_string_aux:wN
689     }
690 }
691 }
692 \cs_new_protected_nopar:Npn \str_bytes_unescape_string_aux_d:N #1
693 {
694     \str_aux_octal_test:NTF #1
695     { \str_bytes_unescape_string_aux_dd:N }
696     {
697         \str_aux_convert_store:
698         \str_bytes_unescape_string_aux:wN #1
699     }
700 }
701 \cs_new_protected_nopar:Npn \str_bytes_unescape_string_aux_dd:N #1
702 {
703     \str_aux_octal_test:NTF #1

```

```

704     {
705         \str_aux_convert_store:
706         \str_bytes_unescape_string_aux:wN
707     }
708     {
709         \str_aux_convert_store:
710         \str_bytes_unescape_string_aux:wN #1
711     }
712 }

```

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

### 1.8.3 Percent encoding

`\str_bytes_percent_encode:NN`

```

713 \tl_const:Nx \c_str_bytes_percent_encode_str { \tl_to_str:n { [] } }
714 \tl_const:Nx \c_str_bytes_percent_encode_not_str { \tl_to_str:n { "-.<> } }
715 \cs_new_protected_nopar:Npn \str_bytes_percent_encode:NN #1#2
716 {
717     \str_set:Nx #1
718     { \str_map_tokens:Nn #2 \str_bytes_percent_encode_aux:N }
719 }
720 \cs_new_nopar:Npn \str_bytes_percent_encode_aux:N #1
721 {
722     \int_compare:nNnTF {'#1} < {"41}
723     {
724         \str_if_contains_char:NNTF \c_str_bytes_percent_encode_not_str #1
725         { #1 }
726         { \c_percent_str \str_aux_byte_to_hexadecimal:N #1 }
727     }
728     {
729         \int_compare:nNnTF {'#1} > {"7E}
730         { \c_percent_str \str_aux_byte_to_hexadecimal:N #1 }
731         {
732             \str_if_contains_char:NNTF \c_str_bytes_percent_encode_str #1
733             { \c_percent_str \str_aux_byte_to_hexadecimal:N #1 }
734             { #1 }
735         }
736     }
737 }

```

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

`\str_bytes_percent_decode:NN`

```

\str_bytes_percent_decode_aux:wN
\str_bytes_percent_decode_aux_ii:NN
738 \group_begin:
739 \char_set_lccode:nn {'\*} {'\%}
740 \tl_to_lowercase:n
741 {
742     \group_end:
743     \cs_new_protected_nopar:Npn \str_bytes_percent_decode:NN #1#2
744     {

```

```

745     \group_begin:
746     \tl_gclear:N \g_str_result_tl
747     \exp_last_unbraced:Nf \str_bytes_percent_decode_aux:wN
748     { \tl_to_other_str:N #2 }
749     * \q_recursion_tail \q_recursion_stop
750     \group_end:
751     \tl_set_eq:NN #1 \g_str_result_tl
752   }
753   \cs_new_protected_nopar:Npn \str_bytes_percent_decode_aux:wN #1 * #2
754   {
755     \tl_gput_right:Nx \g_str_result_tl {#1}
756     \quark_if_recursion_tail_stop:N #2
757     \int_zero:N \l_str_char_int
758     \str_aux_hexadecimal_test:NTF #2
759     { \str_bytes_percent_decode_aux_ii:NN #2 }
760     {
761       \tl_gput_right:Nx \g_str_result_tl { \c_percent_str }
762       \str_bytes_percent_decode_aux:wN #2
763     }
764   }
765 }
766 \cs_new_protected_nopar:Npn \str_bytes_percent_decode_aux_ii:NN #1#2
767 {
768   \str_aux_hexadecimal_test:NTF #2
769   {
770     \str_aux_convert_store:
771     \str_bytes_percent_decode_aux:wN
772   }
773   {
774     \tl_gput_right:Nx \g_str_result_tl {#1}
775     \str_bytes_percent_decode_aux:wN #2
776   }
777 }

```

(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

```

778 \cs_new_protected_nopar:Npn \str_native_from_UTF_viii:NN #1#2
779 {
780   \group_begin:
781   \tl_gclear:N \g_str_result_tl
782   \exp_last_unbraced:Nf \str_native_from_UTF_viii_aux_i:N
783   { \tl_to_other_str:N #2 }
784   \q_recursion_tail \q_recursion_stop
785   \group_end:
786   \tl_set_eq:NN #1 \g_str_result_tl
787 }
788 \cs_new_protected_nopar:Npn \str_native_from_UTF_viii_aux_i:N #1

```

