Writing ETX format font encoding specifications

Lars Hellström

Abstract

This paper explains how one writes formal specifications of font encodings for LaTeX and suggests a ratification procedure for such specifications.

1 Introduction

One of the many difficult problems any creator of a new typesetting system encounters is that of *font construction* — to create fonts that provide all the information that the typesetting system needs to do its job. From the early history of TeX, we learn that this problem is so significant that it motivated the creation of TeX's companion and equal METAFONT, whose implementation proved to be an even greater scientific challenge than TeX was. It is also a tell-tale sign that the `fonts` subtree of the teTeX distribution is about three times as large as the `tex` subtree: fonts are important, and not at all trivial to generate.

The most respected and celebrated part of font construction is *font design* — the creation from practically nothing of new letter (and symbol) shapes, in pursuit of an artistic vision — but it is also something very few people have the time and skill to carry through. More common is the task of *font installation*, where one has to solve the very concrete problem of how to set up an existing font so that it can be used with (LA)TeX. The subproblems in this domain range from the very technical — how to make different pieces of software "talk" to each other, for example making information in file format A available to program B — to the almost artistic — finding values for glyph metrics and kerns that will make them look good in text — but these extremes tend to be clearly defined even if solving them can be hard, so they are not what will be considered here. Rather, this paper is about a class of more subtle problems that have to do with how a font is organised.

The technical name for such a "font organisation" is a *font encoding*. In some contexts, font encodings are assumed to be mere mappings from a set of "slots" to a set of glyph identifiers, but in TeX the concept entails much more; the various aspects are detailed in subsequent sections. For the moment, it is sufficient to observe that the role that a font encoding plays in a typesetting system is that of a standard: it describes what an author can expect from a font, so that a document or macro package can be written that works with a large class of fonts rather than just for one font family. The world of (LA)TeX would be very different if papers published in journal X that is printed in commercial font Y could not use essentially the same sources as the author prepared for typesetting in the free font Z. Fine-tuning of a document (overfull lines, bad page breaks, etc.) depends on the exact font used, but it is a great convenience that one can typeset a well-coded body of text under a rather wide range of layout parameter values (of which the main font family is one) and still expect the result to look decent, often even good. Had font encodings not been standardised, the results might not even have been readable.

When font encodings are viewed as standards, the historical states of most (LA)TeX font encodings becomes rather embarrassing, as they lack something as fundamental as proper specifications! The typical origin of a font encoding has been that someone creates a font that behaves noticeably differently from other fonts, macro packages are then created to support this new font, and in time other people create other fonts that work with the same macros. At the end of this story the new encoding exists, but it is not clear who created it, and there is probably no document that describes all aspects of the encoding. Later contributors have typically had to rely on a combination of imitation of previous works, folklore, and reverse engineering of existing software when trying to figure out what they need to provide, but the results are not always verifiable. Furthermore the errors in this area are usually silent — the classical error being that a '\$' was substituted for a '£' (or vice versa) — which means they can only be discovered through careful proofreading, and then only if a document even exists which exercises all aspects of the font encoding. Since font encodings interact with hyphenation, exhaustive font verification through proofreading is probably beyond the capabilities of any living TeXpert on purely linguistic grounds.

Proper specification of font encodings makes the task of font installation — and to some extent also the task of font design, as it too is subject to the technicalities of font encodings — much simpler, as there is then a document that authoritatively gives all details of a font encoding. This paper even goes one step further, and proposes (i) a standard format for formal specifications of (LA)TeX font encodings and (ii) a process through which such specifications can be ratified as *the* specification of a particular encoding. My hope is that future (LA)TeX font encodings will have proper specifications from the start, as this will greatly simplify making more fonts available in these encodings, and perhaps also make font designers aware of the subtler points of (LA)TEX font design, as many details have been poorly documented.

The proposed file format for encoding specifications is a development of the fontinst [6] ETX format. One reason for this choice was that it is an established format; many of those who are making fonts already use it, even if for a slightly different purpose. Another major reason is that an ETX file is both a LATEX document and a processable data file; this is the same kind of bilinguality that has made the `.dtx` format so useful. Finally the ETX format makes it easy to create experimental font installations when a new encoding is being designed; fontinst can directly read the file, but the file can also be automatically converted to a PostScript encoding vector if that approach seems more convenient. On the other hand, there are some features — most notably the prominent role of the glyph names — of the ETX format that would probably had been done differently in a file format that was built from scratch, but this is necessary for several of the advantages listed above.

2 Points to keep in mind

2.1 Characters, glyphs, and slots

One fundamental difference that must be understood is that between characters and glyphs. A *character* is a semantic entity — it carries some meaning, even if you usually have to combine several characters to make up even one word — whereas a *glyph* simply is a piece of graphics. In printed text, glyphs are used to represent characters and the first step of reading is to determine which character(s) a given glyph is representing.[1]

In the output, TEX neither deals with characters nor glyphs, really (although many of its messages speak of characters), but with *slots*, which essentially are numbered positions in a font. To TEX, a slot is simply something which can have certain metric properties (width, height, depth, etc.) but to the driver which actually does the printing the slot also specifies a glyph. The same slot in two different fonts can correspond to two quite different characters.

For completeness it should also be mentioned that the *input* of TEX is a stream of semantic entities and thus TEX is dealing with characters on that side, but the input is not the subject of this paper.

[1] Some PDF viewers also try to accomplish this, but in general they need extra information to do it right. The generic solution provided is to embed a *ToUnicode CMap* — which is precisely a map from slots to characters — in the PDF font object.

