Contents

The l3str-convert package: string encoding conversions[*]

The LaTeX3 Project[†]

Released 2013/01/08

1 Encoding and escaping schemes

Traditionally, string encodings only specify how strings of characters should be stored as bytes. However, the resulting lists of bytes are often to be used in contexts where only a restricted subset of bytes are permitted (*e.g.*, PDF string objects, URLS). Hence, storing a string of characters is done in two steps.

- The code points ("character codes") are expressed as bytes following a given "encoding". This can be UTF-16, ISO 8859-1, *etc.* See Table 1 for a list of supported encodings.[1]

- Bytes are translated to TeX tokens through a given "escaping". Those are defined for the most part by the `pdf` file format. See Table 2 for a list of escaping methods supported.[2]

[*]This file describes v4339, last revised 2013/01/08.

[†]E-mail: latex-team@latex-project.org

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

Table 1: Supported encodings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the encoding in this list.

⟨*Encoding*⟩	description
`utf8`	UTF-8
`utf16`	UTF-16, with byte-order mark
`utf16be`	UTF-16, big-endian
`utf16le`	UTF-16, little-endian
`utf32`	UTF-32, with byte-order mark
`utf32be`	UTF-32, big-endian
`utf32le`	UTF-32, little-endian
`iso88591`, `latin1`	ISO 8859-1
`iso88592`, `latin2`	ISO 8859-2
`iso88593`, `latin3`	ISO 8859-3
`iso88594`, `latin4`	ISO 8859-4
`iso88595`	ISO 8859-5
`iso88596`	ISO 8859-6
`iso88597`	ISO 8859-7
`iso88598`	ISO 8859-8
`iso88599`, `latin5`	ISO 8859-9
`iso885910`, `latin6`	ISO 8859-10
`iso885911`	ISO 8859-11
`iso885913`, `latin7`	ISO 8859-13
`iso885914`, `latin8`	ISO 8859-14
`iso885915`, `latin9`	ISO 8859-15
`iso885916`, `latin10`	ISO 8859-16
Empty	Native (Unicode) string.

Table 2: Supported escapings. Non-alphanumeric characters are ignored, and capital letters are lower-cased before searching for the escaping in this list.

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

2 Conversion functions

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

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

This function converts the ⟨*string*⟩ from the encoding given by ⟨*name 1*⟩ to the encoding given by ⟨*name 2*⟩, and stores the result in the ⟨*str var*⟩. Each ⟨*name*⟩ can have the form ⟨*encoding*⟩ or ⟨*encoding*⟩/⟨*escaping*⟩, where the possible values of ⟨*encoding*⟩ and ⟨*escaping*⟩ are given in Tables 1 and 2, respectively. The default escaping is to input and output bytes directly. The special case of an empty ⟨*name*⟩ indicates the use of "native" strings, 8-bit for pdfTEX, and Unicode strings for the other two engines.

For example,

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

results in the variable `\l_foo_str` holding the string `FEFF00480065006C006C006F0021`. This is obtained by converting each character in the (native) string `Hello!` to the UTF-16 encoding, and expressing each byte as a pair of hexadecimal digits. Note the presence of a (big-endian) byte order mark "FEFF, which can be avoided by specifying the encoding `utf16be/hex`.

An error is raised if the ⟨*string*⟩ is not valid according to the ⟨*escaping 1*⟩ and ⟨*encoding 1*⟩, or if it cannot be reencoded in the ⟨*encoding 2*⟩ and ⟨*escaping 2*⟩ (for instance, if a character does not exist in the ⟨*encoding 2*⟩). Erroneous input is replaced by the Unicode replacement character "FFFD, and characters which cannot be reencoded are replaced by either the replacement character "FFFD if it exists in the ⟨*encoding 2*⟩, or an encoding-specific replacement character, or the question mark character.

`\str_set_convert:Nnnn`*TF*
`\str_gset_convert:Nnnn`*TF*

`\str_set_convert:NnnnTF` ⟨*str var*⟩ {⟨*string*⟩} {⟨*name 1*⟩} {⟨*name 2*⟩} {⟨*true code*⟩} {⟨*false code*⟩}

As `\str_set_convert:Nnnn`, converts the ⟨*string*⟩ from the encoding given by ⟨*name 1*⟩ to the encoding given by ⟨*name 2*⟩, and assigns the result to ⟨*str var*⟩. Contrarily to `\str_set_convert:Nnnn`, the conditional variant does not raise errors in case the ⟨*string*⟩ is not valid according to the ⟨*name 1*⟩ encoding, or cannot be expressed in the ⟨*name 2*⟩ encoding. Instead, the ⟨*false code*⟩ is performed.

`\c_max_char_int`

The maximum valid character code, 255 for pdfTEX, and 1114111 for XeTEX and LuaTEX.

3 Internal string functions

`\__str_gset_other:Nn`

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

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

`\__str_hexadecimal_use:NTF`

`\__str_hexadecimal_use:NTF ⟨token⟩ {⟨true code⟩} {⟨false code⟩}`

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

TEXhackers note: This function fails on some inputs if the escape character is a hexadecimal digit. We are thus careful to set the escape character to a known (safe) value before using it.

`\__str_output_byte:n` ⋆

`\__str_output_byte:n {⟨intexpr⟩}`

Expands to a character token with category other and character code equal to the value of ⟨*intexpr*⟩. The value of ⟨*intexpr*⟩ must be in the range $[-1, 255]$, and any value outside this range results in undefined behaviour. The special value -1 is used to produce an empty result.

4 Possibilities, and things to do

Encoding/escaping-related tasks.

- Describe the internal format in the code comments. Refuse code points in ["D800, "DFFF] in the internal representation?
- Add documentation about each encoding and escaping method, and add examples.
- The `hex` unescaping should raise an error for odd-token count strings.
- Decide what bytes should be escaped in the `url` escaping. Perhaps `!'()*-./0123456789_` are safe, and all other characters should be escaped?
- Automate generation of 8-bit mapping files.
- Change the framework for 8-bit encodings: for decoding from 8-bit to Unicode, use 256 integer registers; for encoding, use a tree-box.
- More encodings (see Heiko's stringenc). CESU?
- More escapings: shell escapes, lua escapes, etc?

5 l3str implementation

```
⟨*initex | package⟩
⟨@@=str⟩
\ProvidesExplPackage
  {\ExplFileName}{\ExplFileDate}{\ExplFileVersion}{\ExplFileDescription}
\RequirePackage{l3str,l3tl-analysis,l3tl-build,l3flag}
```

5.1 Helpers

5.1.1 A function unrelated to strings

\use_ii_i:nn

A function used to swap its arguments.

```
\cs_if_exist:NF \use_ii_i:nn
  { \cs_new:Npn \use_ii_i:nn #1#2 { #2 #1 } }
```

(*End definition for* \use_ii_i:nn.)

5.1.2 Variables and constants

\c_max_char_int

The maximum valid character code is 255 for pdfTEX, and 1114111 for other engines.

```
\int_const:Nn \c_max_char_int
  { \pdftex_if_engine:TF { "FF } { "10FFFF } }
```

(*End definition for* \c_max_char_int. *This variable is documented on page 4.*)

__str_tmp:w
\l__str_internal_int
\l__str_internal_tl

Internal scratch space for some functions.

```
\cs_new_protected_nopar:Npn \__str_tmp:w { }
\tl_new:N \l__str_internal_tl
\int_new:N \l__str_internal_int
```

(*End definition for* __str_tmp:w. *This function is documented on page* ??.)

\g__str_result_tl

The \g__str_result_tl variable is used to hold the result of various internal string operations (mostly conversions) which are typically performed in a group. The variable is global so that it remains defined outside the group, to be assigned to a user-provided variable.

```
\tl_new:N \g__str_result_tl
```

(*End definition for* \g__str_result_tl. *This variable is documented on page* ??.)

\c__str_replacement_char_int

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

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

(*End definition for* \c__str_replacement_char_int. *This variable is documented on page* ??.)

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

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

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

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

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

```
\prop_new:N \g__str_alias_prop
\prop_gput:Nnn \g__str_alias_prop { latin1 } { iso88591 }
\prop_gput:Nnn \g__str_alias_prop { latin2 } { iso88592 }
\prop_gput:Nnn \g__str_alias_prop { latin3 } { iso88593 }
\prop_gput:Nnn \g__str_alias_prop { latin4 } { iso88594 }
\prop_gput:Nnn \g__str_alias_prop { latin5 } { iso88599 }
\prop_gput:Nnn \g__str_alias_prop { latin6 } { iso885910 }
\prop_gput:Nnn \g__str_alias_prop { latin7 } { iso885913 }
\prop_gput:Nnn \g__str_alias_prop { latin8 } { iso885914 }
\prop_gput:Nnn \g__str_alias_prop { latin9 } { iso885915 }
\prop_gput:Nnn \g__str_alias_prop { latin10 } { iso885916 }
\prop_gput:Nnn \g__str_alias_prop { utf16le } { utf16 }
\prop_gput:Nnn \g__str_alias_prop { utf16be } { utf16 }
\prop_gput:Nnn \g__str_alias_prop { utf32le } { utf32 }
\prop_gput:Nnn \g__str_alias_prop { utf32be } { utf32 }
\prop_gput:Nnn \g__str_alias_prop { hexadecimal } { hex }
```

(*End definition for* `\g__str_alias_prop`. *This variable is documented on page* ??.)

`\g__str_error_bool` In conversion functions with a built-in conditional, errors are not reported directly to the user, but the information is collected in this boolean, used at the end to decide on which branch of the conditional to take.

```
\bool_new:N \g__str_error_bool
```

(*End definition for* `\g__str_error_bool`. *This variable is documented on page* ??.)

`str_byte`
`str_error` Conversions from one ⟨*encoding*⟩/⟨*escaping*⟩ pair to another are done within `x`-expanding assignments. Errors are signalled by raising the relevant flag.

```
\flag_new:n { str_byte }
\flag_new:n { str_error }
```

(*End definition for* `str_byte` *and* `str_error`. *These variables are documented on page* ??.)

5.1.3 Escaping spaces

`\__str_gset_other:Nn`
`\__str_gset_other_loop:w`
`\__str_gset_other_end:w` This function could be done by using `\__str_to_other:n` within an `x`-expansion, but that would take a time quadratic in the size of the string. Instead, we can "leave the result behind us" in the input stream, to be captured into the expanding assignment. This gives us a linear time.

```
\group_begin:
\char_set_lccode:nn { `\* } { `\  }
\char_set_lccode:nn { `\A } { `\A }
\tl_to_lowercase:n
  {
    \group_end:
    \cs_new_protected:Npn \__str_gset_other:Nn #1#2
      {
        \tl_gset:Nx #1
          {
```

```
          \exp_after:wN \__str_gset_other_loop:w \tl_to_str:n {#2} ~ %
          A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ A ~ \q_stop
        }
    }
  \cs_new:Npn \__str_gset_other_loop:w
    #1 ~ #2 ~ #3 ~ #4 ~ #5 ~ #6 ~ #7 ~ #8 ~ #9 ~
    {
      \if_meaning:w A #9
        \__str_gset_other_end:w
      \fi:
      #1 * #2 * #3 * #4 * #5 * #6 * #7 * #8 * #9
      \__str_gset_other_loop:w *
    }
  \cs_new:Npn \__str_gset_other_end:w \fi: #1 * A #2 \q_stop
    { \fi: #1 }
}
```

(*End definition for* `\__str_gset_other:Nn`. *This function is documented on page 4.*)

5.2 String conditionals

`\__str_if_contains_char:NNT`
`\__str_if_contains_char:NNTF`
`\__str_if_contains_char:nNTF`
`\__str_if_contains_char_aux:NN`
`\__str_if_contains_char_true:`

Expects the ⟨*token list*⟩ to be an ⟨*other string*⟩: the caller is responsible for ensuring that no (too-)special catcodes remain. Spaces with catcode 10 are ignored. Loop over the characters of the string, comparing character codes. The loop is broken if character codes match. Otherwise we return "false".

```
\prg_new_conditional:Npnn \__str_if_contains_char:NN #1#2 { T , TF }
  {
    \exp_after:wN \__str_if_contains_char_aux:NN \exp_after:wN #2
      #1 { \__prg_break:n { ? \fi: } }
    \__prg_break_point:
    \prg_return_false:
  }
\prg_new_conditional:Npnn \__str_if_contains_char:nN #1#2 { TF }
  {
    \__str_if_contains_char_aux:NN #2 #1 { \__prg_break:n { ? \fi: } }
    \__prg_break_point:
    \prg_return_false:
  }
\cs_new:Npn \__str_if_contains_char_aux:NN #1#2
  {
    \if_charcode:w #1 #2
      \exp_after:wN \__str_if_contains_char_true:
    \fi:
    \__str_if_contains_char_aux:NN #1
  }
\cs_new_nopar:Npn \__str_if_contains_char_true:
  { \__prg_break:n { \prg_return_true: \use_none:n } }
```

(*End definition for* `\__str_if_contains_char:NNT` *and* `\__str_if_contains_char:NNTF`. *These functions are documented on page* **??**.)

`\__str_octal_use:NTF`

If the ⟨*token*⟩ is an octal digit, it is left in the input stream, *followed* by the ⟨*true code*⟩. Otherwise, the ⟨*false code*⟩ is left in the input stream.

TeXhackers note: This function will fail if the escape character is an octal digit. We are thus careful to set the escape character to a known value before using it. TeX dutifully detects

octal digits for us: if `#1` is an octal digit, then the right-hand side of the comparison is `'1#1`, greater than 1. Otherwise, the right-hand side stops as `'1`, and the conditional takes the `false` branch.