```

789 {
790   \quark_if_recursion_tail_stop:N #1
791   \int_set:Nn \l_str_char_int { '#1 }
792   \str_native_from_UTF_viii_aux_ii:nN {128} \c_one
793   \str_native_from_UTF_viii_aux_ii:nN {64} \c_zero
794   \str_native_from_UTF_viii_aux_ii:nN {32} \c_two
795   \str_native_from_UTF_viii_aux_ii:nN {16} \c_three
796   \str_native_from_UTF_viii_aux_ii:nN {8} \c_four
797   \msg_error:nnx { str } { utf8-invalid-byte } {#1}
798   \use_i:nnn \str_native_from_UTF_viii_aux_i:N
799   \q_stop
800   \str_native_from_UTF_viii_aux_iii:N
801 }
802 \cs_new_protected_nopar:Npn
803   \str_native_from_UTF_viii_aux_ii:nN #1#2
804   {
805     \int_compare:nNnTF \l_str_char_int < {#1}
806     {
807       \int_set_eq:NN \l_str_bytes_int #2
808       \use_none_delimit_by_q_stop:w
809     }
810     { \int_sub:Nn \l_str_char_int {#1} }
811   }
812 \cs_new_protected_nopar:Npn
813   \str_native_from_UTF_viii_aux_iii:N #1
814   {
815     \int_compare:nNnTF \l_str_bytes_int < \c_two
816     { \str_native_from_UTF_viii_aux_iv: #1 }
817     {
818       \if_meaning:w \q_recursion_tail #1
819       \msg_error:nn { str } { utf8-premature-end }
820       \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
821       \fi:
822       \quark_if_recursion_tail_stop:N #1
823       \tex_multiply:D \l_str_char_int by 64 \scan_stop: % no interface??
824       \int_decr:N \l_str_bytes_int
825       \int_compare:nNnTF {'#1} < {128}
826       { \use_ii:nn }
827       { \int_compare:nNnTF {'#1} < {192} }
828       {
829         \int_add:Nn \l_str_char_int { '#1 - 128 }
830         \str_native_from_UTF_viii_aux_iii:N
831       }
832       {
833         \msg_error:nnx { str } { utf8-missing-byte } {#1}
834         \str_native_from_UTF_viii_aux_i:N #1
835       }
836     }
837   }
838 \cs_new_protected_nopar:Npn

```

```

839 \str_native_from_UTF_viii_aux_iv:
840 {
841   \int_compare:nNnTF \l_str_bytes_int = \c_zero
842   {
843     \msg_error:nnx { str } { utf8-extra-byte }
844     { \int_eval:n { \l_str_char_int + 128 } }
845   }
846   {
847     \pdfTeX_if_engine:TF
848     {
849       \int_compare:nNnTF { \l_str_char_int } < { 256 }
850       { \str_aux_convert_store: }
851       {
852         \msg_error:nnx { str } { utf8-pdfTeX-overflow }
853         { \int_use:N \l_str_char_int }
854       }
855     }
856     { \str_aux_convert_store: }
857   }
858   \str_native_from_UTF_viii_aux_i:N
859 }
860 \msg_new:nnn { str } { utf8-invalid-byte }
861 {
862   \int_compare:nNnTF {'#1} < {256}
863   { Byte-number~\int_eval:n {'#1}~invalid-in~utf-8~encoding. }
864   { The~character~number~\int_eval:n {'#1}~is~not~a~byte. }
865 }
866 \msg_new:nnn { str } { utf8-missing-byte }
867 { The~byte~number~\int_eval:n {'#1}~is~not~a~valid~continuation~byte. }
868 \msg_new:nnn { str } { utf8-extra-byte }
869 { The~byte~number~\int_eval:n {'#1}~is~only~valid~as~a~continuation~byte. }
870 \msg_new:nnnn { str } { utf8-premature-end }
871 { Incomplete~last~utf8~character. }
872 {
873   The~sequence~of~byte~that~I~need~to~convert~to~utf8 \
874   ended~before~the~last~character~was~complete.~Perhaps \
875   it~was~mistakenly~truncated?
876 }
877 \msg_new:nnn { str } { utf8-pdfTeX-overflow }
878 { The~character~number~#1~is~too~big~for~pdfTeX. }

```

(End definition for \str\_native\_from\_UTF\_viii:NN. This function is documented on page 5.)

\str\_UTF\_viii\_from\_native:NN

