

The `l3str` package: manipulating strings of characters^{*}

The L^AT_EX3 Project[†]

Released 2011/11/19

L^AT_EX3 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 var>  
\tl_to_str:n ★ \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

<hr/>	
<code>\str_set:Nn</code>	<code>\str_set:Nn <str var> {<token list>}</code>
<code>\str_set:(Nx cn cx)</code>	Converts the $\langle token list \rangle$ to a $\langle string \rangle$, and saves the result in $\langle str var \rangle$.
<hr/>	
<code>\str_gset:Nn</code>	<code>\str_gset:Nn <str var> {<token list>}</code>
<code>\str_gset:(Nx cn cx)</code>	Converts the $\langle token list \rangle$ to a $\langle string \rangle$, and saves the result in $\langle str var \rangle$ globally.
<hr/>	
<code>\str_put_left:Nn</code>	<code>\str_put_left:Nn <str var> {<token list>}</code>
<code>\str_put_left:(Nx cn cx)</code>	Converts the $\langle token list \rangle$ to a $\langle string \rangle$, and prepends the result to $\langle str var \rangle$. The current contents of the $\langle str var \rangle$ are not automatically converted to a string.
<hr/>	
<code>\str_gput_left:Nn</code>	<code>\str_gput_left:Nn <str var> {<token list>}</code>
<code>\str_gput_left:(Nx cn cx)</code>	Converts the $\langle token list \rangle$ to a $\langle string \rangle$, and prepends the result to $\langle str var \rangle$, globally. The current contents of the $\langle str var \rangle$ are not automatically converted to a string.
<hr/>	
<code>\str_put_right:Nn</code>	<code>\str_put_right:Nn <str var> {<token list>}</code>
<code>\str_put_right:(Nx cn cx)</code>	Converts the $\langle token list \rangle$ to a $\langle string \rangle$, and appends the result to $\langle str var \rangle$. The current contents of the $\langle str var \rangle$ are not automatically converted to a string.
<hr/>	
<code>\str_gput_right:Nn</code>	<code>\str_gput_right:Nn <str var> {<token list>}</code>
<code>\str_gput_right:(Nx cn cx)</code>	Converts the $\langle token list \rangle$ to a $\langle string \rangle$, and appends the result to $\langle str var \rangle$, globally. The current contents of the $\langle str var \rangle$ are not automatically converted to a string.

<code>\str_input:Nn</code>	<code>\str_input:Nn <str var> {(token list)}</code>
<code>\str_ginput:Nn</code>	

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 \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\^{\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

<code>\str_length:N</code>	★	<code>\str_length:n {(token list)}</code>
<code>\str_length:n</code>	★	
<code>\str_length_ignore_spaces:n</code>	★	

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.

\TeX hackers 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`.

<code>\str_head:N</code>	★	<code>\str_head:n {⟨token list⟩}</code>
<code>\str_head:n</code>	★	
<code>\str_head_ignore_spaces:n</code>	★	

Converts the $\langle token list \rangle$ into a $\langle string \rangle$. The first character in the $\langle string \rangle$ 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 $\langle token list \rangle$ is empty (or blank in the case of the `_ignore_spaces` variant), then nothing is left on the input stream.

<code>\str_tail:N</code>	★	<code>\str_tail:n {⟨token list⟩}</code>
<code>\str_tail:n</code>	★	
<code>\str_tail_ignore_spaces:n</code>	★	

Converts the $\langle token list \rangle$ to a $\langle string \rangle$, 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.

<code>\str_item:Nn</code>	★	<code>\str_item:nn {⟨token list⟩} {⟨integer expression⟩}</code>
<code>\str_item:nn</code>	★	
<code>\str_item_ignore_spaces:nn</code>	★	

Converts the $\langle token list \rangle$ to a $\langle string \rangle$, and leaves in the input stream the character in position $\langle integer expression \rangle$ of the $\langle string \rangle$. 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 $\langle integer expression \rangle$ is negative, characters are counted from the end of the $\langle string \rangle$. Hence, -1 is the right-most character, *etc.*, while 0 is the first (left-most) character.

<code>\str_substr:Nnn</code>	★	<code>\str_substr:nnn {⟨token list⟩} {⟨start index⟩} {⟨end index⟩}</code>
<code>\str_substr:nnn</code>	★	
<code>\str_substr_ignore_spaces:nnn</code>	★	

Converts the $\langle token list \rangle$ to a $\langle string \rangle$, and leaves in the input stream the characters between $\langle start index \rangle$ (inclusive) and $\langle end index \rangle$ (exclusive). If either of $\langle start index \rangle$ or $\langle end index \rangle$ is negative, then it is incremented by the length of the list. Both $\langle start index \rangle$ and $\langle end index \rangle$ count from 0 for the first (left most) character.

0.3 String conditionals

<code>\str_if_eq_p:NN</code>	★	<code>\str_if_eq_p:nn {<tl₁>} {<tl₂>}</code>
<code>\str_if_eq_p:(nn Vn on no nV VV xx)</code>	★	<code>\str_if_eq:nnTF {<tl₁>} {<tl₂>} {<true code>} {<false code>}</code>
<code>\str_if_eq:NNTF</code>	★	
<code>\str_if_eq:(nn Vn on no nV VV xx)TF</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).

<code>\str_if_contains_char_p:NN</code>	★	<code>\str_if_contains_char:nN {<token list>} {<char>}</code>
<code>\str_if_contains_char_p:nN</code>	★	
<code>\str_if_contains_char:NNTF</code>	★	
<code>\str_if_contains_char:nNTF</code>	★	

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

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

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

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.

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

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.

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

The `escape` variant takes each character in $\langle str\ var1 \rangle$ and converts it into the hexadecimal representation. The resulting string is stored in $\langle str\ var2 \rangle$ within the current \TeX group level. The `unescape` variant reverses this process. Thus for example

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

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

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

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

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

The `escape` variant takes each character in $\langle str\ var1 \rangle$ 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 $\langle str\ var2 \rangle$ within the current \TeX group level. The `unescape` variant reverses this process. Thus for example

```
\str_set:Nn \l_my_a_str { Hello ( ) }
\str_bytes_escape_name:NN \l_my_a_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_my_a_str { Hello#28#29 }
\str_bytes_unescape_name:NN \l_my_a_str \l_myb_str
```

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

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

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

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

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

```
\str_set:Nn \l_my_a_str { Hello\(\) }
\str_bytes_unescape_string:NN \l_my_a_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`.

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

The `escape` variant takes each character in $\langle str\ var1 \rangle$ 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 $\langle str\ var2 \rangle$ within the current \TeX group level. The `unescape` variant reverses this process. Thus for example

```
\str_set:Nn \l_my_a_str { Hello ( ) }
\str_bytes_precent_encode:NN \l_my_a_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_my_a_str { Hello%28%29 }
\str_bytes_unescape_name:NN \l_my_a_str \l_myb_str
```

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

0.6 Internal string functions

<code>\tl_to_other_str:N</code> ★	<code>\tl_to_other_str:n {<token list>}</code>
<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.

\TeX hackers 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.

<hr/> <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 $\langle function \rangle$.
<hr/> <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 $\langle str var \rangle$.
<hr/> <code>\str_aux_escape:NNNn</code>	<code>\str_aux_escape:NNNn</code> $\langle fn1 \rangle$ $\langle fn2 \rangle$ $\langle fn3 \rangle$ $\{\langle token list \rangle\}$
	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 <code>\a</code> , <code>\e</code> , <code>\f</code> , <code>\n</code> , <code>\r</code> , <code>\t</code> and <code>\x</code> are recognized as described for <code>\str_input:Nn</code> , and those are replaced by the corresponding character, then fed to $\langle fn3 \rangle$. The result is assigned globally to <code>\g_str_result_tl</code> .

0.7 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  $\langle *initex | package \rangle$ 
2 \ProvidesExplPackage
3    $\{\backslash ExplFileName\} \{\backslash ExplFileDate\} \{\backslash ExplFileVersion\} \{\backslash ExplFileDescription\}$ 

```

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:N` (unchanged)

1.1 General functions

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

```
4 \cs_if_exist:NF \use_i:nnnnnnnn
5 { \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
\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_put_right:cx
\str_gput_right:Nn
\str_gput_right:Nx
\str_gput_right:cn
\str_gput_right:cx
```

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

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

1.2 Variables and constants

Internal scratch space for some functions.

```
\str_gput_right:Nn
\str_gput_right:Nx
\str_gput_right:cn
\str_gput_right:cx
\str_gput_right:Nn
\str_gput_right:Nx
\str_gput_right:cn
\str_gput_right:cx
```

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

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

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.

```
22 \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`.