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

(*End definition for* `\__str_octal_use:NTF`*.*)

`\__str_hexadecimal_use:NTF`

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

```
\prg_new_conditional:Npnn \__str_hexadecimal_use:N #1 { TF }
  {
    \if_int_compare:w \c_two < "1 \token_to_str:N #1 \exp_stop_f:
      #1 \prg_return_true:
    \else:
      \if_case:w \__int_eval:w
          \exp_after:wN ` \token_to_str:N #1 - `a
        \__int_eval_end:
           A
      \or: B
      \or: C
      \or: D
      \or: E
      \or: F
      \else:
        \prg_return_false:
        \exp_after:wN \use_none:n
      \fi:
      \prg_return_true:
    \fi:
  }
```

(*End definition for* `\__str_hexadecimal_use:NTF`*.*)

5.3 Conversions

5.3.1 Producing one byte or character

\c__str_byte_0_tl
\c__str_byte_1_tl
\c__str_byte_255_tl
\c__str_byte_-1_tl

For each integer N in the range $[0, 255]$, we create a constant token list which holds three character tokens with category code other: the character with character code N, followed by the representation of N as two hexadecimal digits. The value -1 is given a default token list which ensures that later functions give an empty result for the input -1.

```
\group_begin:
  \char_set_catcode_other:n { \c_zero }
  \tl_set:Nx \l__str_internal_tl { \tl_to_str:n { 0123456789ABCDEF } }
  \exp_args:No \tl_map_inline:nn { \l__str_internal_tl " }
    { \char_set_lccode:nn {‘#1} { \c_zero } }
  \tl_map_inline:Nn \l__str_internal_tl
    {
      \tl_map_inline:Nn \l__str_internal_tl
        {
          \char_set_lccode:nn { \c_zero } {"#1##1}
          \tl_to_lowercase:n
            {
              \tl_const:cx
                { c__str_byte_ \int_eval:n {"#1##1} _tl }
                { ^^@ #1 ##1 }
            }
        }
    }
\group_end:
\tl_const:cn { c__str_byte_-1_tl } { { } \use_none:n { } }
```

(*End definition for* \c__str_byte_0_tl *,* \c__str_byte_1_tl *, and* \c__str_byte_255_tl*. These functions are documented on page* **??***.*)

__str_output_byte:n
__str_output_byte:w
__str_output_hexadecimal:n
__str_output_hexadecimal:w
__str_output_end:

Those functions must be used carefully: feeding them a value outside the range $[-1, 255]$ will attempt to use the undefined token list variable \c__str_byte_⟨*number*⟩_tl. Assuming that the argument is in the right range, we expand the corresponding token list, and pick either the byte (first token) or the hexadecimal representations (second and third tokens). The value -1 produces an empty result in both cases.

```
\cs_new:Npn \__str_output_byte:n #1
  { \__str_output_byte:w #1 \__str_output_end: }
\cs_new_nopar:Npn \__str_output_byte:w
  {
    \exp_after:wN \exp_after:wN
    \exp_after:wN \use_i:nnn
    \cs:w c__str_byte_ \int_use:N \__int_eval:w
  }
\cs_new:Npn \__str_output_hexadecimal:n #1
  { \__str_output_hexadecimal:w #1 \__str_output_end: }
\cs_new_nopar:Npn \__str_output_hexadecimal:w
  {
    \exp_after:wN \exp_after:wN
    \exp_after:wN \use_none:n
    \cs:w c__str_byte_ \int_use:N \__int_eval:w
  }
\cs_new_nopar:Npn \__str_output_end:
  { \__int_eval_end: _tl \cs_end: }
```

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

`\__str_output_byte_pair_be:n`
`\__str_output_byte_pair_le:n`
`\__str_output_byte_pair:nnN`

Convert a number in the range $[0, 65535]$ to a pair of bytes, either big-endian or little-endian.

```
\cs_new:Npn \__str_output_byte_pair_be:n #1
  {
    \exp_args:Nf \__str_output_byte_pair:nnN
      { \int_div_truncate:nn { #1 } { "100 } } {#1} \use:nn
  }
\cs_new:Npn \__str_output_byte_pair_le:n #1
  {
    \exp_args:Nf \__str_output_byte_pair:nnN
      { \int_div_truncate:nn { #1 } { "100 } } {#1} \use_ii_i:nn
  }
\cs_new:Npn \__str_output_byte_pair:nnN #1#2#3
  {
    #3
      { \__str_output_byte:n { #1 } }
      { \__str_output_byte:n { #2 - #1 * "100 } }
  }
```

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

5.3.2 Mapping functions for conversions

`\__str_convert_gmap:N`
`\__str_convert_gmap_loop:NN`

This maps the function `#1` over all characters in `\g__str_result_tl`, which should be a byte string in most cases, sometimes a native string.

```
\cs_new_protected:Npn \__str_convert_gmap:N #1
  {
    \tl_gset:Nx \g__str_result_tl
      {
        \exp_after:wN \__str_convert_gmap_loop:NN
        \exp_after:wN #1
          \g__str_result_tl { ? \__prg_break: }
        \__prg_break_point:
      }
  }
\cs_new:Npn \__str_convert_gmap_loop:NN #1#2
  {
    \use_none:n #2
    #1#2
    \__str_convert_gmap_loop:NN #1
  }
```

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

__str_convert_gmap_internal:N
__str_convert_gmap_internal_loop:Nw

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

```
\cs_new_protected:Npn \__str_convert_gmap_internal:N #1
  {
    \tl_gset:Nx \g__str_result_tl
      {
        \exp_after:wN \__str_convert_gmap_internal_loop:Nww
        \exp_after:wN #1
          \g__str_result_tl \s__tl \q_stop \__prg_break: \s__tl
        \__prg_break_point:
      }
  }
\cs_new:Npn \__str_convert_gmap_internal_loop:Nww #1 #2 \s__tl #3 \s__tl
  {
    \use_none_delimit_by_q_stop:w #3 \q_stop
    #1 {#3}
    \__str_convert_gmap_internal_loop:Nww #1
  }
```

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

5.3.3 Error-reporting during conversion

__str_if_flag_error:nnx
__str_if_flag_no_error:nnx

When converting using the function \str_set_convert:Nnnn, errors should be reported to the user after each step in the conversion. Errors are signalled by raising some flag (typically @@_error), so here we test that flag: if it is raised, give the user an error, otherwise remove the arguments. On the other hand, in the conditional functions \str_-set_convert:NnnnTF, errors should be suppressed. This is done by changing __str_-if_flag_error:nnx into __str_if_flag_no_error:nnx locally.

```
\cs_new_protected:Npn \__str_if_flag_error:nnx #1
  {
    \flag_if_raised:nTF {#1}
      { \__msg_kernel_error:nnx { str } }
      { \use_none:nn }
  }
\cs_new_protected:Npn \__str_if_flag_no_error:nnx #1#2#3
  { \flag_if_raised:nT {#1} { \bool_gset_true:N \g__str_error_bool } }
```

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

__str_if_flag_times:nT

At the end of each conversion step, we raise all relevant errors as one error message, built on the fly. The height of each flag indicates how many times a given error was encountered. This function prints #2 followed by the number of occurrences of an error if it occurred, nothing otherwise.

```
\cs_new_protected:Npn \__str_if_flag_times:nT #1#2
  { \flag_if_raised:nT {#1} { #2~(x \flag_height:n {#1} ) } }
```

(*End definition for* `\__str_if_flag_times:nT`.)

5.3.4 Framework for conversions

Most functions in this module expect to be working with "native" strings. Strings can also be stored as bytes, in one of many encodings, for instance UTF8. The bytes themselves can be expressed in various ways in terms of TeX tokens, for instance as pairs of hexadecimal digits. The questions of going from arbitrary Unicode code points to bytes, and from bytes to tokens are mostly independent.

Conversions are done in four steps:

- "unescape" produces a string of bytes;
- "decode" takes in a string of bytes, and converts it to a list of Unicode characters in an internal representation, with items of the form

  ⟨*bytes*⟩ `\s__tl` ⟨*Unicode code point*⟩ `\s__tl`

  where we have collected the ⟨*bytes*⟩ which combined to form this particular Unicode character, and the ⟨*Unicode code point*⟩ is in the range [0, "10FFFF].
- "encode" encodes the internal list of code points as a byte string in the new encoding;
- "escape" escapes bytes as requested.

The process is modified in case one of the encoding is empty (or the conversion function has been set equal to the empty encoding because it was not found): then the unescape or escape step is ignored, and the decode or encode steps work on tokens instead of bytes. Otherwise, each step must ensure that it passes a correct byte string or internal string to the next step.

`\str_set_convert:Nnnn`
`\str_gset_convert:Nnnn`
`\str_set_convert:Nnnn`*TF*
`\str_gset_convert:Nnnn`*TF*
`\__str_convert:nNNnnn`

The input string is stored in `\g__str_result_tl`, then we: unescape and decode; encode and escape; exit the group and store the result in the user's variable. The various conversion functions all act on `\g__str_result_tl`. Errors are silenced for the conditional functions by redefining `\__str_if_flag_error:nnx` locally.

```
\cs_new_protected_nopar:Npn \str_set_convert:Nnnn
  { \__str_convert:nNNnnn { } \tl_set_eq:NN }
\cs_new_protected_nopar:Npn \str_gset_convert:Nnnn
  { \__str_convert:nNNnnn { } \tl_gset_eq:NN }
\prg_new_protected_conditional:Npnn
    \str_set_convert:Nnnn #1#2#3#4 { T , F , TF }
  {
    \bool_gset_false:N \g__str_error_bool
    \__str_convert:nNNnnn
      { \cs_set_eq:NN \__str_if_flag_error:nnx \__str_if_flag_no_error:nnx }
      \tl_set_eq:NN #1 {#2} {#3} {#4}
    \bool_if:NTF \g__str_error_bool \prg_return_false: \prg_return_true:
  }
\prg_new_protected_conditional:Npnn
    \str_gset_convert:Nnnn #1#2#3#4 { T , F , TF }
  {
    \bool_gset_false:N \g__str_error_bool
    \__str_convert:nNNnnn
      { \cs_set_eq:NN \__str_if_flag_error:nnx \__str_if_flag_no_error:nnx }
      \tl_gset_eq:NN #1 {#2} {#3} {#4}
    \bool_if:NTF \g__str_error_bool \prg_return_false: \prg_return_true:
  }
\cs_new_protected:Npn \__str_convert:nNNnnn #1#2#3#4#5#6
  {
    \group_begin:
      #1
      \__str_gset_other:Nn \g__str_result_tl {#4}
      \exp_after:wN \__str_convert:wwwnn
        \tl_to_str:n {#5} /// \q_stop
        { decode } { unescape }
        \prg_do_nothing:
        \__str_convert_decode_:
      \exp_after:wN \__str_convert:wwwnn
        \tl_to_str:n {#6} /// \q_stop
        { encode } { escape }
        \use_ii_i:nn
        \__str_convert_encode_:
    \group_end:
    #2 #3 \g__str_result_tl
  }
```

(*End definition for* `\str_set_convert:Nnnn` *and* `\str_gset_convert:Nnnn`*. These functions are documented on page 4.*)

`\__str_convert:wwwnn`
`\__str_convert:NNnNN`

The task of `\__str_convert:wwwnn` is to split ⟨*encoding*⟩/⟨*escaping*⟩ pairs into their components, `#1` and `#2`. Calls to `\__str_convert:nnn` ensure that the corresponding conversion functions are defined. The third auxiliary does the main work.

- `#1` is the encoding conversion function;
- `#2` is the escaping function;
- `#3` is the escaping name for use in an error message;
- `#4` is `\prg_do_nothing:` for unescaping/decoding, and `\use_ii_i:nn` for encoding/escaping;
- `#5` is the default encoding function (either "decode" or "encode"), for which there should be no escaping.

Let us ignore the native encoding for a second. In the unescaping/decoding phase, we want to do `#2#1` in this order, and in the encoding/escaping phase, the order should be reversed: `#4#2#1` does exactly that. If one of the encodings is the default (native), then the escaping should be ignored, with an error if any was given, and only the encoding, `#1`, should be performed.

```
\cs_new_protected:Npn \__str_convert:wwwnn
    #1 / #2 // #3 \q_stop #4#5
```

```
  {
    \__str_convert:nnn {enc} {#4} {#1}
    \__str_convert:nnn {esc} {#5} {#2}
    \exp_args:Ncc \__str_convert:NNnNN
      { __str_convert_#4_#1: } { __str_convert_#5_#2: } {#2}
  }
\cs_new_protected:Npn \__str_convert:NNnNN #1#2#3#4#5
  {
    \if_meaning:w #1 #5
      \tl_if_empty:nF {#3}
        { \__msg_kernel_error:nnx { str } { native-escaping } {#3} }
      #1
    \else:
      #4 #2 #1
    \fi:
  }
```

(*End definition for* `\__str_convert:wwwnn`*. This function is documented on page 4.*)

`\__str_convert:nnn`
`\__str_convert:nnnn`

The arguments of `\__str_convert:nnn` are: `enc` or `esc`, used to build filenames, the type of the conversion (unescape, decode, encode, escape), and the encoding or escaping name. If the function is already defined, no need to do anything. Otherwise, filter out all non-alphanumerics in the name, and lowercase it. Feed that, and the same three arguments, to `\__str_convert:nnnn`. The task is then to make sure that the conversion function `#3_#1` corresponding to the type `#3` and filtered name `#1` is defined, then set our initial conversion function `#3_#4` equal to that.

How do we get the `#3_#1` conversion to be defined if it isn't? Two main cases.

First, if `#1` is a key in `\g__str_alias_prop`, then the value `\l__str_internal_tl` tells us what file to load. Loading is skipped if the file was already read, *i.e.*, if the conversion command based on `\l__str_internal_tl` already exists. Otherwise, try to load the file; if that fails, there is an error, use the default empty name instead.

Second, `#1` may be absent from the property list. The `\cs_if_exist:cF` test is automatically false, and we search for a file defining the encoding or escaping `#1` (this should allow third-party `.def` files). If the file is not found, there is an error, use the default empty name instead.

