Providing some UTF-8 support via \texttt{inputenc}

David Carlisle    Frank Mittelbach    Chris Rowley\textsuperscript{*}

v1.3c 2022/06/07 printed December 3, 2022

This file is maintained by the \LaTeX{} Project team. Bug reports can be opened (category latex) at \url{https://latex-project.org/bugs.html}.

\section*{Contents}

\section*{1 Introduction} 2
  \subsection*{1.1 Background and general stuff} 2
  \subsection*{1.2 More specific stuff} 2
  \subsection*{1.3 Notes} 3
  \subsection*{1.4 Basic operation of the code} 3

\section*{2 Coding} 4
  \subsection*{2.1 Housekeeping} 4
  \subsection*{2.2 Parsing UTF-8 input} 4
  \subsection*{2.3 Mapping Unicode codes to \LaTeX{} internal forms} 9
  \subsection*{2.4 Loading Unicode mappings at begin document} 14

\section*{3 Mapping characters — based on font (glyph) encodings} 15
  \subsection*{3.1 About the table itself} 15
  \subsection*{3.2 The mapping table} 16
  \subsection*{3.3 Notes} 28
  \subsection*{3.4 Mappings for OT1 glyphs} 28
  \subsection*{3.5 Mappings for OMS glyphs} 29
  \subsection*{3.6 Mappings for TS1 glyphs} 29
  \subsection*{3.7 Mappings for \texttt{latex.ltx} glyphs} 29
  \subsection*{3.8 Old \texttt{utf8.def} file as a temp fix for \TeX{} and friends} 30

\section*{4 A test document} 35

\textsuperscript{*}Borrowing heavily from tables by Sebastian Rahtz; some table and code cleanup by Javier Bezos
1 Introduction

1.1 Background and general stuff

For many reasons what this package provides is a long way from any type of ‘Unicode compliance’.

In stark contrast to 8-bit character sets, with 16 or more bits it can easily be very inefficient to support the full range.\footnote{In fact, \LaTeX{}’s current 8-bit support does not go so far as to make all 8-bit characters into valid input.} Moreover, useful support of character input by a typesetting system overwhelmingly means finding an acceptable visual representation of a sequence of characters and this, for \LaTeX{}, means having available a suitably encoded 8-bit font.

Unfortunately it is not possible to predict exactly what valid UTF-8 octet sequences will appear in a particular file so it is best to make all the unsupported but valid sequences produce a reasonably clear and noticeable error message.

There are two directions from which to approach the question of what to load. One is to specify the ranges of Unicode characters that will result in some sensible typesetting; this requires the provider to ensure that suitable fonts are loaded and that these input characters generate the correct typesetting via the encodings of those fonts. The other is to inspect the font encodings to be used and use these to define which input Unicode characters should be supported.

For Western European languages, at least, going in either direction leads to many straightforward decisions and a few that are more subjective. In both cases some of the specifications are \TeX{} specific whilst most are independent of the particular typesetting software in use.

As we have argued elsewhere, \LaTeX{} needs to refer to characters via ‘seven-bit-text’ names and, so far, these have been chosen by reference to historical sources such as Plain \TeX{} or Adobe encoding descriptions. It is unclear whether this ad hoc naming structure should simply be extended or whether it would be useful to supplement it with standardised internal Unicode character names such as one or more of the following:\footnote{Burkhard und Holger Mittelbach spielen mit mir! Sie haben etwas hier geschrieben.}

\begin{verbatim}
\ltxutwochar <4 hex digits>
\ltxuchar {<hex digits>}
     B H U R R R
\ltxeightchartwo <2 utf8 octets as 8-bit char tokens>
\ltxeightcharthree <3 utf8 octets ...>
\ltxeightcharfour <4 utf8 octets ...>
\end{verbatim}

1.2 More specific stuff

In addition to setting up the mechanism for reading UTF-8 characters and specifying the \LaTeX{}-level support available, this package contains support for some default historically expected \TeX{}-related characters and some example ‘Unicode definition files’ for standard font encodings.
1.3 Notes
This package does not support Unicode combining characters as TeX is not really
equipped to make this possible.
No attempt is made to be useful beyond Latin, and maybe Cyrillic, for European
languages (as of now).

1.4 Basic operation of the code
The \inputenc package makes the upper 8-bit characters active and assigns to all
of them an error message. It then waits for the input encoding files to change this
set-up. Similarly, whenever \inputencoding is encountered in a document, first
the upper 8-bit characters are set back to produce an error and then the definitions
for the new input encoding are loaded, changing some of the previous settings.

