EncTEX

The Extension of TEX For Input Re-encoding

Petr Olšák

www.olsak.net/enctex.html

This text documents the version Feb. 2003 and Jun. 2004

This package is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This package is available on ftp://math.feld.cvut.cz/pub/olsak/enctex/.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

© 1997, 2002, 2003, 2004 RNDr. Petr Olšák

TEX is trademark of the American Mathematical Society.

The author of the TEX is professor Donald Knuth. The TEX is a free software with the specific license. See the documentation of TEX.

The original version of the encTEX documentation (in Czech language) is in `encdoc.tex` file.

Původní česká dokumentace je v souboru `encdoc.tex`.

1. The basic information

The encTEX package is a little extension of TEX. You can install it from source files of TEX by changing the `tex.ch` file in your distribution. The patch to `tex.ch` file for `web2c` distribution is supported.

The encTEX is backward compatible with the original TEX. It adds ten new primitives by which you can set or read the conversion tables used by input processor of TEX or used during output to the terminal, log and `\write` files. These tables are stored to the format files thus, they are reinitialized to the same state as in time of `\dump` command when the format file is read.

This extension is fully tested and it passes the TRIP test with only two differences:

- The banner is different
- The number of "multiletter control sequences" is greater by ten.

1.1. The installation

For install instructions of encTEX – read the `INSTALL.eng` file.

1.2. Versions

I released the first version of encTEX in 1997. This version was able to do the byte to byte conversion only by xord and xchr vector and to assign the characters as "printable" (the `\xordcode`, `\xchrcode` and `\xprncode` primitives).

The TCX tables were renoved in 1998. These tables do the same work as encTEX 1997 thus I closed my support of encTEX. The problem with the missing support of UTF-8 encoding on input processor caused the reinterpretation of my old decision: I implemented the UTF-8 support to the encTEX in December 2002 and I propagate encTEX again.

The new version "Feb. 2003" adds seven new primitives: `\mubyte`, `\endmubyte`, `\mubytein`, `\mubyteout`, `\mubytelog`, `\specialout` and `\noconvert`. They give a possibility to set the conversion from UTF-8 encoded files.

The version "Jun. 2004" fixes a little bugs only. No new features are added. See `enctex.patch-jun2004` file for more information about these bugs.

Of course, the new version is backward compatible with the old one from 1997. I don't plan any new big changes. If a little change will be done then *the backward compatibility with the previous version of encTEX is guaranteed by me.*

1.3. EncTEX in web2c distribution, TCX tables

If your web2c distribution implements encTEX then you can initialize it by the `-enc` option in command line. You have to use this option during iniTEX because encTEX stores its primitives and its data to the format file. When the format is used, the encTEX is initialized from format file automatically and you need not use the `-enc` option again. If you are using a format without encTEX initialized in it and you write `-enc` option then the warning is printed and this option is ignored.

The TCX tables (`-translate-file` option) are working with the same xord and xchr vector as encTEX in web2c distribution. This implies the following little conflicts: If encTEX is used together with TCX table then TCX table may re-write the initial values of `\xordcode`, `\xchrcode` and `\xprncode`. These initial values are documented in section 2.2. If these values are stored in format by encTEX and TCX table is used together with such format then the values from format can be re-written by TCX table too. On the other hand, you can use the `\xordcode`, `\xchrcode` and `\xprncode` primitives for reading or saving of these values after TCX table initialization without problems.

1.4. The TEX license

The encTEX adds the new primitives into TEX so, we cannot call the resulting program by name TEX. On the other hand D. Knuth assumed that TEX internals are filtered always from system dependences. This was a reason why he implemented xord/xchr vectors in TEX. D. Knuth assumed that the parameters of filter from system dependences is set at source code level. EncTEX only moves this setting from source code level to the runtime level. This is nothing new: the TEX memory parameters are possible to set at runtime in

modern TEX distributions too. You can set the conversion tables depend on your system. Then you can say `\let\xordcode=\undefined` etc. (the same for other encTEX primitives) and you can do `\dump`. The format has the conversion tables stored by the system specifications and the user cannot do any more changes. The using of this format acts the same as the using of the original TEX.