In all cases, the conversion based on `\l__str_internal_tl` is defined, so we can set the `#3_#1` function equal to that. In some cases (*e.g.*, `utf16be`), the `#3_#1` function is actually defined within the file we just loaded, and it is different from the `\l__str_internal_tl`-based function: we mustn't clobber that different definition.

```
\cs_new_protected:Npn \__str_convert:nnn #1#2#3
  {
    \cs_if_exist:cF { __str_convert_#2_#3: }
      {
        \exp_args:Nx \__str_convert:nnnn
          { \__str_convert_lowercase_alphanum:n {#3} }
          {#1} {#2} {#3}
      }
  }
\cs_new_protected:Npn \__str_convert:nnnn #1#2#3#4
```

```
  {
    \cs_if_exist:cF { __str_convert_#3_#1: }
      {
        \prop_get:NnNF \g__str_alias_prop {#1} \l__str_internal_tl
          { \tl_set:Nn \l__str_internal_tl {#1} }
        \cs_if_exist:cF { __str_convert_#3_ \l__str_internal_tl : }
          {
            \file_if_exist:nTF { l3str-#2- \l__str_internal_tl .def }
              {
                \group_begin:
                  \__str_load_catcodes:
                  \file_input:n { l3str-#2- \l__str_internal_tl .def }
                \group_end:
              }
              {
                \tl_clear:N \l__str_internal_tl
                \__msg_kernel_error:nnxx { str } { unknown-#2 } {#4} {#1}
              }
          }
        \cs_if_exist:cF { __str_convert_#3_#1: }
          {
            \cs_gset_eq:cc { __str_convert_#3_#1: }
              { __str_convert_#3_ \l__str_internal_tl : }
          }
      }
    \cs_gset_eq:cc { __str_convert_#3_#4: } { __str_convert_#3_#1: }
  }
```

(*End definition for* `\__str_convert:nnn`*. This function is documented on page 4.*)

`\__str_convert_lowercase_alphanum:n`
`\__str_convert_lowercase_alphanum_loop:N`

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

```
\cs_new:Npn \__str_convert_lowercase_alphanum:n #1
  {
    \exp_after:wN \__str_convert_lowercase_alphanum_loop:N
      \tl_to_str:n {#1} { ? \__prg_break: }
    \__prg_break_point:
  }
\cs_new:Npn \__str_convert_lowercase_alphanum_loop:N #1
  {
    \use_none:n #1
    \if_int_compare:w `#1 < \c_ninety_one
      \if_int_compare:w `#1 < \c_sixty_five
        \if_int_compare:w \c_one < 1#1 \exp_stop_f:
          #1
        \fi:
      \else:
        \__str_output_byte:n { `#1 + \c_thirty_two }
      \fi:
    \else:
      \if_int_compare:w '#1 < \c_one_hundred_twenty_three
        \if_int_compare:w '#1 < \c_ninety_seven
        \else:
          #1
        \fi:
      \fi:
    \fi:
    \__str_convert_lowercase_alphanum_loop:N
  }
```

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

`\__str_load_catcodes:` Since encoding files may be loaded at arbitrary places in a TEX document, including within verbatim mode, we set the catcodes of all characters appearing in any encoding definition file.

```
\cs_new_protected:Npn \__str_load_catcodes:
  {
    \char_set_catcode_escape:N \\
    \char_set_catcode_group_begin:N \{
    \char_set_catcode_group_end:N \}
    \char_set_catcode_math_toggle:N \$
    \char_set_catcode_alignment:N \&
    \char_set_catcode_parameter:N \#
    \char_set_catcode_math_superscript:N \^
    \char_set_catcode_ignore:N \ %
    \char_set_catcode_space:N \~
    \tl_map_function:nN { abcdefghijklmnopqrstuvwxyz_:ABCDEFILNPSTUX }
      \char_set_catcode_letter:N
    \tl_map_function:nN { 0123456789"'?*+-.(),'!/<>[];= }
      \char_set_catcode_other:N
    \char_set_catcode_comment:N \%
    \int_set:Nn \tex_endlinechar:D {32}
  }
```

(*End definition for* `\__str_load_catcodes:`.)

5.3.5 Byte unescape and escape

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

`\__str_filter_bytes:n`
`\__str_filter_bytes_aux:N`

In the case of pdfTEX, every character is a byte. For Unicode-aware engines, test the character code; non-bytes cause us to raise the flag `str_byte`. Spaces have already been given the correct category code when this function is called.

```
\pdftex_if_engine:TF
  { \cs_new_eq:NN \__str_filter_bytes:n \use:n }
  {
    \cs_new:Npn \__str_filter_bytes:n #1
      {
        \__str_filter_bytes_aux:N #1
        { ? \__prg_break: }
      \__prg_break_point:
    }
  \cs_new:Npn \__str_filter_bytes_aux:N #1
    {
      \use_none:n #1
      \if_int_compare:w `#1 < 256 \exp_stop_f:
        #1
      \else:
        \flag_raise:n { str_byte }
      \fi:
      \__str_filter_bytes_aux:N
    }
}
```

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

`\__str_convert_unescape_:`
`\__str_convert_unescape_bytes:`

The simplest unescaping method removes non-bytes from `\g__str_result_tl`.

```
\pdftex_if_engine:TF
  { \cs_new_protected_nopar:Npn \__str_convert_unescape_: { } }
  {
    \cs_new_protected_nopar:Npn \__str_convert_unescape_:
      {
        \flag_clear:n { str_byte }
        \tl_gset:Nx \g__str_result_tl
          { \exp_args:No \__str_filter_bytes:n \g__str_result_tl }
        \__str_if_flag_error:nnx { str_byte } { non-byte } { bytes }
      }
  }
\cs_new_eq:NN \__str_convert_unescape_bytes: \__str_convert_unescape_:
```

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

`\__str_convert_escape_:`
`\__str_convert_escape_bytes:`

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

```
\cs_new_protected_nopar:Npn \__str_convert_escape_: { }
\cs_new_eq:NN \__str_convert_escape_bytes: \__str_convert_escape_:
```

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

5.3.6 Native strings

`\__str_convert_decode_:`
`\__str_decode_native_char:N`

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

```
\cs_new_protected_nopar:Npn \__str_convert_decode_:
  { \__str_convert_gmap:N \__str_decode_native_char:N }
\cs_new:Npn \__str_decode_native_char:N #1
  { #1 \s__tl \__int_value:w `#1 \s__tl }
```

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

`\__str_convert_encode_:`

The conversion from an internal string to native character tokens is very different in pdfTEX and in other engines. For Unicode-aware engines, we need the definitions to be read when the null byte has category code 12, so we set that inside a group.

```
\group_begin:
  \char_set_catcode_other:n { 0 }
  \pdftex_if_engine:TF
```

`\__str_encode_native_char:n`

Since pdfTEX only supports 8-bit characters, and we have a table of all bytes, the conversion can be done in linear time within an x-expanding assignment. Look out for character codes larger than 255, those characters are replaced by ?, and raise a flag, which then triggers a pdfTEX-specific error.

```
    {
      \cs_new_protected_nopar:Npn \__str_convert_encode_:
        {
          \flag_clear:n { str_error }
          \__str_convert_gmap_internal:N \__str_encode_native_char:n
          \__str_if_flag_error:nnx { str_error }
            { pdfTeX-native-overflow } { }
        }
      \cs_new:Npn \__str_encode_native_char:n #1
        {
          \if_int_compare:w #1 < \c_two_hundred_fifty_six
            \__str_output_byte:n {#1}
          \else:
            \flag_raise:n { str_error }
            ?
          \fi:
        }
      \__msg_kernel_new:nnnn { str } { pdfTeX-native-overflow }
        { Character~code~too~large~for~pdfTeX. }
        {
          The~pdfTeX~engine~only~supports~8-bit~characters:~
          valid~character~codes~are~in~the~range~[0,255].~
          To~manipulate~arbitrary~Unicode,~use~LuaTeX~or~XeTeX.
        }
    }
```

`\__str_encode_native_loop:w`
`\__str_encode_native_flush:`
`\__str_encode_native_filter:N`

In Unicode-aware engines, since building particular characters cannot be done expandably in TEX, we cannot hope to get a linear-time function. However, we get quite close using the l3tl-build module, which abuses `\toks` to reach an almost linear time. Use the standard lowercase trick to produce an arbitrary character from the null character, and add that character to the end of the token list being built. At the end of the loop, put the token list together with `\__tl_build_end:`. Note that we use an x-expanding assignment because it is slightly faster. Unicode-aware engines will never incur an overflow because the internal string is guaranteed to only contain code points in $[0, "10FFFF]$.

```
    {
      \cs_new_protected_nopar:Npn \__str_convert_encode_:
        {
          \int_zero:N \l__tl_build_offset_int
          \__tl_gbuild_x:Nw \g__str_result_tl
            \exp_after:wN \__str_encode_native_loop:w
              \g__str_result_tl \s__tl { \q_stop \__prg_break: } \s__tl
            \__prg_break_point:
          \__tl_build_end:
        }
      \cs_new_protected:Npn \__str_encode_native_loop:w #1 \s__tl #2 \s__tl
        {
          \use_none_delimit_by_q_stop:w #2 \q_stop
          \tex_lccode:D \l__str_internal_int \__int_eval:w #2 \__int_eval_end:
          \tl_to_lowercase:n { \__tl_build_one:n { ^^@ } }
          \__str_encode_native_loop:w
        }
    }
```

End the group to restore the catcode of the null byte.

```
\group_end:
```

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

5.3.7 8-bit encodings

This section will be entirely rewritten: it is not yet clear in what situations 8-bit encodings are used, hence I don't know what exactly should be optimized. The current approach is reasonably efficient to convert long strings, and it scales well when using many different encodings. An approach based on csnames would have a smaller constant load time for each individual conversion, but has a large hash table cost. Using a range of `\count` registers works for decoding, but not for encoding: one possibility there would be to use a binary tree for the mapping of Unicode characters to bytes, stored as a box, one per encoding.

Since the section is going to be rewritten, documentation lacks.

All the 8-bit encodings which l3str supports rely on the same internal functions.

`\__str_declare_eight_bit_encoding:nnn` This declares the encoding ⟨*name*⟩ to map bytes to Unicode characters according to the ⟨*mapping*⟩, and map those bytes which are not mentionned in the ⟨*mapping*⟩ either to the replacement character (if they appear in ⟨*missing*⟩), or to themselves.

All the 8-bit encoding definition file start with `\__str_declare_eight_bit_encoding:nnn {`⟨*encoding name*⟩`} {`⟨*mapping*⟩`} {`⟨*missing bytes*⟩`}`. The ⟨*mapping*⟩ argument is a token list of pairs `{`⟨*byte*⟩`} {`⟨*Unicode*⟩`}` expressed in uppercase hexadecimal notation. The ⟨*missing*⟩ argument is a token list of `{`⟨*byte*⟩`}`. Every ⟨*byte*⟩ which does not appear in the ⟨*mapping*⟩ nor the ⟨*missing*⟩ lists maps to the same code point in Unicode.

```
\cs_new_protected:Npn \__str_declare_eight_bit_encoding:nnn #1#2#3
  {
    \tl_set:Nn \l__str_internal_tl {#1}
    \cs_new_protected_nopar:cpn { __str_convert_decode_#1: }
      { \__str_convert_decode_eight_bit:n {#1} }
    \cs_new_protected_nopar:cpn { __str_convert_encode_#1: }
      { \__str_convert_encode_eight_bit:n {#1} }
    \tl_const:cn { c__str_encoding_#1_tl } {#2}
    \tl_const:cn { c__str_encoding_#1_missing_tl } {#3}
  }
```

(*End definition for* `\__str_declare_eight_bit_encoding:nnn`.)

__str_convert_decode_eight_bit:n
__str_decode_eight_bit_load:nn
__str_decode_eight_bit_load_missing:n
__str_decode_eight_bit_char:N

```
\cs_new_protected:Npn \__str_convert_decode_eight_bit:n #1
  {
    \group_begin:
      \int_zero:N \l__str_internal_int
      \exp_last_unbraced:Nx \__str_decode_eight_bit_load:nn
        { \tl_use:c { c__str_encoding_#1_tl } }
        { \q_stop \__prg_break: } { }
      \__prg_break_point:
      \exp_last_unbraced:Nx \__str_decode_eight_bit_load_missing:n
        { \tl_use:c { c__str_encoding_#1_missing_tl } }
        { \q_stop \__prg_break: }
      \__prg_break_point:
      \flag_clear:n { str_error }
      \__str_convert_gmap:N \__str_decode_eight_bit_char:N
      \__str_if_flag_error:nnx { str_error } { decode-8-bit } {#1}
    \group_end:
  }
\cs_new_protected:Npn \__str_decode_eight_bit_load:nn #1#2
  {
    \use_none_delimit_by_q_stop:w #1 \q_stop
    \tex_dimen:D "#1 = \l__str_internal_int sp \scan_stop:
    \tex_skip:D \l__str_internal_int = "#1 sp \scan_stop:
    \tex_toks:D \l__str_internal_int \exp_after:wN { \__int_value:w "#2 }
    \tex_advance:D \l__str_internal_int \c_one
    \__str_decode_eight_bit_load:nn
  }
\cs_new_protected:Npn \__str_decode_eight_bit_load_missing:n #1
  {
    \use_none_delimit_by_q_stop:w #1 \q_stop
    \tex_dimen:D "#1 = \l__str_internal_int sp \scan_stop:
    \tex_skip:D \l__str_internal_int = "#1 sp \scan_stop:
    \tex_toks:D \l__str_internal_int \exp_after:wN
      { \int_use:N \c__str_replacement_char_int }
    \tex_advance:D \l__str_internal_int \c_one
    \__str_decode_eight_bit_load_missing:n
  }
\cs_new:Npn \__str_decode_eight_bit_char:N #1
  {
    #1 \s__tl
    \if_int_compare:w \tex_dimen:D `#1 < \l__str_internal_int
      \if_int_compare:w \tex_skip:D \tex_dimen:D `#1 = `#1 \exp_stop_f:
        \tex_the:D \tex_toks:D \tex_dimen:D
      \fi:
    \fi:
```

```
    \__int_value:w `#1 \s__tl
  }
```
(*End definition for* `\__str_convert_decode_eight_bit:n`*. This function is documented on page* **??**.)

__str_convert_encode_eight_bit:n
__str_encode_eight_bit_load:nn
__str_encode_eight_bit_char:n
__str_encode_eight_bit_char_aux:n

```
\cs_new_protected:Npn \__str_convert_encode_eight_bit:n #1
  {
    \group_begin:
      \int_zero:N \l__str_internal_int
      \exp_last_unbraced:Nx \__str_encode_eight_bit_load:nn
        { \tl_use:c { c__str_encoding_#1_tl } }
        { \q_stop \__prg_break: } { }
      \__prg_break_point:
      \flag_clear:n { str_error }
      \__str_convert_gmap_internal:N \__str_encode_eight_bit_char:n
      \__str_if_flag_error:nnx { str_error } { encode-8-bit } {#1}
    \group_end:
  }
\cs_new_protected:Npn \__str_encode_eight_bit_load:nn #1#2
  {
    \use_none_delimit_by_q_stop:w #1 \q_stop
    \tex_dimen:D "#2 = \l__str_internal_int sp \scan_stop:
    \tex_skip:D \l__str_internal_int = "#2 sp \scan_stop:
    \exp_args:NNf \tex_toks:D \l__str_internal_int
      { \__str_output_byte:n { "#1 } }
    \tex_advance:D \l__str_internal_int \c_one
    \__str_encode_eight_bit_load:nn
  }
\cs_new:Npn \__str_encode_eight_bit_char:n #1
  {
    \if_int_compare:w #1 > \c_max_register_int
      \flag_raise:n { str_error }
    \else:
      \if_int_compare:w \tex_dimen:D #1 < \l__str_internal_int
        \if_int_compare:w \tex_skip:D \tex_dimen:D #1 = #1 \exp_stop_f:
          \tex_the:D \tex_toks:D \tex_dimen:D #1 \exp_stop_f:
          \exp_after:wN \exp_after:wN \exp_after:wN \use_none:nn
        \fi:
      \fi:
      \__str_encode_eight_bit_char_aux:n {#1}
    \fi:
  }
\cs_new:Npn \__str_encode_eight_bit_char_aux:n #1
  {
    \if_int_compare:w #1 < \c_two_hundred_fifty_six
      \__str_output_byte:n {#1}
    \else:
      \flag_raise:n { str_error }
    \fi:
  }
```