The 8-bit input encodings currently supported by \inputenc all use declarations
such as \DeclareInputText and the like to map an 8-bit number to some \LaTeX
internal form, e.g. to \texttt{"a}.

The situation when supporting UTF-8 as the input encoding is different, how-
ever. Here we only have to set up the actions of those 8-bit numbers that can be
the first octet in a UTF-8 representation of a Unicode character. But we cannot
simply set this to some internal \LaTeX form since the Unicode character consists
of more than one octet; instead we have to define this starting octet to parse the
right number of further octets that together form the UTF-8 representation
of some Unicode character.

Therefore when switching to utf8 within the inputenc framework the charac-
ters with numbers (hex) from "C2 to "DF are defined to parse for a second octet
following, the characters from "E0 to "EF are defined to parse for two more octets
and finally the characters from "F0 to "F4 are defined to parse for three additional
octets. These additional octets are always in the range "80 to "BF.

Thus, when such a character is encountered in the document (so long as expan-
sion is not prohibited) a defined number of additional octets (8-bit characters) are
read and from them a unique control sequence name is immediately constructed.

This control sequence is either defined (good) or undefined (likely); in the latter
case the user gets an error message saying that this UTF-8 sequence (or, better,
Unicode character) is not supported.

If the control sequence is set up to do something useful then it will expand to
a \LaTeX internal form: e.g. for the utf8 sequence of two octets "C3 "A4 we get \texttt{"a}
as the internal form which then, depending on the font encoding, eventually
resolves to the single glyph 'latin-a-umlaut' or to the composite glyph 'latin-a with
an umlaut accent'.

These mappings from (UTF-8 encoded) Unicode characters to \LaTeX internal
forms are made indirectly. The code below provides a declaration
\texttt{\DeclareUnicodeCharacter} which maps Unicode numbers (as hexadecimal) to
\LaTeX internal forms.

This mapping needs to be set up only once so it is done at \texttt{\begin{document}}
by looking at the list of font encodings that are loaded by the document and
providing mappings related to those font encodings whenever these are available.
Thus at most only those Unicode characters that can be represented by the glyphs
available in these encodings will be defined.
Technically this is done by loading one file per encoding, if available, that is supposed to provide the necessary mapping information.

2 Coding

2.1 Housekeeping

The usual introductory bits and pieces:

```latex
\ProvidesFile{utf8.def}
\ProvidesFile{utf8-test.tex}
\ProvidesFile{1cyenc.dfu}
\ProvidesFile{ly1enc.dfu}
\ProvidesFile{omsenc.dfu}
\ProvidesFile{ot1enc.dfu}
\ProvidesFile{ot2enc.dfu}
\ProvidesFile{t1enc.dfu}
\ProvidesFile{t2aenc.dfu}
\ProvidesFile{t2benc.dfu}
\ProvidesFile{t2cenc.dfu}
\ProvidesFile{ts1enc.dfu}
\ProvidesFile{x2enc.dfu}
\ProvidesFile{utf8enc.dfu}
\ProvidesFile{utf8-2018.def}
\ProvidesFile{utf8-2018.def}
\ProvidesFile{utf8-test.tex}
```

This is a temporary fix for the e-p\TeX / e-up\TeX engines that do not yet have a `\ifincsname` primitive. Once this is available the extra file will be dropped.

```latex
\ifx\ifincsname\@undefined % old e-p\TeX or e-up\TeX engines
\input utf8-2018.def
\expandafter\@firstofone
\else
\expandafter\@gobble
\fi
\endinput
```

We restore the `\catcode` of space (which is set to ignore in `\inputenc`) while reading `.def` files. Otherwise we would need to explicitly use `space` all over the place in error and log messages.

```latex
\makeatletter
\catcode\space=12\relax
```

2.2 Parsing UTF-8 input

A UTF-8 char (that is not actually a 7-bit char, i.e. a single octet) is parsed as follows: each starting octet is an active \TeX character token; each of these is defined below to be a macro with one to three arguments nominally (depending on the starting octet). It calls one of `\UTFviii@two@octets`, `\UTFviii@three@octets`, or `\UTFviii@four@octets` which then actually picks up the remaining octets as the argument(s).
• When typesetting we pick up the necessary number of additional octets, check if they form a command that \TeX knows about (via \csname u8:\string #1\string #2...\endcsname) and if so use that for typesetting. \string is needed as the octets may (all?) be active and we want the literal values in the name.

• If the UTF-8 character is going to be part of a label, then it is essentially becoming part of some csname and with the test \ifincsname we can find this out. If so, we render the whole sequence off octets harmless by using \string too when the starting octet executes (\UTF@...@octets@string).