```

879 \cs_new_protected_nopar:Npn \str_UTF_viii_from_native:NN #1#2
880 {
881   \group_begin:
882   \tl_gclear:N \g_str_result_tl
883   \str_map_tokens:Nn #2 \str_UTF_viii_from_native_aux_i:N
884   \group_end:
885   \tl_set_eq:NN #1 \g_str_result_tl

```

```

886 }
887 \cs_new_protected_nopar:Npn \str_UTF_viii_from_native_aux_i:N #1
888 {
889   \int_set:Nn \l_str_char_int {'#1}
890   \int_compare:nNnTF \l_str_char_int < {128}
891     { \tl_gput_right:Nx \g_str_result_tl {#1} }
892     {
893       \tl_gclear:N \g_str_tmpa_tl
894       \int_set:Nn \l_str_bytes_int { 64 }
895       \str_UTF_viii_from_native_aux_ii:n {32}
896       \str_UTF_viii_from_native_aux_ii:n {16}
897       \str_UTF_viii_from_native_aux_ii:n {8}
898       \ERROR % somehow the unicode char was > "1FFFFFF > "10FFFF
899       \tl_gclear:N \g_str_tmpa_tl
900       \use_none:n \q_stop
901       \int_add:Nn \l_str_char_int { 2 * \l_str_bytes_int }
902       \str_aux_convert_store:
903       \tl_gput_right:Nx \g_str_result_tl { \g_str_tmpa_tl }
904     }
905 }
906 \cs_new_protected_nopar:Npn \str_UTF_viii_from_native_aux_ii:n #1
907 {
908   \str_aux_convert_store:NNn
909   \tl_gput_left:Nx \g_str_tmpa_tl
910     { 128 + \int_mod:nn { \l_str_char_int } {64} }
911   \tex_divide:D \l_str_char_int by 64 \scan_stop: % no interface??
912   \int_add:Nn \l_str_bytes_int {#1}
913   \int_compare:nNnTF \l_str_char_int < {#1}
914     { \use_none_delimit_by_q_stop:w }
915 }

```

(End definition for \str\_UTF\_viii\_from\_native:NN. This function is documented on page 5.)

## 1.9 Deprecated string functions

`\str_length_skip_spaces:N` The naming scheme is a little bit more consistent with “ignore\_spaces” instead of “skip\_spaces”.