```

23 \int_new:N \l_str_char_int
24 \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 25 \tl_const:Nx \c_backslash_str { \cs_to_str:N \ }
\c_rbrace_str 26 \tl_const:Nx \c_lbrace_str { \cs_to_str:N \{ }
\c_hash_str 27 \tl_const:Nx \c_rbrace_str { \cs_to_str:N \} }
\c_percent_str 28 \tl_const:Nx \c_hash_str { \cs_to_str:N \% }
29 \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 30 \group_begin:
\tl_to_other_str_end:w 31 \char_set_lccode:nn { '\* } { '\ }
32 \char_set_lccode:nn { '\A } { '\A }
33 \tl_to_lowercase:n
34 {
35   \group_end:
36   \cs_new:Npn \tl_to_other_str:n #1
37   {
38     \exp_after:wN \tl_to_other_str_loop:w \tl_to_str:n {#1} ~ %
39     A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \q_mark \q_stop
40   }
41   \cs_new_nopar:Npn \tl_to_other_str_loop:w
42   #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 \q_stop
43   {
44     \if_meaning:w A #8
45     \tl_to_other_str_end:w
46     \fi:
47     \tl_to_other_str_loop:w
48     #9 #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * \q_stop
49   }
50   \cs_new_nopar:Npn \tl_to_other_str_end:w \fi: #1 \q_mark #2 * A #3 \q_stop
51   { \fi: #2 }
52 }
53 \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` Here, f-expansion does not lose leading spaces, since they have catcode “other” after
`\str_sanitize_args:Nnn` `\str_sanitize:n`.

```

54 \cs_new:Npn \str_sanitize_args:Nn #1#2
55   { \exp_args:Nf #1 { \tl_to_other_str:n {#2} } }
56 \cs_new:Npn \str_sanitize_args:Nnn #1#2#3
57   {
58     \exp_args:Nff #1
59     { \tl_to_other_str:n {#2} }
60     { \tl_to_other_str:n {#3} }
61   }

```

(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`.

```

62 \cs_new_nopar:Npn \str_length:N { \exp_args:No \str_length:n }
63 \cs_new:Npn \str_length:n { \str_sanitize_args:Nn \str_length_unsafe:n }
64 \cs_new_nopar:Npn \str_length_unsafe:n #1
65   { \str_length_aux:n { \str_length_loop:NNNNNNNNN #1 } }
66 \cs_new:Npn \str_length_ignore_spaces:n #1
67   {
68     \str_length_aux:n
69     { \exp_after:wN \str_length_loop:NNNNNNNNN \tl_to_str:n {#1} }
70   }
71 \cs_new:Npn \str_length_aux:n #1
72   {
73     \int_eval:n
74     {
75       #1
76       { X \c_eight } { X \c_seven } { X \c_six }
77       { X \c_five } { X \c_four } { X \c_three }
78       { X \c_two } { X \c_one } { X \c_zero }
79       \q_stop
80     }
81   }
82 \cs_set:Npn \str_length_loop:NNNNNNNNN #1#2#3#4#5#6#7#8#9
83   {
84     \if_catcode:w X #9
85     \exp_after:wN \use_none_delimit_by_q_stop:w
86     \fi:
87     \c_nine + \str_length_loop:NNNNNNNNN
88   }

```

(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:n` `\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`.

```

89 \cs_new_nopar:Npn \str_head:N { \exp_args:No \str_head:n }
90 \cs_set:Npn \str_head:n #1
91 {
92   \exp_after:wN \str_head_aux:w
93   \tl_to_str:n {#1}
94   { { } } ~ \q_stop
95 }
96 \cs_set_nopar:Npn \str_head_aux:w #1 ~ %
97 { \use_i_delimit_by_q_stop:nw #1 { ~ } }
98 \cs_new:Npn \str_head_ignore_spaces:n #1
99 { \exp_after:wN \use_i_delimit_by_q_stop:nw \tl_to_str:n {#1} { } \q_stop }
100 \cs_new_nopar:Npn \str_head_unsafe:n #1
101 { \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` `\str_tail_ignore_spaces:n` `\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.

```

102 \cs_new_nopar:Npn \str_tail:N { \exp_args:No \str_tail:n }
103 \cs_set:Npn \str_tail:n #1
104 {
105   \exp_after:wN \str_tail_aux:w
106   \reverse_if:N \if_charcode:w
107   \scan_stop: \tl_to_str:n {#1} X X \q_stop
108 }
109 \cs_set_nopar:Npn \str_tail_aux:w #1 X #2 \q_stop { \fi: #1 }
110 \cs_new:Npn \str_tail_ignore_spaces:n #1
111 {
112   \exp_after:wN \str_tail_aux_ii:w
113   \tl_to_str:n {#1} X { } X \q_stop
114 }
115 \cs_new_nopar:Npn \str_tail_unsafe:n #1
116 { \str_tail_aux_ii:w #1 X { } X \q_stop }
117 \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 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.

```

118 \cs_new:Npn \str_skip_do:nn #1
119   {
120     \if_num:w \int_eval:w #1 > \c_seven
121       \exp_after:wN \str_skip_aux:nnnnnnnnn
122     \else:
123       \exp_after:wN \str_skip_end:n
124     \fi:
125     {#1}
126   }
127 \cs_new:Npn \str_skip_aux:nnnnnnnnn #1#2#3#4#5#6#7#8#9
128   { \exp_args:Nf \str_skip_do:nn { \int_eval:n { #1 - \c_seven } } {#2} }
129 \cs_new:Npn \str_skip_end:n #1
130   {
131     \if_case:w \int_eval:w #1 \int_eval_end:
132       \str_skip_end_ii:nwn { XXXXXXX }
133     \or: \str_skip_end_ii:nwn { XXXXXX }
134     \or: \str_skip_end_ii:nwn { XXXXX }
135     \or: \str_skip_end_ii:nwn { XXXX }
136     \or: \str_skip_end_ii:nwn { XXX }
137     \or: \str_skip_end_ii:nwn { XX }
138     \or: \str_skip_end_ii:nwn { X }
139     \or: \str_skip_end_ii:nwn { }
140     \else: \str_skip_end_ii:nwn { XXXXXXX }
141     \fi:
142   }
143 \cs_new:Npn \str_skip_end_ii:nwn #1#2 \fi: #3
144   { \fi: \use_i:nnnnnnnn {#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.