2.2 Ligatures

In typography, a *ligature* is a glyph which has been formed by joining glyphs that represent two or more characters; this joining can involve quite a lot of deformation of the original shapes. Examples of ligatures are the 'fi' ligature (from 'f' and 'i'), the 'Æ' ligature (from 'A' and 'E'), and the '*&*' character (from 'E' and 't'), the latter two of which has evolved to become characters of their own. For those ligatures (such as 'fi') that have not evolved to characters, TEX has a mechanism for forming the ligature out of the characters it is composed from, under the guidance of ligature/kerning programs found in the font.

More technically, what happens is that if the `\char` (or equivalent) for one slot is immediately followed by the `\char` (or equivalent) for another (or the same) slot and there is a ligaturing instruction in the `LIGKERN` table of the current font which applies to this slot pair then this ligaturing instruction is executed. This usually replaces the two slots in the pair with a single new slot specified by the ligaturing instruction (it could also keep one or both of the original slots, but that is less common). TEX has no idea about whether these replacements change the meaning of anything, but TEX assumes that it doesn't, and it is up to the font designer to ensure that this is the case.

Apart from forming ligatures in text, the ligaturing mechanism of TEX is traditionally also employed for another task which is much more problematic. Ligatures are also used to produce certain characters which are not part of visible ASCII — the most common are the endash (typed as `--`) and the emdash (typed as `---`). This is a problem because it violates TEX's assumption that the meaning is unchanged; the classical problem with this appears in the `OT2` encoding, where the Unicode character `U+0446` (CYRILLIC SMALL LETTER TSE) could be typed as `ts`, whilst the `t` and `s` by themselves produced Unicode characters `U+0442` (CYRILLIC SMALL LETTER TE) and `U+0441` (CYRILLIC SMALL LETTER ES) respectively. TEX's hyphenation mechanism can however decompose ligatures, so it sometimes happened that the TSE was hyphenated as TE-ES, which is quite different from what was intended. Since this is such an obvious disadvantage, the use of ligatures for forming non-English letters quickly disappeared after 8-bit input encodings became available. The practice still remains in use for punctuation, however, and the font designer must be aware of this. For many font encodings there is a set of ligatures which must be present and replace two or more characters by a single, different character. These ligatures are called *mandatory ligatures* in this paper.

The use of mandatory ligatures in new font encodings is strongly discouraged, for a number of reasons. The main problem is that they create unhealthy dependencies between input and output encoding, whereas these should ideally be totally independent. Using ligatures in this way complicates the internal representation of text, and it also makes it much harder to typeset text where those ligatures are not wanted (such as verbatim text). Furthermore it creates problems with kerning, since the "ligature" has not yet been formed when a kern to the left of it is inserted. Finally, a much better solution (when it is available) is to use an Omega translation process (see [9, Sec. 8–11]), since that *is* independent of the font, different translations can be combined, and they can easily handle even "abbreviations" much more complicated than those ligatures can deal with.

2.3 Output stages

On its way out of LaTeX towards the printed text, a character passes through a number of stages. The following five seem to cover what is relevant for the present discussion:

1. The *LaTeX Internal Character Representation* (LICR); see [8], Section 7.11, for a full description. At this point the character is a character token (e.g. `a`), a text command (e.g. `\ss`), or a combination (e.g. `\H{o}`).
2. *Horizontal material*; this is what the character is en route from TeX's mouth to its stomach. For most characters this is equivalent to a single `\char` command (e.g. `a` is equivalent to `\char 97`), but some require more than one, some are combined using the `\accent` and `\char` commands, some involve rules and/or kerns, and some are built using boxes that arbitrarily combine the above elements.
3. *DVI commands*; these are the DVI file commands that produce the printed representation of the character.
4. *Printed text*; this is the graphical representation of the character, e.g. as ink on paper or as a pattern on a computer screen. Here the text consists of glyphs.
5. *Interpreted text*; this is essentially printed text modulo equivalence of interpretation, hence the text doesn't really reach this stage until someone reads it. Here the text consists of characters.

In theory there is a universal mapping from LICR to interpreted text, but various technical restrictions make it impossible to simultaneously support the entire mapping. A LaTeX encoding selects a restriction of this mapping to a limited set which will be "well supported" (meaning that kerning and such between characters in the set works), whereas elements outside this set at best can be supported through temporary encoding changes. The encoding also specifies a decomposition of the mapping into one part which maps LICR to horizontal material and one part which maps horizontal material to interpreted text. The first part is realized by the text command definitions usually found in the '⟨*enc*⟩`enc.def`' file for the encoding. The second part is the font encoding, the specification of which is the topic of this paper. It is also worth noticing that an actual font is a mapping of horizontal material to printed text.

An alternative decomposition of the mapping from LICR to interpreted text would be at the DVI command level, but even though this decomposition is realized in most TeX implementations, it has very little relevance for the discussion of encodings. The main reason for this is that it depends not only on the encoding of a font, but also on its metrics. Furthermore it is worth noticing that in e.g. pdfTeX there need not be a DVI command level.

2.4 Hyphenation

There are strong connections between font encoding and hyphenation because TeX's hyphenation mechanism operates on horizontal material; more precisely, the hyphenation mechanism only works on pieces of horizontal material that are equivalent to sequences of `\char` commands. This implies that hyphenation patterns, as selected via the `\language` parameter, are not only for a specific language, they are also for a specific font encoding.

The hyphenation mechanism uses the `\lccode` values to distinguish between three types of slots:

1. lower case letters (`\lccode` $n = n$),
2. upper case letters (`\lccode` $n \notin \{0, n\}$), and
3. non-letters (`\lccode` $n = 0$).

Only the first two types can be part of a hyphenatable word and only lower case letters are needed in the hyphenation patters. This does however place severe restrictions on how letters can be placed in a text font because TeX uses the same `\lccode` values for all text in a paragraph and therefore these values cannot be changed whenever the encoding changes. In LaTeX the `\lccode` table is not allowed to change at all and consequently all text font encodings must work using the standard set of `\lccode` values.

