low level TEX

characters

Contents

1 Introduction

This explanation is part of the low level manuals because in practice users will not have to deal with these matters in MkIV and even less in LMTX. You can skip to the last section for commands.

2 History

If we travel back in time to when TEX was written we end up in eight bit character universe. In fact, the first versions assumed seven bits, but for comfortable use with languages other than English that was not sufficient. Support for eight bits permits the usage of so called code pages as supported by operating systems. Although ascii input became kind of the standard soon afterwards, the engine can be set up for different encodings. This is not only true for TEX, but for many of its companions, like MetaFont and therefore MetaPost.[1]

Core components of a TEX engine are hyphenation of words, applying inter-character kerns and build ligatures. In traditional TEX engines those processes are interwoven into the par builder but in LuaTEX these are separate stages. The original approach is the reason that there is a relation between the input encoding and the font encoding: the character in the input is the slot used in a reference to a glyph. When producing the final result (e.g. pdf) there can also be a mapping to an index in a font resource.

```
input A [tex ->] font slot A [backend ->] glyph index A
```

The mapping that TEX does is normally one-to-one but an input character can undergo some transformation. For instance a character beyond ascii 126 can be made active

[1] This remapping to an internal representation (e.g. ebcdic) is not present in LuaTEX where we assume utf8 to be the input encoding. The MetaPost library that comes with LuaTEX still has that code but in LuaMetaTEX it's gone. There one can set up the machinery to be utf8 aware too.

and expand to some character number that then becomes the font slot. So, it is the expansion (or meaning) of a character that end up as numeric reference in the glyph node. Virtual fonts can introduce yet another remapping but that's only visible in the backend.

Actually, in LuaTEX the same happens but in practice there is no need to go active because (at least in ConTEXt) we assume a Unicode path so there the font slot is the Unicode got from the utf8 input.

In the eight bit universe macro packages (have to) provide all kind of means to deal with (in the perspective of English) special characters. For instance, `\"a` would put a diaeresis on top of the a or even better, refer to a character in the encoding that the chosen font provides. Because there are some limitations of what can go in an eight bit font, and because in different countries the used TEX fonts evolved kind of independent, we ended up with quite some different variants of fonts. It was only with the Latin Modern project that this became better. Interesting is that when we consider the fact that such a font has often also hardly used symbols (like registered or copyright) coming up with an encoding vector that covers most (latin based) European languages (scripts) is not impossible[2] Special symbols could simply go into a dedicated font, also because these are always accessed via a macro so who cares about the input. It never happened.

Keep in mind that when utf8 is used with eight bit engines, ConTEXt will convert sequences of characters into a slot in a font (depending on the font encoding used which itself depends on the coverage needed). For this every first (possible) byte of a multibyte utf sequence is an active character, which is no big deal because these are outside the ascii range. Normal ascii characters are single byte utf sequences and fall through without treatment.

Anyway, in ConTEXt MkII we dealt with this by supporting mixed encodings, depending on the (local) language, referencing the relevant font. It permits users to enter the text in their preferred input encoding and also get the words properly hyphenated. But we can leave these MkII details behind.

3 The heritage

In MkIV we got rid of input and font encodings, although one can still load files in a specific code page.[3] We also kept the means to enter special characters, if only because

[2] And indeed in the Latin Modern project we came up with one but it was already to late for it to become popular.

[3] I'm not sure if users ever depend on an input encoding different from utf8.

text editors seldom support(ed) a wide range of visual editing of those. This is why we still have

```
\"u \^a \v{s} \AE \ij \eacute \oslash
```

and many more. The ones with one character names are rather common in the TEX community but it is definitely a weird mix of symbols. The next two are kind of outdated: in these days you delegate that to the font handler, where turning them into 'single' character references depends on what the font offers, how it is set up with respect to (for instance) ligatures, and even might depend on language or script.