• Another possible case is that \protect has not the meaning of \typeset@protect. In that case we may do a \write or we may do a \protected@edef or ... In all such cases we want to keep the sequence of octets unchanged, but we can’t use \string this time, since at least in the case of \protect@edef the result may later be typeset after all (in fact that is quite likely) and so at that point the starting octet needs to be an active character again (the others could be stringified). So for this case we use \noexpand ((\UTF@...@octets@noexpand).

\UTFviii@two@octets Putting that all together the code for a start octet of a two byte sequence would then look like this:

\begin{verbatim}
26 \long\def\UTFviii@two@octets{\%
27 \ifincsname
28 \expandafter \UTF@two@octets@string
29 \else
30 \ifx \protect\@typeset@protect \else
31 \expandafter\expandafter\expandafter \UTF@two@octets@noexpand
32 \fi
33 \fi
34 \UTFviii@two@octets@combine
35 \}
\end{verbatim}

\ifcsname is tested first because that can be true even if we are otherwise doing typesetting. If this is the case we use \string on the whole octet sequence. \UTF@two@octets@string not only does this but also gets rid of the command \UTFviii@two@octets@combine in the input stream by picking it up as a first argument and dropping it.

If this is not the case and we are doing typesetting (i.e., \protect is \typeset@protect), then we execute \UTFviii@two@octets@combine which picks up all octets and typesets the character (or generates an error if it doesn’t know how to typeset it).

However, if we are not doing typesetting, then we execute the command \UTFviii@two@octets@noexpand which works like \UTF@two@octets@string but uses \noexpand instead of \string. This way the sequence is temporary rendered harmless, e.g., would display as is or stays put inside a \protected@edef. But if the result is later reused the starting octet is still active and so will be able to construct the UTF-8 character again.

\UTFviii@three@octets The definitions for the other starting octets are the same except that they pick up more octets after them.

\begin{verbatim}
36 \long\def\UTFviii@three@octets{\%
\end{verbatim}
These temporarily prevent the active chars from expanding.

And the same with \string for use in \csname constructions.

From the arguments a control sequence with a name of the form u8:#1#2... is constructed where the #1 (i > 1) are the arguments and #1 is the starting octet (as a \TeX active character token). Since some or even all of these characters are active we need to use \string when building the \csname.

The \csname thus constructed can of course be undefined but to avoid producing an unhelpful low-level undefined command error we pass it to \UTFviii@defined which is responsible for producing a more sensible error message (not yet done!!). If, however, it is defined we simply execute the thing (which should then expand to an encoding specific internal \TeX form).

This tests whether its argument is different from \relax: it either calls for a sensible error message (not done), or it gets the \fi out of the way (in case the command has arguments) and executes it.
Test if the sequence is invalid UTF-8 or valid UTF-8 but without a \TeX\ definition.

70 \if\relax\expandafter\UTFviii@checkseq\string#1\relax\relax

The endline character has a special definition within the inputenc package (it is gobbling spaces). For this reason we can’t produce multiline strings without some precaution.

71 \UTFviii@undefined@err{#1}\
72 \else
73 \@latex@error{Invalid UTF-8 byte sequence \expandafter\@gobblefour\string#1}\
74 \UTFviii@invalid@help
75 \fi
76 \else\expandafter
77 \#1\
78 \fi
79 }

\UTFviii@invalid@err
\UTFviii@invalid@help
80 \def\UTFviii@invalid@err#1{\
81 \@latex@error{Invalid UTF-8 byte \UTFviii@hexnumber{#1}}\
82 \UTFviii@invalid@help
83 \def\UTFviii@invalid@help{\
84 The document does not appear to be in UTF-8 encoding.\MessageBreak
85 Try adding \noexpand\UseRawInputEncoding as the first line of the file\MessageBreak
86 or specify an encoding such as \noexpand\usepackage[latin1]{inputenc}\MessageBreak
87 in the document preamble.\MessageBreak
88 Alternatively, save the file in UTF-8 using your editor or another tool}

\UTFviii@undefined@err
89 \def\UTFviii@undefined@err#1{\
90 \@latex@error{Unicode character \UTFviii@splitcsname{#1} not set up for use with \LaTeX}\
91 {You may provide a definition with\MessageBreak
92 \noexpand\DeclareUnicodeCharacter}\
93 }

\UTFviii@checkseq
\UTFviii@check@continue
Check that the csname consists of a valid UTF-8 sequence.