I think that the second line on the terminal and log file is sufficient information about the fact that the program is a modified version of TEX. I think that if the UTF-8 encoding will be used more common then there is no another way than to modify the input processor of TEX otherwise the 8bit TEX will dead in short time.

It is important to say that encTEX has the same default behavior as the original TEX if the new primitives are never used.

IMHO, the new web2c TEX is not exactly the TEX too because you can change its behavior by writing `%&` at the first line of the document. This feature is not documented in *Computers & Typesetting* series.

2. The byte per byte conversion

2.1. The xord and xchr vectors

All text inputs into TEX are mapped by xord vector in input preprocessor (the eyes in TEXbook terminology). If the character has the code x in your system, the same character has the code $y = \mathrm{xord}[x]$ in TEX.

All text outputs from TEX to terminal, log file and files managed by `\write` primitive are filtered by xchr vector and by "printability" feature of the character. If the character with code y is not "printable", then it outputs by `^^code` notation (documented in TEXbook, page 45). If the character with code y is "printable" then the output code of this character on terminal and text files is $z = \mathrm{xchr}[y]$.

2.2. The new primitives with the access to the xord and xchr vectors

The encTEX extension introduces three new primitives with the same syntax as `\lccode`:

- `\xordcode` i ... is xord[i]
- `\xchrcode` i ... is xchr[i]
- the character with the code i is "printable" (not `^notation` on terminal and the log is used) iff (`\xprncode` $i > 0$) or ($i \in \{32, ..., 126\}$).

All setting to `\xordcode`, `\xchrcode` and `\xprncode` are possible in 0...255 range and are *global* every time. It means that the setting inside group are global and it is irrelevant if you write `\global` prefix or you do not.

The initial values at iniTEX state of the mentioned vectors are:

- `\xordcode` $i = i$ for $i \in \{128...255\}$,
- `\xchrcode` $i = i$ for $i \in \{128...255\}$,
- `\xprncode` $i = 0$ for $i \in \{0...31, 127...255\}$,
- `\xprncode` $i = 1$ for $i \in \{32...126\}$.

The `\xordcode` i and `\xchrcode` i for $i \in \{0...127\}$ are system dependent, but on systems with ASCII encoding holds: `\xordcode` $i = i$, `\xchrcode` $i = i$.

3. The multi-byte conversion

Since version Dec 2002, the encTEX is able to convert more bytes to one byte or control sequence on input processor level. This "one byte" is converted back to the original "more bytes" when `\write` is processed or TEX outputs to the terminal or log file. The main reason of this extension of TEX is to serve to work with the UTF-8 encoded input files: we need to assign the `\catcode`s, `\uccode`s etc. to the letters in our alphabet but some letters are encoded in two bytes in UTF-8. The encTEX is able to map other codes from UTF-8 to control sequences thus, the number of UTF-8 codes from input file examined by TEX is unlimited.

There are five new primitives to manage the conversion: `\mubytein`, `\mubyteout`, `\mubytelog`, `\mubyte`, `\endmubyte`. The `\mubytein`, `\mubyteout` and `\mubytelog` are integer registers with zero value by default: it means that no conversion is processed even if the conversion table (created by `\mubyte`, `\endmubyte`) is non empty. If `\mubytein` is positive then the conversion on input processor level is performed by the conversion table. If `\mubyteout` is positive then the conversion for output to the `\write` files is activated by the same conversion table. If `\mubytelog` is positive then the output conversion is activated for log file and terminal output.

The conversion table is empty by default and you can add the new line into this table by the couple of `\mubyte`, `\endmubyte` primitives:

`\mubyte` ⟨*first_token*⟩⟨*one_optional_space*⟩⟨*optional_prefix*⟩⟨*byte_sequence*⟩`\endmubyte`

Each ⟨*byte_sequence*⟩ will be converted to the ⟨*first_token*⟩ at input processor level. There are two possibilities for ⟨*first_token*⟩: it may be a character or a control sequence. If the ⟨*first_token*⟩ is a character then the catcode of it is ignored and the ⟨*first_token*⟩ is interpreted as a ⟨*byte*⟩. This ⟨*byte*⟩ is converted back to the ⟨*byte_sequence*⟩ in `\write` files, log file and terminal.