In ε-TeX each set of hyphenation patterns has its own set of `\lccode` values for hyphenation, so the problem isn't as severe there. The hyphenation mechanism of Omega should become completely independent of the font encoding, although the last time I checked it was still operating on material encoded according to a font encoding.

2.5 Production and specification ETX files

Finally, it is worth pointing out the difference between an ETX file created for the specification of a font encoding and one created to be used in actually producing fonts with this encoding. They are usually not the same. Although specification ETXs certainly may be of direct use in the production of fonts — especially experimental fonts produced as part of the work on a new encoding — they are usually not ideal for the purpose. In particular there is often a need to switch between alternative names for a glyph to accommodate what is actually in the fonts, but such trickeries are undesirable complications in a specification. On the other hand a production ETX file has little need for verbose comments, whereas they are rather an advantage in a specification ETX file.

Therefore one shouldn't be surprised if there are two ETX files for a specific encoding: one which is a specification version and one which is a production version. If both might need to be in the same directory then one should, as a rule of thumb, include a '`spec`' in the name of the specification version.

3 Font encoding specifications

3.1 Basic principles

Most features of the font encoding are categorized as either *mandatory* or *ordinary*. The mandatory features are what macros may rely on, whereas the ordinary simply are something which fonts with this encoding normally provide. Font designers may choose to provide other features than the ordinary, but are recommended to provide the ordinary features to the extent that available resources permit.

Many internal references in the specification are in the form of *glyph names* and the choice of these is a slightly tricky matter. From the point of formal specification, the choices can be completely arbitrary, but from the point of practical usefulness they most likely are not. One of the main advantages of the ETX format for specifications is that such specifications can also be used to make experimental implementations, but this requires that the glyph names in the specification are the same as those used in the fonts from which the experimental implementation should be built. Yet another aspect is that the glyph names are best chosen to be the ones one can expect to find in actual fonts, as that will make things easier for other people that want to make non-experimental implementations later. For this last purpose, a good reference is Adobe's technical note on Unicode and glyph names [3]. For most common glyphs, [3] ends up recommending that one should follow the Adobe glyph list [2], which however has the peculiar trait of recommending names on the form `afii`*ddddd* (rather than the Unicode-based alternative `uni`*xxxx*) for most non-latin glyphs. This is somewhat put in perspective by [1].

3.2 Slot assignments

The purpose of the slot assignments is to specify for each slot the character or characters to which it is mapped. That one slot is mapped to many characters is an unfortunate, but not uncommon, reality in many encodings, as limitations in font size have often encouraged identifications of two characters which are almost the same. It should be avoided in new encodings.

Slot assignments are done using the `\nextslot` command and a `\setslot` ... `\endsetslot` construction as follows:

```
\nextslot{⟨slot number⟩}
\setslot{⟨glyph name⟩}
  ⟨slot commands⟩
\endsetslot
```

A typical example of this is

```
\nextslot{65}
\setslot{A}
  \Unicode{0041}{LATIN CAPITAL LETTER A}
\endsetslot
```

which gets typeset as

Slot 65 '`A`'
Unicode character `U+0041`, LATIN CAPITAL LETTER A.

The `\nextslot` command does not typeset anything; it simply stores the slot number in a counter, for later use by `\setslot`. The `\endsetslot` command increments this counter by one. Hence the `\nextslot` command is unnecessary between `\setslot`s for consecutive slots. Besides `\nextslot`, there is also a command `\skipslots` which increments the slot number counter by a specified amount. The argument of both `\nextslot` and `\skipslots` can be arbitrary fontinst integer expressions (see [5]). All TeX ⟨*number*⟩s that survive full expansion are valid fontinst integer expressions, but for example `` `\~ `` is not, as `\~` is a macro which will break before the expression is typeset. These cases can however be fixed by preceding the TeX ⟨*number*⟩ by `\number`, as `` \number`\~ `` survives full expansion by expanding to `126`.

The main duty of the ⟨*slot commands*⟩ is to specify the target character (or characters) for this slot. The simplest way of doing this is to use the `\Unicode` command, which has the syntax

```
\Unicode{⟨code point⟩}{⟨name⟩}
```

The ⟨*code point*⟩ is the number of the character (in hexadecimal notation, usually a four-digit number) and the ⟨*name*⟩ is the name. Case is insignificant in these arguments. If a slot corresponds to a string of characters rather than to a single character, then one uses the `\charseq` command, which has the syntax

```
\charseq{⟨\Unicode commands⟩}
```

e.g.

```
\nextslot{30}
\setslot{ffi}
  \charseq{
    \Unicode{0066}{LATIN SMALL LETTER F}
    \Unicode{0066}{LATIN SMALL LETTER F}
    \Unicode{0069}{LATIN SMALL LETTER I}
  }
\endsetslot
```

Several `\Unicode` commands not in the argument of a `\charseq` instead mean that each of the listed characters is a valid interpretation of the slot.

If a character cannot be specified in terms of Unicode code points then the specification should simply be a description in text which identifies the character. Such descriptions are written using the `\comment` command

```
\comment{⟨text⟩}
```

It is worth noticing that the ⟨*text*⟩ is technically only an argument of `\comment` when the program processing the ETX file is ignoring `\comment` commands. This means `\verb` and similar catcode-changing commands *can* be used in the ⟨*text*⟩. The `\par` command, on the other hand, is not allowed in the ⟨*text*⟩.

The `\comment` command should also be used for any further piece of explanation of or commentary to the character used for the slot, if the exposition seems to need it. There can be any number of `\comment` commands in the ⟨*slot commands*⟩.