98 \def\UTFviii@checkseq#1:#2#3{\
99 \ifnum'#2<"80 \
100 \@latex@error{Invalid UTF-8 byte \UTFviii@hexnumber{#1}}\
101 \else
102 \ifnum'#2<"C0 \
103 \ UTFviii@splitcsname{#1}\relax\MessageBreak
104 \MessageBreak
105 \noexpand\@expandendafter\expandafter\expandafter\expandafter#3\
106 \expandafter\expandafter\expandafter#3\
107 \expandafter\expandafter\expandafter#3\
108 \fi
109 \fi}
This bit of code derived from `xmltex` defines the active character corresponding to starting octets to call `\UTFviii@two@octets` etc as appropriate. The starting octet itself is passed directly as the first argument, the others are picked up later en route.

The `\UTFviii@loop` loops through the numbers starting at `\count@` and ending at `\@tempcnta – 1`, each time executing the code in `\UTFviii@tmp`.

Store current settings so can restore after the loops without using a group (gh/762).

`\def\reserved@a{\catcode'\noexpand\~\the\catcode'\~\relax \catcode'\noexpand"\the\catcode'"\relax \uccode'\noexpand\~\the\uccode'\~\relax \count@=\the\count@\relax \@tempcnta=\the\@tempcnta\relax \let\noexpand\reserved@a\relax} \catcode'\~13 \catcode'"12 \def\UTFviii@loop{% \uccode'\~\count@ \uppercase\expandafter{\UTFviii@tmp}% \advance\count@\@ne \ifnum\count@<\@tempcnta \expandafter\UTFviii@loop \fi} % Handle the single byte control characters. C0 controls are valid UTF-8 but defined to give the “Character not defined” error They may be defined with \DeclareUnicodeCharacter.

`\def\UTFviii@tmp{\protected\edef{\noexpand\UTFviii@undefined@err{:\string}}} % 0 ^^^null % 9 ^^^I tab % 10 ^^^J nl `\UTFviii@loop \count@11 \@tempcnta12 `\UTFviii@loop % 12 ^^^L % 13 ^^^M % 14 \count@14 \@tempcnta32 `\UTFviii@loop

Bytes with leading bits 10 are not valid UTF-8 starting bytes
Setting up 2-byte UTF-8: The starting bytes is passed as an active character so that it can be reprocessed later!

Setting up 3-byte UTF-8:

Setting up 4-byte UTF-8:

Bytes above F4 are not valid UTF-8 starting bytes as they would encode numbers beyond the Unicode range

Restore values after the loops.

For this case we must disable the warning generated by inputenc if it doesn’t see any new \DeclareInputText commands.

\@inpenc@test

If this file (utf8.def) is not being read while setting up inputenc, i.e. in the preamble, but when \inputencoding is called somewhere within the document, we do not need to input the specific Unicode mappings again. We therefore stop reading the file at this point.

The \fi must be on the same line as \endinput or else it will never be seen!

2.3 Mapping Unicode codes to \LaTeX{} internal forms

\DeclareUnicodeCharacter

The \DeclareUnicodeCharacter declaration defines a mapping from a Unicode character code point to a \LaTeX{} internal form. The first argument is the Unicode number as hexadecimal digits and the second is the actual \LaTeX{} internal form.

We start by making sure that some characters have the right \catcode when they are used in the definitions below.
Next we do the parsing of the number stored in `\count0` and assign the result to `\UTFviii@tmp`. Actually all this could be done in-line, the macro `\parse@XML@charref` is only there to extend this code to parsing Unicode numbers in other contexts one day (perhaps).

Here is an example of what is happening, for the pair "C2 "A3 (which is the utf8 representation for the character £). After `\parse@XML@charref` we have, stored in `\UTFviii@tmp`, a single command with two character tokens as arguments:

\[t_{C2} \text{ and } t_{A3} \text{ are the characters corresponding to these two octets}\]

`\UTFviii@two@octets t_{C2}A3`

what we actually need to produce is a definition of the form

\[\def\u8:{t_{C2}}{t_{A3}}{\text{ELTEX internal form}}.\]

So here we temporarily redefine the prefix commands `\UTFviii@two@octets`, etc. to generate the csname that we wish to define, the `\strings` are added in case these tokens are still active.