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

5.4 Messages

General messages, and messages for the encodings and escapings loaded by default ("native", and "bytes").

```
\__msg_kernel_new:nnn { str } { unknown-esc }
  { Escaping~scheme~'#1'~(filtered:~'#2')~unknown. }
\__msg_kernel_new:nnn { str } { unknown-enc }
  { Encoding~scheme~'#1'~(filtered:~'#2')~unknown. }
\__msg_kernel_new:nnnn { str } { native-escaping }
  { The~'native'~encoding~scheme~does~not~support~any~escaping. }
  {
    Since~native~strings~do~not~consist~in~bytes,~
    none~of~the~escaping~methods~make~sense.~
    The~specified~escaping,~'#1',~will be ignored.
  }
\__msg_kernel_new:nnn { str } { file-not-found }
  { File~'l3str-#1.def'~not~found. }
```

Message used when the "bytes" unescaping fails because the string given to `\str_set_convert:Nnnn` contains a non-byte. This cannot happen for the pdfTEX engine, since that engine only supports 8-bit characters. Messages used for other escapings and encodings are defined in each definition file.

```
\pdftex_if_engine:F
  {
    \__msg_kernel_new:nnnn { str } { non-byte }
      { String~invalid~in~escaping~'#1':~it~may~only~contain~bytes. }
      {
        Some~characters~in~the~string~you~asked~to~convert~are~not~
        8-bit~characters.~Perhaps~the~string~is~a~'native'~Unicode~string?~
        If~it~is,~try~using\\
        \\
        \iow_indent:n
          {
            \iow_char:N\\str_set_convert:Nnnn \\
            \ \ <str~var>~\{~<string>~\}~\{~native~\}~\{~<target~encoding>~\}
          }
      }
  }
```

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

```
\__msg_kernel_new:nnnn { str } { decode-8-bit }
  { Invalid~string~in~encoding~'#1'. }
  {
    LaTeX~came~across~a~byte~which~is~not~defined~to~represent~
    any~character~in~the~encoding~'#1'.
  }
\__msg_kernel_new:nnnn { str } { encode-8-bit }
  { Unicode~string~cannot~be~converted~to~encoding~'#1'. }
  {
    The~encoding~'#1'~only~contains~a~subset~of~all~Unicode~characters.~
    LaTeX~was~asked~to~convert~a~string~to~that~encoding,~but~that~
    string~contains~a~character~that~'#1'~does~not~support.
  }
⟨/initex | package⟩
```

5.5 Escaping definition files

Several of those encodings are defined by the pdf file format. The following byte storage methods are defined:

- `bytes` (default), non-bytes are filtered out, and bytes are left untouched (this is defined by default);
- `hex` or `hexadecimal`, as per the pdfTEX primitive `\pdfescapehex`
- `name`, as per the pdfTEX primitive `\pdfescapename`
- `string`, as per the pdfTEX primitive `\pdfescapestring`
- `url`, as per the percent encoding of urls.

5.5.1 Unescape methods

`\__str_convert_unescape_hex:`
`\__str_unescape_hex_auxi:N`
`\__str_unescape_hex_auxii:N`

Take chars two by two, and interpret each pair as the hexadecimal code for a byte. Anything else than hexadecimal digits is ignored, raising the flag. A string which contains an odd number of hexadecimal digits gets `0` appended to it: this is equivalent to appending a `0` in all cases, and dropping it if it is alone.

```
⟨*hex⟩
\cs_new_protected_nopar:Npn \__str_convert_unescape_hex:
  {
    \group_begin:
      \flag_clear:n { str_error }
      \int_set:Nn \tex_escapechar:D { 92 }
      \tl_gset:Nx \g__str_result_tl
        {
          \__str_output_byte:w "
            \exp_last_unbraced:Nf \__str_unescape_hex_auxi:N
              { \tl_to_str:N \g__str_result_tl }
            0 { ? 0 - \c_one \__prg_break: }
            \__prg_break_point:
          \__str_output_end:
        }
      \__str_if_flag_error:nnx { str_error } { unescape-hex } { }
    \group_end:
  }
\cs_new:Npn \__str_unescape_hex_auxi:N #1
  {
    \use_none:n #1
    \__str_hexadecimal_use:NTF #1
      { \__str_unescape_hex_auxii:N }
      {
        \flag_raise:n { str_error }
        \__str_unescape_hex_auxi:N
      }
  }
\cs_new:Npn \__str_unescape_hex_auxii:N #1
  {
    \use_none:n #1
    \__str_hexadecimal_use:NTF #1
      {
        \__str_output_end:
        \__str_output_byte:w " \__str_unescape_hex_auxi:N
      }
      {
        \flag_raise:n { str_error }
        \__str_unescape_hex_auxii:N
      }
  }
\__msg_kernel_new:nnnn { str } { unescape-hex }
  { String~invalid~in~escaping~'hex':~only~hexadecimal~digits~allowed. }
  {
    Some~characters~in~the~string~you~asked~to~convert~are~not~
    hexadecimal~digits~(0-9,~A-F,~a-f)~nor~spaces.
  }
⟨/hex⟩
```

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

`\__str_convert_unescape_name:`
`\__str_unescape_name_loop:wNN`
`\__str_convert_unescape_url:`
`\__str_unescape_url_loop:wNN`

The `\__str_convert_unescape_name:` function replaces each occurrence of `#` followed by two hexadecimal digits in `\g__str_result_tl` by the corresponding byte. The `url` function is identical, with escape character `%` instead of `#`. Thus we define the two together. The arguments of `\__str_tmp:w` are the character code of `#` or `%` in hexadecimal, the name of the main function to define, and the name of the auxiliary which performs the loop.

The looping auxiliary `#3` finds the next escape character, reads the following two characters, and tests them. The test `\__str_hexadecimal_use:NTF` leaves the uppercase digit in the input stream, hence we surround the test with `\__str_output_byte:w "` and `\__str_output_end:`. If both characters are hexadecimal digits, they should be removed before looping: this is done by `\use_i:nnn`. If one of the characters is not a hexadecimal digit, then feed `"#1` to `\__str_output_byte:w` to produce the escape character, raise the flag, and call the looping function followed by the two characters (remove `\use_i:nnn`).

```
⟨*name | url⟩
\cs_set_protected:Npn \__str_tmp:w #1#2#3
  {
    \cs_new_protected:cpn { __str_convert_unescape_#2: }
```

```
      {
        \group_begin:
          \flag_clear:n { str_byte }
          \flag_clear:n { str_error }
          \int_set:Nn \tex_escapechar:D { 92 }
          \tl_gset:Nx \g__str_result_tl
            {
              \exp_after:wN #3 \g__str_result_tl
                #1 ? { ? \__prg_break: }
              \__prg_break_point:
            }
          \__str_if_flag_error:nnx { str_byte } { non-byte } { #2 }
          \__str_if_flag_error:nnx { str_error } { unescape-#2 } { }
        \group_end:
      }
    \cs_new:Npn #3 ##1#1##2##3
      {
        \__str_filter_bytes:n {##1}
        \use_none:n ##3
        \__str_output_byte:w "
          \__str_hexadecimal_use:NTF ##2
            {
              \__str_hexadecimal_use:NTF ##3
                { }
                {
                  \flag_raise:n { str_error }
                  * \c_zero + `#1 \use_i:nn
                }
            }
            {
              \flag_raise:n { str_error }
              0 + `#1 \use_i:nn
            }
        \__str_output_end:
        \use_i:nnn #3 ##2##3
      }
    \__msg_kernel_new:nnnn { str } { unescape-#2 }
      { String~invalid~in~escaping~'#2'. }
      {
        LaTeX~came~across~the~escape~character~'#1'~not~followed~by~
        two~hexadecimal~digits.~This~is~invalid~in~the~escaping~'#2'.
      }
  }
⟨/name | url⟩
⟨*name⟩
\exp_after:wN \__str_tmp:w \c_hash_str { name }
  \__str_unescape_name_loop:wNN
⟨/name⟩
⟨*url⟩
\exp_after:wN \__str_tmp:w \c_percent_str { url }
```

```
  \__str_unescape_url_loop:wNN
⟨/url⟩
```

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

`\__str_convert_unescape_string:`
`\__str_unescape_string_newlines:wN`
`\__str_unescape_string_loop:wNNN`
`\__str_unescape_string_repeat:NNNNNN`

The `string` escaping is somewhat similar to the `name` and `url` escapings, with escape character `\`. The first step is to convert all three line endings, `^^J`, `^^M`, and `^^M^^J` to the common `^^J`, as per the PDF specification. This step cannot raise the flag.

Then the following escape sequences are decoded.

`\n`	Line feed (10)
`\r`	Carriage return (13)
`\t`	Horizontal tab (9)
`\b`	Backspace (8)
`\f`	Form feed (12)
`\(`	Left parenthesis
`\)`	Right parenthesis
`\\`	Backslash
`\ddd`	(backslash followed by 1 to 3 octal digits) Byte `ddd` (octal), subtracting 256 in case of overflow.

If followed by an end-of-line character, the backslash and the end-of-line are ignored. If followed by anything else, the backslash is ignored, raising the error flag.

```
⟨*string⟩
\group_begin:
  \char_set_lccode:nn {‘\*} {‘\\}
  \char_set_catcode_other:N \^^J
  \char_set_catcode_other:N \^^M
  \tl_to_lowercase:n
    {
      \cs_new_protected_nopar:Npn \__str_convert_unescape_string:
        {
          \group_begin:
            \flag_clear:n { str_byte }
            \flag_clear:n { str_error }
            \int_set:Nn \tex_escapechar:D { 92 }
            \tl_gset:Nx \g__str_result_tl
              {
                \exp_after:wN \__str_unescape_string_newlines:wN
                  \g__str_result_tl \__prg_break: ^^M ?
                \__prg_break_point:
              }
            \tl_gset:Nx \g__str_result_tl
              {
```

```
              \exp_after:wN \__str_unescape_string_loop:wNNN
                \g__str_result_tl * ?? { ? \__prg_break: }
              \__prg_break_point:
            }
          \__str_if_flag_error:nnx { str_byte } { non-byte } { string }
          \__str_if_flag_error:nnx { str_error } { unescape-string } { }
        \group_end:
      }
    \cs_new:Npn \__str_unescape_string_loop:wNNN #1 *#2#3#4
  }
      {
        \__str_filter_bytes:n {#1}
        \use_none:n #4
        \__str_output_byte:w '
          \__str_octal_use:NTF #2
            {
              \__str_octal_use:NTF #3
                {
                  \__str_octal_use:NTF #4
                    {
                      \if_int_compare:w #2 > \c_three
                        - 256
                      \fi:
                      \__str_unescape_string_repeat:NNNNNN
                    }
                    { \__str_unescape_string_repeat:NNNNNN ? }
                }
                { \__str_unescape_string_repeat:NNNNNN ?? }
            }
            {
              \str_case_x:nnn {#2}
                {
                  { \c_backslash_str } { 134 }
                  { ( } { 50 }
                  { ) } { 51 }
                  { r } { 15 }
                  { f } { 14 }
                  { n } { 12 }
                  { t } { 11 }
                  { b } { 10 }
                  { ^^J } { 0 - \c_one }
                }
                {
                  \flag_raise:n { str_error }
                  0 - \c_one \use_i:nn
                }
            }
        \__str_output_end:
        \use_i:nn \__str_unescape_string_loop:wNNN #2#3#4
      }
  \cs_new:Npn \__str_unescape_string_repeat:NNNNNN #1#2#3#4#5#6
    { \__str_output_end: \__str_unescape_string_loop:wNNN }
  \cs_new:Npn \__str_unescape_string_newlines:wN #1 ^^M #2
    {
      #1
      \if_charcode:w ^^J #2 \else: ^^J \fi:
      \__str_unescape_string_newlines:wN #2
    }
  \__msg_kernel_new:nnnn { str } { unescape-string }
    { String~invalid~in~escaping~'string'. }
    {
      LaTeX~came~across~an~escape~character~'\c_backslash_str'~
      not~followed~by~any~of:~'n',~'r',~'t',~'b',~'f',~'(',~')',~
      '\c_backslash_str',~one~to~three~octal~digits,~or~the~end~
      of~a~line.
    }
\group_end:
⟨/string⟩
```

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

5.5.2 Escape methods

Currently, none of the escape methods can lead to errors, assuming that their input is made out of bytes.

`\__str_convert_escape_hex:`
`\__str_escape_hex_char:N`

Loop and convert each byte to hexadecimal.

```
⟨*hex⟩
\cs_new_protected_nopar:Npn \__str_convert_escape_hex:
  { \__str_convert_gmap:N \__str_escape_hex_char:N }
\cs_new:Npn \__str_escape_hex_char:N #1
  { \__str_output_hexadecimal:n { `#1 } }
⟨/hex⟩
```

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

`\__str_convert_escape_name:`
`\__str_escape_name_char:N`
`\__str_if_escape_name:NTF`
`\c__str_escape_name_str`
`\c__str_escape_name_not_str`

For each byte, test whether it should be output as is, or be "hash-encoded". Roughly, bytes outside the range ["2A, "7E] are hash-encoded. We keep two lists of exceptions: characters in `\c__str_escape_name_not_str` are not hash-encoded, and characters in the `\c__str_escape_name_str` are encoded.

```
⟨*name⟩
\str_const:Nn \c__str_escape_name_not_str { ! " $ & ' } %$
\str_const:Nn \c__str_escape_name_str { {}/<>[] }
\cs_new_protected_nopar:Npn \__str_convert_escape_name:
  { \__str_convert_gmap:N \__str_escape_name_char:N }
\cs_new:Npn \__str_escape_name_char:N #1
  {
    \__str_if_escape_name:NTF #1 {#1}
      { \c_hash_str \__str_output_hexadecimal:n {`#1} }
  }
\prg_new_conditional:Npnn \__str_if_escape_name:N #1 { TF }
```

```
  {
    \if_int_compare:w `#1 < "2A \exp_stop_f:
      \__str_if_contains_char:NNTF \c__str_escape_name_not_str #1
        \prg_return_true: \prg_return_false:
    \else:
      \if_int_compare:w `#1 > "7E \exp_stop_f:
        \prg_return_false:
      \else:
        \__str_if_contains_char:NNTF \c__str_escape_name_str #1
          \prg_return_false: \prg_return_true:
      \fi:
    \fi:
  }
⟨/name⟩
```