```

145 \cs_new:Npn \str_collect_do:nn #1#2
146   { \str_collect_aux:n {#1} { \str_collect_end_iii:nwnnnnnnn {#2} } }
147 \cs_new:Npn \str_collect_aux:n #1
148   {
149     \int_compare:nNnTF {#1} > \c_seven
150       { \str_collect_aux:nnnnnnnnn }
151       { \str_collect_end:n }
152     {#1}
153   }
154 \cs_new:Npn \str_collect_aux:nnnnnnnnn #1#2 #3#4#5#6#7#8#9
155   {

```

```

156     \exp_args:Nf \str_collect_aux:n
157     { \int_eval:n { #1 - \c_seven } }
158     { #2 #3#4#5#6#7#8#9 }
159 }
160 \cs_new:Npn \str_collect_end:n #1
161 {
162     \if_case:w \int_eval:w #1 \int_eval_end:
163     \str_collect_end_ii:nwn { { } { } { } { } { } { } { } { } }
164     \or: \str_collect_end_ii:nwn { { } { } { } { } { } { } { } { } }
165     \or: \str_collect_end_ii:nwn { { } { } { } { } { } { } { } { } }
166     \or: \str_collect_end_ii:nwn { { } { } { } { } { } { } { } { } }
167     \or: \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     \else: \str_collect_end_ii:nwn { { } { } { } { } { } { } { } { } }
172     \fi:
173 }
174 \cs_new:Npn \str_collect_end_ii:nwn #1#2 \fi: #3
175 { \fi: #3 \q_stop #1 }
176 \cs_new:Npn \str_collect_end_iii:nwnNNNNNN #1 #2 \q_stop #3#4#5#6#7#8#9
177 { #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.

```

178 \cs_new_nopar:Npn \str_item:Nn { \exp_args:No \str_item:nn }
179 \cs_new:Npn \str_item:nn #1#2
180 {
181     \exp_args:Nf \tl_to_str:n
182     { \str_sanitize_args:Nn \str_item_unsafe:nn {#1} {#2} }
183 }
184 \cs_new:Npn \str_item_ignore_spaces:nn #1
185 { \exp_args:No \str_item_unsafe:nn { \tl_to_str:n {#1} } }
186 \cs_new_nopar:Npn \str_item_unsafe:nn #1#2
187 {
188     \exp_args:Nff \str_item_aux:nn
189     { \int_eval:n {#2} }
190     { \str_length_unsafe:n {#1} }
191     #1
192     \q_stop
193 }
194 \cs_new_nopar:Npn \str_item_aux:nn #1#2
195 {
196     \int_compare:nNnTF {#1} < \c_zero
197     {

```

```

198     \int_compare:nNnTF {#1} < {-#2}
199     { \use_none_delimit_by_q_stop:w }
200     {
201         \str_skip_do:nn { #1 + #2 }
202         { \use_i_delimit_by_q_stop:nw }
203     }
204 }
205 {
206     \int_compare:nNnTF {#1} < {#2}
207     {
208         \str_skip_do:nn {#1}
209         { \use_i_delimit_by_q_stop:nw }
210     }
211     { \use_none_delimit_by_q_stop:w }
212 }
213 }

```

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

<code>\str_substr:Nnn</code> <code>\str_substr:nnn</code> <code>\str_substr_ignore_spaces:nnn</code> <code>\str_substr_unsafe:nnn</code> <code>\str_substr_aux:nnnw</code> <code>\str_substr_aux_ii:nnw</code> <code>\str_aux_eval_args:Nnnn</code> <code>\str_aux_normalize_range:nn</code>	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.	<pre> 214 \cs_new_nopar:Npn \str_substr:Nnn { \exp_args:No \str_substr:nnn } 215 \cs_new:Npn \str_substr:nnn #1#2#3 216 { 217 \exp_args:Nf \tl_to_str:n 218 { \str_sanitiz_e_args:Nn \str_substr_unsafe:nnn {#1} {#2} {#3} } 219 } 220 \cs_new:Npn \str_substr_ignore_spaces:nnn #1 221 { \exp_args:No \str_substr_unsafe:nnn { \tl_to_str:n {#1} } } 222 \cs_new:Npn \str_substr_unsafe:nnn #1#2#3 223 { 224 \str_aux_eval_args:Nnnn \str_substr_aux:nnnw 225 { \str_length_unsafe:n {#1} } 226 {#2} 227 {#3} 228 #1 229 \q_stop 230 } 231 \cs_new:Npn \str_substr_aux:nnnw #1#2#3 232 { 233 \exp_args:Nf \str_substr_aux_ii:nnw 234 { \str_aux_normalize_range:nn {#2} {#1} } 235 { \str_aux_normalize_range:nn {#3} {#1} } 236 } 237 \cs_new:Npn \str_substr_aux_ii:nnw #1#2 238 { 239 \str_skip_do:nn {#1} 240 { </pre>
---	--	---

```

241     \exp_args:Nf \str_collect_do:nn
242     { \int_eval:n { #2 - #1 } }
243     { \use_i_delimit_by_q_stop:nw }
244   }
245 }
246 \cs_new:Npn \str_aux_eval_args:Nnnn #1#2#3#4
247 {
248   \exp_after:wN #1
249   \exp_after:wN { \int_value:w \int_eval:w #2 \exp_after:wN }
250   \exp_after:wN { \int_value:w \int_eval:w #3 \exp_after:wN }
251   \exp_after:wN { \int_value:w \int_eval:w #4 }
252 }
253 \cs_new:Npn \str_aux_normalize_range:nn #1#2
254 {
255   \int_eval:n
256   {
257     \if_num:w #1 < \c_zero
258     \if_num:w #1 < - #2 \exp_stop_f:
259       \c_zero
260     \else:
261       #1 + #2
262     \fi:
263   \else:
264     \if_num:w #1 < #2 \exp_stop_f:
265       #1
266     \else:
267       #2
268     \fi:
269   \fi:
270 }
271 }

```

(End definition for `\str_substr: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
\str_map_tokens:Nn category code 12, not 10.
\str_map_tokens_aux:nn
\str_map_tokens_loop:nN
\str_map_break_do:n
272 \cs_new_nopar:Npn \str_map_tokens:Nn
273 { \exp_args:No \str_map_tokens:nn }
274 \cs_new_nopar:Npn \str_map_tokens:nn
275 { \str_sanitiz_args:Nn \str_map_tokens_aux:nn }
276 \cs_new:Npn \str_map_tokens_aux:nn #1#2
277 {
278   \str_map_tokens_loop:nN {#2} #1
279   { ? \use_none_delimit_by_q_recursion_stop:w } \q_recursion_stop
280 }
281 \cs_new_nopar:Npn \str_map_tokens_loop:nN #1#2
282 {
283   \use_none:n #2

```



```

284     #1 #2
285     \str_map_tokens_loop:nN {#1}
286   }
287 \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 288 \prg_new_conditional:Npnn \str_if_eq:NN #1#2 { p , TF , T , F }
\str_if_eq:xx 289 {
290     \if_int_compare:w \pdftex_strcmp:D { \tl_to_str:N #1 } { \tl_to_str:N #2 }
291     = \c_zero \prg_return_true: \else: \prg_return_false: \fi:
292 }
    (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”.

```

293 \prg_new_conditional:Npnn \str_if_contains_char:NN #1#2 { p , T , F , TF }
294 {
295     \str_map_tokens:Nn #1 { \str_if_contains_char_aux:NN #2 }
296     \prg_return_false:
297 }
298 \prg_new_conditional:Npnn \str_if_contains_char:nN #1#2 { p , T , F , TF }
299 {
300     \str_map_tokens:nn {#1} { \str_if_contains_char_aux:NN #2 }
301     \prg_return_false:
302 }
303 \cs_new_nopar:Npn \str_if_contains_char_aux:NN #1#2
304 {
305     \if_num:w '#1 = '#2 \exp_stop_f:
306     \str_if_contains_char_end:w
307     \fi:
308 }
309 \cs_new_nopar:Npn \str_if_contains_char_end:w \fi: #1 \prg_return_false:
310 { \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 311 \prg_new_conditional:Npnn \str_if_bytes:N #1 { p , T , F , TF }
312 {
313     \str_map_tokens:Nn #1 { \str_if_bytes_aux:N }
314     \prg_return_true:
315 }
316 \cs_new_nopar:Npn \str_if_bytes_aux:N #1
317 {

```

```

318     \int_compare:nNnF {'#1} < \c_two_hundred_fifty_six
319     {
320         \use_i_delimit_by_q_recursion_stop:nw
321         { \prg_return_false: \use_none:n }
322     }
323 }