3.3 Ligatures

There are three classes of ligatures in the font encoding specifications: mandatory, ordinary, and odd. Mandatory ligatures must be present in any font which complies with the encoding, whereas ordinary and odd ligatures need not be. No clear distinction can be made between ordinary and odd ligatures, but a non-mandatory ligature should be categorized as ordinary if it makes sense for the majority of users, and as odd otherwise. Hence the 'fi' ligature is categorized as ordinary in the `T1` encoding (although it makes no sense in Turkish), whereas the 'ij' ligature is odd.

In the ETX format, a ligature is specified using one of the slot commands

```
\Ligature{⟨ligtype⟩}{⟨right⟩}{⟨new⟩}
\ligature{⟨ligtype⟩}{⟨right⟩}{⟨new⟩}
\oddligature{⟨note⟩}{⟨ligtype⟩}
             {⟨right⟩}{⟨new⟩}
```

The `\Ligature` command is used for mandatory ligatures, `\ligature` for ordinary ligatures, and `\oddligature` for odd ligatures. The ⟨*right*⟩ and ⟨*new*⟩ arguments are names of the glyphs being assigned to the slots involved in this ligature. The ⟨*right*⟩ specifies the right part in the slot pair being affected by the ligature, whereas the left part is the one of the `\setslot` ... `\endsetslot` construction in which the ligaturing command is placed. The ⟨*new*⟩ specifies a new slot which will be inserted by the ligaturing instruction. The ⟨*ligtype*⟩ is the actual ligaturing instruction that will be used; it must be `LIG`, `/LIG`, `/LIG>`, `LIG/`, `LIG/>`, `/LIG/`, `/LIG/>`, or `/LIG/>>`. The slashes specify retention of the left or right original character; the `>` signs specify passing over that many slots in the result without further ligature processing. ⟨*note*⟩, finally, is a piece of text which explains when the odd ligature may be appropriate. It is typeset as a footnote.

As an example of ligatures we find the following in the specification of the `T1` encoding:

```
\nextslot{33}
\setslot{exclam}
  \Unicode{0021}{EXCLAMATION MARK}
  \Ligature{LIG}{quoteleft}{exclamdown}
\endsetslot
```

It is typeset as

Slot 33 `‘exclam’`
Unicode character `U+0021`, EXCLAMATION MARK.
Mandatory ligature `exclam`$*$`quoteleft` → `exclamdown`

With other ⟨*ligtype*⟩s there may be more names listed on the right hand side and possibly a '⌊' symbol showing the position at which ligature processing will start afterwards.

3.4 Math font specialities

There are numerous technicalities which are special to math fonts, but only a few of them are exhibited in ETX files.[2] Most of these have to do with the TEX mechanisms that find sufficiently large characters for commands like `\left`, `\sqrt`, and `\widetilde`.

The first mechanism for this is that a character in a font can sort of say "If I'm too small, then try character ... instead". This is expressed in an ETX file using the `\nextlarger` command, which has the syntax

```
\nextlarger{⟨glyph name⟩}
```

The second mechanism constructs a sufficiently large character from smaller pieces; this is known as a 'varchar' or 'extensible character'. This is expressed in an ETX file using an "extensible recipe", the syntax for which is

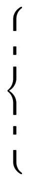
```
\varchar ⟨varchar commands⟩ \endvarchar
```

where each ⟨*varchar command*⟩ is one of

```
\varrep{⟨glyph name⟩}
\vartop{⟨glyph name⟩}
\varmid{⟨glyph name⟩}
\varbot{⟨glyph name⟩}
```

There can be at most one of each and their order is irrelevant. The most important is the `\varrep` command, as that is the part which is repeated until the character is sufficiently large. The `\vartop`, `\varmid`, and `\varbot` commands are used to specify some other part which should be put at the top, middle, and bottom of the extensible character respectively. Not all extensible recipes use all of these, however.

As an example, here is how a very large left brace is constructed:



For `\vartop{bracelefttp}`
For `\varrep{braceex}`
For `\varmid{braceleftmid}`
Again for `\varrep{braceex}`
For `\varbot{braceleftbt}`

Both `\nextlarger` and `\varchar` commands are like `\ligature` in that they describe ordinary features for the encoding; they appear in a specification ETX file mainly to explain the purpose of some ordinary character. There is no such thing as a mandatory `\nextlarger` or `\varchar`, but varchars are occasionally used to a similar effect. In these cases, the character generated by the extensible recipe is something quite different from what a `\char` for that slot would produce. Thus for the slot to produce the expected result it must be referenced using a `\delimiter` or `\radical` primitive, since those are the only ones which make use of the extensible recipe. The effect is that the slot has a *semimandatory* assignment; the result of `\char` is unspecified (as for a slot with an ordinary assignment), but the result for a large delimiter or radical is not (as for a slot with a mandatory assignment).

[2] For an overview of the subject, see for example Vieth [10].

Thus some math fonts have an extra section "Semimandatory characters" between the mandatory and ordinary character sections. In that section for the `OMX` encoding we find for example

```
\nextslot{60}
\setslot{braceleftmid}
  \Unicode{2016}{DOUBLE VERTICAL LINE}
  \comment{This is the large size of the
     |\Arrowvert| delimiter, a glyphic
     variation on |\Vert|. The
     \texttt{braceleftmid} glyph
     ordinarily placed in this slot must
     not be too tall, or else the
     extensible recipe actually
     producing the character might
     sometimes not be used.}
  \varchar
    \varrep{arrowvertex}
  \endvarchar
\endsetslot
```

which is typeset as

> **Slot 60 '`braceleftmid`'**
> Unicode character `U+2016`, DOUBLE VERTICAL LINE.
> This is the large size of the `\Arrowvert` delimiter, a glyphic variation on `\Vert`. The `braceleftmid` glyph ordinarily placed in this slot must not be too tall, or else the extensible recipe actually producing the character might sometimes not be used.
> **Extensible glyph:**
> **Repeated** `arrowvertex`

3.5 Fontdimens

Each TEX font contains a list of fontdimens, numbered from 1 and up, which are accessible via the `\fontdimen` TEX primitive. Quite a few are also used implicitly by TEX and therefore cannot be left out even if they are totally irrelevant, but as one can always include some extra fontdimens in a font — the only bounds on how many fontdimens there may be are the general bound on the size of a TFM file and the amount of font memory TEX has available — this is usually not a problem.