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

__str_convert_escape_string:
__str_escape_string_char:N
__str_if_escape_string:NTF
\c__str_escape_string_str

Any character below (and including) space, and any character above (and including) `del`, are converted to octal. One backslash is added before each parenthesis and backslash.

```
⟨*string⟩
\str_const:Nx \c__str_escape_string_str
  { \c_backslash_str ( ) }
\cs_new_protected_nopar:Npn \__str_convert_escape_string:
  { \__str_convert_gmap:N \__str_escape_string_char:N }
\cs_new:Npn \__str_escape_string_char:N #1
  {
    \__str_if_escape_string:NTF #1
      {
        \__str_if_contains_char:NNT
          \c__str_escape_string_str #1
          { \c_backslash_str }
        #1
      }
      {
        \c_backslash_str
        \int_div_truncate:nn {`#1} {64}
        \int_mod:nn { \int_div_truncate:nn {`#1} \c_eight } \c_eight
        \int_mod:nn {`#1} \c_eight
      }
  }
\prg_new_conditional:Npnn \__str_if_escape_string:N #1 { TF }
  {
    \if_int_compare:w `#1 < "21 \exp_stop_f:
      \prg_return_false:
    \else:
      \if_int_compare:w `#1 > "7E \exp_stop_f:
        \prg_return_false:
      \else:
        \prg_return_true:
      \fi:
```

```
    \fi:
  }
⟨/string⟩
```

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

`\__str_convert_escape_url:`
`\__str_escape_url_char:N`
`\__str_if_escape_url:NTF`

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

```
⟨*url⟩
\cs_new_protected_nopar:Npn \__str_convert_escape_url:
  { \__str_convert_gmap:N \__str_escape_url_char:N }
\cs_new:Npn \__str_escape_url_char:N #1
  {
    \__str_if_escape_url:NTF #1 {#1}
      { \c_percent_str \__str_output_hexadecimal:n { `#1 } }
  }
\prg_new_conditional:Npnn \__str_if_escape_url:N #1 { TF }
  {
    \if_int_compare:w `#1 < "41 \exp_stop_f:
      \__str_if_contains_char:nNTF { "-.<> } #1
        \prg_return_true: \prg_return_false:
    \else:
      \if_int_compare:w `#1 > "7E \exp_stop_f:
        \prg_return_false:
      \else:
        \__str_if_contains_char:nNTF { [ ] } #1
          \prg_return_false: \prg_return_true:
      \fi:
    \fi:
  }
⟨/url⟩
```

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

5.6 Encoding definition files

The `native` encoding is automatically defined. Other encodings are loaded as needed. The following encodings are supported:

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

5.6.1 utf-8 support

```
⟨*utf8⟩
```

__str_convert_encode_utf8:
__str_encode_utf_viii_char:n
__str_encode_utf_viii_loop:wwnnw

Loop through the internal string, and convert each character to its UTF-8 representation. The representation is built from the right-most (least significant) byte to the left-most (most significant) byte. Continuation bytes are in the range $[128, 191]$, taking 64 different values, hence we roughly want to express the character code in base 64, shifting the first digit in the representation by some number depending on how many continuation bytes there are. In the range $[0, 127]$, output the corresponding byte directly. In the range $[128, 2047]$, output the remainder modulo 64, plus 128 as a continuation byte, then output the quotient (which is in the range $[0, 31]$), shifted by 192. In the next range, $[2048, 65535]$, split the character code into residue and quotient modulo 64, output the residue as a first continuation byte, then repeat; this leaves us with a quotient in the range $[0, 15]$, which we output shifted by 224. The last range, $[65536, 1114111]$, follows the same pattern: once we realize that dividing twice by 64 leaves us with a number larger than 15, we repeat, producing a last continuation byte, and offset the quotient by 240 for the leading byte.

How is that implemented? `\__str_encode_utf_vii_loop:wwnnw` takes successive quotients as its first argument, the quotient from the previous step as its second argument (except in step 1), the bound for quotients that trigger one more step or not, and finally the offset used if this step should produce the leading byte. Leading bytes can be in the ranges $[0, 127]$, $[192, 223]$, $[224, 239]$, and $[240, 247]$ (really, that last limit should be 244 because Unicode stops at the code point 1114111). At each step, if the quotient `#1` is less than the limit `#3` for that range, output the leading byte (`#1` shifted by `#4`) and stop. Otherwise, we need one more step: use the quotient of `#1` by 64, and `#1` as arguments for the looping auxiliary, and output the continuation byte corresponding to the remainder $\#2 - 64\#1 + 128$. The bizarre construction `\c_minus_one + \c_zero *` removes the spurious initial continuation byte (better methods welcome).

```
\cs_new_protected_nopar:cpn { __str_convert_encode_utf8: }
  { \__str_convert_gmap_internal:N \__str_encode_utf_viii_char:n }
\cs_new:Npn \__str_encode_utf_viii_char:n #1
  {
    \__str_encode_utf_viii_loop:wwnnw #1 ; \c_minus_one + \c_zero * ;
      { 128 } { \c_zero }
      {  32 } {     192 }
      {  16 } {     224 }
      {   8 } {     240 }
    \q_stop
  }
\cs_new:Npn \__str_encode_utf_viii_loop:wwnnw #1; #2; #3#4 #5 \q_stop
  {
    \if_int_compare:w #1 < #3 \exp_stop_f:
      \__str_output_byte:n { #1 + #4 }
      \exp_after:wN \use_none_delimit_by_q_stop:w
    \fi:
    \exp_after:wN \__str_encode_utf_viii_loop:wwnnw
      \__int_value:w \int_div_truncate:nn {#1} {64} ; #1 ;
      #5 \q_stop
    \__str_output_byte:n { #2 - 64 * ( #1 - \c_two ) }
  }
```
(*End definition for* `\__str_convert_encode_utf8:`. *This function is documented on page* ??.)

`\l__str_missing_flag`
`\l__str_extra_flag`
`\l__str_overlong_flag`
`\l__str_overflow_flag`

When decoding a string that is purportedly in the UTF-8 encoding, four different errors can occur, signalled by a specific flag for each (we define those flags using `\flag_clear_new:n` rather than `\flag_new:n`, because they are shared with other encoding definition files).

- "Missing continuation byte": a leading byte is not followed by the right number of continuation bytes.
- "Extra continuation byte": a continuation byte appears where it was not expected, *i.e.*, not after an appropriate leading byte.
- "Overlong": a Unicode character is expressed using more bytes than necessary, for instance, `"C0"80` for the code point 0, instead of a single null byte.
- "Overflow": this occurs when decoding produces Unicode code points greater than 1114111.

We only raise one LaTeX3 error message, combining all the errors which occurred. In the short message, the leading comma must be removed to get a grammatically correct sentence. In the long text, first remind the user what a correct UTF-8 string should look like, then add error-specific information.

```
\flag_clear_new:n { str_missing }
\flag_clear_new:n { str_extra }
\flag_clear_new:n { str_overlong }
\flag_clear_new:n { str_overflow }
\__msg_kernel_new:nnnn { str } { utf8-decode }
  {
    Invalid~UTF-8~string: \exp_last_unbraced:Nf \use_none:n
    \__str_if_flag_times:nT { str_missing }  { ,~missing~continuation~byte }
    \__str_if_flag_times:nT { str_extra }    { ,~extra~continuation~byte }
    \__str_if_flag_times:nT { str_overlong } { ,~overlong~form }
    \__str_if_flag_times:nT { str_overflow } { ,~code~point~too~large }
    .
  }
  {
    In~the~UTF-8~encoding,~each~Unicode~character~consists~in~
    1~to~4~bytes,~with~the~following~bit~pattern: \\
    \iow_indent:n
      {
        Code~point~\ \ \ \ <~128:~0xxxxxxx \\
        Code~point~\ \ \  <~2048:~110xxxxx~10xxxxxx \\
        Code~point~\ \   <~65536:~1110xxxx~10xxxxxx~10xxxxxx \\
        Code~point~    <~1114112:~11110xxx~10xxxxxx~10xxxxxx~10xxxxxx \\
      }
    Bytes~of~the~form~10xxxxxx~are~called~continuation~bytes.
    \flag_if_raised:nT { str_missing }
      {
        \\\\
        A~leading~byte~(in~the~range~[192,255])~was~not~followed~by~
        the~appropriate~number~of~continuation~bytes.
      }
    \flag_if_raised:nT { str_extra }
      {
        \\\\
        LaTeX~came~across~a~continuation~byte~when~it~was~not~expected.
      }
    \flag_if_raised:nT { str_overlong }
      {
        \\\\
        Every~Unicode~code~point~must~be~expressed~in~the~shortest~
        possible~form.~For~instance,~'0xC0'~'0x83'~is~not~a~valid~
        representation~for~the~code~point~3.
      }
    \flag_if_raised:nT { str_overflow }
      {
        \\\\
        Unicode~limits~code~points~to~the~range~[0,1114111].
      }
  }
```

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

`\__str_convert_decode_utf8:`
`\__str_decode_utf_viii_start:N`
`\__str_decode_utf_viii_continuation:wwN`
`\__str_decode_utf_viii_aux:wNnnwN`
`\__str_decode_utf_viii_overflow:w`
`\__str_decode_utf_viii_end:`

Decoding is significantly harder than encoding. As before, lower some flags, which are tested at the end (in bulk, to trigger at most one LaTeX3 error, as explained above). We expect successive multi-byte sequences of the form ⟨*start byte*⟩ ⟨*continuation bytes*⟩. The `_start` auxiliary tests the first byte:

- [0, "7F]: the byte stands alone, and is converted to its own character code;
- ["80, "BF]: unexpected continuation byte, raise the appropriate flag, and convert that byte to the replacement character "FFFD;
- ["C0, "FF]: this byte should be followed by some continuation byte(s).

In the first two cases, `\use_none_delimit_by_q_stop:w` removes data that only the third case requires, namely the limits of ranges of Unicode characters which can be expressed with 1, 2, 3, or 4 bytes.

We can now concentrate on the multi-byte case and the `_continuation` auxiliary. We expect `#3` to be in the range ["80, "BF]. The test for this goes as follows: if the character code is less than "80, we compare it to −"C0, yielding `false`; otherwise to "C0, yielding `true` in the range ["80, "BF] and `false` otherwise. If we find that the byte is not a continuation range, stop the current slew of bytes, output the replacement character, and continue parsing with the `_start` auxiliary, starting at the byte we just tested. Once we know that the byte is a continuation byte, leave it behind us in the input stream, compute what code point the bytes read so far would produce, and feed that number to the `_aux` function.

The `_aux` function tests whether we should look for more continuation bytes or not. If the number it receives as `#1` is less than the maximum `#4` for the current range, then we are done: check for an overlong representation by comparing `#1` with the maximum `#3` for the previous range. Otherwise, we call the `_continuation` auxiliary again, after shifting the "current code point" by `#4` (maximum from the range we just checkedd).

Two additional tests are needed: if we reach the end of the list of range maxima and we are still not done, then we are faced with an overflow. Clean up, and again insert the code point "FFFD for the replacement character. Also, every time we read a byte, we need to check whether we reached the end of the string. In a correct UTF-8 string, this happens automatically when the `_start` auxiliary leaves its first argument in the input stream: the end-marker begins with `\__prg_break:`, which ends the loop. On the other hand, if the end is reached when looking for a continuation byte, the `\use_none:n #3` construction removes the first token from the end-marker, and leaves the `_end` auxiliary, which raises the appropriate error flag before ending the mapping.

```
\cs_new_protected_nopar:cpn { __str_convert_decode_utf8: }
  {
    \flag_clear:n { str_error }
    \flag_clear:n { str_missing }
    \flag_clear:n { str_extra }
    \flag_clear:n { str_overlong }
    \flag_clear:n { str_overflow }
    \tl_gset:Nx \g__str_result_tl
      {
        \exp_after:wN \__str_decode_utf_viii_start:N \g__str_result_tl
          { \__prg_break: \__str_decode_utf_viii_end: }
        \__prg_break_point:
      }
    \__str_if_flag_error:nnx { str_error } { utf8-decode } { }
  }
\cs_new:Npn \__str_decode_utf_viii_start:N #1
  {
    #1
    \if_int_compare:w `#1 < "C0 \exp_stop_f:
      \s__tl
      \if_int_compare:w `#1 < "80 \exp_stop_f:
        \__int_value:w `#1
      \else:
        \flag_raise:n { str_extra }
        \flag_raise:n { str_error }
        \int_use:N \c__str_replacement_char_int
      \fi:
    \else:
      \exp_after:wN \__str_decode_utf_viii_continuation:wwN
      \int_use:N \__int_eval:w `#1 - "C0 \exp_after:wN \__int_eval_end:
    \fi:
    \s__tl
    \use_none_delimit_by_q_stop:w {"80} {"800} {"10000} {"110000} \q_stop
    \__str_decode_utf_viii_start:N
  }
\cs_new:Npn \__str_decode_utf_viii_continuation:wwN
    #1 \s__tl #2 \__str_decode_utf_viii_start:N #3
  {
    \use_none:n #3
    \if_int_compare:w ‘#3 <
          \if_int_compare:w ‘#3 < "80 \exp_stop_f: - \fi:
          "C0 \exp_stop_f:
      #3
      \exp_after:wN \__str_decode_utf_viii_aux:wNnnwN
      \int_use:N \__int_eval:w
        #1 * "40 + ‘#3 - "80
      \exp_after:wN \__int_eval_end:
    \else:
      \s__tl
      \flag_raise:n { str_missing }
      \flag_raise:n { str_error }
      \int_use:N \c__str_replacement_char_int
    \fi:
    \s__tl
    #2
    \__str_decode_utf_viii_start:N #3
  }
\cs_new:Npn \__str_decode_utf_viii_aux:wNnnwN
    #1 \s__tl #2#3#4 #5 \__str_decode_utf_viii_start:N #6
  {
    \if_int_compare:w #1 < #4 \exp_stop_f:
      \s__tl
      \if_int_compare:w #1 < #3 \exp_stop_f:
        \flag_raise:n { str_overlong }
        \flag_raise:n { str_error }
        \int_use:N \c__str_replacement_char_int
      \else:
        #1
      \fi:
    \else:
      \if_meaning:w \q_stop #5
        \__str_decode_utf_viii_overflow:w #1
      \fi:
      \exp_after:wN \__str_decode_utf_viii_continuation:wwN
      \int_use:N \__int_eval:w #1 - #4 \exp_after:wN \__int_eval_end:
    \fi:
    \s__tl
    #2 {#4} #5
    \__str_decode_utf_viii_start:N
  }
\cs_new:Npn \__str_decode_utf_viii_overflow:w #1 \fi: #2 \fi:
  {
    \fi: \fi:
    \flag_raise:n { str_overflow }
```

```
      \flag_raise:n { str_error }
      \int_use:N \c__str_replacement_char_int
    }
\cs_new_nopar:Npn \__str_decode_utf_viii_end:
  {
    \s__tl
    \flag_raise:n { str_missing }
    \flag_raise:n { str_error }
    \int_use:N \c__str_replacement_char_int \s__tl
    \__prg_break:
  }
```

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

```
⟨/utf8⟩
```

5.6.2 utf-16 support

The definitions are done in a category code regime where the bytes 254 and 255 used by the byte order mark have catcode 12.