```

(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).

```

324 \prg_new_protected_conditional:Npnn \str_aux_hexadecimal_test:N #1 { TF }
325 {
326     \tl_if_in:onTF { \tl_to_str:n { abcdef } } {#1}
327     {
328         \int_set:Nn \l_str_char_int
329         { \c_sixteen * \l_str_char_int + '#1 - 87 }
330         \prg_return_true:
331     }
332     {
333         \if_num:w \c_fifteen < "1 \exp_not:N #1 \exp_stop_f:
334         \int_set:Nn \l_str_char_int
335         { \c_sixteen * \l_str_char_int + "#1 }
336         \prg_return_true:
337         \else:
338             \prg_return_false:
339         \fi:
340     }
341 }

```

(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).

```

342 \prg_new_protected_conditional:Npnn \str_aux_octal_test:N #1 { TF }
343 {
344     \if_num:w \c_seven < '1 \exp_not:N #1 \exp_stop_f:
345     \int_set:Nn \l_str_char_int
346     { \c_eight * \l_str_char_int + '#1 }
347     \prg_return_true:
348     \else:
349         \prg_return_false:
350     \fi:
351 }

```

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

`\str_aux_char_if_octal_digit:NTF`

```
352 \cs_new_protected_nopar:Npn \str_aux_char_if_octal_digit:NTF #1
353   { \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` Simple wrappers around the internal `\str_aux_escape:NNNn`.

```
\str_ginput:Nn
354 \cs_new_protected:Npn \str_input:Nn #1#2
355   {
356     \str_aux_escape:NNNn \token_to_str:N \token_to_str:N \token_to_str:N {#2}
357     \tl_set_eq:NN #1 \g_str_result_tl
358   }
359 \cs_new_protected:Npn \str_ginput:Nn #1#2
360   {
361     \str_aux_escape:NNNn \token_to_str:N \token_to_str:N \token_to_str:N {#2}
362     \tl_gset_eq:NN #1 \g_str_result_tl
363   }
```

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

`\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
364 \cs_new_eq:NN \str_aux_escape_unescaped:N \use:n
365 \cs_new_eq:NN \str_aux_escape_escaped:N \use:n
366 \cs_new_eq:NN \str_aux_escape_raw:N \use:n
367 \cs_new_protected:Npn \str_aux_escape:NNNn #1#2#3#4
368   {
369     \group_begin:
370     \cs_set_nopar:Npn \str_aux_escape_unescaped:N { #1 }
```

```

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

```

(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_/a:w 395 \cs_new_eq:cN
\str_aux_escape_/e:w 396 { str_aux_escape_ \c_backslash_str q_recursion_tail :w }
\str_aux_escape_/f:w 397 \use_none_delimit_by_q_recursion_stop:w
\str_aux_escape_/n:w 398 \cs_new_eq:cN
\str_aux_escape_/r:w 399 { str_aux_escape_ / \c_backslash_str q_recursion_tail :w }
\str_aux_escape_/t:w 400 \use_none_delimit_by_q_recursion_stop:w
401 \cs_new_nopar:cpn { str_aux_escape_~:w } { }
402 \cs_new_nopar:cpx { str_aux_escape_/a:w }
403 { \exp_not:N \str_aux_escape_raw:N \iow_char:N \^^G }
404 \cs_new_nopar:cpx { str_aux_escape_/t:w }
405 { \exp_not:N \str_aux_escape_raw:N \iow_char:N \^^I }
406 \cs_new_nopar:cpx { str_aux_escape_/n:w }
407 { \exp_not:N \str_aux_escape_raw:N \iow_char:N \^^J }
408 \cs_new_nopar:cpx { str_aux_escape_/f:w }
409 { \exp_not:N \str_aux_escape_raw:N \iow_char:N \^^L }
410 \cs_new_nopar:cpx { str_aux_escape_/r:w }
411 { \exp_not:N \str_aux_escape_raw:N \iow_char:N \^^M }
412 \cs_new_nopar:cpx { str_aux_escape_/e:w }
413 { \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>
---	--

```

414 \cs_new_nopar:cpn { str_aux_escape_/x:w } \str_aux_escape_loop:N
415 {
416   \if_false: { \fi: }
417   \int_zero:N \l_str_char_int
418   \str_aux_escape_x_test:N
419 }
420 \cs_new_protected_nopar:Npx \str_aux_escape_x_test:N #1
421 {
422   \exp_not:N \token_if_eq_charcode:NNTF \c_space_token #1
423   { \exp_not:N \str_aux_escape_x_test:N }
424   {
425     \exp_not:N \token_if_eq_charcode:NNTF \c_lbrace_str #1
426     { \exp_not:N \str_aux_escape_x_braced_loop:N }
427     { \exp_not:N \str_aux_escape_x_unbraced_i:N #1 }
428   }
429 }
430 \cs_new_protected_nopar:Npn \str_aux_escape_x_unbraced_i:N #1
431 {
432   \str_aux_hexadecimal_test:NNTF #1
433   { \str_aux_escape_x_unbraced_ii:N }
434   { \str_aux_escape_x_end: #1 }
435 }
436 \cs_new_protected_nopar:Npn \str_aux_escape_x_unbraced_ii:N #1
437 {
438   \token_if_eq_charcode:NNTF \c_space_token #1
439   { \str_aux_escape_x_unbraced_ii:N }
440   {
441     \str_aux_hexadecimal_test:NNTF #1
442     { \str_aux_escape_x_end: }
443     { \str_aux_escape_x_end: #1 }
444   }
445 }
446 \cs_new_protected_nopar:Npn \str_aux_escape_x_braced_loop:N #1
447 {
448   \token_if_eq_charcode:NNTF \c_space_token #1
449   { \str_aux_escape_x_braced_loop:N }
450   {
451     \str_aux_hexadecimal_test:NNTF #1
452     { \str_aux_escape_x_braced_loop:N }
453     { \str_aux_escape_x_braced_end:N #1 }
454   }

```

```

455 }
456 \cs_new_protected_nopar:Npx \str_aux_escape_x_braced_end:N #1
457 {
458   \exp_not:N \token_if_eq_charcode:NNTF \c_rbrace_str #1
459   { \exp_not:N \str_aux_escape_x_end: }
460   {
461     \msg_error:nn { str } { x-missing-brace }
462     \exp_not:N \str_aux_escape_x_end: #1
463   }
464 }
465 \group_begin:
466   \char_set_catcode_other:N ^^@
467   \cs_new_protected_nopar:Npn \str_aux_escape_x_end:
468   {
469     \group_begin:
470     \char_set_lccode:nn { \c_zero } { \l_str_char_int }
471     \tl_to_lowercase:n
472     {
473       \group_end:
474       \tl_gput_right:Nx \g_str_result_tl
475       { \if_false: } \fi:
476       \str_aux_escape_raw:N ^^@
477       \str_aux_escape_loop:N
478     }
479   }
480 \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
481 \group_begin:
482   \char_set_catcode_other:n { '\^^@ }
483   \cs_new_protected_nopar:Npn \str_aux_convert_store:
484   {
485     \char_set_lccode:nn { \c_zero } { \l_str_char_int }
486     \tl_to_lowercase:n { \tl_gput_right:Nx \g_str_result_tl {\^^@} }
487   }
488   \cs_new_protected_nopar:Npn \str_aux_convert_store:NNn #1#2#3
489   {
490     \char_set_lccode:nn { \c_zero } {#3}
491     \tl_to_lowercase:n { #1 #2 {\^^@} }
492   }
493 \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
494 \cs_new_nopar:Npn \str_aux_byte_to_hexadecimal:N #1
495 {