Now we simply:-) need to use the right number of `\expandafters` to finally construct the definition: expanding `\UTFviii@tmp` once to get its contents, a second time to replace the prefix command by its `\csname` expansion, and a third time to turn the expansion into a csname after which the `\gdef` finally gets applied. We add an irrelevant `\IeC` and braces around the definition, in order to avoid any space after the command being gobbled up when the text is written out to an auxiliary file (see `inputenc` for further details)

This macro parses a Unicode number (decimal) and returns its UTF-8 representation as a sequence of non-active TeX character tokens. In the original code it
had two arguments delimited by ; here, however, we supply the Unicode number implicitly.

\begin{verbatim}
\newcommand{\parse@XML@charref}{\ifx
\uccode'
\else
\uccode'
\fi
% We need to keep a few things local, mainly the \uccode's that are set up below. However, the group originally used here is actually unnecessary since we call this macro only within another group; but it will be important to restore the group if this macro gets used for other purposes.
\begingroup
% The original code from xmltex supported the convention that a Unicode slot
number could be given either as a decimal or as a hexadecimal (by starting with x). We do not do this so this code is also removed. This could be reactivated if one wants to support document commands that accept Unicode numbers (but then the first case needs to be changed from an error message back to something more useful again).
\uppercase{\count@\if x\noexpand#1"\else#1\fi#2}\relax
\endgroup
\endverbatim

As \count@ already contains the right value we make \parse@XML@charref work without arguments. In the case single byte UTF-8 sequences, only allow definition if the character os already active. The definition of \UTFviii@tmp looks slightly strange but is designed for the sequence of \expandafter in \DeclareUnicodeCharacter.

\begin{verbatim}
\uppercase{\count@\if x\noexpand#1"\else#1\fi#2}\relax
\endverbatim

The code below is derived from xmltex and generates the UTF-8 byte sequence for the number in \count@.

The reverse operation (just used in error messages) has now been added as \decode@UTFviii.

\begin{verbatim}
\else\ifnum\count@<\texttt{80}\relax
\parse@UTFviii@a,\par
\parse@UTFviii@b C\UTFviii@two@octets.,\par
\else\ifnum\count@<\texttt{1000}\relax
\parse@UTFviii@a;\par
\parse@UTFviii@a,\par
\parse@UTFviii@b E\UTFviii@three@octets.{,;}\par
\else
\texttt{\UTFviii@hexnumber\count@ space too large for Unicode}\par
{Values between 0 and \texttt{10FFFF} are permitted}\par
\fi
\fi
\end{verbatim}

Test added here for out of range values, the 4-octet definitions are still set up so that \texttt{\DeclareUnicodeCharacter} does something sensible if the user scrolls past this error.
In the reverse direction, take a sequence of octets (bytes) representing a character in UTF-8 and construct the Unicode number. The sequence is terminated by `. In this version, if the sequence is not valid UTF-8 you probably get a low level arithmetic error from \numexpr or stray characters at the end. Getting a better error message would be somewhat expensive. As the main use is for reporting characters in messages, this is done just using expansion, so \numexpr is used, A stub returning 0 is defined if \numexpr is not available.

If the input is malformed UTF-8 there may not be enough closing ) so add 5 so there are always some remaining then cleanup and remove any remaining ones at the end. This avoids \numexpr parse errors while outputting a package error.
\UTFviii@hexnumber
\expandafter\dec@de@UTFviii
\fi
\fi
\expandafter\dec@de@UTFviii
\fi
\expandafter\dec@de@UTFviii
\fi\fi

\UTFviii@hexnumber
\def\UTFviii@hexnumber#1{\ifnum#1>15\expandafter\UTFviii@hexnumber\expandafter{\the\numexpr(#1-8)/16\relax}\fi\UTFviii@hexdigit{\numexpr#1\ifnum#1>0-((#1-8)/16)*16\fi\relax}}

Almost but not quite \hexnumber@.

\UTFviii@splitcsname
\UTFviii@hexcodepoint
\def\UTFviii@splitcsname#1:#2\relax{\expandafter\UTFviii@hexcodepoint\expandafter{\the\numexpr\decode@UTFviii#2\relax}}

Need to pre-expand the argument to ensure cleanup in case of mal-formed UTF-8.
These are preamble only as long as we don’t support Unicode charrefs in documents.

2.4 Loading Unicode mappings at begin document

The original plan was to set up the UTF-8 support at \begin{document}; but then any text characters used in the preamble (as people do even though advised against it) would fail in one way or the other. So the implementation was changed and the Unicode definition files for already defined encodings are loaded here.

We loop through all defined font encodings (stored in \cdp@list) and for each load a file nameenc.dfu if it exist. That file is then supposed to contain \DeclareUnicodeCharacter declarations.