```

916 \cs_gset:Npn \str_length_skip_spaces:N
917   { \exp_args:No \str_length_skip_spaces:n }
918 \cs_gset_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 ??.)

```

919 \</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.

Symbols	
<code>\#</code> .....	31, 603
<code>\%</code> .....	32, 739
<code>\*</code> .....	34, 603, 645, 739
<code>\\</code> .....	28, 488, 645
<code>\{</code> .....	29, 488
<code>\}</code> .....	30
<code>\^</code> .....	407, 409, 411, 413, 415, 417, 470, 492, 646, 647
<code>\_</code> .....	34, 873, 874
A	
<code>\A</code> .....	35
C	
<code>\c_backslash_str</code> .....	1, 28, 28, 392, 400, 403, 553, 562, 565, 568, 680
<code>\c_eight</code> .....	82, 350, 518, 519
<code>\c_fifteen</code> .....	337
<code>\c_five</code> .....	83
<code>\c_four</code> .....	83, 796
<code>\c_hash_str</code> .....	1, 28, 31, 533, 542, 545, 548, 625, 639
<code>\c_lbrace_str</code> .....	1, 28, 29, 429, 533
<code>\c_minus_one</code> .....	686
<code>\c_nine</code> .....	93
<code>\c_one</code> .....	84, 792
<code>\c_percent_str</code> .....	..... 1, 28, 32, 533, 726, 730, 733, 761
<code>\c_rbrace_str</code> .....	1, 28, 30, 462, 533
<code>\c_seven</code> .....	82, 126, 132, 153, 161, 348
<code>\c_six</code> .....	82
<code>\c_sixteen</code> .....	333, 339, 508, 509
<code>\c_space_token</code> .....	426, 442, 452
<code>\c_str_bytes_escape_name_str</code> ..	532, 547
<code>\c_str_bytes_escape_string_str</code>	553, 567
<code>\c_str_bytes_percent_encode_not_str</code> .....	714, 724
<code>\c_str_bytes_percent_encode_str</code>	713, 732
<code>\c_three</code> .....	83, 795
<code>\c_two</code> .....	84, 794, 815
<code>\c_two_hundred_fifty_six</code> .....	322
<code>\c_zero</code> .....	84, 200, 261, 263, 295, 474, 495, 500, 793, 841
<code>\char_set_catcode_other:N</code> ..	470, 646, 647
<code>\char_set_catcode_other:n</code> .....	492
<code>\char_set_lccode:nn</code> .....	.. 34, 35, 474, 495, 500, 603, 645, 739
<code>\cs_generate_variant:Nn</code> .....	15
<code>\cs_gset:Npn</code> .....	916
<code>\cs_gset_eq:NN</code> .....	918
<code>\cs_if_exist:cTF</code> .....	7
<code>\cs_if_exist:NF</code> .....	4, 9
<code>\cs_if_exist_use:cF</code> .....	4, 4, 6, 388, 395
<code>\cs_new:Npn</code> ..	6, 10, 38, 57, 62, 69, 72, 77, 104, 116, 124, 131, 133, 147, 149, 151, 158, 164, 178, 180, 183, 188, 219, 224, 226, 235, 241, 250, 257, 280
<code>\cs_new_eq:cN</code> .....	399, 402
<code>\cs_new_eq:NN</code> .....	291, 368–370
<code>\cs_new_nopar:cpn</code> .....	392, 405, 418
<code>\cs_new_nopar:cpx</code> .....	..... 406, 408, 410, 412, 414, 416
<code>\cs_new_nopar:Npn</code> .....	43, 52, 55, 68, 70, 106, 108, 121, 182, 190, 198, 218, 276, 278, 285, 307, 313, 320, 386, 504, 513, 539, 559, 720
<code>\cs_new_protected:cpx</code> .....	13
<code>\cs_new_protected:Npn</code> .....	358, 363, 371
<code>\cs_new_protected_nopar:Npn</code> .....	..... 356, 434, 440, 450, 471, 493, 498, 527, 534, 554, 573, 584, 591, 607, 618, 630, 651, 661, 692, 701, 715, 743, 753, 766, 778, 788, 802, 812, 838, 879, 887, 906
<code>\cs_new_protected_nopar:Npx</code> ...	424, 460
<code>\cs_set:Npn</code> .....	11, 88, 96, 109
<code>\cs_set_nopar:Npn</code> ..	95, 115, 123, 374–376
<code>\cs_set_nopar:Npx</code> .....	102
<code>\cs_set_protected_nopar:Npn</code> .....	23
<code>\cs_to_str:N</code> .....	28–32
E	
<code>\else:</code> ..	144, 175, 264, 267, 270, 295, 341, 352
<code>\ERROR</code> .....	898





\prg_new_protected_conditional:Npnn	328, 346	\str_aux_escape_loop:N	368, 381, 386, 390, 393, 397, 418, 481
\prg_return_false:	295, 300, 305, 313, 325, 342, 353	\str_aux_escape_raw:N	368, 370, 376, 407, 409, 411, 413, 415, 417, 480