```

```

496     \int_compare:nNnTF {'#1} < { 256 }
497     {
498         \int_to_letter:n { \int_div_truncate:nn {'#1} \c_sixteen }
499         \int_to_letter:n { \int_mod:nn {'#1} \c_sixteen }
500     }
501     { \msg_expandable_error:n { Invalid~byte~'#1'. } }
502 }
503 \cs_new_nopar:Npn \str_aux_byte_to_octal:N #1
504 {
505     \int_compare:nNnTF {'#1} < { 64 }
506     {
507         0
508         \int_to_letter:n { \int_div_truncate:nn {'#1} \c_eight }
509         \int_to_letter:n { \int_mod:nn {'#1} \c_eight }
510     }
511     {
512         \int_compare:nNnTF {'#1} < { 256 }
513         { \int_to_octal:n {'#1} }
514         { \msg_expandable_error:n { Invalid~byte~'#1'. } }
515     }
516 }

```

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

\str_bytes_escape_hexadecimal:NN

```

517 \cs_new_protected_nopar:Npn \str_bytes_escape_hexadecimal:NN #1#2
518 {
519     \str_set:Nx #1
520     { \str_map_tokens:Nn #2 \str_aux_byte_to_hexadecimal:N }
521 }

```

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

\str_bytes_escape_name:NN

```

522 \tl_const:Nx \c_str_bytes_escape_name_str
523 { \c_hash_str \c_percent_str \c_lbrace_str \c_rbrace_str ()/<>[] }
524 \cs_new_protected_nopar:Npn \str_bytes_escape_name:NN #1#2
525 {
526     \str_set:Nx #1
527     { \str_map_tokens:Nn #2 \str_bytes_escape_name_aux:N }
528 }
529 \cs_new_nopar:Npn \str_bytes_escape_name_aux:N #1
530 {
531     \int_compare:nNnTF {'#1} < { "21 }
532     { \c_hash_str \str_aux_byte_to_hexadecimal:N #1 }
533     {
534         \int_compare:nNnTF {'#1} > { "7E }
535         { \c_hash_str \str_aux_byte_to_hexadecimal:N #1 }
536         {
537             \str_if_contains_char:NNTF \c_str_bytes_escape_name_str #1
538             { \c_hash_str \str_aux_byte_to_hexadecimal:N #1 }

```

```

539         {#1}
540     }
541 }
542 }

```

(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.

```

543 \tl_const:Nx \c_str_bytes_escape_string_str { \c_backslash_str ( ) }
544 \cs_new_protected_nopar:Npn \str_bytes_escape_string:NN #1#2
545 {
546     \str_set:Nx #1
547     { \str_map_tokens:Nn #2 \str_bytes_escape_string_aux:N }
548 }
549 \cs_new_nopar:Npn \str_bytes_escape_string_aux:N #1
550 {
551     \int_compare:nNnTF {‘#1} < { "21 }
552     { \c_backslash_str \str_aux_byte_to_octal:N #1 }
553     {
554         \int_compare:nNnTF {‘#1} > { "7E }
555         { \c_backslash_str \str_aux_byte_to_octal:N #1 }
556         {
557             \str_if_contains_char:NNT \c_str_bytes_escape_string_str #1
558             { \c_backslash_str }
559             #1
560         }
561     }
562 }

```

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

`\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).

```

563 \cs_new_protected_nopar:Npn \str_bytes_unescape_hexadecimal:NN #1#2
564 {
565     \group_begin:
566     \tl_gclear:N \g_str_result_tl
567     \int_zero:N \l_str_char_int
568     \exp_last_unbraced:Nf \str_bytes_unescape_hexadecimal_aux:N
569     { \tl_to_other_str:N #2 } 0
570     \q_recursion_tail \q_recursion_stop
571     \group_end:
572     \tl_set_eq:NN #1 \g_str_result_tl
573 }
574 \cs_new_protected_nopar:Npn \str_bytes_unescape_hexadecimal_aux:N #1
575 {
576     \quark_if_recursion_tail_stop:N #1
577     \str_aux_hexadecimal_test:NNTF #1
578     { \str_bytes_unescape_hexadecimal_aux_ii:N }

```



```

579     { \str_bytes_unescape_hexadecimal_aux:N }
580   }
581 \cs_new_protected_nopar:Npn \str_bytes_unescape_hexadecimal_aux_ii:N #1
582 {
583   \quark_if_recursion_tail_stop:N #1
584   \str_aux_hexadecimal_test:NTF #1
585   {
586     \str_aux_convert_store:
587     \int_zero:N \l_str_char_int
588     \str_bytes_unescape_hexadecimal_aux:N
589   }
590   { \str_bytes_unescape_hexadecimal_aux_ii:N }
591 }

```

(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
592 \group_begin:
593   \char_set_lccode:nn {'\*} {'\#}
594   \tl_to_lowercase:n
595   {
596     \group_end:
597     \cs_new_protected_nopar:Npn \str_bytes_unescape_name:NN #1#2
598     {
599       \group_begin:
600       \tl_gclear:N \g_str_result_tl
601       \int_zero:N \l_str_char_int
602       \exp_last_unbraced:Nf \str_bytes_unescape_name_aux:wN
603       { \tl_to_other_str:N #2 }
604       * \q_recursion_tail \q_recursion_stop
605       \group_end:
606       \tl_set_eq:NN #1 \g_str_result_tl
607     }
608     \cs_new_protected_nopar:Npn \str_bytes_unescape_name_aux:wN #1 * #2
609     {
610       \tl_gput_right:Nx \g_str_result_tl {#1}
611       \quark_if_recursion_tail_stop:N #2
612       \str_aux_hexadecimal_test:NTF #2
613       { \str_bytes_unescape_name_aux_ii:NN #2 }
614       {
615         \tl_gput_right:Nx \g_str_result_tl { \c_hash_str }
616         \str_bytes_unescape_name_aux:wN #2
617       }
618     }
619   }
620 \cs_new_protected_nopar:Npn \str_bytes_unescape_name_aux_ii:NN #1#2
621 {
622   \str_aux_hexadecimal_test:NTF #2
623   {

```

```

624     \str_aux_convert_store:
625     \int_zero:N \l_str_char_int
626     \str_bytes_unescape_name_aux:wN
627   }
628   {
629     \tl_gput_right:Nx \g_str_result_tl { \c_hash_str #1 }
630     \int_zero:N \l_str_char_int
631     \str_bytes_unescape_name_aux:wN #2
632   }
633 }

```

(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.*

```

634 \group_begin:
635   \char_set_lccode:nn {'\*} {'\\}
636   \char_set_catcode_other:N ^^J
637   \char_set_catcode_other:N ^^M
638   \tl_to_lowercase:n
639   {
640     \group_end:
641     \cs_new_protected_nopar:Npn \str_bytes_unescape_string:NN #1#2
642     {
643       \group_begin:
644         \tl_gclear:N \g_str_result_tl
645         \exp_last_unbraced:Nf \str_bytes_unescape_string_aux:wN
646         { \tl_to_other_str:N #2 }
647         * \q_recursion_tail \q_recursion_stop
648       \group_end:
649       \tl_set_eq:NN #1 \g_str_result_tl

```

```

650     }
651 \cs_new_protected_nopar:Npn \str_bytes_unescape_string_aux:wN #1 * #2
652 {
653     \tl_gput_right:Nx \g_str_result_tl {#1}
654     \quark_if_recursion_tail_stop:N #2
655     \int_zero:N \l_str_char_int
656     \str_aux_octal_test:NTF #2
657     { \str_bytes_unescape_string_aux_d:N }
658     {
659         \int_set:Nn \l_str_char_int
660         {
661             \prg_case_str:xxn {#2}
662             {
663                 { n } { 10 }
664                 { r } { 13 }
665                 { t } { 9 }
666                 { b } { 8 }
667                 { f } { 12 }
668                 { ( } { 40 }
669                 { ) } { 41 }
670                 { \c_backslash_str } { 92 }
671                 { ^^J } { -1 }
672                 { ^^M } { -1 }
673             }
674             {'#2}
675         }
676         \int_compare:nNnF \l_str_char_int = \c_minus_one
677         { \str_aux_convert_store: }
678         \str_bytes_unescape_string_aux:wN
679     }
680 }
681 }
682 \cs_new_protected_nopar:Npn \str_bytes_unescape_string_aux_d:N #1
683 {
684     \str_aux_octal_test:NTF #1
685     { \str_bytes_unescape_string_aux_dd:N }
686     {
687         \str_aux_convert_store:
688         \str_bytes_unescape_string_aux:wN #1
689     }
690 }
691 \cs_new_protected_nopar:Npn \str_bytes_unescape_string_aux_dd:N #1
692 {
693     \str_aux_octal_test:NTF #1
694     {
695         \str_aux_convert_store:
696         \str_bytes_unescape_string_aux:wN
697     }
698     {
699         \str_aux_convert_store:

```