However, we don’t know if there are font encodings still to be loaded (either with fontenc or directly with \input by some package). Font encoding files are loaded only if the corresponding encoding has not been loaded yet, and they always begin with \DeclareFontEncoding. We now redefine the internal kernel version of the latter to load the Unicode file if available.
Mapping characters — based on font (glyph) encodings

This section is a first attempt to provide Unicode definitions for characters whose standard glyphs are currently provided by the standard \LaTeX{} font-encodings T1, OT1, etc. They are by no means completed and need checking.

For example, one should check the already existing input encodings for glyphs that may in fact be available and required, e.g. latin4 has a number of glyphs with the \textasciitilde accent. Since the T1 encoding does not provide such glyphs, these characters are not listed below (yet).

The list below was generated by looking at the current \LaTeX{} font encoding files, e.g., t1enc.def and using the work by Sebastian Rahtz (in ucharacters.sty) with a few modifications. In combinations such as \textasciitilde\i the preferred form is that and not \textasciitilde\i.

This list has been built from several sources, obviously including the Unicode Standard itself. These sources include Passive \TeX{} by Sebastian Rahtz, the \texttt{unicode} package by Dominique P. G. Unruh (mainly for Latin encodings) and \texttt{text4ht} by Eitan Gurari (for Cyrillic ones).

Note that it strictly follows the Mittelbach principles for input character encodings: thus it offers no support for using utf8 representations of math symbols such as $\times$ or $\div$ (in math mode).

3.1 About the table itself

In addition to generating individual files, the table below is, at present, a one-one (we think) partial relationship between the (ill-defined) set of LICRs and the Unicode slots "0080 to "FFFF. At present these entries are used only to define a collection of partial mappings from Unicode slots to LICRs; each of these mappings becomes full if we add an exception value (‘not defined’) to the set of LICRs.

It is probably not essential for the relationship in the full table to be one-one; this raises questions such as: the exact role of LICRs; the formal relationships on the set of LICRs; the (non-mathematical) relationship between LICRs and Unicode (which has its own somewhat fuzzy equivalences); and ultimately what a character is and what a character representation and/or name is.

It is unclear the extent to which entries in this table should resemble the closely related ones in the 8-bit inputenc files. The Unicode standard claims that the
first 256 slots ‘are’ ASCII and Latin-1.

Of course, \TeX{} itself typically does not treat even many perfectly ‘normal text’ 7-bit slots as text characters, so it is unclear whether \TeX{} should even attempt to deal in any consistent way with those Unicode slots that are not definitive text characters.

### 3.2 The mapping table

Note that the first argument must be a hex-digit number greater than 00BF and at most 0FFF.

There are few notes about inconsistencies etc at the end of the table.
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
\[
\text{all}, t_1, \text{ly1}\]
The Cyrillic code points have been recently checked (2007) and extended and corrected by Matthias Noe (a9931078@unet.univie.ac.at) — thanks.
The next two declarations are questionable, the encoding definition should probably contain \CYROTLD and \cyrrotld. Or alternatively, if the characters in the X2 encodings are really meant to represent the historical characters in Ux0472 and Ux0473 (they look like them) then they would need to change instead.

However, their looks are probably a font designer's decision and the next two mappings are wrong or rather the names in OT2 should change for consistency.

On the other hand the names \CYROTLD are somewhat questionable as the Unicode standard only describes “Cyrillic barred O” while TLD refers to a tilde (which is more less what the “Cyrillic FITA looks according to the Unicode book).

While the double grave accent seems to exist in X2, T2A, T2B and T2C encoding, the letter izhitsa exists only in X2 and OT2. Therefore, izhitsa with double grave seems to be possible only using X2.
The character \CYRPalochka is not defined by OT2 and LCY. However it is looking identical to \CYRRII and the Unicode standard explicitly refers to that (and to Latin 1). So perhaps those encodings could get an alias? On the other hand, why are there two distinct slots in the T2 encodings even though they are so pressed for space? Perhaps they don’t always look alike.

According to the Unicode standard Ux04C5 should be an L with “tail” not with descender (which also exists as Ux04A2) but it looks as if the char names do not make this distinction). Should they?
According to the Unicode standard U+04CD should be an M with “tail” not with descender. However this time there is no M with descender in the Unicode standard.
U+2012 should be the width of a digit, endash is OK in many fonts including cm.

U+2015 is Horizontal bar
When doing cut-and-paste from other documents ligatures might show up as individual characters which is fine too. If supported by the font (which is normally the case) they are then reconstructed as ligatures so they come out as desired. Otherwise they will come out as individual characters which is fine too.
3.3 Notes

The following inputs are inconsistent with the 8-bit inputenc files since they will always only produce the ‘text character’. This is an area where inputenc is notoriously confused.