If the ⟨*first_token*⟩ is a control sequence then the ⟨*byte_sequence*⟩ will be converted to this control sequence of the "one token" form at input processor level. It means that the token processor never changes this control sequence. The token processor stays in middle line state after this control sequence is scanned. If `\mubyteout<2` then the output to the `\write` files is not converted back to the ⟨*byte_sequence*⟩ and the control sequence is expanded as usual. If `\mubyteout>=2` then the control sequence declared by `\mubyte` is converted back to the ⟨*byte_sequence*⟩ in `\write` parameters. This works only if the control sequence is not expanded. It means that the control sequence have to be non expandable or it have to be marked by `\noexpand`. If `\mubyteout>=3` then encTEX suppresses the expansion of control sequences declared by `\mubyte` automatically. See section 3.7 for more details.

The control sequences are never converted back to ⟨*byte_sequence*⟩ in log file and on the terminal output.

The syntax and the meaning of ⟨*optional_prefix*⟩ will be explained in section 3.4.

3.1. The conversion table manipulation

The data are stored into conversion table as a global assignment. On the other hand the assignment to `\mubytein`, `\mubyteout` and `\mubytelog` registers are local as usually.

The `\mubyte`, `\endmubyte` primitives work very similar as a well known `\csname`, `\endcsname` pair. The difference is that the ⟨*first_token*⟩ is not expanded and that this token can be followed by ⟨*one_optional_space*⟩ (after expansion). The ⟨*byte_sequence*⟩ is scanned with the full expansion. If the other non expandable control sequence than `\endmubyte` occurs during this process then the error message is printed:

```
! Missing \endmubyte inserted.
```

The `\mubyte` is not performed on the expand processor level: it is a assign primitive. If you write

```
\edef\a{\mubyte X ABC\endmubyte}
```

then the macro `\a` includes the `\mubyte X ABC\endmubyte` tokens.

Examples:

```
\mubyte  ^^c1      ^^c3^^81\endmubyte % \'A
\mubyte  ^^e1      ^^c3^^a1\endmubyte % \'a
% etc. -- the UTF8 implementation

\mubyte  \endash   ^^c4^^f6\endmubyte % the mapping to the control sequence
\mubyte  \integral  INT\endmubyte     % the illustrative example, see below

\mubytein=1  \mubyteout=1  % conversions are activated here

\def\endash {--} % this is good definition for \write files too
\def\integral {\ifmmode \int\else $\int$\fi}
```

We have written more spaces (or tabs) in ⟨*one optional space*⟩ in this example because these characters have the catcode of the space and the token processor converts them to right ⟨*one optional space*⟩.

The word "`INTEGRAL`" is converted to the token `\integral` followed by the letters "`EGRAL`" if the example code is used. The text "`INT something`" is converted to the token `\integral` followed by space and the word "`something`". You can write the following constructions: `\defINT{something}`, `\let INT=\foo`, etc. After `\show INT` we get:

```
> \integral=macro:
->\ifmmode \int \else $\int $\fi .
l.18 \show INT
```

and `\string INT` expands to the text: `\integral`.

Assume the `INT` declaration from the previous example and assume that you write `\INT`. What happens? Strictly speaking, the empty control sequence (`\csname\endcsname`) followed by `\integral` control sequence would be the output from the token processor. But there is an exception in encTEX because to avoid the confusion with the empty control sequences. The `\INT` produces only the control sequence `\integral`, the backslash is ignored in this situation. The token processor stays in middle state after `\INT` is scanned, the letter can follow immediately.

3.2. The features of the conversion process

The input is converted immediately after `\mubytein` is set to the positive value; it means the conversion may start at the same line where the `\mubytein` setting occurs.