The reason fontdimens are part of font encoding specifications is that the meaning of e.g. `\fontdimen 8` varies between different fonts depending on their encoding; thus the encoding specification must define the quantity stored in each `\fontdimen` parameter. This is done using the `\setfontdimen` command, which has the syntax

`\setfontdimen{`⟨*number*⟩`}{`⟨*name*⟩`}`

The ⟨*number*⟩ is the fontdimen number (as a sequence of decimal digits where the first digit isn't zero) and the ⟨*name*⟩ is a symbolic name for the quantity.

The standard list of symbolic names for fontdimen quantities appears below; the listed quantities should always be described using the names in this list. Encoding specifications that employ other quantities as fontdimens should include definitions of these quantities. Those quantities that are defined as "Formula parameter …" have to do with how mathematical formulae are rendered and are beyond our scope here. For exact definitions of these parameters, the reader is referred to Appendix G of *The TEXbook* [7].

acccapheight The height of accented full capitals.

ascender The height of lower case letters with ascenders.

axisheight Formula parameter σ_{22}.

baselineskip The font designer's recommendation for natural length of the TEX parameter `\baselineskip`.

bigopspacing1 Formula parameter ξ_9.

bigopspacing2 Formula parameter ξ_{10}.

bigopspacing3 Formula parameter ξ_{11}.

bigopspacing4 Formula parameter ξ_{12}.

bigopspacing5 Formula parameter ξ_{13}.

capheight The height of full capitals.

defaultrulethickness Formula parameter ξ_8.

delim1 Formula parameter σ_{20}.

delim2 Formula parameter σ_{21}.

denom1 Formula parameter σ_{11}.

denom2 Formula parameter σ_{12}.

descender The depth of lower case letters with descenders.

digitwidth The median width of the digits in the font.

extraspace The natural width of extra interword glue at the end of a sentence. TEX implicitly uses this parameter if `\spacefactor` is 2000 or more and `\xspaceskip` is zero.

interword The natural width of interword glue (spaces). TEX implicitly uses this parameter unless `\spaceskip` is nonzero.

italicslant The slant per point of the font. Unlike all other fontdimens, it is not proportional to the font size.

maxdepth The maximal depth over all slots in the font.

maxheight The maximal height over all slots in the font.

num1 Formula parameter σ_8.

num2 Formula parameter σ_9.

num3 Formula parameter σ_{10}.

quad The quad width of the font, normally approximately equal to the font size and/or the width of an 'M'. Also implicitly available as the length unit `em` and used for determining the size of the length unit `mu`.

shrinkword The (finite) shrink component of interword glue (spaces). TEX implicitly uses this parameter unless `\spaceskip` is nonzero.

stretchword The (finite) stretch component of interword glue (spaces). TEX implicitly uses this parameter unless `\spaceskip` is nonzero.

sub1 Formula parameter σ_{16}.

sub2 Formula parameter σ_{17}.

subdrop Formula parameter σ_{19}.

sup1 Formula parameter σ_{13}.

sup2 Formula parameter σ_{14}.

sup3 Formula parameter σ_{15}.

supdrop Formula parameter σ_{18}.

verticalstem The dominant width of vertical stems. This quantity is meant to be used as a measure of how "dark" the font is.

xheight The x-height (height of lower case letters without ascenders). Also implicitly available as the length unit `ex`.

3.6 The codingscheme

The final encoding-dependent piece of information in a TEX font is the codingscheme, which is essentially a string declaring what encoding the font has. This information is currently only used by programs that convert the information in a TEX font to some other format and these use it to identify the glyphs in the font. Therefore this string should be chosen so that the contents of the slots in the font can be positively identified. Observe that the encoding specification by itself does not provide enough information for this, since there are usually a couple of slots that do not contain mandatory characters. On the other hand, it is not a problem in this context if the font leaves some of the slots (even mandatory ones) empty as that is anyway easily detected. The only problem is with fonts where the slots are assigned to other characters than the ones specified in the encoding.

For that reason, it is appropriate to assign two codingscheme strings to each encoding. The main codingscheme is for fonts where all slots (mandatory and ordinary alike) have been assigned according to the specification or have been left empty. The variant codingscheme is for fonts where some ordinary slots have been assigned other characters than the ones listed in the specification, but where the mandatory slots are still assigned according to the specification or are left empty. The font encoding specification should give the main codingscheme name, whereas the variant codingscheme name could be formed by adding `␣VARIANT` to the main codingscheme name.

Technically the codingscheme is specified by setting the `codingscheme` string variable. This has the syntax

```
\setstr{codingscheme}
{⟨codingscheme name⟩}
```

e.g.

```
\setstr{codingscheme}
{EXTENDED TEX FONT ENCODING - LATIN}
```

which is typeset as (line break is editorial)

Default s(`codingscheme`) = `EXTENDED␣TEX`
`␣FONT␣ENCODING␣-␣LATIN`

A codingscheme name may be at most 40 characters long and may not contain parentheses. If the entire `␣VARIANT` cannot be suffixed to a main name because the result becomes too long (as in the above example) then use the first 40 characters of the result.