The ones with the long names partly are tradition, but as we have a lot of them, in MkII they actually serve a purpose. These verbose names are used in the so called encoding vectors and are part of the utf expansion vectors. They are also used in labels so that we have a good indication if what goes in there: remember that in those times editors often didn't show characters, unless the font for display had them, or the operating system somehow provided them from another font. These verbose names are used for latin, greek and cyrillic and for some other scripts and symbols. They take up quite a bit of hash space and the format file.[4]

4 The LMTX approach

In the process of tagging all (public) macros in LMTX (which happened in 2020-2021) I wondered if we should keep these one character macros, the references to special characters and the verbose ones. When asked on the mailing list it became clear that users still expect the short ones to be present, often just because old bibTEX files are used that might need them. However, in MkIV and LMTX we load bibTEX files in a way that turn these special character references into proper utf8 input so it makes a weak argument. Anyway, although they could go, for now we keep them because users expect them. However, in LMTX the implementation is somewhat different now, a bit more efficient in terms of hash and memory, potentially a bit less efficient in runtime, but no one will notice that.

A new command has been introduced, the very short `\chr`.

```
\chr {a} \chr {a} \chr {a}
\chr {`a} \chr {'a} \chr {"a}
\chr {a acute} \chr {a grave} \chr {a umlaut}
```

[4] In MkII we have an abstract front-end with respect to encodings and also an abstract backend with respect to supported drivers but both approaches no longer make sense today.

```
\chr {aacute}  \chr {agrave}  \chr {aumlaut}
```

In the first line the composed character using two characters, a base and a so called mark. Actually, one doesn't have to use `\chr` in that case because ConTEXt does already collapse characters for you. The second line looks like the shortcuts `\``, `\'` and `\"`. The third and fourth lines could eventually replace the more symbolic long names, if we feel the need. Watch out: in Unicode input the marks come *after*.

à á ä
à á ä
á à a˘mła˘t
á à a˘mła˘t

Currently the repertoire is somewhat limited but it can be easily be extended. It all depends on user needs (doing Greek and Cyrillic for instance). The reason why we actually save code deep down is that the helpers for this have always been there.[5]

The `\"` commands are now just aliases to more verbose and less hackery looking macros:

```
\withgrave         à  \`  à
\withacute         á  \'  á
\withcircumflex    â  \^  â
\withtilde         ã  \~  ã
\withmacron        ā  \=  ā
\withbreve         ĕ  \u  ĕ
\withdotaccent     ċ  \.  .c
\withdiaeresis     ë  \"  ë
\withring          ů  \r  ů
\withhungarumlaut  ű  \H  ű
\withcaron         ĕ  \v  ĕ
\withcedilla       ę  \c  ę
\withogonek        ę  \k  ę
```

Not all fonts have these special characters. Most natural is to have them available as precomposed single glyphs, but it can be that they are just two shapes with the marks anchored to the base. It can even be that the font somehow overlays them, assuming (roughly) equal widths. The `compose` font feature in ConTEXt normally can handle most well.

[5] So if needed I can port this approach back to MkIV, but for now we keep it as is because we then have a reference.

An occasional ugly rendering doesn't matter that much: better have something than nothing. But when it's the main language (script) that needs them you'd better look for a font that handles them. When in doubt, in ConTEXt you can enable checking:

command	**equivalent to**
`\checkmissingcharacters`	`\enabletrackers[fonts.missing]`
`\removemissingcharacters`	`\enabletrackers[fonts.missing=remove]`
`\replacemissingcharacters`	`\enabletrackers[fonts.missing=replace]`
`\handlemissingcharacters`	`\enabletrackers[fonts.missing={decompose,replace}]`

The decompose variant will try to turn a composed character into its components so that at least you get something. If that fails it will inject a replacement symbol that stands out so that you can check it. The console also mentions missing glyphs. You don't need to enable this by default[6] but you might occasionally do it when you use a font for the first time.

In LMTX this mechanism has been upgraded so that replacements follow the shape and are actually real characters. The decomposition has not yet been ported back to MkIV.