\%<all,ts1,t1,ot1,ly1>\DeclareUnicodeCharacter{00A3}{\textsterling}
\%<all,x2,ts1,t2c,t2b,t2a,oms,ly1>\DeclareUnicodeCharacter{00A7}{\textsection}
\%<all,ts1>\DeclareUnicodeCharacter{00A9}{\textcopyright}
\%<all,ts1,oms,ly1>\DeclareUnicodeCharacter{00B1}{\textpm}
\%<all,ts1,oms,ly1>\DeclareUnicodeCharacter{00B6}{\textparagraph}
\%<all,ly1,utf8>\DeclareUnicodeCharacter{2026}{\textellipsis}

The following definitions are in an encoding file but have no direct equivalent in Unicode, or they simply do not make sense in that context (or we have not yet found anything or ...:-). For example, the non-combining accent characters are certainly available somewhere but these are not equivalent to a \TeX accent command.

\\DeclareTextSymbol{\j}{OT1}{17}
\\DeclareTextSymbol{\SS}{T1}{223}
\\DeclareTextSymbol{\textcompwordmark}{T1}{23}

\\DeclareTextAccent{"}{OT1}{127}
\\DeclareTextAccent{"}{OT1}{19}
\\DeclareTextAccent{\_}{OT1}{95}
\\DeclareTextAccent{\=}{OT1}{22}
\\DeclareTextAccent{\H}{OT1}{125}
\\DeclareTextAccent{\^}{OT1}{94}
\\DeclareTextAccent{\'}{OT1}{18}
\\DeclareTextAccent{\r}{OT1}{23}
\\DeclareTextAccent{\u}{OT1}{21}
\\DeclareTextAccent{\v}{OT1}{20}
\\DeclareTextAccent{\~}{OT1}{126}
\\DeclareTextCommand{\hat}{OT1}{1}
\\DeclareTextCommand{\hat}{OT1}{1}
\\DeclareTextCommand{\hat}{OT1}{1}
\\DeclareTextCommand{\hat}{OT1}{1}

3.4 Mappings for OT1 glyphs

This is even more incomplete as again it covers only the single glyphs from OT1 plus some that have been explicitly defined for this encoding. Everything that is
provided in T1, and that could be provided as composite glyphs via OT1, could
and probably should be set up as well. Which leaves the many things that are not
provided in T1 but can be provided in OT1 (and in T1) by composite glyphs.

Stuff not mapped (note that \j (j) is not equivalent to any Unicode character):

\DeclareTextSymbol{\j}{OT1}{17}
\DeclareTextAccent{\'}{OT1}{19}
\DeclareTextAccent{\.'}{OT1}{95}
\DeclareTextAccent{\=}{OT1}{22}
\DeclareTextAccent{\^}{OT1}{94}
\DeclareTextAccent{\'}{OT1}{18}
\DeclareTextAccent{\^}{OT1}{125}
\DeclareTextAccent{\H}{OT1}{21}
\DeclareTextAccent{\v}{OT1}{20}
\DeclareTextAccent{\r}{OT1}{23}
\DeclareTextCommand{\b}{OT1}[1]
\DeclareTextCommand{\c}{OT1}[1]
\DeclareTextCommand{\d}{OT1}[1]

3.5 Mappings for OMS glyphs

Characters like \textbackslash are not mapped as they are (primarily) only in
the lower 127 and the code here only sets up mappings for UTF-8 characters that
are at least 2 octets long.

\DeclareTextSymbol{\textbackslash}{OMS}{110} % "6E
\DeclareTextSymbol{\textbar}{OMS}{106} % "6A
\DeclareTextSymbol{\textbraceleft}{OMS}{102} % "66
\DeclareTextSymbol{\textbraceright}{OMS}{103} % "67

But the following (and some others) might actually lurk in Unicode somewhere...

\DeclareTextSymbol{\textasteriskcentered}{OMS}{3} % "03
\DeclareTextCommand{\textcircled}{OMS}

3.6 Mappings for TS1 glyphs

Exercise for somebody else.

3.7 Mappings for latex.ltx glyphs

There is also a collection of characters already set up in the kernel, one way or the
other. Since these do not clearly relate to any particular font encoding they are
mapped when the utf8 support is first set up.

Also there are a number of \providecommands in the various input encoding
files which may or may not go into this part.