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

`\__str_convert_encode_utf16:`
`\__str_convert_encode_utf16be:`
`\__str_convert_encode_utf16le:`
`\__str_encode_utf_xvi_aux:N`
`\__str_encode_utf_xvi_char:n`

When the endianness is not specified, it is big-endian by default, and we add a byte-order mark. Convert characters one by one in a loop, with different behaviours depending on the character code.

- [0, "D7FF]: converted to two bytes;
- ["D800, "DFFF] are used as surrogates: they cannot be converted and are replaced by the replacement character;
- ["E000, "FFFF]: converted to two bytes;
- ["10000, "10FFFF]: converted to a pair of surrogates, each two bytes. The magic "D7C0 is "D800 − "10000/"400.

For the duration of this operation, `\__str_tmp:w` is defined as a function to convert a number in the range [0, "FFFF] to a pair of bytes (either big endian or little endian), by feeding the quotient of the division of `#1` by "100, followed by `#1` to `\__str_encode_utf_xvi_be:nn` or its `le` analog: those compute the remainder, and output two bytes for the quotient and remainder.

```
  \cs_new_protected_nopar:cpn { __str_convert_encode_utf16: }
    {
      \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n
      \tl_gput_left:Nx \g__str_result_tl { ^^fe ^^ff }
    }
  \cs_new_protected_nopar:cpn { __str_convert_encode_utf16be: }
    { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_be:n }
```

```
\cs_new_protected_nopar:cpn { __str_convert_encode_utf16le: }
  { \__str_encode_utf_xvi_aux:N \__str_output_byte_pair_le:n }
\cs_new_protected:Npn \__str_encode_utf_xvi_aux:N #1
  {
    \flag_clear:n { str_error }
    \cs_set_eq:NN \__str_tmp:w #1
    \__str_convert_gmap_internal:N \__str_encode_utf_xvi_char:n
    \__str_if_flag_error:nnx { str_error } { utf16-encode } { }
  }
\cs_new:Npn \__str_encode_utf_xvi_char:n #1
  {
    \if_int_compare:w #1 < "D800 \exp_stop_f:
      \__str_tmp:w {#1}
    \else:
      \if_int_compare:w #1 < "10000 \exp_stop_f:
        \if_int_compare:w #1 < "E000 \exp_stop_f:
          \flag_raise:n { str_error }
          \__str_tmp:w { \c__str_replacement_char_int }
        \else:
          \__str_tmp:w {#1}
        \fi:
      \else:
        \exp_args:Nf \__str_tmp:w { \int_div_truncate:nn {#1} {"400} + "D7C0 }
        \exp_args:Nf \__str_tmp:w { \int_mod:nn {#1} {"400} + "DC00 }
      \fi:
    \fi:
  }
```

(*End definition for* `\__str_convert_encode_utf16:` *,* `\__str_convert_encode_utf16be:` *, and* `\__str_convert_encode_utf16l`
These functions are documented on page **??***.*)

`\l__str_missing_flag`
`\l__str_extra_flag`
`\l__str_end_flag`

When encoding a Unicode string to UTF-16, only one error can occur: code points in the range ["D800, "DFFF], corresponding to surrogates, cannot be encoded. We use the all-purpose flag `@@_error` to signal that error.

When decoding a Unicode string which is purportedly in UTF-16, three errors can occur: a missing trail surrogate, an unexpected trail surrogate, and a string containing an odd number of bytes.

```
\flag_clear_new:n { str_missing }
\flag_clear_new:n { str_extra }
\flag_clear_new:n { str_end }
\__msg_kernel_new:nnnn { str } { utf16-encode }
  { Unicode~string~cannot~be~expressed~in~UTF-16:~surrogate. }
  {
    Surrogate~code~points~(in~the~range~[U+D800,~U+DFFF])~
    can~be~expressed~in~the~UTF-8~and~UTF-32~encodings,~
    but~not~in~the~UTF-16~encoding.
  }
\__msg_kernel_new:nnnn { str } { utf16-decode }
  {
    Invalid~UTF-16~string: \exp_last_unbraced:Nf \use_none:n
```

```
      \__str_if_flag_times:nT { str_missing }  { ,~missing~trail~surrogate }
      \__str_if_flag_times:nT { str_extra }    { ,~extra~trail~surrogate }
      \__str_if_flag_times:nT { str_end }      { ,~odd~number~of~bytes }
      .
    }
    {
      In~the~UTF-16~encoding,~each~Unicode~character~is~encoded~as~
      2~or~4~bytes: \\
      \iow_indent:n
        {
          Code~point~in~[U+0000,~U+D7FF]:~two~bytes \\
          Code~point~in~[U+D800,~U+DFFF]:~illegal \\
          Code~point~in~[U+E000,~U+FFFF]:~two~bytes \\
          Code~point~in~[U+10000,~U+10FFFF]:~
            a~lead~surrogate~and~a~trail~surrogate \\
        }
      Lead~surrogates~are~pairs~of~bytes~in~the~range~[0xD800,~0xDBFF],~
      and~trail~surrogates~are~in~the~range~[0xDC00,~0xDFFF].
      \flag_if_raised:nT { str_missing }
        {
          \\\\
          A~lead~surrogate~was~not~followed~by~a~trail~surrogate.
        }
      \flag_if_raised:nT { str_extra }
        {
          \\\\
          LaTeX~came~across~a~trail~surrogate~when~it~was~not~expected.
        }
      \flag_if_raised:nT { str_end }
        {
          \\\\
          The~string~contained~an~odd~number~of~bytes.~This~is~invalid:~
          the~basic~code~unit~for~UTF-16~is~16~bits~(2~bytes).
        }
    }
```

(*End definition for* `\l__str_missing_flag`*,* `\l__str_extra_flag`*, and* `\l__str_end_flag`*. These variables are documented on page* **??**.)

`\__str_convert_decode_utf16:`
`\__str_convert_decode_utf16be:`
`\__str_convert_decode_utf16le:`
`\__str_decode_utf_xvi_bom:NN`
`\__str_decode_utf_xvi:Nw`

As for UTF-8, decoding UTF-16 is harder than encoding it. If the endianness is unknown, check the first two bytes: if those are "FE and "FF in either order, remove them and use the corresponding endianness, otherwise assume big-endianness. The three endianness cases are based on a common auxiliary whose first argument is 1 for big-endian and 2 for little-endian, and whose second argument, delimited by the scan mark `\s__stop`, is expanded once (the string may be long; passing `\g__str_result_tl` as an argument before expansion is cheaper).

The `\__str_decode_utf_xvi:Nw` function defines `\__str_tmp:w` to take two arguments and return the character code of the first one if the string is big-endian, and the second one if the string is little-endian, then loops over the string using `\__str_decode_utf_xvi_pair:NN` described below.

```
\cs_new_protected_nopar:cpn { __str_convert_decode_utf16be: }
  { \__str_decode_utf_xvi:Nw 1 \g__str_result_tl \s__stop }
\cs_new_protected_nopar:cpn { __str_convert_decode_utf16le: }
  { \__str_decode_utf_xvi:Nw 2 \g__str_result_tl \s__stop }
\cs_new_protected_nopar:cpn { __str_convert_decode_utf16: }
  {
    \exp_after:wN \__str_decode_utf_xvi_bom:NN
      \g__str_result_tl \s__stop \s__stop \s__stop
  }
\cs_new_protected:Npn \__str_decode_utf_xvi_bom:NN #1#2
  {
    \str_if_eq_x:nnTF { #1#2 } { ^^ff ^^fe }
      { \__str_decode_utf_xvi:Nw 2 }
      {
        \str_if_eq_x:nnTF { #1#2 } { ^^fe ^^ff }
          { \__str_decode_utf_xvi:Nw 1 }
          { \__str_decode_utf_xvi:Nw 1 #1#2 }
      }
  }
\cs_new_protected:Npn \__str_decode_utf_xvi:Nw #1#2 \s__stop
  {
    \flag_clear:n { str_error }
    \flag_clear:n { str_missing }
    \flag_clear:n { str_extra }
    \flag_clear:n { str_end }
    \cs_set:Npn \__str_tmp:w ##1 ##2 { ` ## #1 }
    \tl_gset:Nx \g__str_result_tl
      {
        \exp_after:wN \__str_decode_utf_xvi_pair:NN
          #2 \q_nil \q_nil
        \__prg_break_point:
      }
    \__str_if_flag_error:nnx { str_error } { utf16-decode } { }
  }
```

(*End definition for* `\__str_convert_decode_utf16:` *,* `\__str_convert_decode_utf16be:` *, and* `\__str_convert_decode_utf16l`
These functions are documented on page **??***.*)

`\__str_decode_utf_xvi_pair:NN`
`\__str_decode_utf_xvi_quad:NNwNN`
`\__str_decode_utf_xvi_pair_end:Nw`
`\__str_decode_utf_xvi_error:nNN`
`\__str_decode_utf_xvi_extra:NNw`

Bytes are read two at a time. At this stage, `\@@_tmp:w #1#2` expands to the character code of the most significant byte, and we distinguish cases depending on which range it lies in:

- ["D8, "DB] signals a lead surrogate, and the integer expression yields 1 (ε-TEX rounds ties away from zero);
- ["DC, "DF] signals a trail surrogate, unexpected here, and the integer expression yields 2;
- any other value signals a code point in the Basic Multilingual Plane, which stands for itself, and the `\if_case:w` construction expands to nothing (cases other than 1 or 2), leaving the relevant material in the input stream, followed by another call to the `_pair` auxiliary.

The case of a lead surrogate is treated by the `_quad` auxiliary, whose arguments `#1`, `#2`, `#4` and `#5` are the four bytes. We expect the most significant byte of `#4#5` to be in the range ["DC, "DF] (trail surrogate). The test is similar to the test used for continuation bytes in the UTF-8 decoding functions. In the case where `#4#5` is indeed a trail surrogate, leave `#1#2#4#5 \s__tl` ⟨*code point*⟩ `\s__tl`, and remove the pair `#4#5` before looping with `\__str_decode_utf_xvi_pair:NN`. Otherwise, of course, complain about the missing surrogate.

The magic number "D7F7 is such that "D7F7∗"400 = "D800∗"400+"DC00−"10000.

Every time we read a pair of bytes, we test for the end-marker `\q_nil`. When reaching the end, we additionally check that the string had an even length. Also, if the end is reached when expecting a trail surrogate, we treat that as a missing surrogate.

```
\cs_new:Npn \__str_decode_utf_xvi_pair:NN #1#2
  {
    \if_meaning:w \q_nil #2
      \__str_decode_utf_xvi_pair_end:Nw #1
    \fi:
    \if_case:w
      \__int_eval:w ( \__str_tmp:w #1#2 - "D6 ) / \c_four \__int_eval_end:
    \or: \exp_after:wN \__str_decode_utf_xvi_quad:NNwNN
    \or: \exp_after:wN \__str_decode_utf_xvi_extra:NNw
    \fi:
    #1#2 \s__tl
    \int_eval:n { "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 } \s__tl
    \__str_decode_utf_xvi_pair:NN
  }
\cs_new:Npn \__str_decode_utf_xvi_quad:NNwNN
    #1#2 #3 \__str_decode_utf_xvi_pair:NN #4#5
  {
    \if_meaning:w \q_nil #5
      \__str_decode_utf_xvi_error:nNN { missing } #1#2
      \__str_decode_utf_xvi_pair_end:Nw #4
    \fi:
    \if_int_compare:w
        \if_int_compare:w \__str_tmp:w #4#5 < "DC \exp_stop_f:
          \c_zero = \c_one
        \else:
          \__str_tmp:w #4#5 < "E0 \exp_stop_f:
        \fi:
      #1 #2 #4 #5 \s__tl
      \int_eval:n
        {
          ( "100 * \__str_tmp:w #1#2 + \__str_tmp:w #2#1 - "D7F7 ) * "400
          + "100 * \__str_tmp:w #4#5 + \__str_tmp:w #5#4
        }
      \s__tl
      \exp_after:wN \use_i:nnn
    \else:
      \__str_decode_utf_xvi_error:nNN { missing } #1#2
      \fi:
      \__str_decode_utf_xvi_pair:NN #4#5
    }
  \cs_new:Npn \__str_decode_utf_xvi_pair_end:Nw #1 \fi:
    {
      \fi:
      \if_meaning:w \q_nil #1
      \else:
        \__str_decode_utf_xvi_error:nNN { end } #1 \prg_do_nothing:
      \fi:
      \__prg_break:
    }
  \cs_new:Npn \__str_decode_utf_xvi_extra:NNw #1#2 \s__tl #3 \s__tl
    { \__str_decode_utf_xvi_error:nNN { extra } #1#2 }
  \cs_new:Npn \__str_decode_utf_xvi_error:nNN #1#2#3
    {
      \flag_raise:n { str_error }
      \flag_raise:n { str_#1 }
      #2 #3 \s__tl
      \int_use:N \c__str_replacement_char_int \s__tl
    }
```

(*End definition for* `\__str_decode_utf_xvi_pair:NN`*,* `\__str_decode_utf_xvi_quad:NNwNN`*, and* `\__str_decode_utf_xvi_pai`
These functions are documented on page **??***.*)

Restore the original catcodes of bytes 254 and 255.

```
\group_end:
⟨/utf16⟩
```

5.6.3 utf-32 support

The definitions are done in a category code regime where the bytes 0, 254 and 255 used by the byte order mark have catcode "other".

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

`\__str_convert_encode_utf32:`
`\__str_convert_encode_utf32be:`
`\__str_convert_encode_utf32le:`
`\__str_encode_utf_xxxii_be:n`
`\__str_encode_utf_xxxii_be_aux:nn`
`\__str_encode_utf_xxxii_le:n`
`\__str_encode_utf_xxxii_le_aux:nn`

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

```
  \cs_new_protected_nopar:cpn { __str_convert_encode_utf32: }
    {
      \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_be:n
      \tl_gput_left:Nx \g__str_result_tl { ^^00 ^^00 ^^fe ^^ff }
    }
  \cs_new_protected_nopar:cpn { __str_convert_encode_utf32be: }
    { \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_be:n }
  \cs_new_protected_nopar:cpn { __str_convert_encode_utf32le: }
    { \__str_convert_gmap_internal:N \__str_encode_utf_xxxii_le:n }
```

```
\cs_new:Npn \__str_encode_utf_xxxii_be:n #1
  {
    \exp_args:Nf \__str_encode_utf_xxxii_be_aux:nn
      { \int_div_truncate:nn {#1} { "100 } } {#1}
  }
\cs_new:Npn \__str_encode_utf_xxxii_be_aux:nn #1#2
  {
    ^^00
    \__str_output_byte_pair_be:n {#1}
    \__str_output_byte:n { #2 - #1 * "100 }
  }
\cs_new:Npn \__str_encode_utf_xxxii_le:n #1
  {
    \exp_args:Nf \__str_encode_utf_xxxii_le_aux:nn
      { \int_div_truncate:nn {#1} { "100 } } {#1}
  }
\cs_new:Npn \__str_encode_utf_xxxii_le_aux:nn #1#2
  {
    \__str_output_byte:n { #2 - #1 * "100 }
    \__str_output_byte_pair_le:n {#1}
    ^^00
  }
```

(*End definition for* `\__str_convert_encode_utf32:` *,* `\__str_convert_encode_utf32be:` *, and* `\__str_convert_encode_utf32l`
These functions are documented on page **??**.)