The ⟨*byte_sequence*⟩ is converted only if the whole ⟨*byte_sequence*⟩ is included in the one line. The `\endlinechar` character can be the last part of the ⟨*byte_sequence*⟩.

The sequence `^^c3^^81` is not converted to the letter Á even if the code from the example was used. The reason is that the `^^` conversion is done in token processor after the `\mubyte` conversion.

The `\xordcode` conversion is performed before `\mubyte` conversion in input side and the `\xchrcode` conversion is done after `\mubyte` conversion during output to the files or to the terminal. The following diagram shows the sequence of the conversions:

```
input text -> \xordcode -> \endlinechar appended ->
              \mubyte -> token processor -> expand processor ...
\write argument -> expand processor -> \mubyte -> \xchrcode -> output
```

The ⟨*byte_sequence*⟩ is not converted to the `^^` form during output to the file even if the `\xprncode` of the bytes from ⟨*byte_sequence*⟩ is zero. The ⟨*byte_sequence*⟩ is not converted again even if there exist a character in it which is normally converted by another rule in conversion table.

Let exist two or more ⟨*byte_sequences*⟩ in the conversion table which are equal or which have the same begin part and one sequence is a subsequence of the second. Then the conversion in input processor is done by maximal possible ⟨*byte_sequence*⟩. This feature was implemented in version Feb. 2003. Example:

```
\mubyte X A\endmubyte
\mubyte Y ABC\endmubyte
\mubyte \foo ABCD\endmubyte
```

The letter A is normally converted to X in this example, but if the BC letters immediately follow then ABC is converted to Y with the exception ABCD which is converted to `\foo`.

The order of `\mubyte` settings in this example has no significance.

If the same ⟨*byte_sequences*⟩ are used by `\mubyte` records then the last one has a precedence and the previous records are cleared.

3.3. The conversion to log file and to the terminal

The output to terminal and to log file is not converted if `\mubytelog` is zero. If the `\xprncode` of the character is zero then the character is printed in `^^A` or `^^bc` form. If the `\mubytelog` is positive then the characters stored in conversion table are converted to the ⟨*byte_sequence*⟩ and the bytes from these

⟨*byte_sequence*⟩ are never converted to `^^` form. On the other hand, control sequences are keept unchanged in log and in terminal even if the `\mubytelog` is positive.

The conversion is switched on or off by `\mubytelog` value for both: terminal and log file. You cannot separate these outputs. It means that (for example) the conversion to log and no conversion to terminal is not possible.

There exists a special part of terminal and log output: if the complete line from input is reprinted (for example when the context of an error is shown). We call this situation as "line-reprinting" for the purpose of the following text.

If the `\mubytein` is zero then line-reprinting works as usual in standard TEX. If `\mubytein` is positive and `\mubytelog` is zero then line-reprinting includes the output from the input processor of encTEX. It means that control sequences generated by input processor can be shown here even if they don't actually present in the input line. If both `\mubytein` and `\mubytlog` are positive then line-reprinting works without any multi-byte conversion, only xord is used immediately followed by xchr. No `^^A` form is used in this situation. Note that the error messages can be somewhat peculiar when `\mubytein` and `\mubytlog` are positive:

```
\mubyte \cmd ABC\endmubyte  \let\cmd=\undefined
\mubytein=1 \mubytelog=1
This is test of ABC and another text.
```

We get the following message:

```
! Undefined control sequence.
l.3 This is test of ABC
                        and another text.
?
```

The `\show ABC` can say you more information:

```
> \cmd=undefined.
l.3 \show ABC
```

3.4. Clearing records from the conversion table

There exists only the chance to clear all records which begin with the same first byte of ⟨*byte_sequence*⟩. This is done by the command `\mubyte` ⟨*char*⟩ ⟨*char*⟩`\endmubyte`. For example

```
\mubyte A A\endmubyte
```

clears all ⟨*byte_sequences*⟩ from conversion table which begin with the character A. The following code clears all conversion table:

```
{\catcode`\^^@=12
\gdef\clearmubytes{\bgroup \count255=1
   \loop \uccode`X=\count255
       \uppercase{\mubyte XX\endmubyte}%
       \advance\count255 by1
       \ifnum\count255<256 \repeat
   \mubyte ^^@^^@\endmubyte
   \egroup}
}
\clearmubytes
```