931 (*utf8)
932 % This space is intentionally empty ...
933 (/utf8)
3.8 Old utf8.def file as a temp fix for p\TeX{} and friends

\ProvidesFile{utf8.def}
\makeatletter
\catcode' \saved@space@catcode
\long\def\UTFviii@two@octets#1\string#2{\expandafter\UTFviii@defined\csname u8:#1\string#2\endcsname}
\long\def\UTFviii@three@octets#1\string#2\string#3{\expandafter\UTFviii@defined\csname u8:#1\string#2\string#3\endcsname}
\long\def\UTFviii@four@octets#1\string#2\string#3\string#4{\expandafter\UTFviii@defined\csname u8:#1\string#2\string#3\string#4\endcsname}
\def\UTFviii@defined#1{\ifx#1\relax\if\relax\expandafter\UTFviii@checkseq\string#1\relax\relax \UTFviii@undefined@err{#1}\else\PackageError{inputenc}{Invalid UTF-8 byte sequence}\UTFviii@invalid@help\fi\else#1\fi}
\def\UTFviii@invalid@err#1{\PackageError{inputenc}{Invalid UTF-8 byte \UTFviii@hexnumber{'#1}}{You may provide a definition with \noexpand\DeclareUnicodeCharacter}}
\def\UTFviii@undefined@err#1{\PackageError{inputenc}{Unicode character \UTFviii@splitcsname not set up for use with \LaTeX}{}{You may provide a definition with \noexpand\DeclareUnicodeCharacter}}
\def\UTFviii@checkseq#1:#2#3{\ifnum'#2<"80 \else\expandafter\expandafter\expandafter\UTFviii@check@continue\expandafter\expandafter\expandafter#3\fi}
\def\UTFviii@check@continue#1{\ifnum'#1\relax\UTFviii@checkseq#1\relax\relax \UTFviii@undefined@err{#1}\else\PackageError{inputenc}{Invalid UTF-8 byte \UTFviii@hexnumber{'#1}}{You may provide a definition with \noexpand\DeclareUnicodeCharacter}\UTFviii@invalid@help\fi}
\def\UTFviii@check@continue#1#2#3#4{\ifnum'#2<"80 \else\expandafter\expandafter\expandafter\UTFviii@check@continue\expandafter\expandafter\expandafter\expandafter#3\fi}
\begin{itemize}
\item \noexpand\UseRawInputEncoding
\item \usepackage[latin1]{inputenc}
\end{itemize}
\@tempcntb\count@
\multiply\count@64
\advance\@tempcnta-\count@
\advance\@tempcnta128
\uccode'#1\@tempcnta
\count@\@tempcntb}
\gdef\parse@UTFviii@b#1#2#3#4{%\advance\count#10\relax
\uccode'#3\count@\uppercase{\gdef\UTFviii@tmp{#2#3#4}}
\ifx\numexpr\@undefined\global\let\UTFviii@hexnumber\@firstofone
\global\UTFviii@hexdigit\hexnumber@
\else\gdef\UTFviii@hexnumber#1{%\ifnum#1>15\expandafter\UTFviii@hexnumber\expandafter{\the\numexpr(#1-8)/16\relax}\fi
\UTFviii@hexdigit{\numexpr#1\ifnum#1>0-((#1-8)/16)*16\fi\relax}
}\fi\fi\fi
\fi
\expandafter\dec@de@UTFviii
\fi}
\expandafter\dec@de@UTFviii
\expandafter\UTFviii@cleanup
\the\numexpr\dec@de@UTFviii#1\relax)))))\@empty}
\gdef\UTFviii@cleanup#1)#2\@empty{#1}
\gdef\dec@de@UTFviii#1{%\ifx\relax#1%\else\ifnum'#1>"EF
\ifnum'#1>"DF
\ifnum'#1>"BF
\ifnum'#1>"7F
*64+('#1-"80)%
+('#1-%}
\fi
\fi
\fi\fi\fi
\expandafter\dec@de@UTFviii
\expandafter\UTFviii@cleanup
\the\numexpr\dec@de@UTFviii#1\relax)))))\@empty}
\gdef\UTFviii@cleanup#1)#2\@empty{#1}
A test document

Here is a very small test document which may or may not survive if the current
document is transferred from one place to the other.

German umlauts in UTF-8: "\c3\a4\c3\b6\c3\bc" %\% äöü

German umlauts in UTF-8 but read by latin1 (and will produce one
error since \verb=\textcurrency= is not provided):

\c3\a4\c3\b6\c3\bc

German umlauts in UTF-8: "\c3\a4\c3\b6\c3\bc"

Some codes that should produce errors as nothing is set up
for them: "\c3\F "\e1\a4\b6

And some that are not legal utf8 sequences: "\c3\X "\e1\XY

\showoutput
\tracingstats=2
\stop

\[/test]