3.7 Overall document structure

The overall structure of a font encoding specification should be roughly the following

```
\relax
\documentclass[twocolumn]{article}
\usepackage[specification]{fontdoc}
⟨preamble⟩
\begin{document}
⟨title⟩
⟨manifest⟩
\encoding
⟨body⟩
\endencoding
⟨discussion⟩
⟨change history⟩
⟨bibliography⟩
\end{document}
```

The commands described in the preceding subsections must all go in the ⟨*body*⟩ part of the document, as that is the only part of the file which actually gets processed as a data file. The part before `\encoding` is skipped and the part after `\endencoding` is never even input, so whatever appears there is only part of the LaTeX document. For the purposes of processing as a data file, the important markers in the file are the `\relax`, the `\encoding`, and the `\endencoding` commands.

The ⟨*title*⟩ is the usual `\maketitle` (and the like) stuff. The person or persons who appear as author(s) are elsewhere in this paper described as the *encoding proposers*. The ⟨*title*⟩ should also give the date when the specification was last changed.

The ⟨*manifest*⟩ is an important, although usually pretty short, part of the specification. It is a piece of text which explains the purpose of the encoding (in particular what it can be used for) and the basic ideas (if any) which have been used in its construction. It is often best marked up as an abstract.

The ⟨*discussion*⟩ is the place for any longer comments on the encoding, such as analyses of different implementations, comparisons with other encodings, etc. This is also the place to explain any more general structures in the encoding, such as the arrow kit in the proposed `MS2` encoding [4]. In cases where the specification is mainly a formulation of what is already an established standard the ⟨*discussion*⟩ is often rather short as the relevant discussion has already been published elsewhere, but it is anyway a service to the reader to include this information. References to the original documents should always be given.

It might be convenient to include an FAQ section at the end of the discussion. This is particularly suited for explaining things where one has to look for a while and consult the references to find the relevant information.

The ⟨*change history*⟩ documents how the specification has changed over time. It is preferably detailed, as each detail in an encoding is important, but one should not be surprised if it is nevertheless rather short due to there not having been that many changes.

The ⟨*bibliography*⟩ is an important part of the specification. It should at the very least include all the sources which have been used in compiling the encoding specification, regardless of whether they are printed, available on the net, merely "personal communication", or something else. It is also a service to the reader to include in the bibliography some more general references for related matters.

The ⟨*preamble*⟩ is just a normal LaTeX preamble and there are no restrictions on defining new commands in it, although use of such commands in the ⟨*body*⟩ part is subject to the same restrictions as use of any general LaTeX command. The preamble should however *not* load any packages not part of the required suite of LaTeX packages, as that may prevent users who do not have these packages from typesetting the specification. Likewise, the specification should *not* require that some special font is available. Glyph examples for characters are usually better referenced via Unicode character charts than via special fonts.

An exception to this rule about packages is that the specification must load the fontdoc package, as shown in the outline above, since that defines the `\setslot` etc. commands that should appear in the ⟨*body*⟩. This should not cause any problems, as the fontdoc package can preferably be kept in the same directory as the collection of encoding specifications (see below). The `specification` option should be passed to the package to let it know that the file being processed is an encoding specification — otherwise `\Ligature` and `\ligature` will get the same formatting, for one. It is not actually necessary to use the article document class, and neither must it be passed the `twocolumn` option, but it is customary to do so. In principle any other document class defined in required LaTeX will do just as well.

If you absolutely think that using some non-required package significantly improves the specification, then try writing the code so that it loads the package only if it is available and provide some kind of fallback definition for sites where it is not. E.g. the url package could be loaded as

```
\IfFileExists{url.sty}{\usepackage{url}}{}
\providecommand\url{\verb}
```

The `\url` command defined by this is not equivalent to the command defined by the url package, but it can serve fairly well (with a couple of extra overfull lines as the only ill effect) if its use is somewhat restricted.

Finally, a technical restriction on the ⟨*preamble*⟩, ⟨*title*⟩, and ⟨*manifest*⟩ is that they must not contain any mismatched `\if`s (of any type) or `\fi`s, as TeX conditionals will be used for skipping those parts of the file when it is processed as a data file. If the definition of some macro includes mismatched `\if`s or `\fi`s (this will probably occur only rarely) then include some extra code so that they do match.

3.8 Encoding specification body syntax

The ⟨*body*⟩ part of an encoding specification must adhere to a much stricter syntax than the rest of the file. The ⟨*body*⟩ is a sequence of ⟨*encoding command*⟩s, each of which should be one of the following:

```
\setslot{⟨glyph name⟩} ⟨slot commands⟩
\endsetslot
\nextslot{⟨number⟩}
\skipslots{⟨number⟩}
\setfontdimen{⟨number⟩}{⟨name⟩}
\setstr{codingscheme}
              {⟨codingscheme name⟩}
\needsfontinstversion{⟨version number⟩}
```

The `\needsfontinstversion` command is usually placed immediately after the `\encoding` command. The ⟨*version number*⟩ must be at least `1.918` for many of the features described in this file to be available, and at least `1.928` if the `\charseq` command is used.

The ⟨*slot commands*⟩ are likewise a sequence of ⟨*slot command*⟩s, each of which should be one of the following:

```
\Unicode{⟨code point⟩}{⟨name⟩}
\charseq{⟨\Unicode commands⟩}
\comment{⟨text⟩}
\Ligature{⟨ligtype⟩}{⟨right⟩}{⟨new⟩}
\ligature{⟨ligtype⟩}{⟨right⟩}{⟨new⟩}
\oddligature{⟨note⟩}{⟨ligtype⟩}
              {⟨right⟩}{⟨new⟩}
\nextlarger{⟨glyph name⟩}
\varchar ⟨varchar commands⟩ \endvarchar
```

where ⟨*varchar commands*⟩ is similarly a sequence of ⟨*varchar command*⟩s, each of which should be one of the following:

```
\varrep{⟨glyph name⟩}
\vartop{⟨glyph name⟩}
\varmid{⟨glyph name⟩}
\varbot{⟨glyph name⟩}
```

Finally, one can include any number of ⟨*comment command*⟩s between any two encoding, slot, or varchar commands. The comment commands are

```
\begincomment ⟨LaTeX text⟩ \endcomment
\label{⟨reference label⟩}
```

The ⟨*LaTeX text*⟩ can be pretty much any LaTeX code that can appear in conditional text. (`\begincomment` is either `\iffalse` or `\iftrue` depending on whether the encoding specification is processed as a data file or typeset as a LaTeX document respectively. `\endcomment` is always `\fi`.) The `\label` command is just the normal LaTeX `\label` command; when it is used in a ⟨*slot commands*⟩ string it references that particular slot (by number and glyph name).