```

700     \str_bytes_unescape_string_aux:wN #1
701   }
702 }

```

(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`

```

703 \tl_const:Nx \c_str_bytes_percent_encode_str { \tl_to_str:n { [] } }
704 \tl_const:Nx \c_str_bytes_percent_encode_not_str { \tl_to_str:n { "-.<> } }
705 \cs_new_protected_nopar:Npn \str_bytes_percent_encode:NN #1#2
706 {
707   \str_set:Nx #1
708     { \str_map_tokens:Nn #2 \str_bytes_percent_encode_aux:N }
709 }
710 \cs_new_nopar:Npn \str_bytes_percent_encode_aux:N #1
711 {
712   \int_compare:nNnTF {'#1} < { "41 }
713   {
714     \str_if_contains_char:NNTF \c_str_bytes_percent_encode_not_str #1
715       { #1 }
716     { \c_percent_str \str_aux_byte_to_hexadecimal:N #1 }
717   }
718   {
719     \int_compare:nNnTF {'#1} > { "7E }
720     { \c_percent_str \str_aux_byte_to_hexadecimal:N #1 }
721     {
722       \str_if_contains_char:NNTF \c_str_bytes_percent_encode_str #1
723       { \c_percent_str \str_aux_byte_to_hexadecimal:N #1 }
724       { #1 }
725     }
726   }
727 }

```

(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 728 \group_begin:
\str_bytes_percent_decode_aux_ii:NN 729   \char_set_lccode:nn {'\*} {'\%}
730   \tl_to_lowercase:n
731   {
732     \group_end:
733     \cs_new_protected_nopar:Npn \str_bytes_percent_decode:NN #1#2
734     {
735       \group_begin:
736       \tl_gclear:N \g_str_result_tl
737       \exp_last_unbraced:Nf \str_bytes_percent_decode_aux:wN
738       { \tl_to_other_str:N #2 }
739       * \q_recursion_tail \q_recursion_stop
740     \group_end:

```

```

741         \tl_set_eq:NN #1 \g_str_result_tl
742     }
743     \cs_new_protected_nopar:Npn \str_bytes_percent_decode_aux:wN #1 * #2
744     {
745         \tl_gput_right:Nx \g_str_result_tl {#1}
746         \quark_if_recursion_tail_stop:N #2
747         \int_zero:N \l_str_char_int
748         \str_aux_hexadecimal_test:NTF #2
749         { \str_bytes_percent_decode_aux_ii:NN #2 }
750         {
751             \tl_gput_right:Nx \g_str_result_tl { \c_percent_str }
752             \str_bytes_percent_decode_aux:wN #2
753         }
754     }
755 }
756 \cs_new_protected_nopar:Npn \str_bytes_percent_decode_aux_ii:NN #1#2
757 {
758     \str_aux_hexadecimal_test:NTF #2
759     {
760         \str_aux_convert_store:
761         \str_bytes_percent_decode_aux:wN
762     }
763     {
764         \tl_gput_right:Nx \g_str_result_tl {#1}
765         \str_bytes_percent_decode_aux:wN #2
766     }
767 }

```

(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:
768 \cs_new_protected_nopar:Npn \str_native_from_UTF_viii:NN #1#2
769 {
770     \group_begin:
771     \tl_gclear:N \g_str_result_tl
772     \exp_last_unbraced:Nf \str_native_from_UTF_viii_aux_i:N
773     { \tl_to_other_str:N #2 }
774     \q_recursion_tail \q_recursion_stop
775     \group_end:
776     \tl_set_eq:NN #1 \g_str_result_tl
777 }
778 \cs_new_protected_nopar:Npn \str_native_from_UTF_viii_aux_i:N #1
779 {
780     \quark_if_recursion_tail_stop:N #1
781     \int_set:Nn \l_str_char_int {'#1}
782     \str_native_from_UTF_viii_aux_ii:nN { 128 } \c_one
783     \str_native_from_UTF_viii_aux_ii:nN { 64 } \c_zero
784     \str_native_from_UTF_viii_aux_ii:nN { 32 } \c_two

```

```

785 \str_native_from_UTF_viii_aux_ii:nN { 16 } \c_three
786 \str_native_from_UTF_viii_aux_ii:nN { 8 } \c_four
787 \msg_error:nnx { str } { utf8-invalid-byte } {#1}
788 \use_i:nnn \str_native_from_UTF_viii_aux_i:N
789 \q_stop
790 \str_native_from_UTF_viii_aux_iii:N
791 }
792 \cs_new_protected_nopar:Npn \str_native_from_UTF_viii_aux_ii:nN #1#2
793 {
794 \int_compare:nNnTF \l_str_char_int < {#1}
795 {
796 \int_set_eq:NN \l_str_bytes_int #2
797 \use_none_delimit_by_q_stop:w
798 }
799 { \int_sub:Nn \l_str_char_int {#1} }
800 }
801 \cs_new_protected_nopar:Npn \str_native_from_UTF_viii_aux_iii:N #1
802 {
803 \int_compare:nNnTF \l_str_bytes_int < \c_two
804 { \str_native_from_UTF_viii_aux_iv: #1 }
805 {
806 \if_meaning:w \q_recursion_tail #1
807 \msg_error:nn { str } { utf8-premature-end }
808 \exp_after:wN \use_none_delimit_by_q_recursion_stop:w
809 \fi:
810 \quark_if_recursion_tail_stop:N #1
811 \tex_multiply:D \l_str_char_int by 64 \scan_stop:
812 \int_decr:N \l_str_bytes_int
813 \int_compare:nNnTF {'#1} < { 128 }
814 { \use_ii:nn }
815 { \int_compare:nNnTF {'#1} < { 192 } }
816 {
817 \int_add:Nn \l_str_char_int { '#1 - 128 }
818 \str_native_from_UTF_viii_aux_iii:N
819 }
820 {
821 \msg_error:nnx { str } { utf8-missing-byte } {#1}
822 \str_native_from_UTF_viii_aux_i:N #1
823 }
824 }
825 }
826 \cs_new_protected_nopar:Npn \str_native_from_UTF_viii_aux_iv:
827 {
828 \int_compare:nNnTF \l_str_bytes_int = \c_zero
829 {
830 \msg_error:nnx { str } { utf8-extra-byte }
831 { \int_eval:n { \l_str_char_int + 128 } }
832 }
833 { \str_native_from_UTF_viii_aux_v: }
834 \str_native_from_UTF_viii_aux_i:N

```

```

835 }
836 \pdfTeX_if_engine:TF
837 {
838   \cs_new_protected_nopar:Npn \str_native_from_UTF_viii_aux_v:
839   {
840     \int_compare:nNnTF { \l_str_char_int } < { 256 }
841     { \str_aux_convert_store: }
842     {
843       \msg_error:nnx { str } { utf8-pdfTeX-overflow }
844       { \int_use:N \l_str_char_int }
845     }
846   }
847 }
848 {
849   \cs_new_protected_nopar:Npn \str_native_from_UTF_viii_aux_v:
850   { \str_aux_convert_store: }
851 }

```

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

\str_UTF_viii_from_native:NN