`str_overflow` `str_end` There can be no error when encoding in UTF-32. When decoding, the string may not have length $4n$, or it may contain code points larger than "10FFFF. The latter case often happens if the encoding was in fact not UTF-32, because most arbitrary strings are not valid in UTF-32.

```
\flag_clear_new:n { str_overflow }
\flag_clear_new:n { str_end }
\__msg_kernel_new:nnnn { str } { utf32-decode }
  {
    Invalid~UTF-32~string: \exp_last_unbraced:Nf \use_none:n
    \__str_if_flag_times:nT { str_overflow } { ,~code~point~too~large }
    \__str_if_flag_times:nT { str_end }      { ,~truncated~string }
    .
  }
  {
    In~the~UTF-32~encoding,~every~Unicode~character~
    (in~the~range~[U+0000,~U+10FFFF])~is~encoded~as~4~bytes.
    \flag_if_raised:nT { str_overflow }
      {
        \\\\
        LaTeX~came~across~a~code~point~larger~than~1114111,~
        the~maximum~code~point~defined~by~Unicode.~
        Perhaps~the~string~was~not~encoded~in~the~UTF-32~encoding?
      }
    \flag_if_raised:nT { str_end }
```

```
        {
          \\\\
          The~length~of~the~string~is~not~a~multiple~of~4.~
          Perhaps~the~string~was~truncated?
        }
    }
```

(*End definition for* `str_overflow` *and* `str_end`*. These variables are documented on page* **??***.*)

`\__str_convert_decode_utf32:`
`\__str_convert_decode_utf32be:`
`\__str_convert_decode_utf32le:`
`\__str_decode_utf_xxxii_bom:NNNN`
`\__str_decode_utf_xxxii:Nw`
`\__str_decode_utf_xxxii_loop:NNNN`
`\__str_decode_utf_xxxii_end:w`

The structure is similar to UTF-16 decoding functions. If the endianness is not given, test the first 4 bytes of the string (possibly `\s__stop` if the string is too short) for the presence of a byte-order mark. If there is a byte-order mark, use that endianness, and remove the 4 bytes, otherwise default to big-endian, and leave the 4 bytes in place. The `\__str_decode_utf_xxxii:Nw` auxiliary recieves 1 or 2 as its first argument indicating endianness, and the string to convert as its second argument (expanded or not). It sets `\__str_tmp:w` to expand to the character code of either of its two arguments depending on endianness, then triggers the `_loop` auxiliary inside an `x`-expanding assignment to `\g__str_result_tl`.

The `_loop` auxiliary first checks for the end-of-string marker `\s__stop`, calling the `_end` auxiliary if appropriate. Otherwise, leave the ⟨*4 bytes*⟩ `\s__tl` behind, then check that the code point is not overflowing: the leading byte must be 0, and the following byte at most 16.

In the ending code, we check that there remains no byte: there should be nothing left until the first `\s__stop`. Break the map.

```
  \cs_new_protected_nopar:cpn { __str_convert_decode_utf32be: }
    { \__str_decode_utf_xxxii:Nw 1 \g__str_result_tl \s__stop }
  \cs_new_protected_nopar:cpn { __str_convert_decode_utf32le: }
    { \__str_decode_utf_xxxii:Nw 2 \g__str_result_tl \s__stop }
  \cs_new_protected_nopar:cpn { __str_convert_decode_utf32: }
    {
      \exp_after:wN \__str_decode_utf_xxxii_bom:NNNN \g__str_result_tl
        \s__stop \s__stop \s__stop \s__stop \s__stop
    }
  \cs_new_protected:Npn \__str_decode_utf_xxxii_bom:NNNN #1#2#3#4
    {
      \str_if_eq_x:nnTF { #1#2#3#4 } { ^^ff ^^fe ^^00 ^^00 }
        { \__str_decode_utf_xxxii:Nw 2 }
        {
          \str_if_eq_x:nnTF { #1#2#3#4 } { ^^00 ^^00 ^^fe ^^ff }
            { \__str_decode_utf_xxxii:Nw 1 }
            { \__str_decode_utf_xxxii:Nw 1 #1#2#3#4 }
        }
    }
  \cs_new_protected:Npn \__str_decode_utf_xxxii:Nw #1#2 \s__stop
    {
      \flag_clear:n { str_overflow }
      \flag_clear:n { str_end }
      \flag_clear:n { str_error }
      \cs_set:Npn \__str_tmp:w ##1 ##2 { ‘ ## #1 }
      \tl_gset:Nx \g__str_result_tl
        {
          \exp_after:wN \__str_decode_utf_xxxii_loop:NNNN
            #2 \s__stop \s__stop \s__stop \s__stop
          \__prg_break_point:
        }
      \__str_if_flag_error:nnx { str_error } { utf32-decode } { }
    }
  \cs_new:Npn \__str_decode_utf_xxxii_loop:NNNN #1#2#3#4
    {
      \if_meaning:w \s__stop #4
        \exp_after:wN \__str_decode_utf_xxxii_end:w
      \fi:
      #1#2#3#4 \s__tl
      \if_int_compare:w \__str_tmp:w #1#4 > \c_zero
        \flag_raise:n { str_overflow }
        \flag_raise:n { str_error }
        \int_use:N \c__str_replacement_char_int
      \else:
        \if_int_compare:w \__str_tmp:w #2#3 > \c_sixteen
          \flag_raise:n { str_overflow }
          \flag_raise:n { str_error }
          \int_use:N \c__str_replacement_char_int
        \else:
          \int_eval:n
            { \__str_tmp:w #2#3*"10000 + \__str_tmp:w #3#2*"100 + \__str_tmp:w #4#1 }
        \fi:
      \fi:
      \s__tl
      \__str_decode_utf_xxxii_loop:NNNN
    }
  \cs_new:Npn \__str_decode_utf_xxxii_end:w #1 \s__stop
    {
      \tl_if_empty:nF {#1}
        {
          \flag_raise:n { str_end }
          \flag_raise:n { str_error }
          #1 \s__tl
          \int_use:N \c__str_replacement_char_int \s__tl
        }
      \__prg_break:
    }
```

(*End definition for* `\__str_convert_decode_utf32:` , `\__str_convert_decode_utf32be:` , *and* `\__str_convert_decode_utf32l`
These functions are documented on page **??**.)

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

```
\group_end:
⟨/utf32⟩
```