The full syntax of the ETX format can be found in the fontinst manual [5], but font encoding specifications only need a subset of that.

3.9 Additional fontdoc features

The `\textunicode` command is an "in comment paragraph" form of `\Unicode`. Both commands have the same syntax, but `\textunicode` is only allowed in "comment" contexts. A typical use of `\textunicode` is

```
\comment{An ...
... this is
\textunicode{2012}{FIGURE DASH}; in ...
}
```

which is typeset as

An ... this is U+2012 (FIGURE DASH); in ...

The fontdoc package inputs a configuration file `fontdoc.cfg` if that exists. This can be used to pass additional options to the package. The only currently available options that may be of interest are the `hypertex` and `pdftex` options, which hyperlinks each U+... generated by `\Unicode` or `\textunicode` (using HyperTEX or pdfTEX conventions[3] respectively) to a corresponding glyph image on the Unicode consortium website. To use this feature one should put the line

```
\ExecuteOptions{hypertex}
```

or

```
\ExecuteOptions{pdftex}
```

in the `fontdoc.cfg` file. *Please* do not include this option in the `\usepackage{fontdoc}` of an encoding specification file as that can be a severe annoyance for people whose TEX program or DVI viewers do not support the necessary extensions.

4 Font encoding ratification

This section describes a suggested ratification process for font encoding specifications. As there are fewer technical matters that impose restrictions on what it may look like, it is probably more subjective than the other parts of this paper.

A specification in the process of being ratified can be in one of three different stages: *draft*, *beta*, or *final*. Initially the specification is in the draft stage, during which it will be scrutinized and can be subject to major changes. A specification which is in the beta stage has received a formal approval but the encoding in question may still be subject to some minor changes if weighty arguments present themselves. Once the specification has reached the final stage, the encoding may not change at all.

[3] One could just as well do the same thing using some other convention if a suitable definition of `\FD@codepoint` is included in `fontdoc.cfg`. See the fontinst sources [6] for more details.

4.1 Getting to the draft stage

The process of taking an encoding to the draft stage can be summarized in the following steps. Being in the draft stage doesn't really say anything about whether the encoding is in any way correct or useful, except that some people (the encoding proposers) believe it is and are willing to spend some time on ratifying it.

Write an encoding specification The first step is to write a specification for the font encoding in question. This document must not only technically describe the encoding but also explain what the encoding is for and why it was created. See Subsection 3.7 for details on how the document is preferably organised.

Request an encoding name The second step is to write to the LATEX3 project and request a LATEX encoding name for the encoding. This mail should be in the form of a LATEX bug report, it must be sent to

`latex-bugs@latex-project.org`,

and it must include the encoding specification file. Suggestions for an encoding name are appreciated, but not necessarily accepted. The purpose of this mail is *not* to get an approval of the encoding, but only to have a reasonable name assigned to it.

Upload the specification to CTAN The third step is make the encoding specification publicly available by uploading it to CTAN. Encoding specifications are collected in the

`info/encodings`

directory (which should also contain the most recent version of this paper). The name of the uploaded file should be '⟨*encoding name*⟩`draft.etx`'. The reason for this naming is that it must be clear that the specification has not yet been ratified.

Announce the encoding When the upload has been confirmed, it is time to announce the encoding by posting a message about it to the relevant forums. Most important is the `tex-fonts` mailing list, since that is where new encodings should be debated. Messages should also be posted to the `comp.text.tex` newsgroup and any forums related to the intended use of the encoding: an encoding for Sanskrit should be announced on Indian TEX users forums, an encoding for printing chess positions should be announced on some chess-with-TEX user forum, etc., to the extent that such forums exist.

The full address of the `tex-fonts` mailing list is

`tex-fonts@math.utah.edu`

This list rejects postings from non-members, so you need to subscribe to it before you can post your announcement. This is done by sending a 'subscribe me' mail to

`tex-fonts-request@math.utah.edu`

The list archives can be found at

HTTP:`//www.math.utah.edu/mailman/listinfo/tex-fonts`

A tip is to read through the messages from a couple of months before you write up your announcement, as that should help you get acquainted with the normal style on the list. Please do not send messages encoded in markup languages (notably, HTML, XML, and word processor formats) to the list.

Experimental encodings There is a point in going through the above procedure even for experimental encodings, i.e., encodings whose names start with an `E`. Of course there is no point in ratifying a specification of an experimental encoding, as it is very likely to frequently change, but having a proper name assigned to the encoding and uploading its specification to CTAN makes it much simpler for other people to learn about and make references to the encoding.

4.2 From draft to beta stage

The main difference between a draft and beta stage specification respectively is that beta stage specifications have been scrutinized by other people and found to be free of errors. The practical implementation of this is that a debate is held (in the normal anarchical manner of mailing list debates) on the `tex-fonts` mailing list. In particular the following aspects of the specification should be checked:

1. *Is the encoding technically correct?* There are many factors which affect what TEX does and it is easy to overlook some. (The `\lccode`s seem to be particularly troublesome in this respect.) Sometimes fonts simply cannot work as an encoding specifies they should and it is important that such defects in the encoding are discovered on an early stage.
2. *Are there any errors in the specification?* A font encoding specification is largely a table and typos are easy to make. Proofreading may be boring, but it is very, very important.
3. *Is the specification sufficiently precise?* Are there any omissions, ambiguities, inaccuracies, or completely irrelevant material in the specification? There shouldn't be.