```

852 \cs_new_protected_nopar:Npn \str_UTF_viii_from_native:NN #1#2
853 {
854   \group_begin:
855   \tl_gclear:N \g_str_result_tl
856   \str_map_tokens:Nn #2 \str_UTF_viii_from_native_aux_i:N
857   \group_end:
858   \tl_set_eq:NN #1 \g_str_result_tl
859 }
860 \cs_new_protected_nopar:Npn \str_UTF_viii_from_native_aux_i:N #1
861 {
862   \int_set:Nn \l_str_char_int {'#1}
863   \int_compare:nNnTF \l_str_char_int < { 128 }
864   { \tl_gput_right:Nx \g_str_result_tl {#1} }
865   {
866     \tl_gclear:N \g_str_tmpa_tl
867     \int_set:Nn \l_str_bytes_int { 64 }
868     \str_UTF_viii_from_native_aux_ii:n { 32 }
869     \str_UTF_viii_from_native_aux_ii:n { 16 }
870     \str_UTF_viii_from_native_aux_ii:n { 8 }
871     \ERROR % somehow the unicode char was > "1FFFFFF > "10FFFF
872     \tl_gclear:N \g_str_tmpa_tl
873     \use_none:n \q_stop
874     \int_add:Nn \l_str_char_int { 2 * \l_str_bytes_int }
875     \str_aux_convert_store:
876     \tl_gput_right:Nx \g_str_result_tl { \g_str_tmpa_tl }
877   }
878 }
879 \cs_new_protected_nopar:Npn \str_UTF_viii_from_native_aux_ii:n #1
880 {
881   \str_aux_convert_store:NNn

```

```

882 \tl_gput_left:Nx \g_str_tmpa_tl
883 { 128 + \int_mod:nn { \l_str_char_int } { 64 } }
884 \tex_divide:D \l_str_char_int by 64 \scan_stop:
885 \int_add:Nn \l_str_bytes_int {#1}
886 \int_compare:nNnT \l_str_char_int < {#1}
887 { \use_none_delimit_by_q_stop:w }
888 }

```

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

1.9 Messages

```

889 \msg_new:nnnn { str } { x-missing-brace }
890 { Missing-closing-brace-in~ \token_to_str:N \x ~byte-sequence. }
891 {
892   You-wrote-something-like~
893   '\iow_char:N\{ \int_to_hexadecimal:n { \l_str_char_int }' .~
894   The-closing-brace-is-missing.
895 }
896 \msg_new:nnn { str } { utf8-invalid-byte }
897 {
898   \int_compare:nNnTF {'#1} < { 256 }
899   { Byte-number~ \int_eval:n {'#1} ~invalid-in-utf-8-encoding. }
900   { The-character-number~ \int_eval:n {'#1} ~is-not-a-byte. }
901 }
902 \msg_new:nnn { str } { utf8-missing-byte }
903 { The-byte-number~ \int_eval:n {'#1} ~is-not-a-valid-continuation-byte. }
904 \msg_new:nnn { str } { utf8-extra-byte }
905 { The-byte-number~ \int_eval:n {'#1} ~is-only-valid-as-a-continuation-byte. }
906 \msg_new:nnnn { str } { utf8-premature-end }
907 { Incomplete-last-UTF8-character. }
908 {
909   The-sequence-of-byte-that-to-be-converted-to-UTF8-
910   ended-before-the-last-character-was-complete.~Perhaps~
911   it-was-mistakenly-truncated?
912 }
913 \msg_new:nnn { str } { utf8-pdftex-overflow }
914 { The-character-number-#1-is-too-big-for-pdfTeX. }

```

1.10 Deprecated string functions

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