\prg_return_true:	295, 314, 318, 334, 340, 351	\str_aux_escape_unescaped:N	368, 368, 374, 389
\ProvidesExplPackage	2	\str_aux_escape_x_braced_end:N	418, 457, 460
<b>Q</b>			
\q_mark	41, 52	\str_aux_escape_x_braced_loop:N	418, 430, 450, 453, 456
\q_recursion_stop	283, 382, 580, 614, 657, 749, 784	\str_aux_escape_x_end:	418, 438, 446, 447, 463, 466, 471
\q_recursion_tail	382, 580, 614, 657, 749, 784, 818	\str_aux_escape_x_test:N	418, 422, 424, 427
\q_stop	41, 44, 50, 52, 85, 100, 105, 107, 113, 115, 119, 122, 123, 179, 180, 196, 233, 799, 900	\str_aux_escape_x_unbraced_i:N	418, 431, 434
\quark_if_recursion_tail_stop:N	586, 593, 621, 664, 756, 790, 822	\str_aux_escape_x_unbraced_ii:N	418, 437, 440, 443
<b>R</b>			
\reverse_if:N	112	\str_aux_eval_args:Nnnn	218, 228, 250
<b>S</b>			
\scan_stop:	113, 823, 911	\str_aux_hexadecimal_test:N	328, 328
\str_aux_byte_to_hexadecimal:N	504, 504, 530, 542, 545, 548, 726, 730, 733	\str_aux_hexadecimal_test:NTF	436, 445, 455, 587, 594, 622, 632, 758, 768
\str_aux_byte_to_octal:N	504, 513, 562, 565	\str_aux_normalize_range:nn	218, 238, 239, 257
\str_aux_char_if_octal_digit:NTF	356, 356	\str_aux_octal_test:N	346, 346
\str_aux_convert_store:	491, 493, 596, 634, 687, 697, 705, 709, 770, 850, 856, 902	\str_aux_octal_test:NTF	666, 694, 703
\str_aux_convert_store:NNn	491, 498, 908	\str_bytes_escape_hexadecimal:NN	5, 527, 527
\str_aux_escape:NNNn	7, 360, 365, 368, 371	\str_bytes_escape_name:NN	6, 532, 534
\str_aux_escape_□:w	399	\str_bytes_escape_name_aux:N	537, 539
\str_aux_escape_/\q_recursion_tail:w	399	\str_bytes_escape_string:NN	6, 553, 554
\str_aux_escape_/a:w	399	\str_bytes_escape_string_aux:N	557, 559
\str_aux_escape_/e:w	399	\str_bytes_percent_decode:NN	6, 738, 743
\str_aux_escape_/f:w	399	\str_bytes_percent_decode_aux:wN	738, 747, 753, 762, 771, 775
\str_aux_escape_/n:w	399	\str_bytes_percent_decode_aux_ii:NN	738, 759, 766
\str_aux_escape_/r:w	399	\str_bytes_percent_encode:NN	6, 713, 715
\str_aux_escape_/t:w	399	\str_bytes_percent_encode_aux:N	718, 720
\str_aux_escape_/x:w	418	\str_bytes_unescape_hexadecimal:NN	5, 573, 573
\str_aux_escape_\.w	368	\str_bytes_unescape_hexadecimal_aux:N	573, 578, 584, 589, 598
\str_aux_escape_\.q_recursion_tail:w	399	\str_bytes_unescape_hexadecimal_aux_ii:N	573, 588, 591, 600
\str_aux_escape_escaped:N	368, 369, 375, 396	\str_bytes_unescape_name:NN	6, 602, 607
		\str_bytes_unescape_name_aux:wN	602, 612, 618, 626, 636, 641
		\str_bytes_unescape_name_aux_ii:N	602

<code>\str_bytes_unescape_name_aux_ii:NN</code> .	<code>\str_if_contains_char:nNTF</code> . . . . .	5
. . . . . 623, 630	<code>\str_if_contains_char_aux:NN</code> . . . . .	
<code>\str_bytes_unescape_string:NN</code> 6, 644, 651	. . . . . 297, 299, 304, 307	
<code>\str_bytes_unescape_string_aux:wN</code> . .	<code>\str_if_contains_char_end:w</code> 297, 310, 313	
. . . . . 655, 661, 688, 698, 706, 710	<code>\str_if_eq:NN</code> . . . . . 292, 292	
<code>\str_bytes_unescape_string_aux_d:N</code> .	<code>\str_if_eq:nn</code> . . . . . 292	
. . . . . 667, 692	<code>\str_if_eq:NNTF</code> . . . . . 4	
<code>\str_bytes_unescape_string_aux_dd:N</code>	<code>\str_if_eq:nnTF</code> . . . . . 4	
. . . . . 695, 701	<code>\str_if_eq:xx</code> . . . . . 292	
<code>\str_collect_aux:n</code> . . . . 149, 150, 151, 160	<code>\str_if_UTF_viii:NTF</code> . . . . . 5	
<code>\str_collect_aux:nnNNNNNNN</code> 149, 154, 158	<code>\str_input:Nn</code> . . . . . 2, 358, 358	
<code>\str_collect_do:nn</code> . . . . . 149, 149, 245	<code>\str_item:Nn</code> . . . . . 4, 182, 182	