3.5. Input and output sides of the conversion table

The conversion table consists from two independent parts: input side used by input processor and output side used during `\write` or printing to the log and terminal. You can save the record only to one of this parts by using the nonempty ⟨*optional_prefix*⟩. If the ⟨*optional_prefix*⟩ is empty then the same record is stored twice: into input and output sides. If ⟨*optional_prefix*⟩ is a token of catcode 8 (usually the _ character) then

the record is stored only into input side . If ⟨*optional_prefix*⟩ is a pair of tokens catcode 8 (usually `__`) then the record is stored only into output side.

If the optional prefix has a form of `__` then the following ⟨*byte_sequence*⟩ can be empty. EncTEX removes the record corresponding to the ⟨*first_token*⟩ from output side in such situation.

The macro code for clearing the conversion table from previous section clears all records from input side but only the records concerned to the ⟨*first_token*⟩ in "one byte form" from output side. You can remove the record concerned to control sequence from output side only by `\mubyte \foo __\endmubyte`.

3.6. Inserted control sequences

If the ⟨*first_token*⟩ is the control sequence and the ⟨*optional_prefix*⟩ is one token of catcode 6 (usually the `#` character) followed by ⟨*number*⟩ then the ⟨*number*⟩ bytes are kept by input processor (it means they are no converted again) and the declared control sequence is inserted before ⟨*byte_sequence*⟩. The example:

```
\def\abc{ABC}
\mubyte X BC\endmubyte \mubytein=1
\mubyte \foo #3 \abc\endmubyte  ABC is converted to \foo ABC
\mubyte \foo #1 \abc\endmubyte  ABC is converted to \foo AX
```

The ⟨*number*⟩ has the same syntax as ⟨*number*⟩ from TEXbook. It means that "one optional space" can work as a separator of digit(s). See the previous example.

If ⟨*number*⟩ is zero then the control sequence is inserted and the whole ⟨*byte_sequence*⟩ is unchanged. This has the same effect as if the ⟨*number*⟩ equals to the length of the ⟨*byte_sequence*⟩.

The ⟨*number*⟩ is accepted only in the range 0 to 50. The negative ⟨*numbers*⟩ are silently interpreted as zero and the numbers greater than 50 mean that the rest of the converted line will be unchanged by input processor.

More practical example follows. Note, that the ⟨*number*⟩ is greater than the length of the ⟨*byte_sequence*⟩ here.

```
\mubyte \warntwobytes  #2^^c3\endmubyte
\mubyte \warntwobytes  #2^^c4\endmubyte
\mubyte \warntwobytes  #2^^c5\endmubyte
% atd...
\def\warntwobytes #1#2{\message{WARNING: the UTF8 code:
   \noconvert#1\noconvert#2 is not defined i my macros.}}
```

The new primitive `\noconvert` is used in this example (see chapter 5). The similar code is used in the file `utf8unkn.tex`.

3.7. The virtual mark of line begin

If `\mubytein>0` and if the first byte in ⟨*byte_sequence*⟩ is equal to `\endlinechar` (it means ⟨*byte_sequence*⟩ has a format ⟨*endlinechar*⟩⟨*rest*⟩) then input processor checks the matching of the ⟨*rest*⟩ with the begin of every line. If it matches then the given conversion is done. The example:

```
\bgroup \uccode`X=\endlinechar \uppercase{\gdef\echar{X}}\egroup
\mubyte \fooB \echar ABC\endmubyte % ABC matches at begin of line
\mubyte \fooE ABC\echar \endmubyte % ABC matches at end of line
\mubyte \fooW \spce\space ABC\space \endmubyte
     % ABC matches as a word with spaces before and after
\mubyte \foo #\echar XYZ\endmubyte %
     % if XYZ is at begin of line the \foo is inserted before them
```

