EncTEX

The Extension of TEX For Input Reencoding

6. 9. 1997, 3. 1. 2003 Petr Olšák

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 seven 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 reinitialised 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 seven.

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).

I incorporated the possibility to multibyte to one byte or control sequence conversion in december 2002 (the functional version was finisched a few minutes before midnight 12/31/2002). This version is called "Dec. 2002" and it adds four new primitives `\mubyte`, `\endmubyte`, `\mubytein` and `\mubyteout`. They give a possibility to set the conversion from UTF-8 encoded files.

1.3. The conflict with TCX tables

The both encTEX and TCX tables manipulates with the same data (xord and xchr vectors). It means that the conflict may be occur. This was a reason why I took back the encTEX after the TCX tables renovation was released in 1998. But TCX are not able to convert from UTF-8 so I upgraded my encTEX in 2002 and I am propagate it again.

Attention: If you are using encTEX then the TCX tables are ignored even if the `*.tcx` file is given in `-translate-file` option. The encTEX is implemented only in `tex.ch` file but TCX are in additional C sources. We are working on possibility of cooperating of the encTeX and TCX tables.

1.4. The TEX licension

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 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 writting `%&` 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 preporcessor (the eyes in TEXbook terminology). If the character has the code x in your system, the same character has the code y = 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 = xchr[y].

2.2. The new primitives with the acces 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 multibyte 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 four new primitives to manage the conversion: `\mubytein`, `\mubyteout`, `\mubyte`, `\endmubyte`. The `\mubytein` and `\mubytout` 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, the log file and the terminal is activated by the same conversion table. The conversion table is empty by defauult and you can add the new line into this table by the couple of `\mubyte`, `\endmubyte` primitives:

`\mubyte` ⟨*first_token*⟩⟨*one_optional_space*⟩⟨*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 interpredet 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*⟩ is converted to this control sequence at input processor level to the "one token"

form. 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. In this case, the output is not converted back to the ⟨*byte_sequence*⟩ because the control sequence is expanded before the output conversion is active.

The data are stored into conversion table as a global assigment. On the other hand the assigment to `\mubytein` and `\mubyteout` registers are local as usuall.

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 expanable control sequence than `\endmubyte` occurs during this process then the error message is printed:

```
! Missing \endmubyte inserted.
\begtt

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

\begtt
\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   {\protect\ptintegral}
\def\ptintegral {\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. If the `\integral` is undefined control sequence then the error message is printed if you write the "`INT`". The error message has a peculiar form:

```
! Undefined control sequence.
l.13 this is a INT
                  EGRAL.
```

You can type `\show INT` with the following answer:

```
> \integral=undefined.
l.13 \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 sequneces. The `\INT` produces only the control sequnece `\integral`, the

backslash is ignored in this situation. The token processor stays in middle state after `\INT` is scanned, the letter can follow immediatelly.

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

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 converted ⟨*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.

There exists an exeption from output conversion process to the log file and to the terminal. If the complete line from input is reprinted to the log or to the terminal and if `\mubytein` is positive then there is no conversion of such line on input side nor back on output side. This line is reprinted byte per byte in the same form. Only `\xchrcode` followed by `\xordcode` conversion is active. The conversion to `^^aa` form is deactivated in this situation even if `\xprncode` is zero. This exeption concerns to the error messages where the place of the problem is shown on the terminal and log file by printing the actual line and splitting it to two parts. This exeption does not concern to any `\write` or `\message` output. Arguments of these commands are always expanded, translated by `\mubyte` conversion table and translated by `\xchrcode`.

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 precedence has the last occurence of such ⟨*byte_sequences*⟩ in the conversion table and all others are ignored. Example:

```
\mubyte X ABC\endmubyte
{\mubytein=1 the ABC is converted to X}
\mubyte Y ABCDE\endmubyte
\mubyte W ABFG\endmubyte
{\mubytein=1 now, the ABC stay unchanged and ABCDE is converted to Y
             and ABFG is converted to W}
\mubyte Z AB\endmubyte
{\mubytein=1 now, the ABCDE is converted to ZCDE}
```

This convention servers the possibility of removing all lines from conversion table with the same first byte in ⟨*byte_sequence*⟩. It is sufficient to do it if you write the next line to conversion table with the one byte length of ⟨*byte_sequence*⟩ and with the same ⟨*first_token*⟩ as this ⟨*byte_sequence*⟩. For example:

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

removes all lines from conversion table with the ⟨*byte_sequence*⟩ beginning by the letter A. If ⟨*first_token*⟩ is equal to ⟨*byte_sequence*⟩ then `\mubyte` primitive really clears the lines with ⟨*byte_sequence*⟩ beginning by ⟨*first_token*⟩ from the conversion table and frees the main memory of TEX where these data are stored. In all other cases, `\mubyte` primitive only adds the new line into conversion table.

The following code clears the whole 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
```

If the ⟨*first_token*⟩ is the control sequence and the token with catcode 6 (usually the `#` character) is followed instead ⟨*one_optional_space*⟩ then the ⟨*byte_sequence*⟩ is keep by input processor and only the declared control sequence is inserted before ⟨*byte_sequence*⟩. The example of usage:

```
\mubyte \warntwobytes  #^^c3^^80\endmubyte
\mubyte \warntwobytes  #^^c3^^82\endmubyte
\mubyte \warntwobytes  #^^c3^^83\endmubyte
% etc...
\def\warntwobytes #1#2{\bgroup\mubyteout=0
   \message{WARNING: the UTF8 code: #1#2 is not defined i my macros.}
   \egroup}
```

If `\mubytein=1` and if the control sequence is inserted before ⟨*byte_sequence*⟩ then the ⟨*byte_sequence*⟩ stays unchanged absolutly. It means no part of ⟨*byte_sequence*⟩ is converted again. On the other hand, if `\mubytein>1` then the parts of ⟨*byte_sequence*⟩ can be converted by other rules given in conversion table. Example:

```
\mubyte \foo #ABC\endmubyte  \mubyte X BC\endmubyte
\mubytein=1  ABC is converted to \foo ABC
\mubytein=2  ABC is converted to \foo AX
```

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 \space\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
```

4. The macro files

The encTEX package includes some encoding tables inputted by `\input` during format generation. These tables support encodings widelly 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.