During the debate, the encoding proposers should hear what other people have to say about the encoding draft, revise it accordingly when some flaw is pointed out, and upload the revised version. This cycle may well have to be repeated several times before everyone is content. It is worth pointing out that in practice the debate should turn out to be more of a collective authoring of the specification than a defense of its validity. There is no point in going into it expecting the worst.

Unfortunately, it might happen that there never is a complete agreement on an encoding specification — depending on what side one takes, either the encoding proposers refuse to correct obvious flaws in it, or someone on the list insists that there is a flaw although there is obviously not — but hopefully that will never happen. If it anyway does happen then the person objecting should send a mail whose subject contains the phrase "formal protest against XXX encoding" (with XXX replaced by whatever the encoding is called) to the list. Then it will be up to the powers that be to decide on the fate of the encoding (see below).

Summarize the debate When the debate on the encoding is over — e.g. a month after anyone last posted anything new on the subject — then the encoding proposers should summarize the debate on the encoding specification draft and post this summary as a follow-up on the original mail to `latex-bugs`. This summary should list the changes that have been made to the encoding, what suggestions there were for changes which have not been included, and whether there were any formal protests against the encoding. The summary should also explain what the proposers want to have done with the encoding. In the usual case this is having it advanced to beta stage, but the proposers might alternatively at this point have reached the conclusion that the encoding wasn't such a good idea to start with and therefore withdraw it, possibly to come again later with a different proposal.

In response to this summary, the LATEX-project people may do one of three things:

- If the proposers want the encoding specification advanced and there are no formal protests against this, then the encoding should be advanced to the beta stage. The LaTeX-project people do this by adding the encoding to the list of approved (beta or final stage) encodings that they [presumably] maintain.

- If the proposers want to withdraw the encoding specification then the name assigned to it should once again be made available for use for other encodings.
- If the proposers want the encoding specification advanced but there is some formal protest against this, then the entire matter should be handed over to some suitable authority, as a suggestion some technical TUG committee, for resolution.

Update the specification on CTAN When the specification has reached the beta stage, its file on CTAN should be updated to say so. In particular the file name should be changed from '⟨*encoding name*⟩`draft.etx`' to '⟨*encoding name*⟩`spec.etx`'.

Modifying beta stage encodings If a beta stage encoding is modified then the revised specification should go through the above procedure of ratification again before it can replace the previous '⟨*encoding name*⟩`spec.etx`' file on CTAN. The revised version should thus initially be uploaded as ⟨*encoding name*⟩`draft.etx`, reannounced, and redebated. It can however be expected that such debates will not be as extensive as the original debates.

4.3 From beta stage to final stage

The requirements for going from beta stage to final stage are more about showing that the encoding has reached a certain maturity than about demonstrating technical merits. The main difference in usefulness between a beta stage encoding and a final stage encoding is that the latter can be considered safe for archival purposes, whereas one should have certain reservations against such use of beta stage encodings.

It seems reasonable that the following conditions should have to be fulfilled before a beta stage encoding can be made a final stage encoding:

- At least one year must have passed since the last change was made to the specification.
- At least two people other than the proposer must have succeeded in implementing the encoding in a font.

It is quite possible that some condition should be added or some of the above conditions reformulated.

References

[1] Adobe Systems Incorporated: *Adobe Standard Cyrillic Font Specification*, Adobe Technical Note #5013, 1998; HTTP://`www.adobe.com/devnet/font/pdfs/5013.Cyrillic_Font_Spec.pdf`.

[2] Adobe Systems Incorporated: *Adobe Glyph List*, text file, 2002, HTTP://`www.adobe.com/devnet/opentype/archives/glyphlist.txt`.

[3] Adobe Systems Incorporated: *Adobe Solutions Network: Unicode and Glyph Names*, web page, 2007, HTTP://`www.adobe.com/devnet/opentype/archives/unicodegn.html`.

[4] Matthias Clasen and Ulrik Vieth: *Towards a New Math Font Encoding for (La)TeX*, March 1998, presented at EuroTeX'98; HTTP://`tug.org/twg/mfg/papers/current/mfg-euro-all.ps.gz`.

[5] Alan Jeffrey, Rowland McDonnell, Ulrik Vieth, and Lars Hellström: *fontinst — font installation software for TeX* (manual), 2004, CTAN:`fonts/utilities/fontinst/doc/manual/fontinst.pdf`.

[6] Alan Jeffrey, Sebastian Rahtz, Ulrik Vieth, and Lars Hellström: *The fontinst utility*, documented source code, v 1.9xx, CTAN:`fonts/utilities/fontinst/source/`.

[7] Donald E. Knuth, Duane Bibby (illustrations): *The TeXbook*, Addison-Wesley, 1991; ISBN 0-201-13447-0, volume A of *Computers and Typesetting.*

[8] Frank Mittelbach and Michel Goossens, with Johannes Braams, David Carlisle, and Chris Rowley: *The LaTeX Companion* (second edition), Addison–Wesley, 2004; ISBN 0-201-36299-6.

[9] John Plaice and Yannis Haralambous: *Draft documentation for the Omega system*, version 1.12, 1999; HTTP://`omega.enstb.org/roadmap/doc-1.12.ps`.

[10] Ulrik Vieth: *Math typesetting in TeX: The good, the bad, the ugly*, in the proceedings of EuroTeX 2001; HTTP://`www.ntg.nl/maps/pdf/26_27.pdf`.

◇ Lars Hellström
LaTeX3 project