5.6.4 iso 8859 support

The ISO-8859-1 encoding exactly matches with the 256 first Unicode characters. For other 8-bit encodings of the ISO-8859 family, we keep track only of differences, and of unassigned bytes.

```
⟨*iso88591⟩
\__str_declare_eight_bit_encoding:nnn { iso88591 }
  {
  }
  {
  }
⟨/iso88591⟩
```

```
⟨*iso88592⟩
\__str_declare_eight_bit_encoding:nnn { iso88592 }
  {
    { A1 } { 0104 }
    { A2 } { 02D8 }
    { A3 } { 0141 }
    { A5 } { 013D }
    { A6 } { 015A }
    { A9 } { 0160 }
    { AA } { 015E }
    { AB } { 0164 }
    { AC } { 0179 }
    { AE } { 017D }
    { AF } { 017B }
    { B1 } { 0105 }
    { B2 } { 02DB }
    { B3 } { 0142 }
    { B5 } { 013E }
    { B6 } { 015B }
    { B7 } { 02C7 }
    { B9 } { 0161 }
    { BA } { 015F }
    { BB } { 0165 }
    { BC } { 017A }
    { BD } { 02DD }
    { BE } { 017E }
    { BF } { 017C }
    { C0 } { 0154 }
    { C3 } { 0102 }
    { C5 } { 0139 }
    { C6 } { 0106 }
    { C8 } { 010C }
    { CA } { 0118 }
    { CC } { 011A }
    { CF } { 010E }
    { D0 } { 0110 }
    { D1 } { 0143 }
    { D2 } { 0147 }
    { D5 } { 0150 }
    { D8 } { 0158 }
    { D9 } { 016E }
    { DB } { 0170 }
    { DE } { 0162 }
    { E0 } { 0155 }
    { E3 } { 0103 }
    { E5 } { 013A }
    { E6 } { 0107 }
    { E8 } { 010D }
    { EA } { 0119 }
    { EC } { 011B }
    { EF } { 010F }
    { F0 } { 0111 }
    { F1 } { 0144 }
    { F2 } { 0148 }
    { F5 } { 0151 }
    { F8 } { 0159 }
    { F9 } { 016F }
    { FB } { 0171 }
    { FE } { 0163 }
    { FF } { 02D9 }
  }
  {
  }
⟨/iso88592⟩
⟨*iso88593⟩
\__str_declare_eight_bit_encoding:nnn { iso88593 }
  {
    { A1 } { 0126 }
    { A2 } { 02D8 }
    { A6 } { 0124 }
    { A9 } { 0130 }
    { AA } { 015E }
    { AB } { 011E }
    { AC } { 0134 }
    { AF } { 017B }
    { B1 } { 0127 }
    { B6 } { 0125 }
    { B9 } { 0131 }
    { BA } { 015F }
    { BB } { 011F }
    { BC } { 0135 }
    { BF } { 017C }
    { C5 } { 010A }
    { C6 } { 0108 }
    { D5 } { 0120 }
    { D8 } { 011C }
    { DD } { 016C }
    { DE } { 015C }
    { E5 } { 010B }
    { E6 } { 0109 }
    { F5 } { 0121 }
    { F8 } { 011D }
    { FD } { 016D }
    { FE } { 015D }
    { FF } { 02D9 }
  }
  {
    { A5 }
    { AE }
    { BE }
    { C3 }
    { D0 }
    { E3 }
    { F0 }
  }
⟨/iso88593⟩
⟨*iso88594⟩
\__str_declare_eight_bit_encoding:nnn { iso88594 }
  {
    { A1 } { 0104 }
    { A2 } { 0138 }
    { A3 } { 0156 }
    { A5 } { 0128 }
    { A6 } { 013B }
    { A9 } { 0160 }
    { AA } { 0112 }
    { AB } { 0122 }
    { AC } { 0166 }
    { AE } { 017D }
    { B1 } { 0105 }
    { B2 } { 02DB }
    { B3 } { 0157 }
    { B5 } { 0129 }
    { B6 } { 013C }
    { B7 } { 02C7 }
    { B9 } { 0161 }
    { BA } { 0113 }
    { BB } { 0123 }
    { BC } { 0167 }
    { BD } { 014A }
    { BE } { 017E }
    { BF } { 014B }
    { C0 } { 0100 }
    { C7 } { 012E }
    { C8 } { 010C }
    { CA } { 0118 }
    { CC } { 0116 }
    { CF } { 012A }
    { D0 } { 0110 }
    { D1 } { 0145 }
    { D2 } { 014C }
    { D3 } { 0136 }
    { D9 } { 0172 }
    { DD } { 0168 }
    { DE } { 016A }
    { E0 } { 0101 }
    { E7 } { 012F }
    { E8 } { 010D }
    { EA } { 0119 }
    { EC } { 0117 }
    { EF } { 012B }
    { F0 } { 0111 }
    { F1 } { 0146 }
    { F2 } { 014D }
    { F3 } { 0137 }
    { F9 } { 0173 }
    { FD } { 0169 }
    { FE } { 016B }
    { FF } { 02D9 }
  }
  {
  }
⟨/iso88594⟩
⟨*iso88595⟩
\__str_declare_eight_bit_encoding:nnn { iso88595 }
  {
    { A1 } { 0401 }
    { A2 } { 0402 }
    { A3 } { 0403 }
    { A4 } { 0404 }
    { A5 } { 0405 }
    { A6 } { 0406 }
    { A7 } { 0407 }
    { A8 } { 0408 }
    { A9 } { 0409 }
    { AA } { 040A }
    { AB } { 040B }
    { AC } { 040C }
    { AE } { 040E }
    { AF } { 040F }
    { B0 } { 0410 }
    { B1 } { 0411 }
    { B2 } { 0412 }
    { B3 } { 0413 }
    { B4 } { 0414 }
    { B5 } { 0415 }
    { B6 } { 0416 }
    { B7 } { 0417 }
    { B8 } { 0418 }
    { B9 } { 0419 }
    { BA } { 041A }
    { BB } { 041B }
    { BC } { 041C }
    { BD } { 041D }
    { BE } { 041E }
    { BF } { 041F }
    { C0 } { 0420 }
    { C1 } { 0421 }
    { C2 } { 0422 }
    { C3 } { 0423 }
    { C4 } { 0424 }
    { C5 } { 0425 }
    { C6 } { 0426 }
    { C7 } { 0427 }
    { C8 } { 0428 }
    { C9 } { 0429 }
    { CA } { 042A }
    { CB } { 042B }
    { CC } { 042C }
    { CD } { 042D }
    { CE } { 042E }
    { CF } { 042F }
    { D0 } { 0430 }
    { D1 } { 0431 }
    { D2 } { 0432 }
    { D3 } { 0433 }
    { D4 } { 0434 }
    { D5 } { 0435 }
    { D6 } { 0436 }
    { D7 } { 0437 }
    { D8 } { 0438 }
    { D9 } { 0439 }
    { DA } { 043A }
    { DB } { 043B }
    { DC } { 043C }
    { DD } { 043D }
    { DE } { 043E }
    { DF } { 043F }
    { E0 } { 0440 }
    { E1 } { 0441 }
    { E2 } { 0442 }
    { E3 } { 0443 }
    { E4 } { 0444 }
    { E5 } { 0445 }
    { E6 } { 0446 }
    { E7 } { 0447 }
    { E8 } { 0448 }
    { E9 } { 0449 }
    { EA } { 044A }
    { EB } { 044B }
    { EC } { 044C }
    { ED } { 044D }
    { EE } { 044E }
    { EF } { 044F }
    { F0 } { 2116 }
    { F1 } { 0451 }
    { F2 } { 0452 }
    { F3 } { 0453 }
    { F4 } { 0454 }
    { F5 } { 0455 }
    { F6 } { 0456 }
    { F7 } { 0457 }
    { F8 } { 0458 }
    { F9 } { 0459 }
    { FA } { 045A }
    { FB } { 045B }
    { FC } { 045C }
    { FD } { 00A7 }
    { FE } { 045E }
    { FF } { 045F }
  }
  {
  }
⟨/iso88595⟩
⟨*iso88596⟩
\__str_declare_eight_bit_encoding:nnn { iso88596 }
  {
    { AC } { 060C }
    { BB } { 061B }
    { BF } { 061F }
    { C1 } { 0621 }
    { C2 } { 0622 }
    { C3 } { 0623 }
    { C4 } { 0624 }
    { C5 } { 0625 }
    { C6 } { 0626 }
    { C7 } { 0627 }
    { C8 } { 0628 }
    { C9 } { 0629 }
    { CA } { 062A }
    { CB } { 062B }
    { CC } { 062C }
    { CD } { 062D }
    { CE } { 062E }
    { CF } { 062F }
    { D0 } { 0630 }
    { D1 } { 0631 }
    { D2 } { 0632 }
    { D3 } { 0633 }
    { D4 } { 0634 }
    { D5 } { 0635 }
    { D6 } { 0636 }
    { D7 } { 0637 }
    { D8 } { 0638 }
    { D9 } { 0639 }
    { DA } { 063A }
    { E0 } { 0640 }
    { E1 } { 0641 }
    { E2 } { 0642 }
    { E3 } { 0643 }
    { E4 } { 0644 }
    { E5 } { 0645 }
    { E6 } { 0646 }
    { E7 } { 0647 }
    { E8 } { 0648 }
    { E9 } { 0649 }
    { EA } { 064A }
    { EB } { 064B }
    { EC } { 064C }
    { ED } { 064D }
    { EE } { 064E }
    { EF } { 064F }
    { F0 } { 0650 }
    { F1 } { 0651 }
    { F2 } { 0652 }
  }
  {
    { A1 }
    { A2 }
    { A3 }
    { A5 }
    { A6 }
    { A7 }
    { A8 }
    { A9 }
    { AA }
    { AB }
    { AE }
    { AF }
    { B0 }
    { B1 }
    { B2 }
    { B3 }
```

```
    { B4 }
    { B5 }
    { B6 }
    { B7 }
    { B8 }
    { B9 }
    { BA }
    { BC }
    { BD }
    { BE }
    { C0 }
    { DB }
    { DC }
    { DD }
    { DE }
    { DF }
  }
⟨/iso88596⟩
⟨*iso88597⟩
\__str_declare_eight_bit_encoding:nnn { iso88597 }
  {
    { A1 } { 2018 }
    { A2 } { 2019 }
    { A4 } { 20AC }
    { A5 } { 20AF }
    { AA } { 037A }
    { AF } { 2015 }
    { B4 } { 0384 }
    { B5 } { 0385 }
    { B6 } { 0386 }
    { B8 } { 0388 }
    { B9 } { 0389 }
    { BA } { 038A }
    { BC } { 038C }
    { BE } { 038E }
    { BF } { 038F }
    { C0 } { 0390 }
    { C1 } { 0391 }
    { C2 } { 0392 }
    { C3 } { 0393 }
    { C4 } { 0394 }
    { C5 } { 0395 }
    { C6 } { 0396 }
    { C7 } { 0397 }
    { C8 } { 0398 }
    { C9 } { 0399 }
    { CA } { 039A }
    { CB } { 039B }
    { CC } { 039C }
    { CD } { 039D }
    { CE } { 039E }
    { CF } { 039F }
    { D0 } { 03A0 }
    { D1 } { 03A1 }
    { D3 } { 03A3 }
    { D4 } { 03A4 }
    { D5 } { 03A5 }
    { D6 } { 03A6 }
    { D7 } { 03A7 }
    { D8 } { 03A8 }
    { D9 } { 03A9 }
    { DA } { 03AA }
    { DB } { 03AB }
    { DC } { 03AC }
    { DD } { 03AD }
    { DE } { 03AE }
    { DF } { 03AF }
    { E0 } { 03B0 }
    { E1 } { 03B1 }
    { E2 } { 03B2 }
    { E3 } { 03B3 }
    { E4 } { 03B4 }
    { E5 } { 03B5 }
    { E6 } { 03B6 }
    { E7 } { 03B7 }
    { E8 } { 03B8 }
    { E9 } { 03B9 }
    { EA } { 03BA }
    { EB } { 03BB }
    { EC } { 03BC }
    { ED } { 03BD }
    { EE } { 03BE }
    { EF } { 03BF }
    { F0 } { 03C0 }
    { F1 } { 03C1 }
    { F2 } { 03C2 }
    { F3 } { 03C3 }
    { F4 } { 03C4 }
    { F5 } { 03C5 }
    { F6 } { 03C6 }
    { F7 } { 03C7 }
    { F8 } { 03C8 }
    { F9 } { 03C9 }
    { FA } { 03CA }
    { FB } { 03CB }
    { FC } { 03CC }
    { FD } { 03CD }
    { FE } { 03CE }
  }
```

```
  {
    { AE }
    { D2 }
  }
⟨/iso88597⟩
⟨*iso88598⟩
\__str_declare_eight_bit_encoding:nnn { iso88598 }
  {
    { AA } { 00D7 }
    { BA } { 00F7 }
    { DF } { 2017 }
    { E0 } { 05D0 }
    { E1 } { 05D1 }
    { E2 } { 05D2 }
    { E3 } { 05D3 }
    { E4 } { 05D4 }
    { E5 } { 05D5 }
    { E6 } { 05D6 }
    { E7 } { 05D7 }
    { E8 } { 05D8 }
    { E9 } { 05D9 }
    { EA } { 05DA }
    { EB } { 05DB }
    { EC } { 05DC }
    { ED } { 05DD }
    { EE } { 05DE }
    { EF } { 05DF }
    { F0 } { 05E0 }
    { F1 } { 05E1 }
    { F2 } { 05E2 }
    { F3 } { 05E3 }
    { F4 } { 05E4 }
    { F5 } { 05E5 }
    { F6 } { 05E6 }
    { F7 } { 05E7 }
    { F8 } { 05E8 }
    { F9 } { 05E9 }
    { FA } { 05EA }
    { FD } { 200E }
    { FE } { 200F }
  }
  {
    { A1 }
    { BF }
    { C0 }
    { C1 }
    { C2 }
    { C3 }
    { C4 }
    { C5 }
    { C6 }
    { C7 }
    { C8 }
    { C9 }
    { CA }
    { CB }
    { CC }
    { CD }
    { CE }
    { CF }
    { D0 }
    { D1 }
    { D2 }
    { D3 }
    { D4 }
    { D5 }
    { D6 }
    { D7 }
    { D8 }
    { D9 }
    { DA }
    { DB }
    { DC }
    { DD }
    { DE }
    { FB }
    { FC }
  }
⟨/iso88598⟩
⟨*iso88599⟩
\__str_declare_eight_bit_encoding:nnn { iso88599 }
  {
    { D0 } { 011E }
    { DD } { 0130 }
    { DE } { 015E }
    { F0 } { 011F }
    { FD } { 0131 }
    { FE } { 015F }
  }
  {
  }
⟨/iso88599⟩
⟨*iso885910⟩
\__str_declare_eight_bit_encoding:nnn { iso885910 }
  {
    { A1 } { 0104 }
    { A2 } { 0112 }
    { A3 } { 0122 }
    { A4 } { 012A }
    { A5 } { 0128 }
    { A6 } { 0136 }
    { A8 } { 013B }
    { A9 } { 0110 }
    { AA } { 0160 }
    { AB } { 0166 }
    { AC } { 017D }
    { AE } { 016A }
    { AF } { 014A }
    { B1 } { 0105 }
    { B2 } { 0113 }
    { B3 } { 0123 }
    { B4 } { 012B }
    { B5 } { 0129 }
    { B6 } { 0137 }
    { B8 } { 013C }
    { B9 } { 0111 }
    { BA } { 0161 }
    { BB } { 0167 }
    { BC } { 017E }
    { BD } { 2015 }
    { BE } { 016B }
    { BF } { 014B }
    { C0 } { 0100 }
    { C7 } { 012E }
    { C8 } { 010C }
    { CA } { 0118 }
    { CC } { 0116 }
    { D1 } { 0145 }
    { D2 } { 014C }
    { D7 } { 0168 }
    { D9 } { 0172 }
    { E0 } { 0101 }
    { E7 } { 012F }
    { E8 } { 010D }
    { EA } { 0119 }
    { EC } { 0117 }
    { F1 } { 0146 }
    { F2 } { 014D }
    { F7 } { 0169 }
    { F9 } { 0173 }
    { FF } { 0138 }
  }
  {
  }
⟨/iso885910⟩
⟨*iso885911⟩
\__str_declare_eight_bit_encoding:nnn { iso885911 }
```

```
{
  { A1 } { 0E01 }
  { A2 } { 0E02 }
  { A3 } { 0E03 }
  { A4 } { 0E04 }
  { A5 } { 0E05 }
  { A6 } { 0E06 }
  { A7 } { 0E07 }
  { A8 } { 0E08 }
  { A9 } { 0E09 }
  { AA } { 0E0A }
  { AB } { 0E0B }
  { AC } { 0E0C }
  { AD } { 0E0D }
  { AE } { 0E0E }
  { AF } { 0E0F }
  { B0 } { 0E10 }
  { B1 } { 0E11 }
  { B2 } { 0E12 }
  { B3 } { 0E13 }
  { B4 } { 0E14 }
  { B5 } { 0E15 }
  { B6 } { 0E16 }
  { B7 } { 0E17 }
  { B8 } { 0E18 }
  { B9 } { 0E19 }
  { BA } { 0E1A }
  { BB } { 0E1B }
  { BC } { 0E1C }
  { BD } { 0E1D }
  { BE } { 0E1E }
  { BF } { 0E1F }
  { C0 } { 0E20 }
  { C1 } { 0E21 }
  { C2 } { 0E22 }
  { C3 } { 0E23 }
  { C4 } { 0E24 }
  { C5 } { 0E25 }
  { C6 } { 0E26 }
  { C7 } { 0E27 }
  { C8 } { 0E28 }
  { C9 } { 0E29 }
  { CA } { 0E2A }
  { CB } { 0E2B }
  { CC } { 0E2C }
  { CD } { 0E2D }
  { CE } { 0E2E }
  { CF } { 0E2F }
  { D0 } { 0E30 }
  { D1 } { 0E31 }
```

```
    { D2 } { 0E32 }
    { D3 } { 0E33 }
    { D4 } { 0E34 }
    { D5 } { 0E35 }
    { D6 } { 0E36 }
    { D7 } { 0E37 }
    { D8 } { 0E38 }
    { D9 } { 0E39 }
    { DA } { 0E3A }
    { DF } { 0E3F }
    { E0 } { 0E40 }
    { E1 } { 0E41 }
    { E2 } { 0E42 }
    { E3 } { 0E43 }
    { E4 } { 0E44 }
    { E5 } { 0E45 }
    { E6 } { 0E46 }
    { E7 } { 0E47 }
    { E8 } { 0E48 }
    { E9 } { 0E49 }
    { EA } { 0E4A }
    { EB } { 0E4B }
    { EC } { 0E4C }
    { ED } { 0E4D }
    { EE } { 0E4E }
    { EF } { 0E4F }
    { F0 } { 0E50 }
    { F1 } { 0E51 }
    { F2 } { 0E52 }
    { F3 } { 0E53 }
    { F4 } { 0E54 }
    { F5 } { 0E55 }
    { F6 } { 0E56 }
    { F7 } { 0E57 }
    { F8 } { 0E58 }
    { F9 } { 0E59 }
    { FA } { 0E5A }
    { FB } { 0E5B }
  }
  {
    { DB }
    { DC }
    { DD }
    { DE }
  }
⟨/iso885911⟩
⟨*iso885913⟩
\__str_declare_eight_bit_encoding:nnn { iso885913 }
  {
    { A1 } { 201D }
    { A5 } { 201E }
    { A8 } { 00D8 }
    { AA } { 0156 }
    { AF } { 00C6 }
    { B4 } { 201C }
    { B8 } { 00F8 }
    { BA } { 0157 }
    { BF } { 00E6 }
    { C0 } { 0104 }
    { C1 } { 012E }
    { C2 } { 0100 }
    { C3 } { 0106 }
    { C6 } { 0118 }
    { C7 } { 0112 }
    { C8 } { 010C }
    { CA } { 0179 }
    { CB } { 0116 }
    { CC } { 0122 }
    { CD } { 0136 }
    { CE } { 012A }
    { CF } { 013B }
    { D0 } { 0160 }
    { D1 } { 0143 }
    { D2 } { 0145 }
    { D4 } { 014C }
    { D8 } { 0172 }
    { D9 } { 0141 }
    { DA } { 015A }
    { DB } { 016A }
    { DD } { 017B }
    { DE } { 017D }
    { E0 } { 0105 }
    { E1 } { 012F }
    { E2 } { 0101 }
    { E3 } { 0107 }
    { E6 } { 0119 }
    { E7 } { 0113 }
    { E8 } { 010D }
    { EA } { 017A }
    { EB } { 0117 }
    { EC } { 0123 }
    { ED } { 0137 }
    { EE } { 012B }
    { EF } { 013C }
    { F0 } { 0161 }
    { F1 } { 0144 }
    { F2 } { 0146 }
    { F4 } { 014D }
    { F8 } { 0173 }
    { F9 } { 0142 }
    { FA } { 015B }
    { FB } { 016B }
    { FD } { 017C }
    { FE } { 017E }
    { FF } { 2019 }
  }
  {
  }
⟨/iso885913⟩
⟨*iso885914⟩
\__str_declare_eight_bit_encoding:nnn { iso885914 }
  {
    { A1 } { 1E02 }
    { A2 } { 1E03 }
    { A4 } { 010A }
    { A5 } { 010B }
    { A6 } { 1E0A }
    { A8 } { 1E80 }
    { AA } { 1E82 }
    { AB } { 1E0B }
    { AC } { 1EF2 }
    { AF } { 0178 }
    { B0 } { 1E1E }
    { B1 } { 1E1F }
    { B2 } { 0120 }
    { B3 } { 0121 }
    { B4 } { 1E40 }
    { B5 } { 1E41 }
    { B7 } { 1E56 }
    { B8 } { 1E81 }
    { B9 } { 1E57 }
    { BA } { 1E83 }
    { BB } { 1E60 }
    { BC } { 1EF3 }
    { BD } { 1E84 }
    { BE } { 1E85 }
    { BF } { 1E61 }
    { D0 } { 0174 }
    { D7 } { 1E6A }
    { DE } { 0176 }
    { F0 } { 0175 }
    { F7 } { 1E6B }
    { FE } { 0177 }
  }
  {
  }
⟨/iso885914⟩
⟨*iso885915⟩
```

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

Index

The italic numbers denote the pages where the corresponding entry is described, numbers underlined point to the definition, all others indicate the places where it is used.