The full list of commands can be queried when a tracing module is loaded:

```
\usemodule[s][characters-combinations]

\showcharactercombinations
```

We get this list:

acute	U+00301	◌́	`\withacute`
breve	U+00306	◌̆	`\withbreve`
caron	U+0030C	◌̌	`\withcaron`
caron below	U+0032C	◌̬	`\withcaronbelow`
cedilla	U+00327	◌̧	`\withcedilla`
circumflex	U+00302	◌̂	`\withcircumflex`
circumflex below	U+0032D	◌̭	`\withcircumflexbelow`
comma below	U+00326	◌̦	`\withcommabelow`
diaeresis	U+00308	◌̈	`\withdiaeresis`
dieresis	U+00308	◌̈	`\withdieresis`
dot	U+00307	◌̇	`\withdot`
dot below	U+00323	◌̣	`\withdotbelow`
double acute	U+0030B	◌̋	`\withdoubleacute`

[6] There is some overhead involved here.

double grave	U+0030F	̏	\withdoublegrave
double vertical line	U+0030E	̎	\withdoubleverticalline
grave	U+00300	̀	\withgrave
hook	U+00309	̉	\withhook
hook below	U+1FA9D		\withhookbelow
hungarumlaut	U+0030B	̋	\withhungarumlaut
inverted breve	U+00311	̑	\withinvertedbreve
line	U+00304	̄	\withline
line below	U+00331	̲	\withlinebelow
macron	U+00304	̄	\withmacron
macron below	U+00331	̲	\withmacronbelow
middle dot	U+000B7	·	\withmiddledot
ogonek	U+00328	̨	\withogonek
overline	U+00305	̅	
ring	U+0030A	̊	\withring
ring below	U+00325	̥	\withringbelow
slash	U+0002F	/	\withslash
stroke	U+0002F	/	\withstroke
tilde	U+00303	̃	\withtilde
tilde below	U+00330	̰	\withtildebelow
vertical line	U+0030D	̍	\withverticalline

Some combinations are special for ConTEXt because Unicode doesn't specify decomposition for all composed characters.

5 spaces

The engine has no real concept of a space. When the input has one it is turned into a glue, likely with some stretch and shrink. When `\nospaces` is set to one, no glue will be inserted. A value of two will inject a zero width glue. When set to three a glyph will be inserted with the character code set by `\spacechar`.

```
\nospaces\plusthree
\spacechar\underscoreasciicode
\hccode\underscoreasciicode\underscoreasciicode
Where are the spaces?
```

The `\hccode` tells the machinery that the underscore is a valid word separator (think compound words).

Where_are_the_spaces?_

6 Casing

We started by mentioning that TEX was primarily made and used for typesetting English although quickly other Latin scripts were supported. As long as a language uses a script that fits nicely in a few hundred characters we're okay. Of course mixing scripts and languages is more demanding, but Latin, Cyrillic and Greek were no problem.

A characteristic of western languages and these scripts is that there is a mix of upper and lowercase characters. Now, in most cases the input has the right usage of casing so there is little for TEX to worry about. However, in the traditional approach every character has a so called `\lccode` and `\uccode`. These are used to map from upper- to lowercase or reverse and the lowercase code, when set, is also a signal that controls hyphenation. In LuaTEX and LuaMetaTEX that role is taken over by the `\hccode`.

Good old TEX has two primitives that deal with case swapping.

```
\lowercase{Don't get fooled!}
\uppercase{Don't get fooled!}
```

This does what one expects:

don't get fooled!
DON'T GET FOOLED!

So, what do you expect here?

```
\edef\fooled{\lowercase{Don't get fooled!}}\meaning\fooled
\edef\fooled{\uppercase{Don't get fooled!}}\meaning\fooled
```

macro:\lowercase {Don't get fooled!}
macro:\uppercase {Don't get fooled!}

And here?

```
\lowercase{\edef\fooled{Don't get fooled!}}\meaning\fooled
\uppercase{\edef\fooled{Don't get fooled!}}\meaning\fooled
```

macro:don't get fooled!
macro:DON'T GET FOOLED!

These primitives are non-expandable but act on a token list when interpretation takes place.

```
\begingroup
```

```
    \lowercase{\edefcsname fooled\endcsname{Guess}}
    \meaningfull\fooled
    \meaningfull\FOOLED
\endgroup
\begingroup
    \uppercase{\edefcsname fooled\endcsname{Guess}}
    \meaningfull\fooled
    \meaningfull\FOOLED
\endgroup
```

```
macro:guess
undefined
undefined
macro:GUESS
```

It will be clear that using these primitives in a situation where you want expansion you really need to make sure that in getting there no side effects take place. Instead of these primitives, that we keep around for compatibility reasons, you can better use the converters that ConTEXt provides. Even these are probably of little used when typesetting is your main focus, because when case mapping of various kind is needed there are high level mechanism for just that.

```
\edef\fooled{\utflower{Don't get fooled!}}\meaning\fooled
\edef\fooled{\utfupper{Don't get fooled!}}\meaning\fooled
```

```
macro:don't get fooled!
macro:DON'T GET FOOLED!
```

It will be clear that in an Unicode environment there is more than an English alphabet involved. The primitive case mapping as well as the ConTEXt variants use of course Unicode specified mapping but the last two macros are just wrappers around a lot more character related properties.