<code>\str_collect_end:n</code> . . . . . 155, 164	<code>\str_item:nn</code> . . . . . 4, 182, 182, 183	
<code>\str_collect_end:nn</code> . . . . . 149	<code>\str_item_aux:nn</code> . . . . . 182, 192, 198	
<code>\str_collect_end_ii:nwn</code> 149, 167–175, 178	<code>\str_item_ignore_spaces:nn</code> . . 4, 182, 188	
<code>\str_collect_end_iii:nwNNNNNNN</code> . . . .	<code>\str_item_unsafe:nn</code> . . . 182, 186, 189, 190	
. . . . . 149, 150, 180	<code>\str_length:N</code> . . . . . 3, 68, 68	
<code>\str_from_to:Nnn</code> . . . . . 4, 218, 218	<code>\str_length:n</code> . . . . . 3, 68, 68, 69	
<code>\str_from_to:nnn</code> . . . . . 4, 218, 218, 219	<code>\str_length_aux:n</code> . . . . . 68, 71, 74, 77	
<code>\str_from_to_aux:nnnw</code> . . . . 218, 228, 235	<code>\str_length_ignore_spaces:n</code> 3, 68, 72, 918	
<code>\str_from_to_aux_ii:nnw</code> . . 218, 237, 241	<code>\str_length_loop:NNNNNNNNN</code> . . . . .	
<code>\str_from_to_ignore_spaces:nnn</code> . . . .	. . . . . 68, 71, 75, 88, 93	
. . . . . 4, 218, 224	<code>\str_length_skip_spaces:N</code> . . . . 916, 916	
<code>\str_from_to_unsafe:nnn</code> 218, 222, 225, 226	<code>\str_length_skip_spaces:n</code> . 916, 917, 918	
<code>\str_ginput:Nn</code> . . . . . 2, 358, 363	<code>\str_length_unsafe:n</code> . 68, 69, 70, 194, 229	
<code>\str_gput_left:cn</code> . . . . . 11	<code>\str_map_break_do:n</code> . . . . . 276, 291	
<code>\str_gput_left:cx</code> . . . . . 11	<code>\str_map_tokens:Nn</code> . . . . . 6, 276,	
<code>\str_gput_left:Nn</code> . . . . . 2, 11	276, 299, 317, 530, 537, 557, 718, 883	
<code>\str_gput_left:Nx</code> . . . . . 11	<code>\str_map_tokens:nn</code> . . . . 276, 277, 278, 304	
<code>\str_gput_right:cn</code> . . . . . 11	<code>\str_map_tokens_aux:nn</code> . . . 276, 279, 280	
<code>\str_gput_right:cx</code> . . . . . 11	<code>\str_map_tokens_loop:nN</code> 276, 282, 285, 289	
<code>\str_gput_right:Nn</code> . . . . . 2, 11	<code>\str_native_from_UTF_viii:NN</code> 5, 778, 778	
<code>\str_gput_right:Nx</code> . . . . . 11	<code>\str_native_from_UTF_viii_aux_i:N</code> . .	
<code>\str_gset:cn</code> . . . . . 11	. . . . . 782, 788, 798, 834, 858	
<code>\str_gset:cx</code> . . . . . 11	<code>\str_native_from_UTF_viii_aux_ii:nN</code>	
<code>\str_gset:Nn</code> . . . . . 2, 11	. . . . . 792–796, 803	
<code>\str_gset:Nx</code> . . . . . 11	<code>\str_native_from_UTF_viii_aux_iii:N</code>	
<code>\str_head:N</code> . . . . . 3, 95, 95	. . . . . 800, 813, 830	
<code>\str_head:n</code> . . . . . 3, 95, 95, 96	<code>\str_native_from_UTF_viii_aux_iv:</code> . .	
<code>\str_head_aux:w</code> . . . . . 95, 98, 102	. . . . . 816, 839	
<code>\str_head_ignore_spaces:n</code> . . . . 3, 95, 104	<code>\str_put_left:cn</code> . . . . . 11	
<code>\str_head_unsafe:n</code> . . . . . 95, 106	<code>\str_put_left:cx</code> . . . . . 11	
<code>\str_if_bytes:N</code> . . . . . 315, 315	<code>\str_put_left:Nn</code> . . . . . 2, 11	
<code>\str_if_bytes:NTF</code> . . . . . 5	<code>\str_put_left:Nx</code> . . . . . 11	
<code>\str_if_bytes_aux:N</code> . . . . . 315, 317, 320	<code>\str_put_right:cn</code> . . . . . 11	
<code>\str_if_contains_char:NN</code> . . . . . 297, 297	<code>\str_put_right:cx</code> . . . . . 11	
<code>\str_if_contains_char:nN</code> . . . . . 297, 302	<code>\str_put_right:Nn</code> . . . . . 2, 11	
<code>\str_if_contains_char:NNTF</code> . . . . . 567	<code>\str_put_right:Nx</code> . . . . . 11	
<code>\str_if_contains_char:NNTF</code> . . . . .	<code>\str_sanitize_args:Nn</code> . . . . .	
. . . . . 5, 547, 724, 732	. . . . . 6, 57, 57, 69, 186, 222, 279	