3.8. The suppression of the expansion in write parameters

If you need to convert the control sequences back to its ⟨*byte_sequences*⟩ then the expansion of such control sequences is not welcome. You can suppress the expansion by `\let\macro=\relax` before `\write` starts the expansion of its parameter. But `\write` works asynchronously in most situations and you can manipulate

with hundreds or thousands control sequences declared as UTF-8 codes. The encTEX serves a simple tool to solve this problem: If `\mubyteout>=3` then encTEX gives the `\relax` meaning to each control sequence declared in output side of the conversion table before the `\write` starts its expansion and it returns back these control sequences to their original meaning immediately after `\write` finish its work. Example:

```
\mubyte \foo ABC\endmubyte  \def\foo{macro body}
\mubyteout=2
\immediate\write16{testwrite: \foo}  % prints "testwrite: macro body"
\immediate\write16{testwrite: \noexpand\foo} % prints "testwrite: ABC"
\mubyteout=3
\immediate\write16{testwrite: \foo}  % prints "testwrite: ABC"
\message{testmessage: \foo}          % prints "testmessage: macro body"
\message{testmessage: \noexpand\foo} % prints "testmessage: \foo"
\edef\a{testedef: \foo}              % expands to macro body
\foo                                 % expands to macro body
\immediate\write16{\meaning\foo}     % prints "\relax"
\message{\meaning\foo}               % prints "macro:->macro body"
```

Note the difference between `\message` and `\immediate\write16`. The control sequences in `\message` parameter are always expanded and never converted to the ⟨*byte_sequence*⟩.

You can set the "noexpand" flag (for `\write` parameters only) to any ⟨*control_sequence*⟩ and you need not declare the ⟨*byte_sequence*⟩ for it. Write `\mubyte` ⟨*control˙sequence*⟩ `\relax \endmubyte` for this purposes. This has the same effect as `\mubyte` ⟨*control_sequence*⟩ `__\string` ⟨*control_sequence*⟩`\space\endmubyte`, but this second solution is more memory consuming because TEX has to store the ⟨*byte_sequence*⟩ as a string to the pool.

You can write your own macros which expand to one code in normal situation and to different code in write parameters. The declaration of `\writeparameter` control sequence is recommended:

```
\mubyte \writeparameter \relax \endmubyte \def\writeparameter{}
\def\mymacro{\ifx\writeparameter\relax THIS CODE IS USED IN WRITE.
             \else THIS CODE IS USED IN NORMAL EXPANSION.\fi}
```

3.9. The asynchronous write command and the mubyteout value

If you don't use `\immediate` then the `\write` command first gets its parameter but it expands and prints this parameter at another time. The `\write` command stores the actual value of the `\mubyteout` register when it gets its parameter. This value is used late when parameter is expanded and written to the file.

This feature gives the possibility to write to more files, first (for table of contents, for example) is written with conversion to UTF-8 and another files are written without this conversion, because (for example) this file is an input for a program which cannot read the UTF-8 encoding. You can try:

```
\newwrite\tocfile \newwrite\indexfile
\immediate\openout\tocfile=\jobname.toc
\immediate\openout\indexfile=\jobname.idx
\mubyteout=3
\write\tocfile{this parameter will be converted to UTF-8}
{\mubyteout=0 \write\indexfile{this parameter stay unchanged}}
\write\tocfile{this parameter will be converted to UTF-8}
\end % now, all three writes are actually done
```

3.10. Summary of the mubyteout values

Apart from the values 0, 1, 2 and 3, you can set the `\mubyteout` register to the value −1 or −2. The summary table of meanings of these values follows:

\mubyteout	⟨*byte*⟩->⟨*byte_sequence*⟩	⟨*cs_name*⟩->⟨*byte_sequence*⟩	noexpanding
0	off	off	off
1	on	off	off
2	on	on	off
3	on	on	on
-1	on	off	on
-2	off	off	on

If the ⟨*byte*⟩->⟨*byte_sequence*⟩ conversion is on then all texts written to the `\write` files, log file and to the terminal are converted. On the other hand, the ⟨*cs_name*⟩->⟨*byte_sequence*⟩ conversion and the noexpanding are related only to the `\write` arguments (and the `\special` arguments, see the following chapter).