In MkII we supported various encodings and each came with a case mapping. We gradually moved to utf8 and for instance hyphenation patterns came in that format so that they could easily be shared.[7]

[7] This was reason for Arthur and Mojca to start the hyphenation project that resulted in utf patterns being that standard now. It's a nice example of prototyping and exploring in ConTEXt and applying it to a wider audience.

When we started in 2005 with a follow up a major change in approach to dealing with characters between MkII and MkIV was that we could get rid of, if really needed, remapping at the TEX end. We delegate much to Lua which also has the advantage that we operate on the text as intended, possibly resulting from generated input. This means that right from the start we had support for Unicode, although of course that standard also evolved over time. Some was driven by specific user need: for the oriental TEX project we needed bidirectional interpretation, for some scripts we needed (de)composition, we needed sorting, tracing, special purpose conversions, etc.[8]

Most of this is rather stable and as it has been present from the start and it hasn't been touched much, although we do keep up with Unicode. But one has to keep in mind that much is about control and user input and demand so we can deviate from what is considered 'standard': we target typesetting, not programming and text manipulation! We keep most (not all) information in the `char-def.lua` file and actually although it gets adapted to Unicode updates (for which of course we use a script) it hasn't changed much, apart from an update with respect to math classes and such when we upgraded the math engine. It can be an interesting read.

That said: if users want some more explanations or low level examples, just let us know and we will add them here (or to the low level manual that makes sense).

7 Specials

Traditionally TEX has a few specially interpreted characters. Most noticeably the backslash starts a control sequence. Any programming languages has some escape and this the one that TEX uses. Of course any character can be an escape character.

The dollar is well known for bounding inline math. A pair of dollars wraps display math. In ConTEXt you can use the dollars but double ones act as single ones and trigger inline display style math. In practice one will use `\im{...}` and `\dm{...}` and for display math `\startformula` and `\stopformula`.

The ampersand is, unlike in other macro packages, just that and not a table cell separator. The circumflex character is also just that. However, in math upto four are used for super scripts (pre, post and index) while the underscore(s) handle the subscripts. The single quote is in math signaling a prime (multiple primes are supported).

The end of line character(s) are in fact spaces unless configured to be obeyed, what can be the case when you use `\startlines` and `\stoplines` or display verbatim.

[8] If manipulation of character properties in the TEX language is your hobby, maybe MkII is a better choice as it pretty much belongs to that period.

The tilde is normally active and equivalent to a `\nbsp` character. In ConTEXt it as an `\amcode` set so that it will never expand in for instance an `\edef` like situation. The hash mark is used to signal a parameter in a macro body and is followed by a (hexadecimal) digit. They get duplicated in nested macro definitions. The percent sign starts comment that then runs till the end of the line.

Hyphens are also special, as they can become a discretionary (hyphenation point), or when entered double or triple can become an en- of emdash, either or not with similar hyphenation behavior (think compound words).

Quotation and explanation marks and some punctuation like periods, commas, colons and semicolons can get different spacing depending on the `\sfcode`. Here 'french spacing' is the buzzword.

In ConTEXt the vertical bar has always been an active character used for compound word marks and other word bounding characters. They come in pairs where the second one is a delimiter of a possible argument. In math it behaves as normal bar, although bars in math are sort special too. In fact, all characters in math get a treatment depending on what they represent. More about that can be read in the math manual.

The various spacing characters in Unicode are interpreted but normally can't be seen in an editor. Instead one can use commands. *todo: list them here*

Of course all can be changed to ones need but be careful and aware of interferences, especially when loading code that assumes otherwise.

7 Colofon

Author	Hans Hagen
ConTEXt	2025.08.16 17:36
LuaMetaTEX	2.11.07 | 20250813
Support	www.pragma-ade.com
	contextgarden.net
	ntg-context@ntg.nl