```

915 \cs_set:Npn \str_length_skip_spaces:N
916 { \exp_args:No \str_length_skip_spaces:n }
917 \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 ??.)

```

918 </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.

Symbols	
\#	28, 593
\%	29, 729
*	31, 593, 635, 729
\\	25, 635, 893
\{	26, 893
\}	27
\^	403, 405, 407, 409, 411, 413, 466, 482, 636, 637
_	31
A	
\A	32
C	
\c_backslash_str	1, <u>25</u> , 25, 388, 396, 399, 543, 552, 555, 558, 670
\c_eight	76, 346, 508, 509
\c_fifteen	333
\c_five	77
\c_four	77, 786
\c_hash_str	1, <u>25</u> , 28, 523, 532, 535, 538, 615, 629
\c_lbrace_str	1, <u>25</u> , 26, 425, 523
\c_minus_one	676
\c_nine	87
\c_one	78, 782
\c_percent_str	1, <u>25</u> , 29, 523, 716, 720, 723, 751
\c_rbrace_str	1, <u>25</u> , 27, 458, 523
\c_seven	76, 120, 128, 149, 157, 344
\c_six	76
\c_sixteen	329, 335, 498, 499
\c_space_token	422, 438, 448
\c_str_bytes_escape_name_str	522, 537
\c_str_bytes_escape_string_str	543, 557
\c_str_bytes_percent_encode_not_str	704, 714
\c_str_bytes_percent_encode_str	703, 722
\c_three	77, 785
\c_two	78, 784, 803
\c_two_hundred_fifty_six	318
\c_zero	78, 196, 257, 259, 291, 470, 485, 490, 783, 828
\char_set_catcode_other:N	466, 636, 637
\char_set_catcode_other:n	482
\char_set_lccode:nn	31, 32, 470, 485, 490, 593, 635, 729
\cs_generate_variant:Nn	11
\cs_if_exist:Nf	4
\cs_if_exist_use:cF	384, 391
\cs_new:Npn	5, 36, 54, 56, 63, 66, 71, 98, 110, 118, 127, 129, 143, 145, 147, 154, 160, 174, 176, 179, 184, 215, 220, 222, 231, 237, 246, 253, 276
\cs_new_eq:cN	395, 398
\cs_new_eq:NN	287, 364–366
\cs_new_nopar:cpn	388, 401, 414
\cs_new_nopar:cpx	402, 404, 406, 408, 410, 412
\cs_new_nopar:Npn	41, 50, 53, 62, 64, 89, 100, 102, 115, 117, 178, 186, 194, 214, 272, 274, 281, 303, 309, 316, 382, 494, 503, 529, 549, 710
\cs_new_protected:cpx	9
\cs_new_protected:Npn	354, 359, 367
\cs_new_protected_nopar:Npn	20, 352, 430, 436, 446, 467, 483, 488, 517, 524, 544, 563, 574, 581, 597, 608, 620, 641, 651, 682, 691, 705, 733, 743, 756, 768, 778, 792, 801, 826, 838, 849, 852, 860, 879
\cs_new_protected_nopar:Npx	420, 456
\cs_set:Npn	82, 90, 103, 915
\cs_set_eq:NN	917
\cs_set_nopar:Npn	96, 109, 370–372
\cs_set_protected_nopar:Npn	7
\cs_to_str:N	25–29
E	
\else:	122, 140, 171, 260, 263, 266, 291, 337, 348
\ERROR	871
\exp_after:wN	38, 69, 85, 92, 99, 105, 112, 121, 123, 248–251, 377, 808
\exp_args:Nc	11

\prg_new_protected_conditional:Npnn	\str_aux_escape_loop:N
..... 324, 342	364, 377, 382, 386, 389, 393, 414, 477
\prg_return_false:	\str_aux_escape_raw:N
..... 291, 296, 301, 309, 321, 338, 349	364, 366, 372, 403, 405, 407, 409, 411, 413, 476
\prg_return_true:	\str_aux_escape_unescaped:N
..... 291, 310, 314, 330, 336, 347 364, 364, 370, 385
\ProvidesExplPackage	\str_aux_escape_x_braced_end:N
2 414, 453, 456
Q	\str_aux_escape_x_braced_loop:N
\q_mark 414, 426, 446, 449, 452
39, 50	\str_aux_escape_x_end:
\q_recursion_stop 414, 434, 442, 443, 459, 462, 467
..... 279, 378, 570, 604, 647, 739, 774	\str_aux_escape_x_test:N
\q_recursion_tail 414, 418, 420, 423
..... 378, 570, 604, 647, 739, 774, 806	\str_aux_escape_x_unbraced_i:N
\q_stop 414, 427, 430
39, 42, 48, 50, 79, 94, 99, 101, 107, 109, 113, 116, 117, 175, 176, 192, 229, 789, 873	\str_aux_escape_x_unbraced_ii:N
\quark_if_recursion_tail_stop:N 414, 433, 436, 439
..... 576, 583, 611, 654, 746, 780, 810	\str_aux_eval_args:Nnnn
R 214, 224, 246
\reverse_if:N	\str_aux_hexadecimal_test:N
..... 106 324
S	\str_aux_hexadecimal_test:NTF
\scan_stop: 324, 432, 441, 451, 577, 584, 612, 622, 748, 758
107, 811, 884	\str_aux_normalize_range:nn
\str_aux_byte_to_hexadecimal:N 214, 234, 235, 253
494, 494, 520, 532, 535, 538, 716, 720, 723	\str_aux_octal_test:N
\str_aux_byte_to_octal:N 342
..... 494, 503, 552, 555	\str_aux_octal_test:NTF
\str_aux_char_if_octal_digit:NTF 342, 656, 684, 693
..... 352, 352	\str_bytes_escape_hexadecimal:NN
\str_aux_convert_store: 6, 517, 517
..... 481, 483, 586, 624, 677, 687, 695, 699, 760, 841, 850, 875	\str_bytes_escape_name:NN
\str_aux_convert_store:NNn 6, 522, 524
481, 488, 881	\str_bytes_escape_name_aux:N
\str_aux_escape:NNNn 527, 529
8, 356, 361, 364, 367	\str_bytes_escape_string:NN
\str_aux_escape_□:w 7, 543, 544
..... 395	\str_bytes_escape_string_aux:N
\str_aux_escape_/\q_recursion_tail:w 547, 549
..... 395	\str_bytes_percent_decode:NN
\str_aux_escape_/a:w 7, 728, 733
..... 395	\str_bytes_percent_decode_aux:wN
\str_aux_escape_/e:w 728, 737, 743, 752, 761, 765
..... 395	\str_bytes_percent_decode_aux_ii:NN
\str_aux_escape_/f:w 728, 749, 756
..... 395	\str_bytes_percent_encode:NN
\str_aux_escape_/n:w 7, 703, 705
..... 395	\str_bytes_percent_encode_aux:N
\str_aux_escape_/r:w 708, 710
..... 395	\str_bytes_unescape_hexadecimal:NN
\str_aux_escape_/t:w 6, 563, 563
..... 395	\str_bytes_unescape_hexadecimal_aux:N
\str_aux_escape_/x:w 563, 568, 574, 579, 588
..... 414	\str_bytes_unescape_hexadecimal_aux_ii:N
\str_aux_escape_□:w 563, 578, 581, 590
..... 364	\str_bytes_unescape_name:NN
\str_aux_escape_□\q_recursion_tail:w 6, 592, 597
..... 395	\str_bytes_unescape_name_aux:wN
\str_aux_escape_escaped:N 592, 602, 608, 616, 626, 631
..... 364, 365, 371, 392	\str_bytes_unescape_name_aux_ii:N
 592

\str_bytes_unescape_name_aux_ii:NN .	\str_input:Nn	3, 354, 354
. 613, 620	\str_item:Nn	4, 178, 178
\str_bytes_unescape_string:NN 7, 634, 641	\str_item:nn	178, 178, 179
\str_bytes_unescape_string_aux:wN .	\str_item_aux:nn	178, 188, 194
. 645, 651, 678, 688, 696, 700	\str_item_ignore_spaces:nn . .	4, 178, 184
\str_bytes_unescape_string_aux_d:N .	\str_item_unsafe:nn . . .	178, 182, 185, 186
. 657, 682	\str_length:N	3, 62, 62
\str_bytes_unescape_string_aux_dd:N	\str_length:n	62, 62, 63
. 685, 691	\str_length_aux:n	62, 65, 68, 71
\str_collect_aux:n 145, 146, 147, 156	\str_length_ignore_spaces:n	3, 62, 66, 917
\str_collect_aux:nnNNNNNNN 145, 150, 154	\str_length_loop:NNNNNNNNN	62, 65, 69, 82, 87
\str_collect_do:nn 145, 145, 241	\str_length_skip_spaces:N	915, 915
\str_collect_end:n 151, 160	\str_length_skip_spaces:n .	915, 916, 917
\str_collect_end:nn 145	\str_length_unsafe:n .	62, 63, 64, 190, 225
\str_collect_end_ii:nwn 145, 163–171, 174	\str_map_break_do:n	272, 287
\str_collect_end_iii:nwNNNNNNN	\str_map_tokens:Nn	8, 272,
. 145, 146, 176	272, 295, 313, 520, 527, 547, 708, 856	
\str_ginput:Nn 3, 354, 359	\str_map_tokens:nn 272, 273, 274, 300	
\str_gput_left:cn 6	\str_map_tokens_aux:nn . . .	272, 275, 276
\str_gput_left:cx 6	\str_map_tokens_loop:nN 272, 278, 281, 285	
\str_gput_left:Nn 2, 6	\str_native_from_UTF_viii:NN 5, 768, 768	
\str_gput_left:Nx 6	\str_native_from_UTF_viii_aux_i:N . .	768, 772, 778, 788, 822, 834
\str_gput_right:cn 6	\str_native_from_UTF_viii_aux_ii:nN	768, 782–786, 792
\str_gput_right:cx 6	\str_native_from_UTF_viii_aux_iii:N	768, 790, 801, 818
\str_gput_right:Nn 2, 6	\str_native_from_UTF_viii_aux_iv: . .	768, 804, 826
\str_gput_right:Nx 6	\str_native_from_UTF_viii_aux_v: . . .	768, 833, 838, 849
\str_gset:cn 6	\str_put_left:cn 6	
\str_gset:cx 6	\str_put_left:cx 6	
\str_gset:Nn 2, 6	\str_put_left:Nn 2, 6	
\str_gset:Nx 6	\str_put_left:Nx 6	
\str_head:N 4, 89, 89	\str_put_right:cn 6	
\str_head:n 89, 89, 90	\str_put_right:cx 6	
\str_head_aux:w 89, 92, 96	\str_put_right:Nn 2, 6	
\str_head_ignore_spaces:n 4, 89, 98	\str_put_right:Nx 6	
\str_head_unsafe:n 89, 100	\str_sanitiz_e_args:Nn	8, 54, 54, 63, 182, 218, 275
\str_if_bytes:N 311, 311	\str_sanitiz_e_args:Nnn	54, 56
\str_if_bytes:NTF 5	\str_set:cn 6	
\str_if_bytes_aux:N 311, 313, 316	\str_set:cx 6	
\str_if_contains_char:NN 293, 293	\str_set:Nn 2, 6	
\str_if_contains_char:nN 293, 298	\str_set:Nx 6, 519, 526, 546, 707	
\str_if_contains_char:NNT 557	\str_skip_aux:nnnnnnnnn . .	118, 121, 127
\str_if_contains_char:NNTF	\str_skip_do:nn 118, 118, 128, 201, 208, 239	
. 5, 537, 714, 722	\str_skip_end:n	123, 129
\str_if_contains_char_aux:NN		
. 293, 295, 300, 303		
\str_if_contains_char_end:w 293, 306, 309		
\str_if_eq:NN 288, 288		
\str_if_eq:nn 288		
\str_if_eq:NNTF 5		
\str_if_eq:xx 288		