4. The arguments of the special primitive

The plain texts in non-english languages can occur in `\special` arguments. The PDF-outlines are a good example of this situation. May be, you need to save these arguments in UTF-8 encoding. The encTEX gives the possibility to do it.

The argument of `\special` is processed by the value of the integer primitive register `\specialout`. This register is introduced by encTEX and its default value is zero.

- `\specialout=0` – no conversion, the same as in the original TEX.
- `\specialout=1` – only the xchr conversion.
- `\specialout=2` – only the `\mubyteout` conversion.
- `\specialout=3` – the `\mubyteout` conversion followed by the xchr conversion.

The `\special` primitive expands its argument immediately. If `\specialout` is 2 or 3 then the expansion is done by `\mubyteout` value in the same manner as during the `\write` expansion. Moreover, `\special` saves the current values of `\specialout` and `\mubyteout` registers to its memory and use them at the time of the output to the `dvi` file.

5. The noconvert primitive

The `\noconvert` primitive is introduced by encTEX. This primitive suppress the possible conversion of the following character or the control sequence. More exactly: the `\noconvert` is non expandable primitive and does nothing in typesetting output (the same as `\relax`). If this primitive is used in `\message` or `\errmessage` argument then the control sequence of this primitive is never printed and the following character is not converted to ⟨*byte_sequence*⟩ even if the `\mubytelog` is positive and the character is recorded in output side of conversion table.

The primitive `\noconvert` does the same in `\write` and `\special` parameters. Moreover, if a control sequence follows then this control sequence is normally printed even if the `\mubyteout` is positive and this control sequence is recorded in output side of conversion table.

The `\noconvert\noconvert` yields to one `\noconvert` in the output.

The `\noconvert` primitive is normally printed to the log and to the terminal in the another situation than `\message` and `\errmessage` parameters. For example when `\tracing...` is used.

6. Summary of the encTEX's primitives

- `\mubyte` — new record to the conversion table, see chapter 3.
- `\endmubyte` — the `\mubyte` separator.
- `\mubytein` — integer register. 0: multi-byte input conversion suppressed, 1 and more: multi-byte input conversion activated.

- `\mubyteout` — integer register, the level of the output conversion in `\write` and `\special` parameters, see 3.10.
- `\mubytelog` — integer register, 0: multi-byte output conversion to log and terminal suppressed, 1 and more: multi-byte output conversion activated.
- `\specialout` — integer register, the mode of the `\special` parameter conversion, see chapter 4.
- `\noconvert` — similar as `\noexpand`, but for conversion process. See chapter 5.
- `\xordcode` — access to xord vector, see 2.1.
- `\xchrcode` — access to xchr vector, see 2.1.
- `\xprncode` — access to "printability" vector, see 2.2.

The summary of ⟨*optional_prefixes*⟩ in `\mubyte` primitive follows. The character "`#`" means the token of category 6 and "`_`" means the token of category 7 here.

- no prefix — records to the input and output side of the conversion table.
- `_` — records only to the input side of the conversion table.
- `__` — records only to the output side of the conversion table.
- `#`⟨*number*⟩ — records to the input side of the conversion table, the control sequence will be inserted and ⟨*number*⟩ bytes will be kept without conversion again.
- `\relax` — the control sequence will be not expand in `\write` parameters.

More prefixes may be implemented in next versions of encTEX. The prefix has its first character different from catcode 10, 11 or 12. This rule will be kept in next versions thus it is sufficient to use the first character of ⟨*byte_sequence*⟩ with this category if no prefix is needed.

7. The macro files

The encTEX package includes some encoding tables inputted by `\input` during format generation. These tables support encodings widely used in Czech texts. The more information about these macro files are in comments of these files and in the Czech version of the documentation.